Improved Basic Block Reordering[1]

Andy Newell and Sergey Pupyrev

Group 20: Zijian Zhang, Huiruo Zou, Yunhao Wang, Zhaoyuan Zhang

Overview

1. Introduction
2. Approach
3. Related Work
4. Optimization Model
5. ExtTSP
6. Heuristic Algorithm
7. Evaluation
8. Limitations and Potential Future Directions
9. References
10. Q & A

Introduction

PGO: Profile-Guided Binary Optimization

- Function and **Basic Block Reordering**
- Identical Code Folding
- Function Inlining
- Unreachable Code Elimination
- Register Allocation
- ……

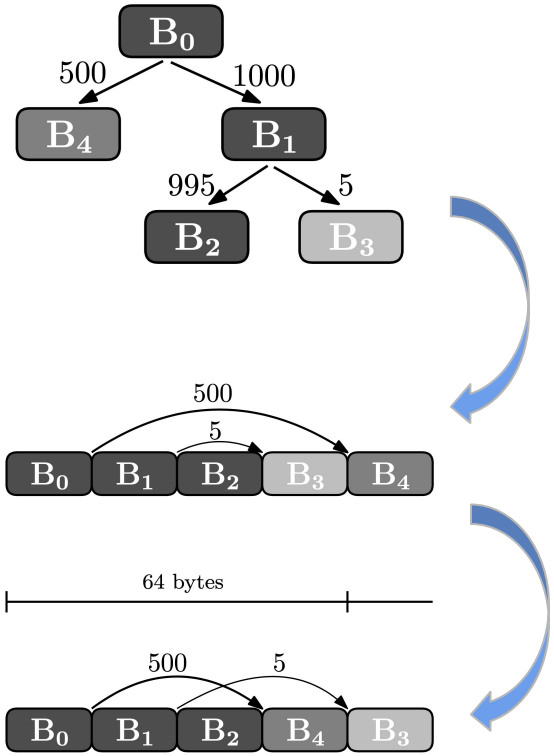
Introduction

Current techniques for basic block reordering:

- increasing the average number of instructions executed per cache line
- reducing the number of mispredicted branches
- minimizing cache line conflicts

This paper:

Design and implement an algorithm ***reordering*** basic blocks layouts in memory that ***directly*** optimizes the performance of an application.



** colored according to their hotness in the profile*

Approach

(1) Learn a ***proxy metric*** that describes the relationship between the performance of a binary and the ordering of its basic blocks.
   (a) ***identifying a set of features*** representing how basic block ordering can influence performance,
   (b) ***collecting training data*** by running extensive experiments and measuring the performance, and
   (c) ***using machine learning*** to select the best combination of the features for a score that best predicts CPU performance.

(2) Suggest an ***efficient algorithm*** that, given a control flow graph for a procedure, builds an improved ordering of the basic blocks optimizing the learned metric.

Related Work

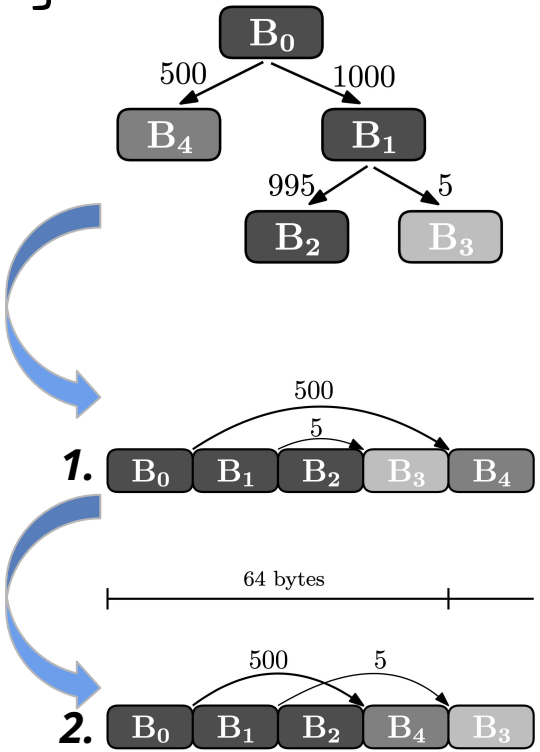
Pettis and Hansen[2]:

- Basis for the majority of modern code reordering techniques.
- An approach greedily merges chains of functions and is designed to primarily reduce ITLB misses.

