Spill Code Minimization via Interference Region Spilling [*]

Peter Bergner[†] Peter Dahl[‡] David Engebretsen Matthew O'Keefe

University of Minnesota
Department of Electrical Engineering
Minneapolis, MN 55455
bergner@vnet.ibm.com

Abstract

Many optimizing compilers perform global register allocation using a Chaitin-style graph coloring algorithm. Live ranges that cannot be allocated to registers are spilled to memory. The amount of code required to spill the live range depends on the spilling heuristic used. Chaitin's spilling heuristic offers some guidance in reducing the amount of spill code produced. However, this heuristic does not allow the partial spilling of live ranges and the reduction in spill code is limited to a local level. In this paper, we present a global technique called *interference region spilling* that improves the spilling granularity of any local spilling heuristic. Our technique works above the local spilling heuristic, limiting the normal insertion of spill code to a portion of each spilled live range. By partially spilling live ranges, we can achieve large reductions in dynamically executed spill code; up to 75% in some cases and an average of 33.6% across the benchmarks tested.

1 Introduction

Global register allocation can be modeled as a graph coloring problem. Nodes in the *interference graph* represent the live ranges which need to be allocated to machine registers. Edges between nodes specify constraints on the allocation. Specifically, if two nodes are connected by an edge, then their associated live ranges cannot be allocated to the same machine register. The register allocation problem then becomes: Does a proper k-coloring[1] of the interference graph exist? If the interference graph is k-colorable, then the colors can be seen as representing registers and the coloring as a valid register assignment. Although minimal graph coloring is NP-complete, fast and powerful heuristics exist that produce very efficient colorings in practice [6, 3, 7].

1.1 Chaitin-Style Allocators

Chaitin *et al.* at IBM Yorktown were the first to implement a global register allocator based on a graph coloring algorithm [5, 6]. Chaitin's coloring heuristic is fast, simple and it relies on the following seemingly obvious but very powerful graph theoretic property:

> *Given a graph G and a node v such that $degree(v) < k$, then G is k-colorable if and only if $G - v$ is k-colorable.*

This property states that no matter how the reduced graph $G - v$ is colored, there will always be at least one color left for v. Chaitin's coloring heuristic utilizes this property to recursively *simplify* the interference graph by removing unconstrained nodes[2] until the graph is either empty or all the nodes in the reduced graph are constrained[3]. If the graph is empty, then Chaitin's coloring heuristic has reduced the problem of finding a k-coloring of the interference graph to finding a k-coloring of the empty graph. Chaitin then re-inserts the nodes into the graph in the reverse order that they were removed, giving each node a color not used by any of its neighbors. Since each node was unconstrained when removed, each node is guaranteed to be colorable when it is re-inserted.

[*]This work was supported in part by the Office of Naval Research under grant no. N00014-94-1-0846 and grant no. N00019-95-1-0611, by the NSF through grant no. ASC-9523480, and in part by IBM Graduate Fellowships and an IBM Faculty Development Award.

[†]Currently with IBM Rochester.

[‡]Currently with SGI, Mountain View CA.

[1]Where k equals the number of machine registers.

[2]Nodes with $degree < k$.

[3]Nodes with $degree \geq k$.

