Optimal Basic Block Reordering via Hammock Decomposition

Oleg Medvedev
Fac. of Mathematics and Mechanics
St. Petersburg State University
Universitetski pr., 28, 198504, St.Petersburg, Russia
Email: dours@mail.ru

***Abstract*—Many optimizing compilers use basic block reordering to reduce conditional branch misprediction penalties, decrease the number of unconditional branches and to improve the instruction cache performance. To perform this optimization one has to solve an NP-hard problem of finding a covering of a weighted control flow graph with a node-disjoint set of simple paths with maximal total weight; the problem is usually solved heuristically. We present the precise method which decomposes the initial graph into the set of subgraphs of certain kind — *hammocks* — and then solves a subproblem for each subgraph by an exponential branch-and-bound algorithm. The algorithm was tested on several real-world C programs, and its running time was as small as the rest of the compilation time for 91% of the tested graphs. Unfortunately it still failed to finish in a reasonable time for the remaining graphs since they didn't have enough hammocks. We consider fixing up this drawback as a future work.**

I. Introduction

Basic block reordering is a well-known optimization which allows to reduce the number of pipeline flushes caused by conditional jumps, improve instruction cache performance and decrease the number of unconditional jump executions. It gives much better results when applied to a program together with its profiling information.

This optimization is typically reduced to the following graph problem: given a control flow graph of a function with integer weights on its edges, find its covering with a set of node-disjoint simple paths with maximal total weight. Here a weight of a path is a sum of weights of its edges. A weight of an edge is actually a result of profiling — it is equal to a number of times a control passed through the edge during function execution on a set of tests.

This problem can be approximately solved by a greedy approximation algorithm, which is fast and gives answers that are close to precise or by reducing it to a traveling salesman problem. The latter algorithm works rather fast in practice and gives an answer that is very close to the precise one, but *is not* precise.

This paper presents an approach to solve the problem *precisely* and fast. This approach uses *hammocks* to decompose the problem and construct an answer incrementally starting from the smallest hammocks and proceeding to the whole graph using simple branch-and-bounds method on each step.

This approach leads to an algorithm that usually works fast, but may fail to finish for graphs that don't have enough hammocks.

II. Terminology

The following notation is used throughout this paper:

- a control flow graph is a directed graph with a dedicated starting node from which all other nodes are reachable;
- for an edge e, $src(e)$ denotes its source and $dst(e)$ — its destination;
- for a node v, $ins(v)$ denotes a set of incoming edges and $outs(v)$ — a set of outgoing edges;
- "the path covering problem" is a problem to cover a weighted graph with a node-disjoint set of simple paths with maximal total weight of edges in those paths.

III. The optimization and its reduction to the graph problem

Consider a control flow graph $G = (V, E)$ of some function with integer weights on its edges given by a function $w : E \to \mathbb{N}$. For an edge e $w(e)$ is a number of times a flow of control passed from $src(e)$ to $dst(e)$ during function execution on a set of tests. Thus, it is a kind of profiling information. First, consider a following simple model of processor jump penalties: every non-fallthrough jump (either conditional or not) takes one cycle. That is, a jump from a previous instruction to the next one takes 0 cycles to be performed and every other one takes 1 cycle. Suppose that we want to minimize a number of cycles a processor spends on jumps while executing the function on the same set of tests. If the set is representative this will also accelerate execution on an average test. Suppose that the only action allowed to achieve this is to reorder the basic blocks of the function in its machine code.

Let $\{p_1, \ldots, p_l\}$ be the covering of G with node-disjoint paths of maximal total weight. Let $p_i = (u_{i,1}, u_{i,2}, \ldots, u_{i,k_i})$. Then $(u_{1,1}, u_{1,2}, \ldots, u_{1,k_1}, u_{2,1}, \ldots, u_{l,k_l})$ is a reordering we need, because all the edges in the covering correspond to jumps that take 0 cycles to be executed while all other edges take 1 cycle for each time they are passed. Thus the maximal amount of cycles is saved by such an ordering.

The processor model where each non-fallthrough jump takes 1 cycle is too simple. In this paper a more reasonable model is used which is defined by the three constants:

- a — a number of cycles a processor spends to execute a taken conditional jump (the "pipeline flush" case);
- b — a number of cycles spent on a not taken conditional jump;
- c — a number of cycles spent on an unconditional jump.

The optimal reordering of nodes of a graph $G = (V, E)$ for this model can be reduced to the path covering problem for a graph $G' = (V', E')$, where $|V| = |V'|$ and $|E'| \leq \frac{3}{2} \cdot |E|$. This reduction is provided in [7].

