Register Allocation via Chordal Graph Coloring for a HOT Compiler

Daniel Lee Bryan Mills
Carnegie Mellon University

May 18, 2006

1 Introduction

[Note: this is an early draft of this write-up. A more polished version with citations and so forth is on the way shortly.]

The problem of creating efficient type-preserving compilers for programming langauges with higher-order functions and rich type systems (HOT) has been studied with great interest in recent years, particularly at CMU. In Fall 2005, a course on HOT Compilation was offered for advanced undergraduates. Due to time constraints, students were never able to write code generators for their compilers. This project extends the vision of HOT Compilation by generating code for a simplified TAL, which we have also written a simulator for. In keeping with the spirit of optimizing compilers, we have chosen as the central point of our implementation a very new register allocation algorithm that is surprisingly appropriate for HOT backends.

A typical compilation strategy for a HOT compiler is to eventually employ an intermediate representation known as continuation passing style. In CPS all control flow is represented by jumping to continuations. There are no join points and (in proper functional style) all variables are only declared once. This turns out to be a very simplified version of SSA form. The main difference is that SSA code can have join points, and variables are "joined" via ϕ nodes. ϕ nodes are internalized by CPS style as arguments into continuations. CPS converted code enjoys many of the benefits of SSA without the drawback of having to go through SSA elimination. This is paid for by the cost of passing around heap allocated data as arguments to continuations.

For this project, we have chosen a register allocation algorithm based on optimal coloring for chordal graphs. As we will see, CPS converted programs have chordal interference graphs; there are strong theoretical reasons why it is appropriate to pair the two. However, although coloring is optimal, when spilling is necessary we resort to non-optimal heuristics.

2 Register Alloc. via Chordal Graph Coloring

In order to talk about chordal graphs, it is necessary to first define some terminology.

Definition 2.1 A *clique* is an undirected graph $G = (V, E)$ such that forall $v_1, v_2 \in V$ such that $v_1 \neq v_2$ it is the case that $(v_1, v_2) \in E$.

Definition 2.2 The *neighborhood* of a vertex $v \in V$ in a graph $G = (V, E)$ is the subgraph of G induced by the neighbors of v.

Definition 2.3 A vertex $v \in V$ is *simplicial* if its neighborhood in $G = (V, E)$ is a clique.

Definition 2.4 A *Simplicial Elimination Ordering* of G is an ordering of vertices in G such that every vertex v_i is a simplicial vertex in the subgraph induced by $v_j | 0 < j \leq i$.

See Pereira and Palsberg for an example.

Definition 2.5 An undirected graph without self-loops is chordal if and only if it has a simplicial elimination order.

The original Pereira and Palsberg paper cites this result as a theorem, but for our purposes we will simply use it as a definition, i.e. we will refer to graphs with simplicial elimination orderings as chordal. Chordal graphs are optimally colorable in with linear time algorithms. The algorithm used in Pereira and Palsberg's allocator uses *maximum cardinality search* in

order to find a simplicial elimination ordering. A greedy coloring on a simplicial elimination ordering is optimal. [cite]

A more recent result is that programs in strict-SSA form have chordal interference graphs. [cite Hack] Without going into details, this result is based on the fact that in strict-SSA form every variable has a single contiguous live range. Variables with overlapping live ranges form cliques in the interference graph. This contributes to a variable being simplicial in its interference graph.

As we stated before, the crucial property that a variable is assigned only once is maintained by CPS code; programs that re-assign variables can be alpha-varied into programs that do not. Strictness guarantees the well-formedness of the CPS code. (Programs with variables that are used before they are declared would not type-check).

The Pereira-Palsberg algorithm first builds the interference graph using liveness analysis. It then performs maximum cardinality search to derive a simplicial elimination ordering. Using the simplicial elimination ordering, it greedily colors the interference graph. If coloring fails, it then performs post-spilling to handle variables that could not be colored to registers. Finally, coalescing occurs. For the most part we were able to adopt this algorithm off the shelf.