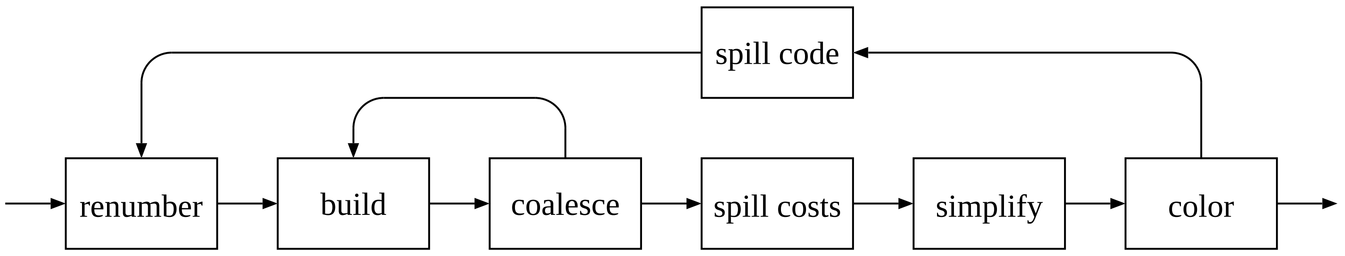

Figure 1: Flow Graph of Briggs' Allocator

1.2 The Need for Spilling

Register allocation becomes difficult when there are not enough registers for all the live ranges (*i.e.*, the interference graph is not k-colorable). Since the number of machine registers is fixed, a valid allocation for the intermediate language (IL) does not exist. The problem then is to modify the IL in such a way that its interference graph is k-colorable and the increase in program execution time is minimized.

In Chaitin-style allocators, the IL is modified by spilling the live ranges of uncolorable nodes to memory during portions of the program when they are not needed. Using Chaitin's spilling heuristic, this roughly translates to inserting a store after each *def* of the live range and a load before every *use* (Section 2.1 describes Chaitin's spilling heuristic in more detail). A shortcoming of this heuristic is that it inserts spill code throughout the entire spilled live range. The live range is either entirely allocated or it is entirely spilled, no middle ground exists. This spilling heuristic results in the live range being broken into many very short live ranges, all of which are live for only one basic block (*i.e.*, the new live ranges do not extend across basic block boundaries.). In fact, many of the new live ranges are live for only a few IL statements.

In this paper, we present a global technique called *interference region spilling* that improves the spilling granularity of Chaitin's spilling heuristic[4]. Our technique works above the local spilling heuristic, limiting the normal insertion of spill code to a portion of each live range. This allows live ranges to be partially spilled so that the new live ranges produced by our technique may extend across basic block boundaries, maybe even entire loops. By partially spilling live ranges, we can achieve large reductions in dynamically executed spill code; up to 75% in some cases and an average of 33.6% across the benchmarks tested.

[4]Our technique can actually improve the spilling granularity of any local spilling heuristic.

2 Briggs' Allocator

Briggs *et al.* developed an improvement to Chaitin's allocator when faced with a reduced graph that only contains constrained nodes [3]. Normally, Chaitin's allocator would remove one of the constrained nodes from the graph and mark it for later spilling. However, Briggs takes this node and removes it from the graph as if it were unconstrained, optimistically hoping there will be a color for it during the coloring phase. This spill candidate is chosen using Chaitin's spill cost heuristic which selects the constrained node with the smallest spill cost divided by current degree. This heuristic attempts to satisfy the goal of minimizing spill costs while at the same time trying to simplify the graph by reducing the degrees of many nodes (hopefully making some of the neighboring nodes unconstrained). After the node has been removed from the graph, the simplification stage continues the process of removing unconstrained nodes or choosing another spill candidate if there are none, until the graph becomes empty.

Once the interference graph has been reduced to the empty graph, Briggs re-inserts the nodes in reverse order of deletion, attempting to give each node a color that is different than any of its neighbors. For unconstrained nodes, Briggs is guaranteed to find a color. For constrained nodes, if two or more non-interfering neighbors have received the same color, then some color may still be available for assignment to the constrained node. If Briggs finds an available color, he gives the node that color and continues. However, if the neighboring nodes have been assigned all k colors, then Briggs marks this node for later spilling. The process of coloring the nodes continues until all nodes have been re-inserted with a color or marked for spilling.

If all of the nodes receive a color, then the colors can be seen as representing machine registers and the coloring as a valid register assignment. If however, some of the nodes did not receive a color, then Briggs inserts spill code using Chaitin's spilling heuristic for all the live ranges whose nodes were marked for spilling.

Since spill code requires some register resources, the entire process of building, simplifying and coloring the interference graph is repeated until no further spilling is needed. The flow graph for Briggs' allocator is shown in Figure 1.

2.1 Chaitin's Spilling Heuristic

Once we have determined which live ranges are to be spilled, we must now decide where to place the spill code. The simplest and roughest technique is to insert a store after every *def* of the live range and a load before every *use*. Although this spill-everywhere technique works, it usually generates much more spill code than is necessary. In Chaitin's spilling heuristic, he mentions several local optimizations that can reduce the amount of spill code produced when compared to the spill-everywhere technique [6, Pages 100–101].