Most of the existing works focus on *field reordering* and *structure splitting* based on the field hotness and data affinities.

Optimization Model - Basic Block Reordering

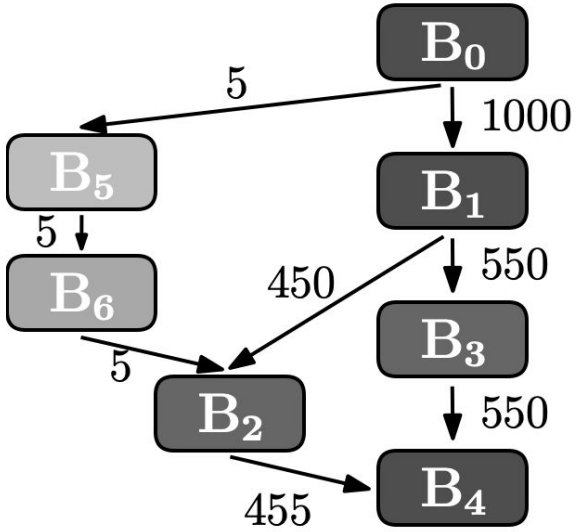
- Position blocks so that the **hottest** successor of a block will most likely be a **fall-through** branch, that is, located right next to the predecessor in memory layout. (**B0-B1-B2**)
- Reduces the number of **taken branches** and the working set **size of the I-cache**, while relieving pressure from the branch predictor unit. (**Order 1. to 2.**)



** colored according to their hotness in the profile*

Optimization Model - Data Collection

- Extract a weighted directed **control flow graph** for every function in the profiled binary.
- **Vertices** (basic blocks) and **Edges** (branches) of the graph are extracted via the BOLT infrastructure, which is based on LLVM.
- **Weights** between the basic blocks correspond to the total number of times the jumps appear in collected Last Branch Records (LBR), a list of the last 16 taken branches, in Intel X86 processor.



** colored according to their hotness in the profile*

Optimization Model - TSP

Reordering Problem Formulation: **TRAVELING SALESMAN PROBLEM** (TSP)

Definition: Given a directed control flow graph comprising of basic blocks and frequencies of jumps between the blocks, find an ordering of the blocks such that the number of fall-through jumps is maximized.

TSP score: $$\text{TSP} = \sum_{(s,t)} w(s,t) \times \begin{cases} 1 & \text{if } \text{len}(s,t) = 0, \\ 0 & \text{if } \text{len}(s,t) > 0, \end{cases}$$

Where ***w(s,t)*** is the frequency and **len(*s, t*)** is the jump length from branch s to t.

*jump length: the distance (in bytes) between the end of the source block to the beginning of the target block

Optimization Model - More Characteristics

The performance might also depend on other characteristics of a branch, which are **NOT** included in TSP.

- The length of a jump impacts the performance of instruction caches.
- The direction of a branch plays a role for branch
- The branches can be classified into *unconditional* (if the out-degree is one) and *conditional* (if the out-degree is two)

We need a new score to include these characteristics!

ExtTSP

As for scoring the ordering, we also need to consider some characteristics of branch

$$\text{ExtTSP} = \sum_{(s,t)} w(s,t) \times K_{s,t} \times h_{s,t}\big(\text{len}(s,t)\big)$$

1. $w(s,t)$: the frequency of branch s ➡ t
2. $0 \le K_{s,t} \le 1$: the weight coefficient modeling the relative importance of the branch optimization
   a. Six categories
      i. Conditional and Unconditional of fall-through, forward, and backward
3. $h_{s,t}$: a function accounts for the importance of branch length
   a. 1 for zero-length jumps
   b. 0 for jumps exceeding a prescribed length
   c. Monotonically decreased between these two values

$$(1 - (\frac{\text{len}(jump)}{M})^{\alpha})$$

Learning Parameters

The parameters for ExtTSP cannot be decided manually

- Run experiments on:
  - Clang and HipHop Virtual Machine (HHVM)
- Each experiment consists of:
  - Constructing A distinct ordering of BB
  - Running a binaring
  - Measuring performance metrics vis the Linux perf tool
- This paper evaluated 50 distinct block orderings and conducted 250 exp. (5 different algorithms).
- Trying to find the parameters that have the highest correlation with the performance.

