UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

**Asier Fernández De Lecea
and Javier Salamero**

Improving Instruction Caches

Background

- Memory accesses are a performance bottleneck in most workloads
  An access to main memory could require hundreds of cycles
- Traditionally mitigated through memory hierarchy organization
  TLBs, L1, L2, data-instruction splitted caches, etc.
- Lower levels in the hierarchy must be small and fast to match processor frequency
  However, small and simple designs implies a generally high miss rate
- Increasing cache size and/or design complexity comes at a trade-off
  Higher block capacity and/or more complex designs raises HW costs and access times
- Instructions misses present up to 20% performance loss in certain applications
  Most of the work has been focused on dealing with data accesses

Context

At the time of paper publications, late 80s start of 90s, memory hierarchy was a hot topic of discussion:

- Whether split caches in instruction and data (reduced sizes, greater access speeds)
- The use of set-associativity to avoid some conflicts
- Off-chip or on-chip caches, etc.

An orthogonal solution was proposed by compiler people:

- **Compiler guys:** “Hey, hardware guys! If you give us direct-mapped caches, we can apply some cool techniques so that programs run much faster :)”
- **Hardware guys:** “Okay 👌 “

Compiler-based solution approach

Objective: Keep a small, direct-mapped instruction cache coupled with compiler-based optimizations in order to maintain low miss rates.

DM caches imply:

- Simple hardware design
  - Cheap and fast
- Straightforward control and placement logic
  - The memory blocks are placed in cache only depending on their address
- Most misses come from competing instructions for the same sets in cache
  - This is referred-to as aliasing
- Misses related to first appearance of the block or the lack of capacity in the cache are not studied in this work
  - Prefetching could help in this area

Optimization techniques

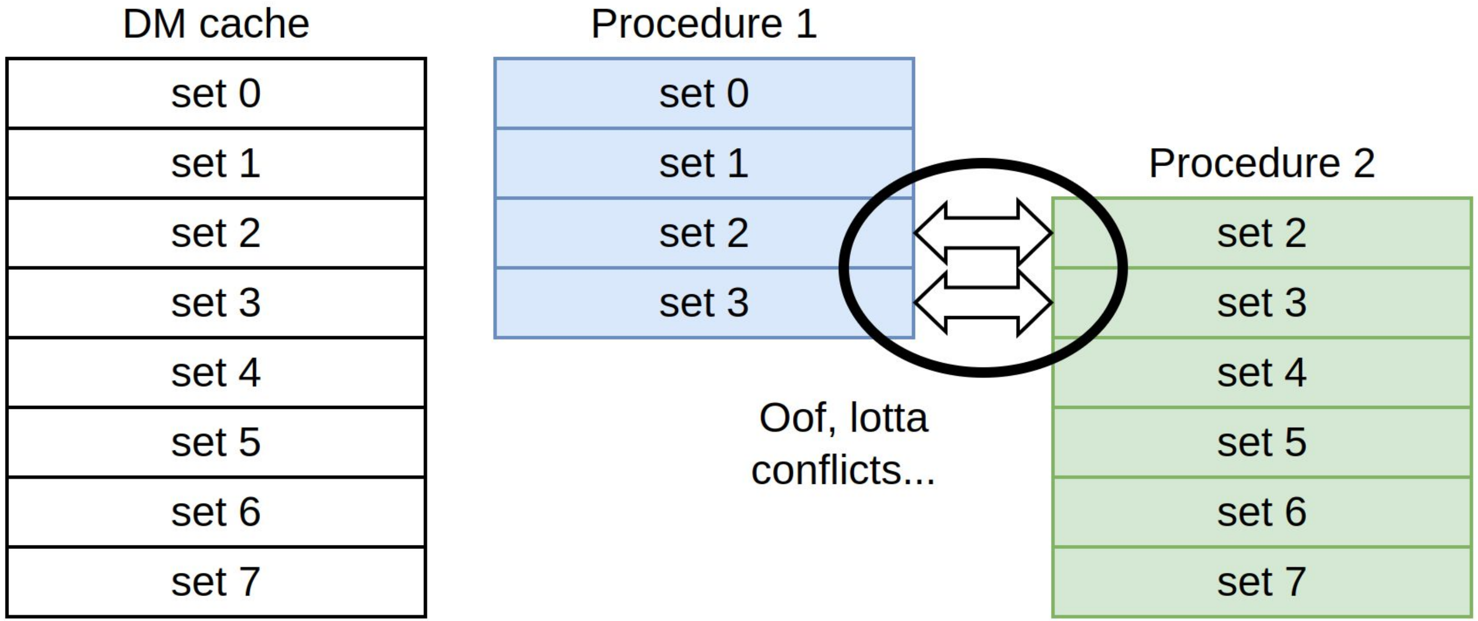
General goal: Reduce **aliasing conflicts** in the instruction cache by **carefully mapping instructions in memory** using the compiler