- If a *use* of a live range is easy to recompute, then it should not be reloaded, but recomputed.
- If a *use* of a live range is "close" to its definition, then it is unnecessary to reload the live range at the *use*.
- If two *uses* of a live range are close, then it is unnecessary to reload the live range at the second *use*.
- Live ranges whose *uses* are *all* close to their definition should never be spilled.

Two references to a live range are defined by Chaitin to be "close" if no other live range dies between them. In other words, if no new register resources are made available by the death of a live range, then the load at the second *use* will gain nothing and should be avoided. Spilling a live range whose *uses* are all close to its definition will not help make the interference graph more easily colored. Therefore, Chaitin gives these live ranges an infinite spill cost. This ensures they are never spilled.

Later work, such as Bernstein *et al.*'s *spill-almost-everywhere* [1] and Briggs *et al.*'s *rematerialization* [4] techniques further reduced the amount of spill code generated. However, the drawback of all of these techniques is that if a live range has been marked for spilling, it is spilled entirely.

3 Interference Region Spilling

To enable the partial spilling of live ranges, we introduce a new concept called the *interference region*. For two interfering live ranges, we define their interference region to be the portion of the program where they are live simultaneously. By eliminating (*i.e.*, spilling) this region from one of the live ranges through the addition of spill code, they will no longer be live simultaneously anywhere in the program; thus they will no longer interfere. This effectively removes the edge between the two nodes in the interference graph, making the graph more easily colored. An example of a simple interference region can be seen in Figure 2.

To spill an interference region from a live range, we simply limit the insertion of normal spill code to the *uses* of the live range that occur inside the interference region and to the definition points of the live range that can reach, in the data-flow sense, the interference region. Unlike Chaitin's and Bernstein's local spilling heuristics, our technique can reduce the number of stores as well as the number of loads inserted for non-rematerializable live ranges.

In the event the spilled live range will be used again after the interference region, as is the case in our example, we must introduce a new form of spill code to *reload* the live range back into a register for further reuse. If the next *use* of the spilled live range after the interference region is within the current basic block, then our reloads will be inserted at the same location where Chaitin's spilling heuristic places a load. However, if the next *use* is in a successor basic block, then we will insert a reload where Chaitin has none.

This has important ramifications when computing interference region spill costs. Specifically, it may be more expensive, in terms of weighted loads and stores to spill an interference region from a live range than to spill the live range entirely. In the next section, we discuss how to easily ensure interference region spilling never generates more expensive spill code than the local spilling heuristic.

Returning to our simple example, Figure 3 shows what live range `B` might look like after spilling it using Chaitin's spilling heuristic, while Figure 4 shows what it might look like after interference region spilling. Notice that with interference region spilling and after renumbering, the majority of `B`'s original live range is still intact and it no longer interferes with `A`.

