Interference Graph Trimming

Cliff Click
Michael Paleczny
Christopher Vick

Sun Microsystems
901 San Antonio Road, Palo Alto, CA 94303

Abstract

Graph-coloring allocators have a reputation for being slow and producing high quality allocations. HotSpot uses a modified Briggs-Chaitin allocator to portably produce high quality code. Since the compiler executes at runtime, allocation speed is at a premium. One of the slower tasks in a coloring allocator is constructing the interference graph. We present a technique that produces a more than 3-fold reduction in interference graph edges, leading to a significant reduction in allocation time.

Introduction

Graph-coloring allocators have a reputation for being slow and producing high quality allocations. HotSpot targets a variety of CPUs, including register-rich RISC chips for which a high quality allocation can make the most of a large register set. HotSpot uses a modified Briggs-Chaitin [B92, C82] allocator to portably produce high quality allocations. Since the compiler executes at runtime, allocation speed is at a premium. One of the slower tasks in a coloring allocator is constructing the interference graph. We present a technique that produces a more than 3-fold reduction in interference graph, edges leading to a significant reduction in allocation time.

The key observation is that procedure calls and certain instructions bind arguments to fixed registers, and this binding is common. Live ranges that span a binding site cannot use the bound register at that site. Live ranges that are bound **must** use the bound register at the binding site. These two classes of live ranges cannot interfere and their interference edge does not need to be represented in the interference graph.

The Interference Graph

Interference Graphs (IFGs) represent *interferences* between *live ranges*. Live ranges represent the lifetimes of individual values. Each live range is eventually assigned a single register. An interference happens when two live ranges overlap, carry different values, and may legally reside in the same register set. These two live ranges must be assigned to different registers, and the interference represents the requirement for two different register assignments.

In an IFG, live ranges are the vertices and interferences are the edges. Graph-coloring allocators attempt to *color* the IFG vertices in fewer than k colors, where k is the number of registers on the target CPU. A graph is colored if every vertex is assigned a color that differs from all neighboring vertices. If the coloring succeeds the colors are

mapped to CPU registers and allocation is complete. Frequently, IFGs are not *k*-colorable on the first attempt. If the coloring attempt fails, the allocator *spills*. After spilling the IFG is rebuilt and another coloring is attempted.

Spilling breaks large live ranges with many interferences into many small live ranges with fewer interferences. A common spill technique is to store values into *stack-slots* at each def and load prior to each use; the bulk of the original live range is kept on the stack. The inserted instructions are called *spill instructions*. There is a large body of work on spilling techniques, [BCKT89,BCT94,BDEO97,LG97], most focusing on which live ranges to spill and not where to spill. HotSpot *splits* live ranges rather than spilling them, which allows us finer control on where spill instructions are placed [B92]. After splitting, large portions of the original live range may remain in registers.

HotSpot allocates and colors stack-slots as if they were registers. Split instructions appear to HotSpot as a special kind of a register-register copy (one that targets a machine register on one side and a stack-slot or machine register on the other). Like other copies, our allocator will attempt to coalesce splits. If successful, a copy, a load or a store is avoided. If a live range is colored to a stack-slot, inserted split instructions will be converted into loads or stores just prior to code emission. Otherwise, they remain as register-register copy instructions.

Building the IFG uses a substantial portion of allocation time. In general, the set of live ranges is proportional to the size of the compilation unit ($O(n)$), and the edge set is $O(n^2)$. Once the IFG is built, the allocator uses both membership tests and for-all-neighbors iteration. A fast membership test is usually done with a bitset implementation, and the for-all-neighbors iteration is usually done with a list implementation. Thus the IFG will often use two distinct implementations.

Our implementation of IFG trimming reduces the number of edges present by a factor of 2 to 3. This allows a single list implementation to achieve good time bounds for all uses. The single implementation simplifies the allocator, and we are not required to convert from one form to the other.

