Computing Runtime Attributed Call Graphs and Displaying Call Graphs as Trees

Stefan Kaes

Version 0.4

The problem

Suppose we are given a call tree of a program as a graph $G_0 = (V, E)$. V is the set of call frame identifiers, and $(x, y) \in E$ iff x directly precedes y on the call stack during runtime. The function name of call frame x is given by a mapping $\mathsf{N} : V \to A^*$ and the cost of executing frame x (excluding called functions) is given by $c_0 : V \to \mathbb{R}$.

How do we determine a call graph $G_t = (V_t, E_t)$, and cost functions c_i, c_t, such that $c_i(f)$ gives the total amount of time spent in function f, excluding called functions, and $c_t(f)$ yields the total runtime of f, including called functions?

Basic definitions

For a given graph G, let functions p_G and s_G denote the set of predecessors and successors in G:

$$p_G(x) = \{\, y \mid (y, x) \in E \,\}, \qquad s_G(x) = \{\, y \mid (x, y) \in E \,\}\,.$$

Since G_0 is a tree, we have $|p_{G_0}(x)| \leq 1$ for all $x \in V$ and no path $p = p_1, \ldots, p_n$ exists with $p_1 = p_n$, where $(p_i, p_{i+1}) \in E$ for $i \in \{1, \ldots, n-1\}$.

The total cost $c(x)$ of frame x can be defined as

$$c(x) = c_0(x) + \sum_{y \in s_{G_0}(x)} c(y)\,.$$

Note that c is well defined since G_0 is acyclic, and can be computed in linear time, i.e. $O(|V| + |E|)$.

We can also easily compute the relative cost $c_r(x) = c(x)/c(r) * 100$ of a given call frame, where r is the root of G_0, i.e. $p_{G_0}(r) = \emptyset$.

Computing the call graph

The call graph $G_t = (V_t, E_t)$ is now obtained by merging nodes of the call tree with identical names. Define an equivalence relation $\approx$ on V using the function name mapping f:

$$x \approx y \iff \mathsf{N}(x) = \mathsf{N}(y)\,.$$

Let $[x]_\approx$ denote the equivalence class of $x \in V$ w.r.t. $\approx$, i.e.

$$[x]_\approx = \{\, y \mid x \approx y \,\}\,.$$

Since there is exactly on equivalence class for a given function name $\mathsf{N}(x)$, we can identify $\mathsf{N}(x)$ with $[x]_\approx$. Then $G_t = (V_t, E_t)$ is given as

$$V_t = \{\, \mathsf{N}(x) \mid x \in V \,\} \quad \text{and} \quad E_t = \{\, (\mathsf{N}(x), \mathsf{N}(y) \mid (x, y) \in E \,\}\,.$$

Cost functions c_i and c_t are defined as

$$c_i(\mathsf{N}(x)) = \sum_{y \in [x]_\approx} c_0(y)\,, \qquad c_t(\mathsf{N}(x)) = \sum_{y \in \mathsf{Min}_E([x]_\approx)} c(y)\,.$$

where $\mathsf{Min}_E(X) = \{x \in X \mid \neg\exists\, y \in X : (y, x) \in E^+\}$ and E^+ is the transitive closure of E.

Displaying the call graph as a tree

Visualizing the call graph G_t on screen as a real graph would use enormous amounts of screen estate and would be very cumbersome to manipulate for the end user. We therefore propose a tree display as follows:

For each function f there will be $|p_{G_t}(f)|$ nodes in the display tree $G_T = (V_T, E_T)$. One of them, let's call it master node of f ($m(f)$), will have child nodes for all functions called by f, the other nodes will just link to the master node (otherwise we would not get a tree). For each function f we will have nodes $f_{g_1}, \ldots, f_{g_n}$, where $g_1, \ldots, g_n$ are the callers of f. If f has no callers (is called on top level), we will have just one node named f. Thus,

$$V_T = \{\, f_g \mid (g, f) \in E_t \,\} \cup \{\, f \mid p_{G_t}(f) = \emptyset \,\}$$

and

$$E_T = \{\, (m(g), f_g) \mid (g, f) \in E_t \,\}\,.$$

It is easy to see that G_T is indeed a graph. The big question is of course: given function name f, which will be the master node of f? For top level functions, we have obviously $m(f) = f$. For the remaining functions, the choice is less obvious. One possible approach is to use the contribution of

f's callers to the total program run time as an oracle and resolve ties using lexicographic ordering of function names. Hence,

$$m(f) = \begin{cases} f & \text{if } p_{G_t}(f) = \emptyset, \\ max_<\{\ (c_t(g), g)\ \mid\ f_g \in V_T\ \} & \text{otherwise.} \end{cases}$$

Relation $<$ is the usual strict ordering on tuples, defined as

$$(a, b) < (a', b') \Leftrightarrow a < a' \vee (a = a' \wedge b < b')\,.$$

Implementation notes

G_t can be constructed in expected linear time over the size of G_0; i.e. in $O(|V| + |E|)$ steps, if we implement an expected $O(1)$ lookup routine for testing whether $\mathsf{N}(x)$ has $\mathsf{N}(y)$ as a successor (predecessor) in G_t. This can be done e.g., by using hashes to store the successors and predecessors of a node in V_T.

Computation of c_i is obviously linear and c_t is also linear if Min_E can be computed in linear time. For the computation of Min_E we simply traverse the tree in preorder, passing along a hash of function names that occured so far along the path. Then x is minimal, if $\mathsf{N}(x)$ is not in the hash. If it is, we add it to the hash and traverse the child nodes with the modified hash. If it is not minimal, then we traverse the childs with the unmodified hash.

Finally, G_T can be constructed from G_t in $O(|V_t| + |E_t|)$ steps.