Learning Parameters

$$\text{ExtTSP} = \sum_{(s,t)} w(s,t) \times \begin{cases} 1 & \text{if } \text{len}(s,t) = 0, \\ 0.1 \cdot \left(1 - \frac{\text{len}(s,t)}{1024}\right) & \text{if } 0 < \text{len}(s,t) \leq 1024 \\ & \text{and } s < t, \\ 0.1 \cdot \left(1 - \frac{\text{len}(s,t)}{640}\right) & \text{if } 0 < \text{len}(s,t) \leq 640 \\ & \text{and } t < s, \\ 0 & \text{otherwise.} \end{cases}$$

- After combining similar weights(diff< **0.05**) and excluding small values(value < **0.05**)
- Here is the final unified model

Heuristic for Basic Block Reordering in Memory

Greedy Heuristic

Algorithm 1: Basic Block Reordering

Input : control flow graph $G = (V, E, w)$,
the entry point $v^* \in V$

Output: ordering of basic blocks $(v^* = B_1, B_2, \ldots, B_{|V|})$

Function `ReorderBasicBlocks`

- **for** $v \in V$ **do** `/* initial chain creation */`
  - $Chains \leftarrow Chains \cup (v)$;
- **while** $|Chains| > 1$ **do** `/* chain merging */`
  - **for** $c_i, c_j \in Chains$ **do**
    - $gain[c_i, c_j] \leftarrow$ `ComputeMergeGain`(c_i, c_j);
  - `/* find best pair of chains */`
  - $src, dst \leftarrow \arg\max_{i,j} gain[c_i, c_j]$;
  - `/* merge the pair and update chains */`
  - $Chains \leftarrow Chains \cup$ `Merge`$(src, dst) \setminus \{src, dst\}$;
- **return** *ordering given by the remaining chain;*

Step 1: Initialize Chains

Function `ComputeMergeGain`(src, dst)

- `/* try all ways to split chain` src `*/`
- **for** $i = 1$ **to** blocks(src) **do**
  - `/* break the chain at index` i `*/`
  - $s_1 \leftarrow src[1 : i]$;
  - $s_2 \leftarrow src[i + 1 : \text{blocks}(src)]$;
  - `/* try all valid ways to concatenate */`
  - $score_i \leftarrow \max \begin{cases} \text{ExtTSP}(s_1, s_2, dst) \text{ if } v^* \notin dst \\ \text{ExtTSP}(s_1, dst, s_2) \text{ if } v^* \notin dst \\ \text{ExtTSP}(s_2, s_1, dst) \text{ if } v^* \notin s_1, dst \\ \text{ExtTSP}(s_2, dst, s_1) \text{ if } v^* \notin s_1, dst \\ \text{ExtTSP}(dst, s_1, s_2) \text{ if } v^* \notin src \\ \text{ExtTSP}(dst, s_2, s_1) \text{ if } v^* \notin src \end{cases}$
- `/* the gain of merging chains` src `and` dst `*/`
- **return** $\max_i score_i - \text{ExtTSP}(src) - \text{ExtTSP}(dst)$;

Heuristic for Basic Block Reordering in Memory

Step 2: Iterate through all pairs of Chains

Greedy Heuristic

Algorithm 1: Basic Block Reordering

Input : control flow graph $G = (V, E, w)$,
the entry point $v^* \in V$
Output: ordering of basic blocks $(v^* = B_1, B_2, \ldots, B_{|V|})$

Function `ReorderBasicBlocks`

```
for v ∈ V do                                   /* initial chain creation */
    Chains ← Chains ∪ (v);
while |Chains| > 1 do                          /* chain merging */
    for c_i, c_j ∈ Chains do
        gain[c_i, c_j] ← ComputeMergeGain(c_i, c_j);
    /* find best pair of chains                                          */
    src, dst ← arg max_{i,j} gain[c_i, c_j];
    /* merge the pair and update chains                                 */
    Chains ← Chains ∪ Merge(src, dst) \ {src, dst};
return ordering given by the remaining chain;
```