- Requires code profiling
- Frequently executed code is placed/repositioned close in memory
- Reduces cache misses
- Reduces page and TLB misses

The presented compiler-related techniques motivating the use of direct-mapped caches are:

- *Repositioning of instructions to avoid cache interferences*
- *Procedure merging, callee-with-caller merging*
- *Procedure Positioning*
- *Basic Block Positioning*
- *Procedure Splitting*

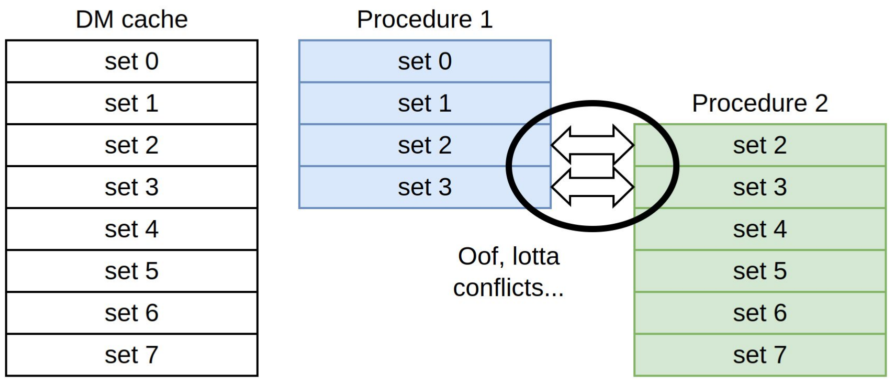
Repositioning of instructions to avoid cache interferences