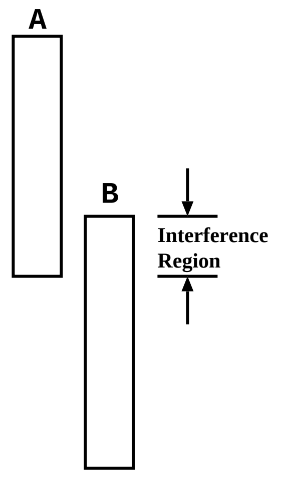

Figure 2: Live Ranges

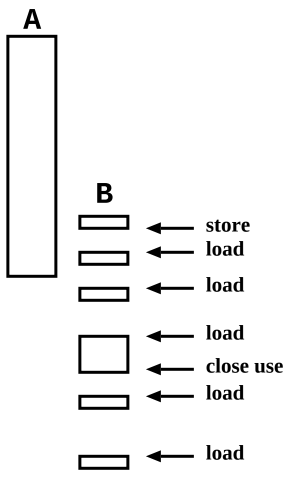

Figure 3: Chaitin Spilling

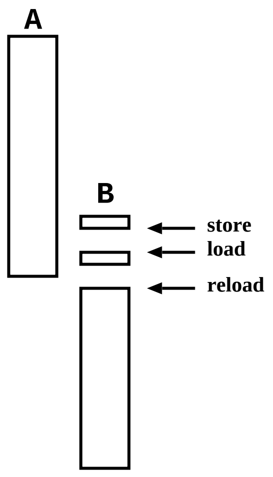

Figure 4: IR Spilling

3.1 Choosing Interference Regions to Spill

Now that we are able to remove arbitrary edges from the interference graph by spill interference regions, the question becomes: which interference regions should we spill? Our solution utilizes the k-colored subgraph of the interference graph and the list of uncolorable nodes from the output of Briggs' coloring phase.

In Briggs' allocator, the k-colored subgraph represents the live ranges that can be fully allocated, while the list of uncolorable nodes represents the live ranges that could not be fully allocated (*i.e.*, there may have been some register resources available, but none were sufficient to entirely satisfy their needs). With interference region spilling, we refine the meaning of the k-colored subgraph to represent the live ranges that can be fully or partially allocated. We start with the k-colored subgraph produced by Briggs and we attempt to increase its size by attaching onto the subgraph each of the uncolorable nodes, such that the enlarged subgraph is still k-colorable. Since these nodes are uncolorable, we cannot include all of their edges. For each uncolorable node, we group its edges into k sets – one set for each color – where each set contains the edges that lead to neighboring nodes with the same color. A color is then chosen that minimizes spill costs, and the set of edges associated with that color are not inserted back into the graph with the node. By not inserting the edges from one of the sets, we have made available a color with which that node can be colored. The edges not inserted represent the interference regions we must spill.

By increasing the size of the k-colored subgraph, we are allowing portions of live ranges that were spilled by Chaitin's spilling heuristic to be allocated, thus reducing the amount of spill code for each spilled live range. An important consequence of interference region spilling is that we insert a subset of the loads and stores generated by Chaitin's spilling heuristic working alone, meaning any load or store we insert will be inserted by Chaitin's heuristic, while some loads and stores Chaitin inserts may not be inserted by our technique. However, our technique may also insert some reloads which Chaitin's spilling heuristic does not insert. Therefore, the amount of spill code reduction, if any, depends on the number of uses of the live range that lie outside the spilled interference regions, plus the number of definition points that do not reach those interference regions, minus any reloads we are forced to insert[5].

In the event the interference region covers the entire live range, then our technique reduces to Chaitin's spilling heuristic. However, if we insert any reloads, it may actually be cheaper (in terms of weighted loads and stores) to spill the entire live range. However, since our heuristic works on top of Chaitin's spilling heuristic, we can simply revert to using Chaitin's spilling heuristic for any live range which is less expensive to spill entirely than it is to spill any of its interference regions. Therefore, for a given spilling decision, we are guaranteed never to produce more spill code than Chaitin's spilling heuristic working alone.

[5]Actually, only reloads which are inserted in locations where Chaitin's spilling heuristic has not inserted a load count against us.

```
A = input();
B = A + 1;
if ( A ) {
     C = A + 2;
     B = A + C;
     if ( C ) {
         B = B + C;
         C = B + C;
     }
     A = B + C;
}
D = A + B;
```

Figure 5: Code Example

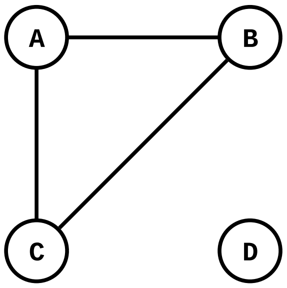

Figure 6: Interference Graph

node	cost
A	8
B	12
C	12
D	∞

Figure 7: Spill Costs

3.2 Interference Region Spilling Example

To demonstrate the effectiveness of interference region spilling, we present a simple example to compare the spill code generated by interference region spilling versus Chaitin's spilling heuristic. The code example and its corresponding interference graph are shown in Figures 5 and 6. In this example, we will attempt a 2-coloring using the spill costs[6] given in Figure 7.