Function `ComputeMergeGain` (src, dst)

```
/* try all ways to split chain src                                      */
for i = 1 to blocks(src) do
    /* break the chain at index i                                       */
    s_1 ← src[1 : i];
    s_2 ← src[i + 1 : blocks(src)];
    /* try all valid ways to concatenate                                */
    score_i ← max { ExtTSP(s_1, s_2, dst)      if v* ∉ dst
                  { ExtTSP(s_1, dst, s_2)      if v* ∉ dst
                  { ExtTSP(s_2, s_1, dst)      if v* ∉ s_1, dst
                  { ExtTSP(s_2, dst, s_1)      if v* ∉ s_1, dst
                  { ExtTSP(dst, s_1, s_2)      if v* ∉ src
                  { ExtTSP(dst, s_2, s_1)      if v* ∉ src
/* the gain of merging chains src and dst                               */
return max_i score_i − ExtTSP(src) − ExtTSP(dst);
```

Heuristic for Basic Block Reordering in Memory

Step 3: Compute Merge Gain

Greedy Heuristic

Algorithm 1: Basic Block Reordering

Input : control flow graph $G = (V, E, w)$,
the entry point $v^* \in V$
Output: ordering of basic blocks $(v^* = B_1, B_2, \ldots, B_{|V|})$

Function `ReorderBasicBlocks`

- **for** $v \in V$ **do** `/* initial chain creation */`
  - $Chains \leftarrow Chains \cup (v)$;
- **while** $|Chains| > 1$ **do** `/* chain merging */`
  - **for** $c_i, c_j \in Chains$ **do**
    - $gain[c_i, c_j] \leftarrow$ `ComputeMergeGain`(c_i, c_j);
  - `/* find best pair of chains */`
  - $src, dst \leftarrow \arg\max_{i,j} gain[c_i, c_j]$;
  - `/* merge the pair and update chains */`
  - $Chains \leftarrow Chains \cup$ `Merge`$(src, dst) \setminus \{src, dst\}$;
- **return** *ordering given by the remaining chain;*

Function `ComputeMergeGain`(src, dst)

- `/* try all ways to split chain` src `*/`
- **for** $i = 1$ **to** blocks(src) **do**
  - `/* break the chain at index` i `*/`
  - $s_1 \leftarrow src[1 : i]$;
  - $s_2 \leftarrow src[i + 1 : \text{blocks}(src)]$;
  - `/* try all valid ways to concatenate */`
  - $$score_i \leftarrow \max \begin{cases} \text{ExtTSP}(s_1, s_2, dst) \text{ if } v^* \notin dst \\ \text{ExtTSP}(s_1, dst, s_2) \text{ if } v^* \notin dst \\ \text{ExtTSP}(s_2, s_1, dst) \text{ if } v^* \notin s_1, dst \\ \text{ExtTSP}(s_2, dst, s_1) \text{ if } v^* \notin s_1, dst \\ \text{ExtTSP}(dst, s_1, s_2) \text{ if } v^* \notin src \\ \text{ExtTSP}(dst, s_2, s_1) \text{ if } v^* \notin src \end{cases}$$
- `/* the gain of merging chains` src `and` dst `*/`
- **return** $\max_i score_i - \text{ExtTSP}(src) - \text{ExtTSP}(dst)$;

Heuristic for Basic Block Reordering in Memory

Step 4: Merge the pair of Chains with highest ExtTSP Gain

Greedy Heuristic

Algorithm 1: Basic Block Reordering

Input : control flow graph $G = (V, E, w)$,
the entry point $v^* \in V$
Output: ordering of basic blocks $(v^* = B_1, B_2, \ldots, B_{|V|})$