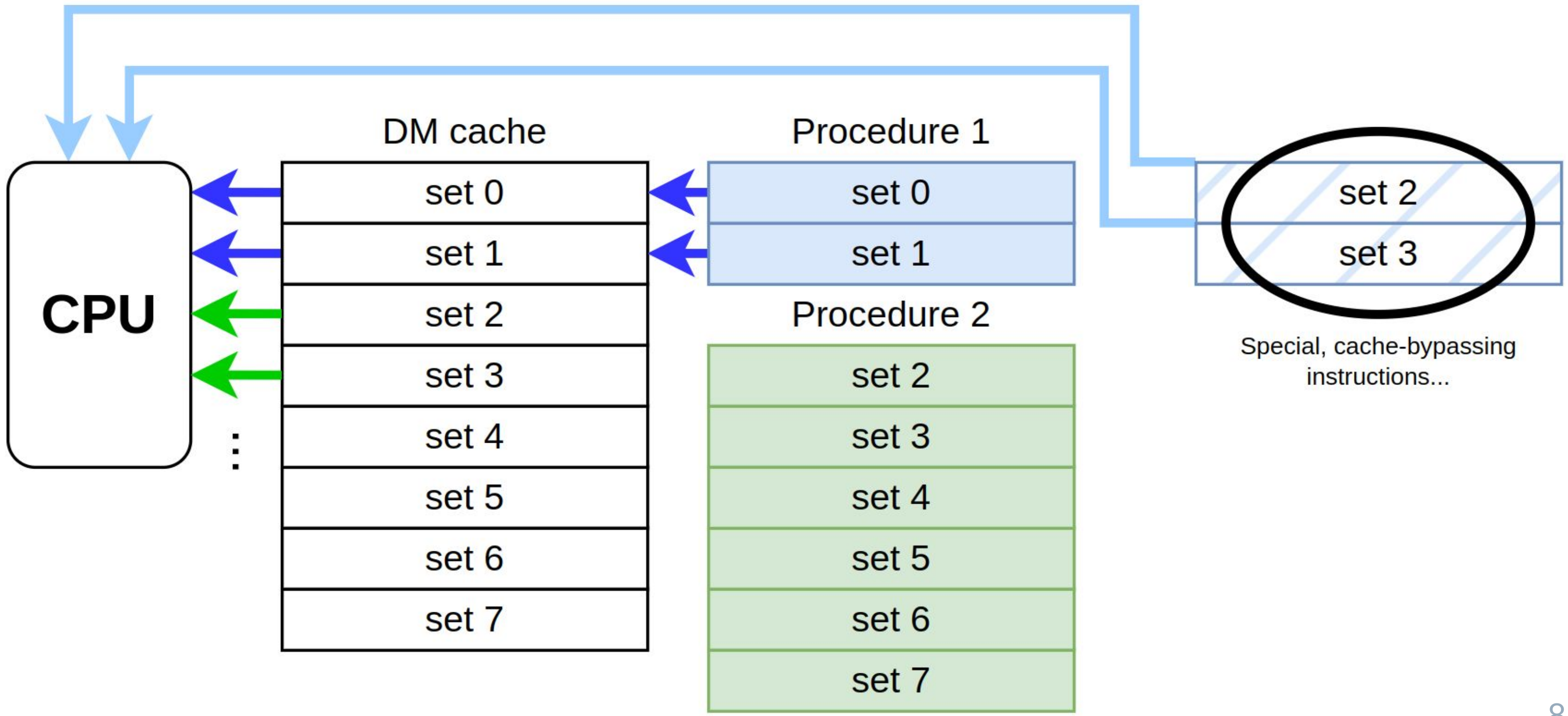
Repositioning of instructions to avoid cache interferences

This first technique has two ways of dealing with procedure interference:

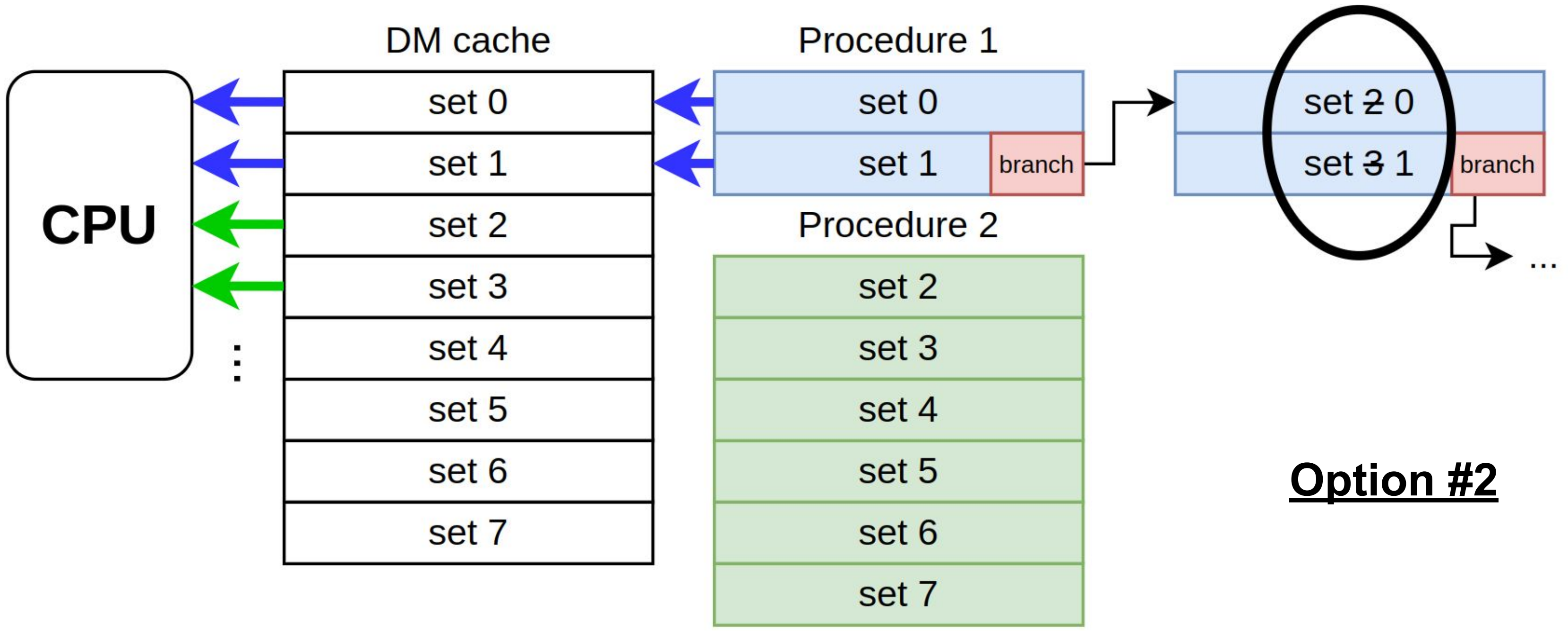
- **Option 1**: mark conflictive sets of a procedure as “bypass instructions”, not needing to enter into cache altogether
- **Option 2** change the address of conflictive sets of a procedure so that a procedure always occupies the same sets and doesn’t interfere with other procedures