Register Bindings

The key observation that makes IFG trimming work is that certain instructions, most notably procedure calls, bind values to fixed registers, and that these bindings are common. Live ranges that span a binding site **cannot** be assigned to the bound register at that site. Live ranges that are bound **must** be assigned the bound register at the binding site. These two classes of live ranges cannot interfere, and their interference edge does not need to be represented in the IFG. The allocation quality is not lowered by trimming. Trimming removes interferences that are not required for any legal allocation.

In the HotSpot allocator, each live range carries with it a set of allowable registers. The set is implemented as a short fixed-length bitvector which includes elements for all machine registers, some stack slots and a bit to summarize the infinite stack. Typically the set contains bits for all the integer (or float) registers. If a live range is bound, e.g. by being an argument to a call instruction, the set will be reduced to the single allowed register which may be a machine register or a stack-slot. Conflicting bindings will empty the bitset completely. This is a fairly common occurrence and simply indicates that the live range needs to be split.

```
x = ...+...;                        ADD  LR1:any,...
y = ...-...;                        SUB  LR2:any,...
                                    CALL      # call uses LR1:ECX
y.field1 = x.call();                # call defs LR3:EAX
                                    # call kills EBX,ECX,EDX
...x...                             MOV [LR2:any +8],LR3:any
                                    ...LR1:any...
```

Figure 1: Java code and pseudo-assembly

A *binding site* is a program point where some live range (either a value definition or use) is bound to a specific register. Subroutine calls are common binding sites. Procedure entry and exit also bind several live ranges. Some machine instructions on certain CPUs are also binding sites.[1] Live ranges that span a binding site will have the bound registers removed from their allowed-register bitset. This prevents them from coloring to the bound register. If the spanning live range were to color to the binding register, the bound live range would not get its required color and spilling would be required.

When the IFG is built, if two live ranges are simultaneously live but have no registers in common, *they do not interfere*. Since they have no registers in common those values cannot appear in the same register at the same time. No edge between them is placed in the IFG. Consider the example in Figure 1. A short piece of Java code is on the left along with pseudo-assembly on the right. Register bindings for the Intel IA32 are shown as lists of registers. The notation LR1:ECX indicates that live range LR1 must reside in register ECX at that point. We have chosen to pass the first two call arguments in ECX and EDX, hence the call generates the binding LR1:ECX. Registers ESI, EDI and EBP are preserved across a call; EBX, ECX and EDX are killed; EAX is returned. Java variable `x` is assigned to live range LR1 by an instruction with no register bindings, then `y` is assigned to LR2. Variable `x` is passed in ECX to the call. The call's result, returned in EAX, is stored into address `y`+8. Finally, `x` remains live across the call.

From this example we see that LR1 has a direct conflict. It is needed in ECX for the call, ECX is killed by the call and LR1 is alive after the call. LR1's allowed register set is empty and it must be split. LR2 spans a call site which kills EAX, EBX, ECX and EDX. Its allowed register set is {ESI,EDI,EBP}. LR3 is set by the call to EAX. Since LR1's register set is empty it has no interferences in the IFG. LR2 cannot use EAX and LR3 must use EAX so they do not interfere either. For this code snippet, the trimmed IFG is empty.

After a round of splitting we have the situation shown in Figure 2. Here LR1 remains bound to ECX by the call. LR2 is bound to one of {ESI, EDI, EBP} (the only registers to survive the call). LR3 is bound to EAX (call return value) and the new live range LR4 is also bound to one of {ESI, EDI, EBP}. The IFG only has an interference between LR2 and LR4. This graph is trivially colorable.

A key action taken while coloring the IFG is finding and removing *trivial* live ranges. A trivial live range has fewer neighbors (degree) than allowed colors (registers). Trivial live ranges can always get a color (register). Even if all neighbors choose different registers there is always one left over for the trivial live range. Commonly the definition of *trivial* is fixed by the number of integer (or float) registers on the CPU. In our model, *trivial* is defined by the size of the allowed register bitset. Live ranges with zero allowed registers can never color, even with no neighbors. Live ranges with 1 allowed re-

1 Most commonly executed instructions on an Intel IA32 do not bind registers.

```
x = ...+...;                        ADD  LR1:any,...
y = ...-...;                        SUB  LR2:any,...
lr4 = x; // spill instruction       MOV  LR4:any,LR1:any
lr3 = x.call();                     CALL # call uses LR1:ECX
                                    # call defs LR3:EAX
y.field = lr3;                      # call kills EBX,ECX,EDX
...lr4...                           MOV [LR2:any +8],LR3:any
                                    ...LR4:any...
```

Figure 2: Java code and pseudo-assembly

gister can only color with zero neighbors. If they interfere with even one other live range, that live range could use the only register available. Thus, in our allocator a trivial live range is defined as having fewer neighbors than the number of allowed registers in its bit-set.