IV. Existing methods to solve the path covering problem

This section gives an overview of algorithms to solve the path covering problem for the special case where weights of all edges are equal to 1 ("unweighted case") as well as for the case of arbitrary non-negative integer weights.

A. The unweighted case

The NP-hardness of the problem even for this case is proven in [3]. An algorithm to solve it in polynomial time for dags is given there too. It can also be polynomially solved for all reducible control flow graphs, which is proven in [4]. An interesting linear algorithm for the graphs representing "structural" programs is provided in [5]. Unfortunately the class of such graphs is very small.

B. The weighted case

Two approximation algorithms are known.

The greedy algorithm [1] starts with a covering containing no edges, where each node is a separate path. Then it processes all the edges in order of decreasing of their sizes. It adds an edge (u, v) to the covering if v is a first node of some path and u is a last node of some *other* path, thus merging those two paths into a one. It skips the edge otherwise.

Traveling salesman. The authors of [2] reduced the path covering problem to the directed traveling salesman problem, which can be approximately solved by a special solver. They claim that their approach gives an answer that averages within 0.3% of a provable optimum on their benchmarks. The speed of their algorithm is less than the speed of other compilation passes. The difference between the greedy algorithm and their approach is about 0-10%.

V. The proposed approach

This section first describes a simple branch-and-bounds algorithm that cannot solve the path covering problem for general control flow graphs because it works too slowly. Then a definition of a hammock is given along with a way to decompose the problem using hammocks. Since each part of the decomposed problem is small enough it can be solved by the branch-and-bounds algorithm in a reasonable time.

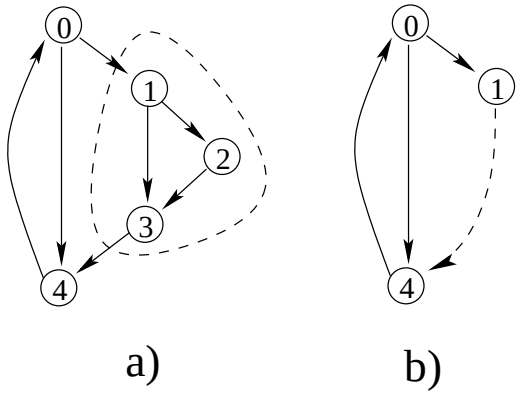

Fig. 1. A graph and a hammock on the left, the graph with the collapsed hammock on the right

A. Branch-and-bounds

Suppose we are given a graph $G = (V, E)$ where $V = \{v_1, \ldots, v_n\}$; let $w : E \to \mathbb{N}$ be a weighting function.

The algorithm is a recursive procedure of one parameter i — a number of a node to be processed. It works as follows:

1) Case $i < n$. Suppose that we have already constructed A — a path covering of the subgraph of G consisting of nodes $v_1, \ldots, v_i$. We compute an upper bound for a weight of the best covering containing A using the bounding function described below. If this bound doesn't exceed the current best we do nothing. Otherwise for each $e \in out(v_i)$ we try to add e to A. If this edge doesn't break the covering (i.e. doesn't enter a node that some other edge enters and doesn't transform any path to a cycle) we recursively proceed to the node v_{i+1};
2) Case $i = n$. A is a covering of the whole graph in this case. It is compared to the current best answer, which is updated if needed.

The bounding function is defined as follows: let A be the current path covering for the nodes $\{v_1, \ldots, v_i\}$. Then a weight of any covering that contains A will not exceed $\sum_{e \in A} w(e) + \sum_{j=i}^{n} \max_{e \in out(v_j)} w(e)$.

This algorithm works fast only for small graphs (several tenths of nodes).

B. Decomposition

In this section we present a way to decompose the problem using subgraphs of a certain kind called "hammocks".

1) Definition of a hammock: Let $G = (V, E)$ be a graph and $G' = (V', E')$ — its subgraph. A node $v' \in V'$ is called an *entry* node of G' iff it is the start node of G or $\exists e \in ins(v') : src(e) \notin V'$. A node $u \in V$ is called a *final* node iff $u \notin V'$ and $\exists e \in ins(u) : src(e) \in V'$. Informally, if we are traveling from the start node of G then we can enter G' only through some of its entry nodes and can leave it only to some of its final nodes. G' is called a *hammock* iff it has exactly one entry node and at most one final node. For example, a subgraph consisting of nodes $\{1, 2, 3\}$ on a figure 1a) has one entry node — 1 — and one final node — 4 — and, thus, is a