Function `ReorderBasicBlocks`

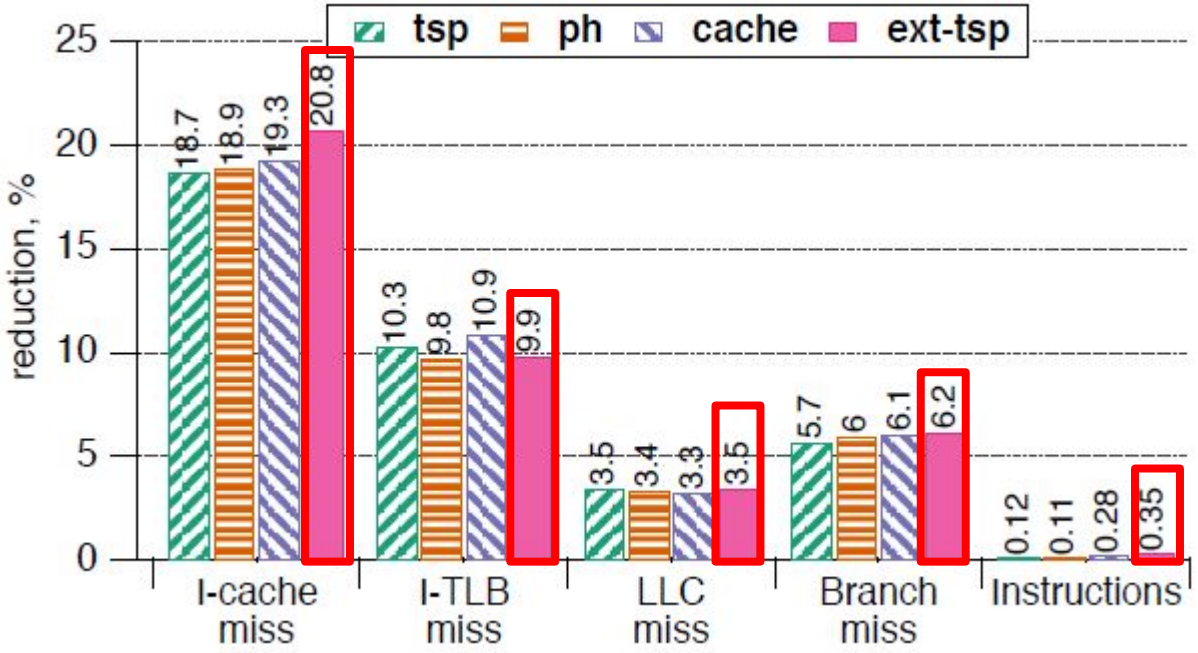
```
for v ∈ V do                                   /* initial chain creation */
    Chains ← Chains ∪ (v);
while |Chains| > 1 do                          /* chain merging */
    for c_i, c_j ∈ Chains do
        gain[c_i, c_j] ← ComputeMergeGain(c_i, c_j);
    /* find best pair of chains                                          */
    src, dst ← arg max_{i,j} gain[c_i, c_j];
    /* merge the pair and update chains                                 */
    Chains ← Chains ∪ Merge(src, dst) \ {src, dst};
return ordering given by the remaining chain;
```

Function `ComputeMergeGain` (src, dst)

```
/* try all ways to split chain src                                      */
for i = 1 to blocks(src) do
    /* break the chain at index i                                       */
    s_1 ← src[1 : i];
    s_2 ← src[i + 1 : blocks(src)];
    /* try all valid ways to concatenate                                */
    score_i ← max { ExtTSP(s_1, s_2, dst)    if v* ∉ dst
                  { ExtTSP(s_1, dst, s_2)    if v* ∉ dst
                  { ExtTSP(s_2, s_1, dst)    if v* ∉ s_1, dst
                  { ExtTSP(s_2, dst, s_1)    if v* ∉ s_1, dst
                  { ExtTSP(dst, s_1, s_2)    if v* ∉ src
                  { ExtTSP(dst, s_2, s_1)    if v* ∉ src
/* the gain of merging chains src and dst                               */
return max_i score_i − ExtTSP(src) − ExtTSP(dst);
```