Repositioning of instructions to avoid cache interferences



Repositioning of instructions to avoid cache interferences

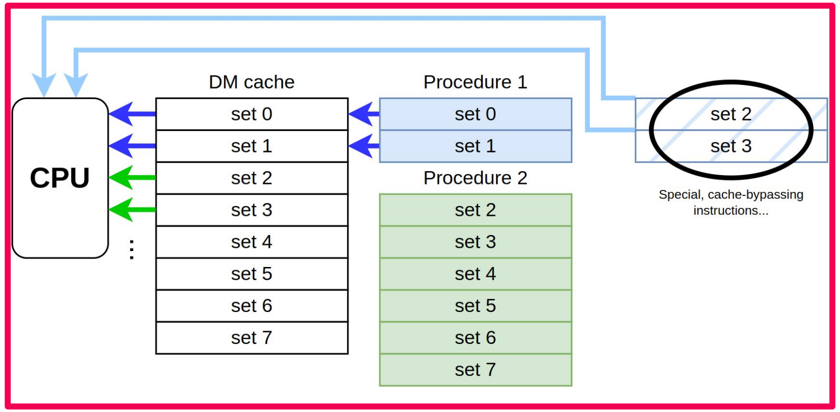


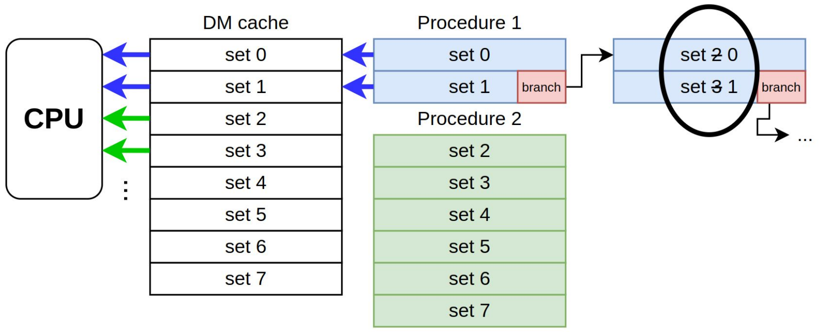
Option #2

Repositioning of instructions to avoid cache interferences

Option 1 requires that the icache be cable of dynamic shut-offs, i.e. hardware extensions to identify which cache lines are not to be included, thus allowing cache lines to go directly into the processor