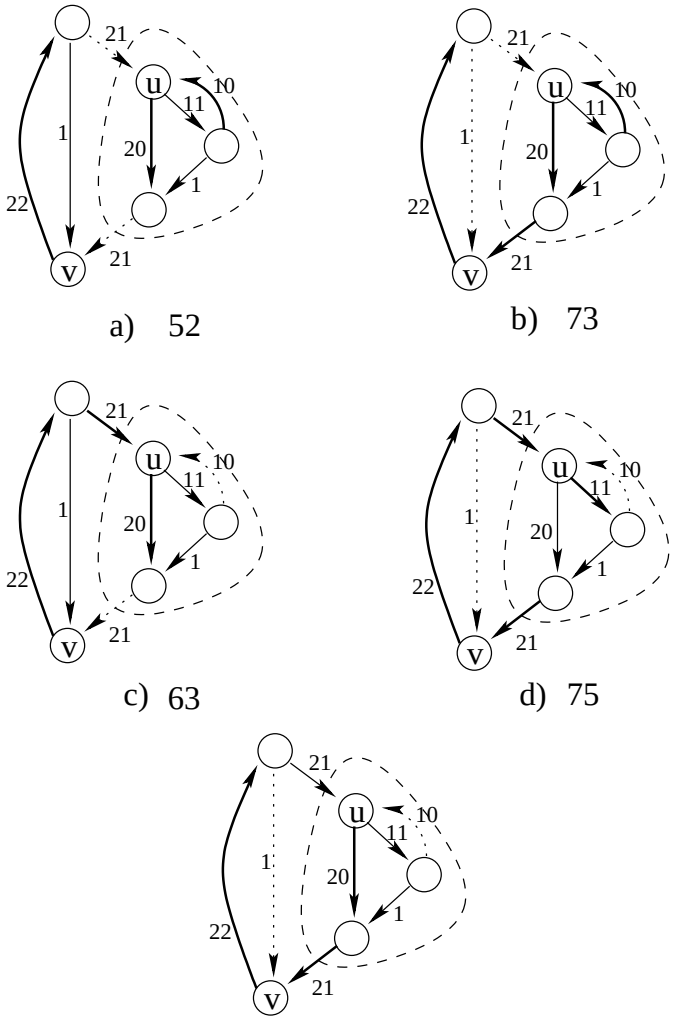

Fig. 2. the five subproblems to be considered to find an optimal answer

hammock. On the contrary a subgraph consisting only of node 1 is not a hammock because it has two final nodes — 2 and 3.

The important idea is that a hammock with entry node u and final node v can be replaced by an edge (u, v), which simplifies the graph. A graph with a selected hammock is shown on figure 1a). A corresponding simplified graph is shown on figure 1b).

2) How to decompose the path covering problem for a given hammock: suppose we have some control flow graph G and a hammock G' inside it. Suppose we also can find an optimal solution for G' and for G without G'. The main idea of our approach is to use these capabilities to obtain an optimal answer for the whole graph G. We will show this idea using the graph and the hammock on figure 1 as an illustration. The hammock there consists of nodes 1, 2, 3 and the rest of the graph consists of nodes 0 and 4.

The idea is to compute five pairs of optimal coverings of the hammock and the rest of the graph, satisfying different boundary conditions, then join the coverings in each of the pairs and take the best one of the resulting five coverings of the whole graph. Let's call the hammock G', the rest of the graph — G'', the entry node of the hammock u and the final one — v. Those five cases are described below and are shown on respective subfigures of figure 2. The edges that are not allowed to be covered in the respective case are dotted and those contained in the optimal covering are bold. Each edge's weight is written near it. A total weight of an optimal covering for each case is written below an image.

The five pairs of coverings are:

a) an optimal covering of G' that is allowed to enter u but is not allowed to enter v; an optimal covering of G'' that is not allowed to enter u but is allowed to enter v;
b) an optimal covering of G' that is allowed to enter u and is allowed to enter v; an optimal covering of G'' that is not allowed to enter u and is not allowed to enter v;
c) an optimal covering of G' that is not allowed to enter u and is not allowed to enter v; an optimal covering of G'' that is allowed to enter u and is allowed to enter v;
d) an optimal covering of G' that is not allowed to enter u but is allowed to enter v and is not allowed to contain any path from u to v; an optimal covering of G'' that is allowed to enter u but is not allowed to enter v and is allowed to contain a path from v to u;
e) an optimal covering of G' that is not allowed to enter u but is allowed to enter v and is allowed to contain a path from u to v; an optimal covering of G'' that is allowed to enter u but is not allowed to enter v and is not allowed to contain any path from v to u.

