A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs

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

McGill University, Montréal, Québec, Canada

Abstract. In this paper, we present a new register allocation framework based on *hierarchical cyclic interval graphs*. We motivate our approach by demonstrating that cyclic interval graphs provide a feasible and effective representation to characterize sequences of live ranges of variables in successive iterations of a loop. Based on this representation we provide a new heuristic algorithm for minimum register allocation, the *fat cover* algorithm. In addition, we present a spilling algorithm 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 demonstrate the effectiveness of our approach on a collection of loops by comparing the results of our algorithm to the results produced by three state-of-the-art optimizing compilers.

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 [13]. Furthermore, as VLSI technology permits the integration of more and more general-purpose registers on a processor chip, the benefit of keeping variables in registers will increase.

Register allocators in many modern compilers employ the classical graph coloring method originally proposed by Chaitin [4, 5, 6, 2]. With this method, an *interference graph* is built to direct register allocation and 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. This *interference graph* representation of the register allocation problem approach has some weaknesses. The most notable is that 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. This lack of information about the exact relationships among live ranges leads to some limitations. For example, it is known that Chaitin's heuristic may fail to find the minimum coloring even for some simple cases. In addition, 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

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

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. To present a complete framework for our approach, we have structured our paper as follows. In Section 2 we discuss the background and challenges of the register allocation problem in more detail. In Section 3 we formally introduce the problem statement and our alternative representation, cyclic interval graphs. Using the additional information available with this representation we present a new heuristic algorithm for minimum register allocation, the *fat cover* algorithm (Section 4), and a new spilling algorithm (section 5). We briefly discuss how our scheme naturally extends to a hierarchical method that can handle nested loops and conditionals in Section 6. In Section 7 we demonstrate the effectiveness of our approach by comparing our method to the register allocators used in state-of-the-art optimizing compilers (the SUN Sparc C compiler, the MIPS C compiler, and the IBM RS6000 compiler). Finally, we present our conclusions in Section 8.

2 Register Allocation: Background and Challenges

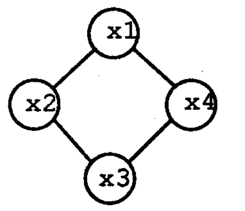
In this section, we present a simple motivating example to illustrate the weaknesses of the classical interference graph approach. We then outline how these problems can be handled by an alternative representation, cyclic interval graphs.

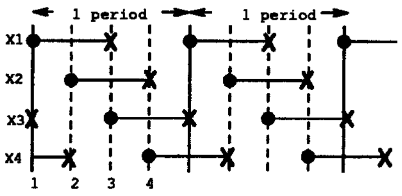
2.1 Interference Graphs and Chaitin's Heuristics

