Optimization of Invocation Interfaces Using Pruning and Annealing

VERSION 0.1

Cortland D. Starrett
cort@ecn.purdue.edu
Graduate School of Electrical Engineering
Purdue University

September 15, 2008

1 Scope

Some study and a great deal of experimentation have been performed on techniquess for optimizing program invocation interfaces for speed. Intermediate results of experimentation are presented along with the algorithms and methods used. Plans for continued work are discussed. The problem studied could likely benefit from some of the work in the area of graph coloring (and specifically "arc coloring").

The object of this project is to streamline the allocation of high speed storage (registers specifically) in blocks of code making invocations. Passed parameters and local variables are considered. The algorithms used attempt to cause the variable allocations in one block of code to seek the speed optimum storage in consideration of other affected blocks of code. Variable allocation collisions are minimized across call/return interfaces.

Some examples:

given:

$f(a,b)g(b);$

It is possible to allocate storage such that f allocates b to the same high speed storage location as g allocates b. Thus, when f invokes g, no "swapping" or saving/restoring is required.

given:

$f(a,b)g(a);g(b);g(a);$

In this instance, g's allocations should optimize for a because of invocation frequency. This connotes some sort of algorithm which assigns weights and probabilistically determines optimum allocation.

given:

$f(a,b)return(a*b)/*multfunction*/$
$g(c)return(f(c,c))/*squarefunction*/$

This example show that no static allocation algorithm can always achieve perfect optimization. Assuming one and only one variable assigned to each one and only one physical storage unit, a collision is guaranteed. However, games may be played with additional storage and function replication.

2 Experimentation

For experimentation, a live system of embedded control code modules was used. This code contained 745 code modules (invocable functions and procedures), with [approximately 990] calls, passing [approximately 2000] parameters and local variables. No recursion was present in the code.

The original compiler (actually an extensive macro processor and assembler) allocated variables *positionally*. In $f(a, b)$, a would get $R1$; b would get $R2$. In $f(a, b)declc$;, a gets $R1$; b gets $R2$, and c gets $R3$.

The optimizing compiler allowed allocation in arbitrary order and storage skipping. For example, in $f(a, b)$, a might get $R3$ and b might get $R1$.

A call tree with variable declaration information was extracted statically from the source. A digraph was constructed from the call tree information. With this data structure, it was a simple matter to identify and measure the number of local variables and parameter collisions across invocation interfaces.

3 Recursive Pruning

A pruning technique was applied to the digraph. It was noted that any module having only a single invocation interface could arbitrarily adapt to the one interface. Put another way, a module which makes only one invocation or (exclusive) is invoked in only one place may defer the allocation of parameters to the module on the other end of the interface. In graph terms, arcs are eliminated from the digraph.

This pruning can recurse. After the elimination of "single interface" modules, the graph can be repruned unitl all modules remaining are interrelated with at least two interfaces each.

At this point, a significant number of collisions have been eliminated (depending on the call tree digraph). For the remaining allocation, collisions must be statistically minimized.

4 Static/Dynamic Analysis

One approach to minimization is to treat all interfaces equivalently. A reduction method is applied to achieve the absolute least number of interface variable collisions. This approach may or may not produce the fastest code. It may turn out that interface variables among low priority modules become optimized at the expense of higher priority interfaces.

Additional statistics can be gathered either statically or dynamically to dertermine weighting factors to be used as input to the optimization. For example, call frequencies, variable usage statistics, module priority and code path information can be used to give some interfaces and/or specific variables preference over others.

In my experimentation, all of the above were used as input to the statistical optimization portion of the work.

5 Shake Down

At this point, with all of the weighting statistics factored in, a simulated annealing was performed to attempt to achieve a minimum (in this case, minimum number of collisions).

Several variations of the shake down were tried. The simulated annealing uses a voting scheme which uses collision and weighting statistics to determine which variable gets allocated to each

internal register. The choice of a starting point is arbitrary and can have an affect on the allocation. Several starting points were tried.

The voting algorithm indentified the optimum allocation for a *single* interface. Often this allocation proved to catalyze optimization on other interfaces. However, occasionally one optimum interface can cause inefficiencies in other interfaces. To ameliorate this, a population of optimizations was chosen. Five permutations of an interface optimization were performed, the two best and the one worst survived.

6 Results

In the experiment code library, after pruning, 225 variable collisions remained in the source. The shake down was able to reduce this to 45 absolute collisions. Numbers higher than 45 were experienced when weighting factors were considered.

The average number of instructions per call was reduced from five to less than two. $20\mu sec$ was eliminated from the critical path on a benchmark run. Performance curves were visibly shifted during stress runs on the code.

7 Future

- Formalize the shake down algorithm using graph theory.
- Apply methods to a simple C-like compiler.
- Use daVinci (graph visualization tool for Linux) to provide clear abstractions of the algorithms.
- Consider recursion.
- Allow control flow with invocable modules.
- Allow aliasing.
- Package into a real compiler.