3 Post-Spilling Heuristics

Although the Pereira-Palsberg paper gives two reasonable heuristics for post-spilling, their presentation contains a notable omission. The motivation behind their spilling algorithm is captured by the following quotation: "We propose to spill nodes in a single iteration, by removing in each step all nodes of a chosen color from the colored interference graph. The idea is that given a K-colored graph, if all the vertices sharing a certain color are removed, the resulting subgraph can be colored with $K - 1$ colors."

Although nothing in this statement is untrue, there is no mention of whether such a procedure is safe with respect to the program transformation induced by the coloring and spilling. Spilled registers are spilled onto stack locations. It is not unreasonable for an architecture to limit which operands to particular instructions may be memory references or dereferences. Consequently, it may be necessary to shuttle data from the stack location into a temporary register in order to perform a particular instruction. Consider the situation where a RISC machine with two registers in which all operands to addition must be in registers and spilling has resulted in temps a and b being spilled to stack offsets 4 and 8 respectively. When transforming the instruction $x = a+b$, the values of a and b must be shuttled in from the stack via registers. However, if there are live values in the two machine registers, then there are actually no registers available for shuttling.

If we are concerned with safe code transformations, then the quoted spilling procedure is inadequate. Much of the implementational and theoretical work of this project with creating and coding up spilling heuristics that were safe with respect to program transformation. As a starting point, we chose to simply use the Pereira-Palsberg spill heuristics, but instead of K-coloring the interference graph, we would (K-2)-color the interference graph while reserving 2 registers for shuttle. The number 2 is architecture specific. In particular, it is the maximum number of input operands that must be registers for any instruction used by the compiler in the target instruction set.

Exploring the area of efficient post-spilling algorithm lead to the development of a number of different spilling algorithms, which we will refer to as spillers.

In the following subsections, we will describe our spillers. The two base spillers are implemented are functions with type `heuristic`. Abstractly, a heuristic takes in an interference graph, a coloring, a maximum number of colors, and returns an updated coloring, a coloring of the spilled registers to stack values, and identifiers for the two shuttle registers. This is enough information that a program transformation function can re-write the input program using the computed allocation. We describe our optimizations on heuristics as functions of type `heuristic -> heuristic`. In our implementation, it is safe to arbitrarily chain together these base heuristics and their optimization functions to create new heuristics. Consequently, we refer to them as spill combinators.

3.1 Highest

The spill-highest heuristic is the naive heuristic given in the Pereira-Palsberg paper. It is very simple to implement, as the colors to be spilled can be chosen simply by identifying the colors that are greater than K. In order to make this heuristic safe with respect to program transformation, we must reserve two registers for shuttling spilled values. This conservatism is often not necessary and later improvements seek to

address this issue.

3.2 Least Used

The second heuristic given is Pereira and Palsberg is spill-least-used, where the most used colors are allocated to registers and the least used colors are spilled. With respect to the total number of spills, this is in some sense optimal, because it allocates the greatest number of registers and spills the least. However, this is ignorant to how often spilled registers are used, so this is not necessarily optimal in terms of total number of loads and stores arising from spilling. Like the previous heuristic, we are conservative and reserve two registers for shuttling.

3.3 Try-in-k

Our first spill optimization, called try-in-k, is based on the idea that if an interference graph is K-colorable, then there is no point in saving two registers for shuttling. As such, it is a higher order spill combinator that takes in a heuristic and returns a heuristic. By counting the number of colors used in the coloring, it is able to determine whether an interference graph is K-colorable. If it is, then no spilling occurs. Otherwise, the input spilling heuristic is used.