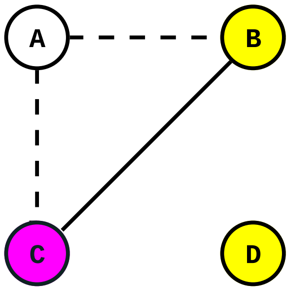

Figure 8: 2-Colored Sub-Graph

After Briggs' coloring phase has finished, we are left with the 2-colored subgraph shown in Figure 8 and the uncolored live range A. Normally we would spill the live range A using Chaitin's spilling heuristic resulting in the code in Figure 10. However with interference region spilling, we first insert the uncolorable node A back into the graph. Now we must make A colorable by choosing a color for it and removing all of its edges leading to neighboring nodes with that color. For our example, we have two choices. We may either omit the edge A↔B or we can omit the edge A↔C.

[6]In Briggs' spill cost phase, loads and stores are charged a cost of 2 versus 1 for rematerializable instructions [2].

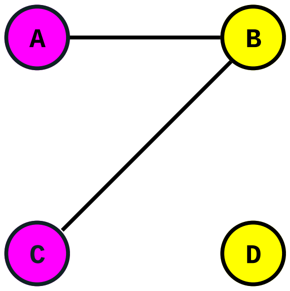

Figure 9: Spilling A↔C

The estimated spill cost of spilling interference region A↔C is less than the cost of spilling interference region A↔B (1 store + 1 load vs. 2 stores + 1 load + 1 reload), so we omit this edge from the graph giving us the enlarged 2-colored subgraph shown in Figure 9. The code resulting from spilling interference region A↔C from live range A is shown in Figure 11.

Once spilling is completed, we need to recompute live ranges and rebuild the interference graph so that we can attempt another coloring. However, for this example, we will find that the new interference graphs for the codes in Figures 10 and 11 are both 2-colorable and further spilling is not necessary. Therefore, the final result is that interference region spilling inserted 1 store and 1 load of live range A while Chaitin's spilling heuristic generated 2 stores and 2 loads. This is a 50% reduction in total spill code for this example.

```
A = input();
store A;
B = A + 1;
if ( A ) {
     load A1;
     C = A1 + 2;
     B = A1 + C;
     if ( C ) {
          B = B + C;
          C = B + C;
     }
     A2 = B + C;
     store A2;
}
load A3;
D = A3 + B;
```

Figure 10: After Spilling `A` Entirely

```
A = input();
store A;
B = A + 1;
if ( A ) {
     load A1;
     C = A1 + 2;
     B = A1 + C;
     if ( C ) {
          B = B + C;
          C = B + C;
     }
     A = B + C;

}

D = A + B;
```

Figure 11: After Spilling `A`↔`C`

3.3 Implementation Details

To implement interference region spilling, we modified Briggs' allocator in two ways. First, we added an additional *interference region spill costs* stage that, for each spilled live range, determines whether we should spill the entire live range or which set of interference regions we need to spill. This stage attempts to choose the interference regions that will minimize the amount of spill code needed.

Secondly, the spill code stage was modified to limit the insertion of spill code to the uses inside the spilled interference regions and to the loads needed to reload the live range for any further uses outside the interference regions. We also altered this stage so that only definition points that reach the new loads and reloads will have stores inserted for them. Figure 12 shows the flow graph for Briggs' allocator modified with interference region spilling.

In Briggs' implementation of Chaitin's spilling heuristic, a single pass is made over the control flow graph and each instruction in the block is visited in reverse order. As each instruction is visited, the instruction operands are examined to determine whether any spill loads or spill stores are needed. Three sets are utilized: `live` indicates which live ranges are currently live, `markedLR` contains the live ranges that have been marked for spilling and `needLoad` specifies all the live ranges which have been marked for spilling and have been used since the last death.

```
foreachBasicBlock(CFG, blk) {
    copySet(live, blk->liveOut);
    clearSet(needLoad);
    foreachMember(markedLR, reg) {
        < init range[reg].numLiveNeighbors info >
        if ( member?(live, reg) &&
             !range[reg].spillEntirely &&
             range[reg].numLiveNeighbors == 0 )
            addMember(needReload, reg);
    }
    // Scan instructions from bottom to top...
    foreachInsnB2T(blk, insn) {
        < handle definitions of insn >
        < check for deaths in insn >
        < handle uses of insn >
    }
    // Add loads at top of block
    foreachMember(needLoad, reg)
        < insert load of reg >
    // Add some reloads at top of block
    foreachMember(needReload, reg)
        if ( at IR boundary for reg )
             < insert reload of reg >
}
< insert stores for defs that reach the new loads/reloads >
```