Note that live ranges LR2 and LR4 will presumably be allocated to callee-save registers, requiring some prolog and epilog code. We shall see in the next section how HotSpot's treatment of callee-save registers can lead to better allocations.

Callee-Save Registers

The usual treatment of callee-save registers marks them as available for general use. After all registers are selected, callee-save registers that get used are spilled and reloaded in prolog and epilog code. Since that spilling is not free, there is often some weighting heuristic used to prevent unnecessary uses of callee-save registers. Prolog and epilog code can be further processed to *sink* it deeper into the procedure, e.g., if the procedure has a quick test and exit that requires no extra registers [Chow88,CL96] on one path and complex computations on another path.

In the HotSpot allocator, callee-save registers are treated as an additional live range created at the procedure entry and used at procedure exits. Callee-save live ranges are pre-bound at entry and exit to the register they are saving. After adding these extra live ranges to the IFG, the allocator attempts allocations normally. Callee-save live ranges are not further treated specially in any way. They make ideal split candidates (single def, single use, span a large area) so they will tend to split early, on an as-needed basis. Thus the usual prolog/epilog location of callee-save register spills is avoided by the normal splitting mechanism. A split-everywhere technique (analogous to the common spill-everywhere technique) would produce a result similar to the usual handling of callee-save registers, except that the splits do not necessarily produce a memory reference since they permit a simple register-register move to resolve the conflict.

In addition to taking arguments bound to fixed registers, call sites generally kill a large number of registers (all the non-callee-save ones!). These kills act as if there is an unused pre-bound def of all non-callee-save registers at the call site. These bound defs remove the non-callee-save registers from any live range which spans the call. This, in turn, forces spanning live ranges into the callee-save registers. Since the callee-save live ranges also need those registers some live ranges will need to split. At this point the normal split heuristics will choose the most profitable live range to split. It is common, but not required, for the callee-save registers to split.

A more complete example showing the effect of the callee-save registers is shown in Figure 3. The live range bindings and interferences are shown in Figure 4. Live ranges

```
static A foo( A x, B y ) {
                                  y.field = lr3;
                                  return x;
                                }
    lr3 = x.call();

                                # Start defs LR1:ECX as parameter 0
```

Figure 3: Java code and pseudo-assembly

1 and 2 have no allowed registers and must be split. Live ranges 5, 6, and 7 represent the callee-save registers.

Assume the allocator splits the live ranges as shown in Figure 5. The exact technique used to split or spill is beyond the scope of this paper. The notation `any+stack`

Live Range	Registers allowed	Interferences
LR1	[ ]	none
LR2	[ ]	none
LR3	[EAX]	none
LR5	[ESI]	none
LR6	[EDI]	none
LR7	[EBP]	none

Figure 4: IFG for Figure 3

means the live range can be colored into any register or a stack slot. Live ranges 4 and 8 are introduced to carry live ranges 1 and 2 across the call and are expected (but not required) to color into the stack.