3.4 Partition Coloring

The partition-optimize optimization is a more sophisticated version of try-in-k. It is developed on the observation that once you allocate $K - 2$ colors, then the problem of making better use of the final two registers is essentially 2-coloring all the spilled registers ignoring the registers that were already colored. It is also likely that the interference graph of the spilled registers is not connected, so the problem can be further subdivided into coloring the connected subgraphs of the interference graph for spilled registers. To capitalize on this, we partition the subgraph of the interference graph induced by the spilled registers into connected subgraphs. These subgraphs represent shared live-ranges. If the initial graphs were generated from strict-SSA code, this is actually a single contiguous live range. In order for shuttling registers to be needed in a particular live range, there must be a spilled value in use over that live range. Our algorithm proceeds by attempting to 2-color each individual subgraph. If one is 2-colorable, then it is safe to color it using the shuttle registers, as we know the shuttle registers will not be needed for retrieving stack values over relevant live range. If the subgraph is not 2-colorable, then we simply spill it again. For the sake of simplicity, we re-used our coloring code to attempt to find two colorings. A more aggressive implementation should probably use an algorithm based detecting finding bi-partite graphs.

3.5 Coalescing

The greedy coalescing algorithm given in Pereira and Palsberg is a transformation on the coloring that is dependent on interference information. For each two move-related nodes, it attempts to find free a color that is not used by the union of the set of their neighbors colors. If one exists, the two nodes are then colored to the free color. Assuming the interference graph abstract data type is able to provide a list of copies, then we may implement coalescing as a spill heuristic transformer. Operationally, it would run the input heuristic and then apply greedy coalescing on the output. Implementationally the `coalesce` combinator has the same type as any of the other spill optimizations. Noting that the problem is basically the same for spilled registers (except that we can always find some colors), we also coalesce spilled values in a similar fashion. When successful, this avoids expensive memory to memory copies.

3.6 More Optimizations

There are additional aggressive optimizations that could be done based on determining when only one shuttle register is necessary. Such analyses presumably begin with information about which pairs of registers are both inputs to the same instruction. From, this one can determine which, if any, pairs of registers will need to be shuttled simultaneously. In live ranges where no such pairs exist (but some shuttling is still necessary) one of the shuttle registers can be reclaimed. An optimization based on this concept has not been implemented, but its possible such a transformation could be useful. In our particular instruction set there are actually very few operators that require two input operands, so its possible such optimizations could regularly reclaim a register.

4 TAL + Simulator

4.1 TAL Design

The design of our typed assembly language was strongly motivated by the structure of the preceding interme-

diate language in our compiler's translation chain. We mapped binding operations in that intermediate language to equivalent operations using registers instead of values, mapped values to instructions that load values into registers, and mapped flow-control expressions to branching instructions. The resulting assembly language bears a strong similarity to a RISC-like assembly language with additional type-changing copy operations. Most values are conceptually represented as pointers into a heap, so all values can be represented by a single register or stack slot. One instruction, the product-formation instruction (`prod`), is considered as a macro for a sequence of instructions to write values into a tuple, and counted accordingly in the simulator's statistical instrumentation.

4.2 Simulator

We wrote a simulator (virtual machine) for our typed assembly language using small-step transitions between simulated machine states. The simulated machine, though untyped, detects operations on incompatible values as well as reads to uninitialized registers and stack slots. Many common register allocation errors produce simulation errors of this form, so successful evaluation on the simulator provides a good (albeit approximate) indication of successful register allocation.

We instrumented the simulator with mechanisms for tallying instructions executed, including individual counts for stack reads and writes, register-register copies, and type-only operations. In addition, we wrote a small static analysis to obtain similar information for instructions emitted in the generated assembly code.

5 Empirical Results

We tested and timed our register allocator's performance on a number of test cases generated from previous phases of the compiler. The register allocator itself ran to completion on all tests; the resulting code ran successfully on the simulator in 555 of 560 cases.

The Pereira-Palsburg coloring algorithm is supposed to run in linear time; even with our modified spilling algorithms, the performance of our register allocator followed an approximately linear pattern. As expected, our spilling optimizations tended to decrease the number of spills generated; the additional complication of the spilling heuristics often required a small amount of additional time, though the reduced spill generation actually improved compilation times for a few cases.

6 Conclusion