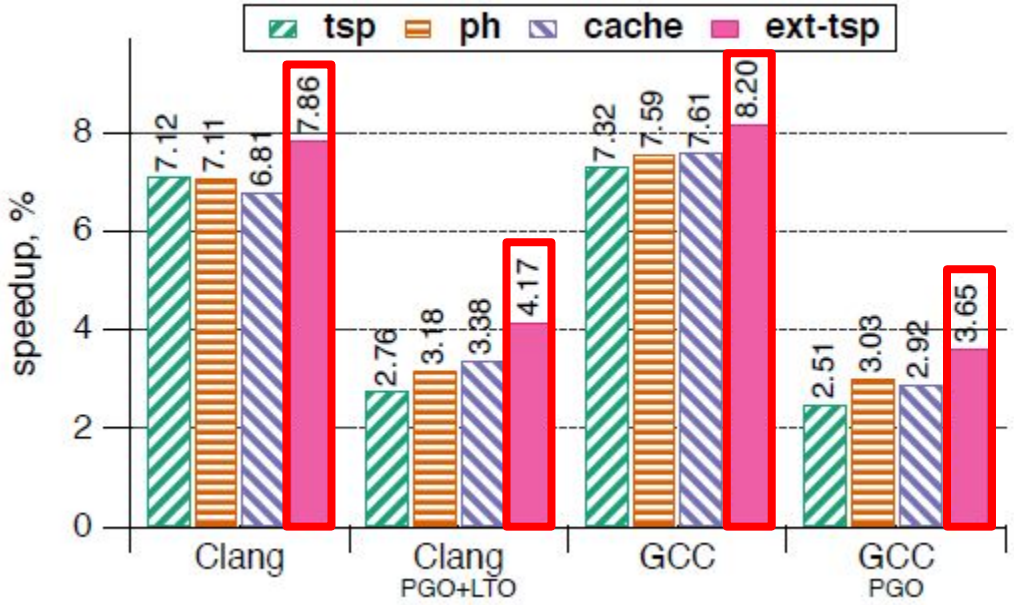
Evaluation

perf on HipHop Virtual Machine(HHVM) - reduction rate of misses



Evaluation

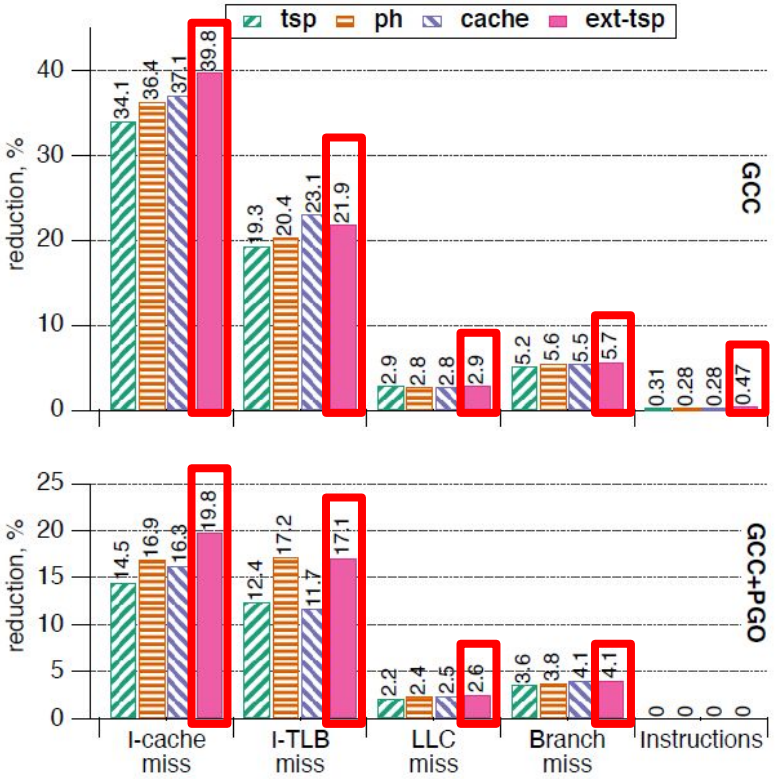
Performance Improvement - Clang and GCC



Evaluation

Perf - reduction of misses

(with/without PGO)



Limitations & Potential Future Directions

- **Limitations**

  1. Many *complementary optimizations are not investigated* in detail.

  2. The measurable gains from the heuristics algorithm is only produced *within the scope* of each individual function.

- **Potential Future Directions**

  1. *Integrate complementary optimizations* into the algorithm, such as checking unrolling loops and/or duplicating blocks to avoid extra jumps.

  2. After doing further research, the refinement of the heuristics algorithm can probably take advantage of the *cross-procedure reordering*[3].

References

[1] A. Newell and S. Pupyrev. “Improved Basic Block Reordering.” IEEE Transactions on Computers, vol. 69, no. 12, 2020, pp. 1784–1794.

[2] K. Pettis and R. C. Hansen, “Profile guided code positioning,” in SIGPLAN Notices, vol. 25, no. 6. ACM, 1990, pp. 16–27.

[3] J. Torrellas, C. Xia, and R. L. Daigle, “Optimizing the instruction cache performance of the operating system,” IEEE Transactions on Computers, vol. 47, no. 12, pp. 1363–1381, 1998.

Q&A

Improved Basic Block Reordering[1]

Andy Newell and Sergey Pupyrev

Group 20: Zijian Zhang, Huiruo Zou, Yunhao Wang, Zhaoyuan Zhang