Figure 6 shows the resulting live ranges, register sets and interferences. Live ranges 1, 2, 9 and 10 are trivial to color and simplifying LR9 allows LR3 to become trivial. Live ranges 4 and 8 are also trivial, but only if we color them to the stack (i.e., spill them). Since no other live ranges are trivial, we must simplify live ranges 4 and 8. At this point the remaining live ranges (5,6,7) are all trivial and we are guaranteed a coloring. Final register selection is shown in the last column, and final code is shown in Figure 7. Note that the classic treatment of callee-save registers would allocate the two live ranges that survive the call (LR3 and LR4) to callee-save registers, requiring 2 copies to get the values there. In addition, two callee-save registers would need to be saved and restored, requiring 2 loads and 2 stores. The HotSpot allocation requires the 2 loads and 2 stores but avoids the 2 copies.

```
                                lr10 = lr4;
                                return lr10;
static A foo( A x, B y ) {    }
                              # Start defs LR1:ECX as parameter 0
                              # Start defs LR2:EDX as parameter 1
  lr4 = x;                    # Start defs LR5:ESI as callee-save ESI
  lr8 = y;                    # Start defs LR6:EDI as callee-save EDI
  lr3 = x.call();             # Start defs LR7:EBP as callee-save EBP
                              MOV LR4:any+stack, LR1:any+stack
                              MOV LR8:any+stack, LR2:any+stack
  lr9 = lr8;                  CALL  # call uses LR1:ECX
  lr9.field = lr3;            # call defs LR3:EAX
                              # call kills EBX,ECX,EDX
```

Figure 5: Java code and pseudo-assembly

Numbers

HotSpot is a dynamic system. For example, variation in context switches can cause methods to be compiled in a different order or with different inlining patterns. The heuristics guiding compilation act to stabilize this process so that repeated runs produce the same average behavior. The HotSpot runtime monitors the ratio of time spent compiling versus time spent executing. If compilation speeds up, due to faster allocation, more methods can be compiled. Total time spent allocating is nearly the same whether

Live Range	Registers allowed	Interferences	Register selected
LR1	[ECX]	none	ECX
LR2	[EDX]	none	EDX
LR3	[EAX]	9	EAX
LR4	[ESI,EDI,EBP,stack]	5,6,7,8,9	[ESP+0]
LR5	[ESI]	4,8,9	ESI
LR6	[EDI]	4,8,9	EDI
LR7	[EBP]	4,8,9	EBP
LR8	[ESI,EDI,EBP,stack]	4,5,6,7	[ESP+4]
LR9	[any]	3,4,5,6,7	EBX
LR10	[EAX]	none	EAX

Figure 6: IFG for Figure 5

we trim the IFG or not, but the allocation rate improves. All our results are expressed as per byte-of-bytecode rates.

We ran HotSpot on the applications which comprise the SpecJVM98 [Spec] benchmark suite,[2] and collected the size of every IFG built. We ran the entire benchmark

```
static A foo( A x, B y ) {          ADD  ESP,8
                                    MOV  [ESP+0],ECX  ; lr4 <- lr1
                                    MOV  [ESP+4],EDX  ; lr8 <- lr2
  y.field = x.call();               CALL
                                    MOV  EBX,[ESP+4]  ; lr9 <- lr8
                                    MOV  [EBX+8],EAX  ; store lr3
                                    MOV  EAX,[ESP+0]  ; lr10 <- lr1
  return x;                         SUB  ESP,8
}                                   RET
```

Figure 7: Java code and pseudo-assembly

suite five times and report the average sizes. The standard deviations are small in all cases, around 1 or 2%. We measured these five runs for both the SPARC and IA32 CPUs, with and without IFG trimming. Each run compiles an average of 650 methods with extensive inlining and an average of 358,700 bytes-of-bytecodes.

Each individual compile builds a number of IFGs. The design of HotSpot is such that at least two IFGs are required for any compile. The first IFG is used to transform from SSA form to a normal CFG representation; the second IFG is used for the first coloring attempt. Typically there are direct register conflicts which prevent a coloring; e.g. a value is defined in one register but required in another. Such live ranges are split without a coloring attempt and then a third IFG is built. Many capacity failures are detected while attempting to color using the third IFG. The live ranges which fail are then split, and we repeat the coloring process. The number of compiles requiring more than six coloring attempts is small enough to not be statistically relevant.

Figure 9 shows the IFG edge density, the number of edges (interferences) added to the IFG for each byte-of-bytecodes, broken down by color attempt. On average, trimming lowers edge density by 73% on Intel and 49% on SPARC. After the first few attempts, splitting makes a great many unconstrained live ranges. These live ranges all mutually interfere, leading to a more dense IFG in the later attempts.