- Specific hardware requirements and extensions needed
- Icache needs to be “smart” enough shut off, i.e. detect blocks that are not to be added
- This results in added hardware complexity for the cache module, when the objective was to keep it as simple and fast as possible…

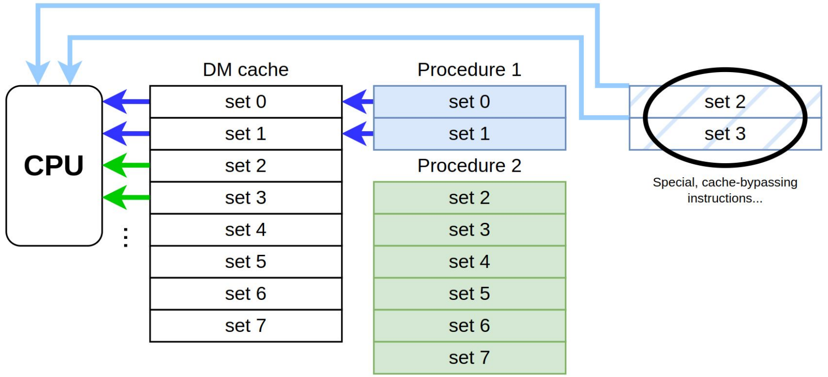


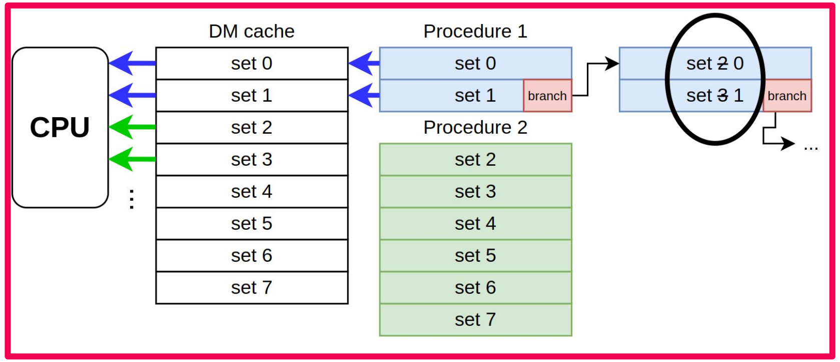


Repositioning of instructions to avoid cache interferences

Option 2 assumes a “normal” icache, so the address of conflicting sets must be modified…

- Certain addresses of a procedure will require to be changed, dividing the procedure into different sections across memory
- In order to connect different sections of the splitted procedure, branches/jumps should be added…
- …which results in an increase of code size of the repositioned procedures with “empty calories” instructions…

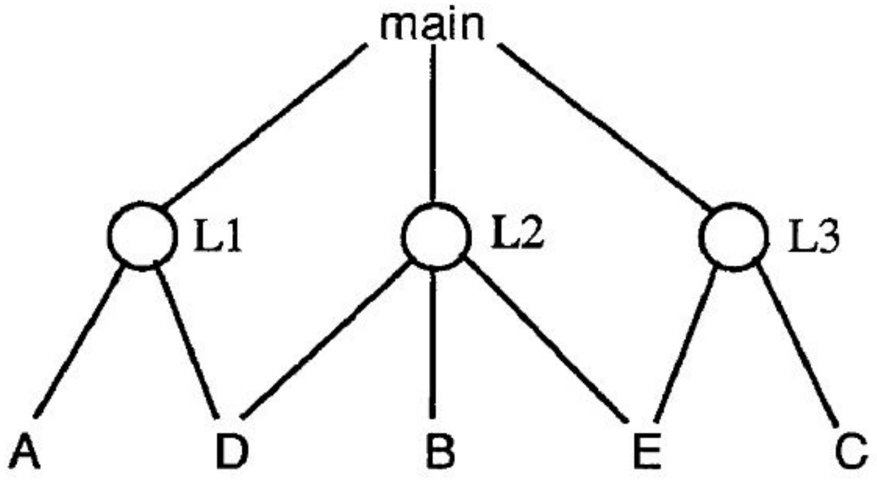




Repositioning of instructions to avoid cache interferences

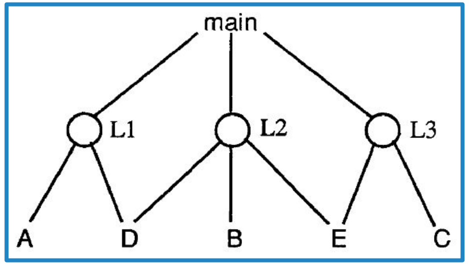
Let’s see an example of how this technique functions:

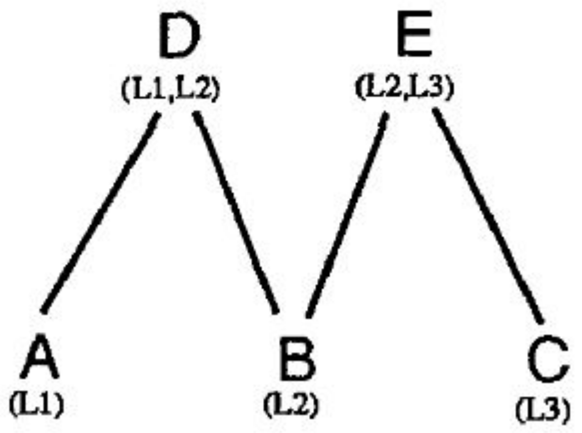
- Figure on the right represents an example “main” program
- “Main” has three loops, L1 L2 and L3
- Loop L1 calls procedures A and D
- Loop L2 calls procedures D B and E
- Loop L3 calls procedures E and C
- The question now is, **how do we position these procedures in our DM cache so that they do not interfere with each other?**



Repositioning of instructions to avoid cache interferences

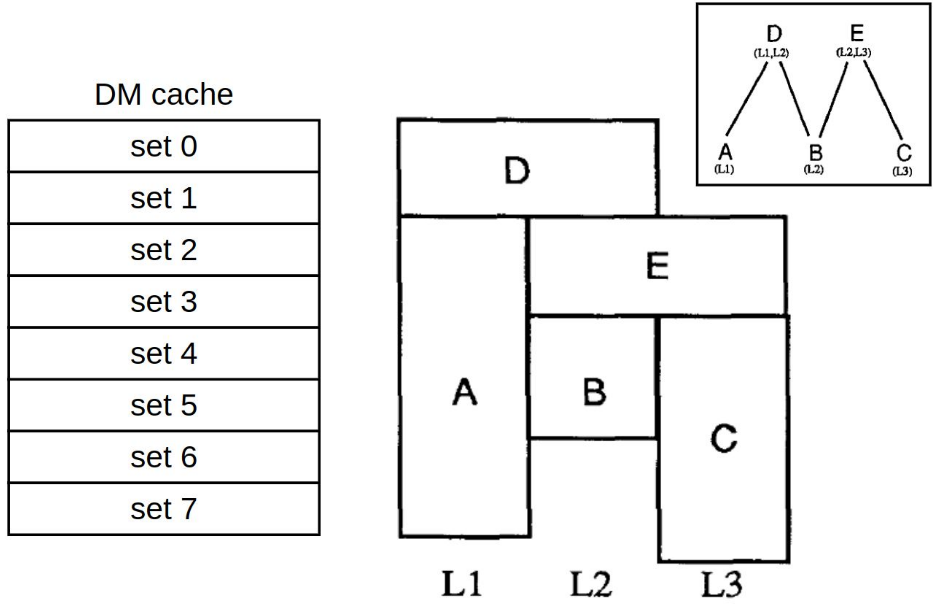
- …
- Dependencies between procedures must first be established, i.e. which procedures are called by which loop
- A is called in L1, and so is D, so they are related to each other
- D E and B are related due to loop L2
- E and C are related due to loop L3
- Fitting all these procedures in icache, one loop at a time, becomes a bin-packing problem solved by a **greedy algorithm**…