Figure 13: Interference Region Spill Code

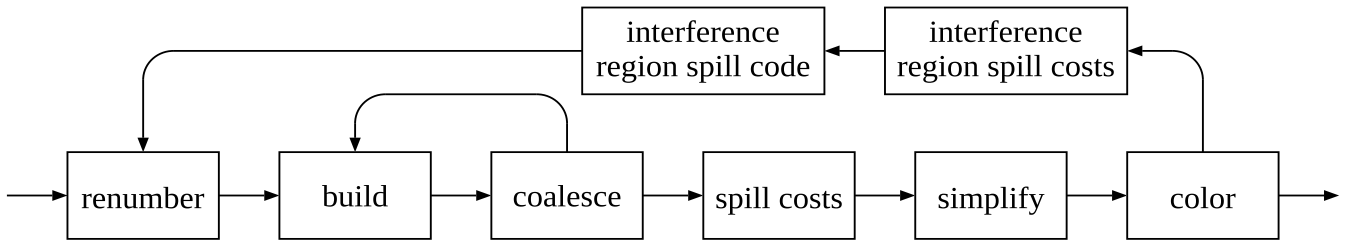
interference region spill code
interference region spill costs
renumber
build
coalesce
spill costs
simplify
color

Figure 12: Briggs' Allocator Modified with Interference Region Spilling

Our implementation of interference region spilling builds upon this framework in several ways. We have added three new fields to Briggs' `range` structure. The `range[reg].spillEntirely` field indicates whether a live range is cheaper to spill entirely or spill partially. If a live range is partially spilled, `range[reg].spillColor` specifies the interference regions that need to be spilled (*i.e.*, the interference regions associated with edges leading to neighboring nodes that were assigned the color `range[reg].spillColor`). To easily detect when an interference region boundary is encountered, we incrementally update the `range[reg].numLiveNeighbors` field to count the number of neighbors that are assigned the color `range[reg].spillColor` and that are currently live. Since reloads are inserted at interference region boundaries, a `needReload` set was added to track the live ranges that have been marked for spilling but have only been used outside of any spilled interference region.

While scanning the definitions, Briggs' implementation updates the `live` and `needLoad` sets and inserts stores for definitions of spilled live ranges. With interference region spilling, we include an update of the `needReload` set and delay the insertion of stores until until all loads and reloads have been inserted.

```
// Handle definitions of insn
foreachDefinedReg(insn, reg) {
    < update range[...].numLiveNeighbors info >
    deleteMember(needLoad, reg);
    deleteMember(needReload, reg);
    deleteMember(live, reg);
}
```

Figure 14: Handle Definitions

In Briggs' implementation of Chaitin's spilling heuristic, if a death is detected, loads are inserted for every spilled live range that has been used since the last death (*i.e.*, members of `needLoad`). With interference region spilling, we also need to insert reloads if this death is an interference region boundary for any member of `needReload`.

```
// Insert loads/reloads only at deaths...
foreachUsedReg(insn, reg) {
    if ( !member?(live, reg) ) {
        color = range[reg].color;
        foreachMember(needReload, mem)
            if ( color == range[mem].spillColor &&
                 range[mem].numLiveNeighbors == 0 &&
                 interfere(reg, mem)) {
                < insert reload of mem >
                deleteMember(needReload, mem);
            }
        foreachMember(needLoad, mem)
            < insert load of mem >
        clearSet(needLoad);
    }
}
```

Figure 15: Check for Deaths

Finally, after any loads and reloads have been inserted, the `live`, `needLoad` and `needReload` sets need to be updated. If a used live range is a member of `markedLR`, then it is added to the `needLoad` set if is cheaper to spill entirely or this use occurred within a spilled interference region. Otherwise, if the live range is not already a member of `needLoad`, it is added to the `needReload` set.