The intuition behind these cases is rather simple: we cannot just find optimal coverings for G' and G'' and join them, because we are at hazard to obtain a covering that has two paths simultaneously entering u or v or a cycle that goes from u to v completely inside G' and back in G''. Note that we need to explore all possible ways to include u and v in path coverings for G' and G''.

C. How the decomposition and the branch-and-bounds algorithm are fit together

The proposed algorithm to solve the path covering problem works as follows:

- given a graph G find all the hammocks with more than 4 nodes in it and sort them in order of increasing of their sizes. Let $H_1, \ldots, H_k$ be the sorted hammocks;
- process the hammocks in this order and for each hammock H_i
  - find coverings for the 5 cases described above using the branch-and-bounds algorithm;
  - let u be the entry node of H_i and v be the final one. Replace H_i with one "edge" (u, v) in all the hammocks that are not yet processed. This edge has 5 different weights corresponding to weights of the respective coverings;
- the biggest hammock H_k is the graph G, thus we obtain the answer for G as the answer for H_k.

TABLE I
TIME SPENT TO COMPILE AND OPTIMIZE THE TEST PROGRAMS, IN SECONDS

name	gcc time	opt time	number of big graphs	avg size of big graphs	max size of good graph	number of bad graphs	avg size of bad graphs
gzip	1.32	0.52	8	110.75	149	0	0
bzip2	2.24	7.87	16	240.25	648	1	648
awhttpd	0.78	1.39	2	84	84	0	0
gnuchess	6.681	12.84	40	191.5	552	5	210.2

In other words the algorithm solves the problem for some hammock and collapses it to one "edge" in the enclosing hammocks thus simplifying the problem for them. Note that a hammock is collapsed not to a regular edge, but to an edge that must be treated in a special way: the branch-and-bounds algorithm has to consider five different cases for such an edge paying respect to boundary conditions mentioned above.

VI. RESULTS

The algorithm was implemented on basis of the PRAN-LIB [6] library for control flow optimizations. It parses the assembler output of gcc, reorders basic blocks and prints assembler code back.

We tested our implementation on four open-source programs:

- `gzip` — general-purpose data compressor;
- `bzip2` — another data compressor (a bit better, but slower than gzip);
- `awhttpd` — a tiny web server;
- `gnuchess` — a well known program that plays chess.

The average improvement of our path covering algorithm over the greedy approach is about 3% for these tests, which corresponds to results reported in [2]. We are mostly interested in the running time of the algorithm. The table I gives the following information:

- time spent by `gcc` to compile the program (without our optimization);
- time spent by our optimization for this program;
- a number of graphs with more than 80 nodes in the program;
- average size of these graphs;
- size of the maximal graph that our algorithm can process fast;
- a number of graphs that our algorithm failed to process fast;
- average size of these graphs.

Where "to process fast" means to process in less than 6 seconds on an Intel Core-Duo 1.83MHz processor. We didn't provide the results for the graphs with less than 80 nodes because we believe them to be insignificant.

It occurred that the algorithm failed to finish fast on 9% of the big graphs considered. The reason for that is not the size of those graphs but the lack of hammocks in them — this forces the algorithm to run the branch-and-bounds method on some large graph for which it fails to finish. The run time of the algorithm is comparable to that of the compiler for the majority of the graphs.

VII. FUTURE WORK

The algorithm described can easily be generalized to process subgraphs which have more entry and/or final nodes than hammocks. To use the generalized algorithm we may for example borrow some edges from a graph, then find a set of hammocks in it, then add the edges back and use the resulting large set of more complex subgraphs. This may help to find answers for the graphs which don't have enough hammocks.

REFERENCES

[1] K. Pettis and R. C. Hansen. "Profile guided code positioning". In Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, pp 16-27. ACM, June 1990

[2] C. Young, D. S. Johnson, M. D. Smith, D. R. Karger, "Near-optimal intraprocedural branch alignment". In Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation, pp 183-193. ACM, 1997

[3] F. T. Boesch and J. F. Gimpel. "Covering the points of a digraph with point-disjoint paths and its application to code optimization", Journal of the Association for Computing Machinery, Vol 24, No 2, pp 192-198, April 1977

[4] T. Hirata, A. Maruoka and M. Kimura. "A polynomial time algorithm to find a path cover of a reducible flow graph". Syst. Comput. Control 10, 3 (May-June 1979), 71-78.

[5] M. V. S. Ramanath and M. Solomon. "Jump Minimization in Linear Time". ACM Trans. Program. Lang. Syst. 6, 4, pp 527-545, 1984

[6] http://oops.tepkom.ru/projects/pranlib

[7] O. Medvedev. "Linearization of a control flow graph with respect to a profiling information". System Programming, Issue 2, 2006 (to appear).