Compile Time Benefits

The goal of IFG trimming is to improve the allocation time of a graph-coloring allocator. Figure 8 shows the allocation rate as bytes-of-bytecodes per second for trimmed and untrimmed IFGs. Both measured platforms, SPARC Ultra2 and Pentium III, show significant improvement. Allocation using the trimmed IFG achieved a rate of 22680 bytes-of-bytecodes/sec on a 700Mhz Pentium III. The untrimmed run achieved a rate of 13440 bytes-of-bytecodes/sec. This makes the trimmed IFG nearly 70% faster to allocate over the untrimmed IFG when compiling for a Pentium III. Trimming improves allocation speed by 31% when compiling for a SPARC Ultra2. We fell that this speed increase

Figure 9: IFG Edge Density

makes a graph-coloring allocator tenable in a dynamic compilation system.

Platform	Normal IFG	Trimmed IFG	Improvement
450Mhz SPARC U2	6800	8930	31%
700Mhz Pentium 3	13440	22680	69%

Figure 8: Allocation rate, in bytes-of-bytecodes/second

Related Work

Chaitin-style [C82] graph color allocators have been around awhile. Briggs, et. al. [B92, BCT94] proposes a number of enhancements. There has been a lot of work in producing better allocations from a Briggs-Chaitin allocator [BCKT89, V99], or better spill code placement [BDEO97, LG97] or disambiguating pointers to allow hoisting into re-

gisters [CL97, SJ98, LCKLT98], or removing more copies [GA96]. Relatively little work has been done on speeding up graph-coloring allocators. Speedy allocator work has been focused on linear scan allocators [THS98, PEK97, ACLPS98].

References

[ACLPS98] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. Parikh, and J. Stichnoth. Fast, Effective Code Generation in a Just-In-Time Java Compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation,* pages 280-290, June 1998.

[B92] P. Briggs, *Register Allocation via Graph Coloring*. Ph.D. Thesis, Rice University, 1992.

[BCKT89] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* pages 275-284, July 1989.

[BCT94] P. Briggs, K. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428-455, May 1994.

[BDEO97] P. Bergner, P. Dahl, D. Engebretsen, and M. O'Keefe. Spill Code Minimization via Interference Region Spilling. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 287-295, June 1997.

[C82] G. J. Chaitin. 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*.

[Chow88] F. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* pages 84-95, June 1989.

[CL96] R. Cohn and P. Lowney. Hot Cold Optimization of Large Windows/NT Applications. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 80-89, December, 1996.

[CL97] K. Cooper and J. Lu. Register promotion in C programs. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation,* pages 308-319, June 1997.

[GA96] L. George and A. Appel. Iterated Register Coalescing. *ACM Transactions on Programming Languages and Systems*, 12(4):300-324, May 1996.

[LCKLT98] R. Lo, F. Chow, R. Kennedy, S. Liu, and P. Tu. Register promotion by Sparse Partial Redundancy Elimination of Loads and Stores. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation,* pages 25-37, June 1998.

[LG97] G. Lueh, and T. Gross. Call-Cost Directed Register Allocation. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation,* pages 2967-307, June 1997.

[PEK97] M. Poletto, D. Engler, and M. Kaashoek, "tcc: a System for Fast, Flexible and High Level Dynamic Code Generation," *SIGPLAN Notices*, 32(5):109-121, May 1997.

[SJ98] A. Sasry, and R. Ju. A New Algorithm for Scalar Register Promotion Based on SSA Form. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation,* pages 15-25, June 1998.

[Spec] SPECjvm98 Benchmarks. August 19, 1998. http://www.spec.org/osg/jvm98.

[THS98] O. Traub, G. Holloway, and M. Smith. Quality and Speed in a Linear-scan Register Allocator. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation,* pages 142-151, June 1998.

[V99] S. Vegdahl. Using Node Merging to Enhance Graph Coloring. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation,* pages 150-154, June 1999.