An important question with regard to interference region spilling is how much effect does it have on allocation time when compared to Chaitin's spilling heuristic. Currently, we only have limited allocation time measurements comparing interference region spilling and Chaitin's spilling heuristic. Preliminary data compiling *tomcatv* indicates that the increase in register allocation time due to interference region spilling is approximately 20 - 40%. However, analyzing our implementation, we believe tuning can reduce this performance penalty. The modifications to Briggs' allocator have been localized to the spilling phase, which means that

```
// Handle uses of insn
foreachUsedReg(insn, reg) {
    if ( member?(markedLR, reg) ) {
        if ( range[reg].spillEntirely ||
             range[reg].numLiveNeighbors != 0 )
            addMember(needLoad, reg);
        else if ( !member?(needLoad, reg) )
            addMember(needReload, reg);
    } else if ( !member?(live, reg) )
        < update range[...].numLiveNeighbors info >
    addMember(live, reg);
}
```

Figure 16: Handle Uses

the allocation time for routines that do not need any spilling is unchanged. For routines that require spill code, we now must compute interference region spill costs. However, we need only compute these for live ranges that have been marked for spilling.

4 Results

In order to experiment with our spill code minimization heuristics, we have modified a version of the GNU C compiler (`gcc` version 2.7.2 targeted to a MIPS II processor) to contain our implementation of Briggs' optimistic coloring allocator. We then modified Briggs' allocator so that we can choose between the normal spill code stage which uses Chaitin's spilling heuristic and our new spill code stage which uses interference region spilling. We then compiled several integer and floating point intensive programs from the SPEC 92 suite of benchmarks. To simulate varying levels of register pressure, we compiled each benchmark multiple times varying the number of registers available to the register allocator. To accurately measure the amount of executed spill code, a MIPS II instruction level simulator was used to count the spill code that was inserted by our spilling phases.

Our results are given in Table 17. The first two columns of the table specify the benchmark compiled and the number of registers available for allocation[7]. The third and fourth columns give results using Briggs' implementation of Chaitin's spilling heuristic[2]. Column 3 shows the dynamic spill cost which is computed by counting each spill instruction executed[8] and column 4 shows the percentage of all instructions executed that were spill code. The fifth and sixth columns give the same information as columns 3 and 4 for interference region spilling. The next three columns show the percent improvement in terms of dynamic spill costs for loads, stores and rematerialized instructions. Finally, the last two columns show the percent reduction in dynamic spill code executed and execution time[9].

For example, the first row of data shows we compiled the benchmark *compress*, allowing the allocator only 8 registers for coloring. The spill cost using Chaitin's spilling heuristic was 128 million weighted spill instructions executed and 19.5% of all instructions executed were spill instructions. For interference region spilling, our dynamic spill cost dropped to 63 million weighted spill instructions, which now comprise only 11.7% of all instructions executed. This gives us a 51.2% reduction in spill code and an 18.6% execution time improvement. Note that all percentages less than one tenth of one percent have been left blank and register file sizes for which spill code comprised less than one percent of all instructions executed have been omitted.

Examining these results, we notice that although interference region spilling is not guaranteed to generate less spill code than Chaitin's spilling heuristic, for every benchmark compiled, interference region spilling *always* produced better spill code than Chaitin's spilling heuristic. Secondly, interference region spilling averaged a 33.6% reduction in dynamic spill costs and an 8.3% improvement in execution time over all of the benchmarks compiled.

5 Conclusion

We have introduced a new fine granularity spilling technique called *interference region spilling* that can significantly reduce the amount of spill code generated in Chaitin-style graph coloring register allocators. Interference region spilling relies on our definition of an interference region, which specifies the portion of the program where two interfering live ranges are live simultaneously. Spilling this region from one of the live ranges breaks their interference, allowing them to be allocated to the same register. The results demonstrate the effectiveness of interference region spilling. Comparing against Chaitin's spilling heuristic, interference region spilling reduced dynamic spill costs an average of 33.6% across all register files sizes and benchmarks tested and up to 75% in some cases. Our results also indicate that under heavy register pressure, interference region spilling significantly outperforms Chaitin's spilling heuristic.