The traditional register allocation 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 [5], 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. The simple program given in Figure 1(a) illustrates two problems with the traditional approach: (1) lack of optimality, and (2) how to represent live ranges of loop variables.

```
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

Fig. 1. An Interference Graph and an Interval Graph

Let us first address the problem of optimality. 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. For example, consider the interference graph given in Figure 1(b).[2] 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.

The other challenge is how to represent the live ranges of loop variables. Our example in 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'.

2.2 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 in 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*. We formally define this notion of cyclic intervals in the next section.

[2] This interference graph is similar to the one presented in [2].

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.

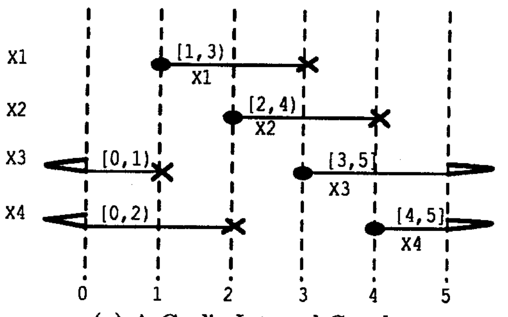
3.1 Lifetime Intervals and Cyclic Interval Graphs

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.

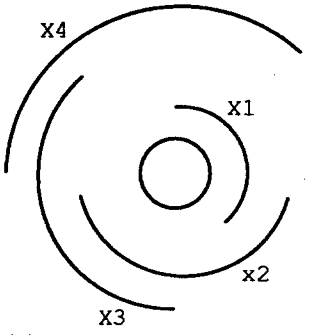
We assume that each machine operation is in the form of a quadruple, e.g. `x = y + z`, which begins at some time point t. To be precise, we say that variable `x` is defined at time point t. The live range of `x` will continue to the time point t', $(t' > t)$, where it is last used in a statement, e.g. `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 into one period. Note that the times 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 Cyclic Interval Graph



(b) A Circular Arc Graph

Fig. 2. A Cyclic Interval Graph

3.2 The Problem Statement

We can now state the main problems to be studied in this paper.

Problem 1 (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 (k-coloring of a Cyclic Interval Graph): 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. Firstly, 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 [15]. Secondly, 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 [14, 10]. 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 $O(nk!klogk)$ [10]. 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 [10]. 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 define formally below.

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

Definition 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 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 is intuitively clear, and proofs are presented fully in [12]. The following theorem addresses the problem of optimal coloring of acyclic interval graphs.

Theorem 5. *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 6. *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.*

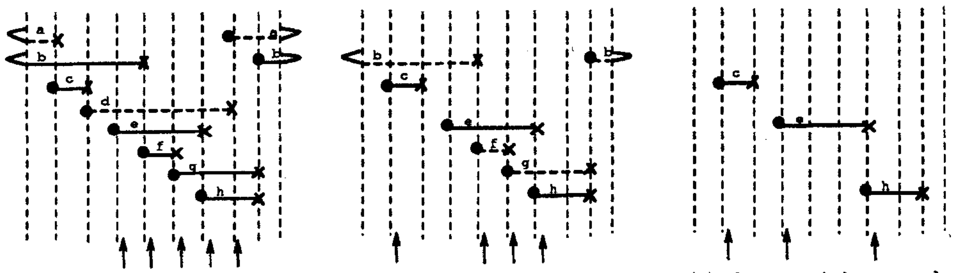
4 Finding a Minimal Coloring of Cyclic Interval Graphs

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 6), 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 cover 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 covers all of the fat spots and 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 non-cyclic 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

Fig. 3. An example of the fat cover algorithm

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.

Definition 7. 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 8. 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 9. *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 8. 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 8. Secondly, it is clear that by removing F from G, we are left with a graph that is colored with $k - 1$ colors.

Our *fat cover* algorithm is inspired by Theorems 5 and 9. 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 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 (Theorem 5). 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[4]. 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 directly 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 $I_m, I_{(m-1)}, \ldots, I_1$.

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 [12].

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 $k' > k$. We can summarize the main features and advantages of our cyclic interval graphs approach as follows.

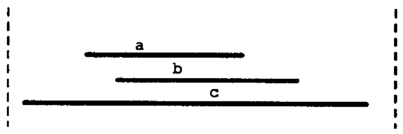
Separation of the spill phase from the coloring phase: Given that we developed a good algorithm for coloring a graph G with maximum thickness $W_{max}(G)$, we take the approach that the register allocation should proceed in two phases. Given $k < W_{max}(G)$, the first phase transforms G to an equivalent 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. It should be noted that the approach given in [7] 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. Some of 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 [1]. In this approach the "width" (number of variables live at the same time) of the interference graph was used for one of the spill heuristics. 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 exploit this information.

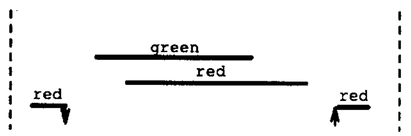
Register Floats: Our approach uses a two-level mechanism: (1) floating registers and (2) spilling registers. A register float corresponds to moving a value from one register to another register, while a register spill corresponds to moving a value from a register to a memory location and back. Clearly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

5.1 Chameleon Intervals, Register Floats, and Register Spills

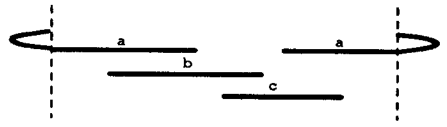
By carefully studying the structure of cyclic interval graphs, one can see that there are two quite different constraints that make a graph not colorable in k registers. The first is the most evident. If a graph G has some time, t_i, where there are more than k intervals covering t_i, then it is impossible to allocate a different color to each interval at t_i. For example, consider the graph given in Figure 4(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 4(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.[3]