[7]Here, "8" registers indicates 8 integer and 8 floating point registers were used for allocation. Note the MIPS II ISA only contains 16 usable floating point registers.

[8]Spill loads and stores are weighted twice as much as simple rematerialized spill instructions.

[9]Timings were taken on an MIPS R5000. Data shown is an average of 10 trials.

Program	# Regs	Dynamic Spill Costs				Percentage Reduction				
		Chaitin	% total	IR Spill	% total	load	store	remat	Total Spill	Run Time
Compress	8	128222726	19.5	62578192	11.7	51.2			51.2	18.6%
	12	128222726	19.5	62578192	11.8	51.2			51.2	16.1%
	16	128222726	19.5	62578192	11.7	51.2			51.2	16.4%
	20	26421276	6.6	26421272	6.6					0.5%
Espresso	8	966806867	14.9	763558471	12.0	16.9	-0.4	4.5	21.0	6.8%
	12	733166332	12.0	502212430	8.7	27.6	0.5	3.4	31.5	8.7%
	16	563277100	9.7	334234983	6.1	35.2	0.1	5.4	40.7	6.7%
	20	99429247	2.0	74511956	1.5	24.1		1.0	25.1	0.9%
Li	8	2330448245	39.6	1470516987	29.5	35.6	1.3		36.9	9.0%
	12	2340525803	41.4	1465031307	30.4	36.1	1.3		37.4	9.3%
	16	2345019255	42.5	1465032073	30.7	36.3	1.3		37.5	8.8%
	20	142346501	3.8	119978216	2.9	0.1		15.7	15.7	-0.1%
Alvinn	8	1824793308	46.2	463650268	17.9	74.6			74.6	15.0%
	12	1817620508	46.1	458937468	17.8	74.8			74.8	13.5%
	16	1816336508	46.1	457653468	17.7	74.8			74.8	15.7%
Tomcatv	8	2154304766	42.1	2121383158	41.7	1.2		0.3	1.5	0.0%
	12	1321829706	31.7	1262873006	30.1	2.0	-1.0	3.5	4.5	7.7%
	16	580301462	18.7	501863262	16.2	4.5	2.2	6.8	13.5	10.7%
	20	404606462	14.2	365004062	12.6	3.2		6.6	9.7	1.1%
	24	261551462	10.7	208587562	8.3	5.1		15.2	20.3	0.0%
	28	261551462	10.7	208587562	8.3	5.1		15.2	20.3	0.0%
	32	261551462	10.7	208587562	8.3	5.1		15.2	20.3	0.0%

Figure 17: SPEC'92 Benchmarks

References

[1] Bernstein, D., Goldin, D. Q., Golumbic, M. C., Krawczyk, H., Mansour, Y., Nahshon, I., and Pinter, R. Y. Spill code minimization techniques for optimizing compilers. *SIGPLAN Notices 24*, 7 (July 1989), 258–263. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[2] Briggs, P. Register allocation via graph coloring. Ph.D. Thesis Rice COMP TR92-183, Department of Computer Science, Rice University, 1992.

[3] Briggs, P., Cooper, K. D., Kennedy, K., and Torczon, L. Coloring heuristics for register allocation. *SIGPLAN Notices 24*, 7 (July 1989), 275–284. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[4] Briggs, P., Cooper, K. D., and Torczon, L. Rematerialization. *SIGPLAN Notices 27*, 7 (July 1992), 311–321. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[5] Chaitin, G., Auslander, M., Chandra, A., Cocke, J., Hopkins, M., and Markstein, P. Register allocation via coloring. *Computer Languages 6* (1981), 47–57.

[6] Chaitin, G. J. Register allocation and spilling via graph coloring. *SIGPLAN Notices 17*, 6 (June 1982), 98–105. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[7] Chow, F. C., and Hennessy, J. L. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst. 12*, 4 (Oct. 1990), 501–536.