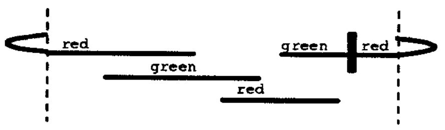
(a) a graph with $W_{max} = 3$ *(3-colorable)*



(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)*

Fig. 4. An Example of Register Spilling and Register Floating

[3] Note that it is important to choose c to spill. Spilling either a or b would not result in reducing the $W_{max}(G)$ to 2.

The second situation is more subtle. Consider the graph given in Figure 4(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 4(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.[4]

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 4(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 4(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.[5] 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.

[4] 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.

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

Thus, the only difficulty is to develop 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.[6] 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 structures.

Let us first consider the case of nested loops as illustrated in Figure 5(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 5(a). As shown in Figure 5(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.

Nested conditionals can also be handled in a hierarchical fashion using the concept of cyclic interval graphs. We create a structure similar to nested loops by introducing the proper constraints between the two branches in the conditional. We refer the interested reader to [12] for more details of this process.

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,

[6] 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'.

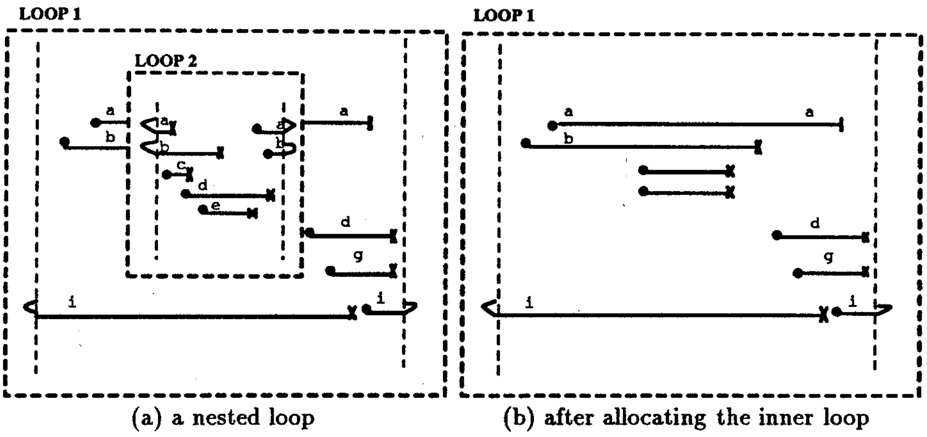


Fig. 5. Nested Loops as Hierarchical Structured Interval Graphs

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 is the Livermore Loop 8 the other is taken from the Tomcat[7] SPEC 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.

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` is 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.

Figure 6 compares the result of applying the *sweep and split* spilling algorithm (the column labeled IntGr) to the performance of the production compilers. Let us first consider the case when only 16 floating point registers are available. As can be seen in Figure 6(a) and Figure 6(b), 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

[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.

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

(a) 16 Registers - 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

(b) 16 Registers - Loop 8

	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

(c) 32 Registers - Tomcat and Loop 8

Fig. 6. Number of Double Precision Loads and Stores

Figure 6(c) gives analogous results when 32 registers are available on the RS6000. The increased number of registers alleviates the need for many of the spills. In fact, the interval graph method allows Loop 8 and the rolled version of Tomcat to execute with no load or store spills. 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.) Referring back to the 16 register tests we note the largest (absolute) reduction in spills occurred in for the case of 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 one 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 Figure 7.

	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

Fig. 7. Interval Graph Statistics (after Spilling).

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.

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 [16][17]. He also noted that the related concept of circular arc graphs could be applied to program loops. Other previous uses of interval graphs include: (1) channel routing in VLSI layouts [3][8], and, (2) the computation of overlays for arrays to minimize program memory requirements [9]. 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 [1]. A great deal of theoretical work has been done, a good summary of which may be found in [11].

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, 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. Finally, we have quantitatively demonstrated the effectiveness of our spilling and coloring algorithms by comparing our results to three production compilers.

References

1. 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.
2. 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.
3. M. Burstein. Channel Routing. In T. Ohtsuki, editor, *Layout Design and Verification*, pages 132–167. North-Holland, 1986.
4. 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.
5. 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.
6. 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.
7. 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.
8. I. Dagan, M.C. Golumbic, and R.Y. Pinter. Trapezoid Graphs and their Coloring. *Discrete Applied Mathematics*, 21:35–46, 1988.
9. J. Fabri. *Automatic Storage Optimization.* PhD thesis, University of Michigan, 1982.
10. 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.
11. M.C. Golumbic. Interval Graphs and Related Topics. *Discrete Mathematics*, 55(2):113–121, 1985.
12. Laurie J. Hendren, Guang R. Gao, Erik Altman, and Chandrika Mukerji. Register allocation using cyclic interval graphs. Technical Report ACAPS-MEMO 33, McGill University, 1991.
13. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.
14. V. Klee. What Are the Intersection Graphs of Arcs in a Circle? *American Mathematics Monthly*, 76:810–813, 1969.
15. 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.
16. A. Tucker. Coloring a Family of Circular Arcs. *SIAM Journal of Applied Mathematics*, 29(3):493–502, November 1975.
17. Alan Tucker. *Applied Combinatorics.* John Wiley and Sons, Inc., 2nd edition, 1984.