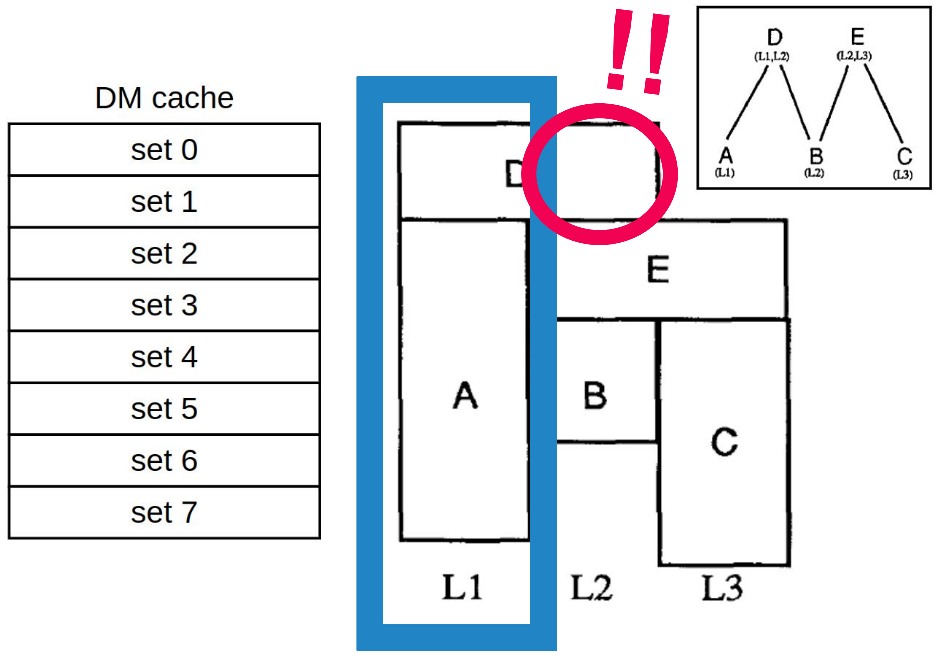
Repositioning of instructions to avoid cache interferences

- ...
- From the previous procedure-relation graph, a procedure placement configuration emerges!

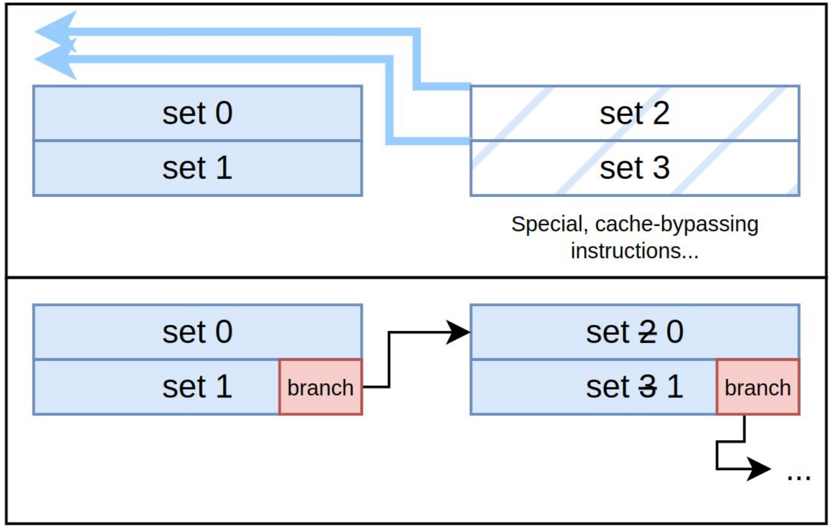


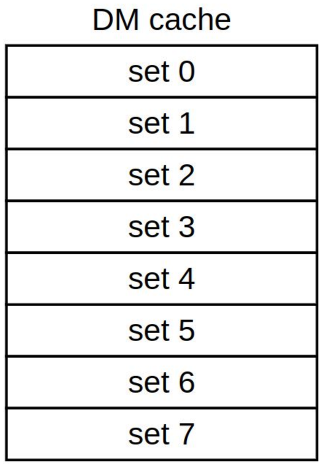
Repositioning of instructions to avoid cache interferences

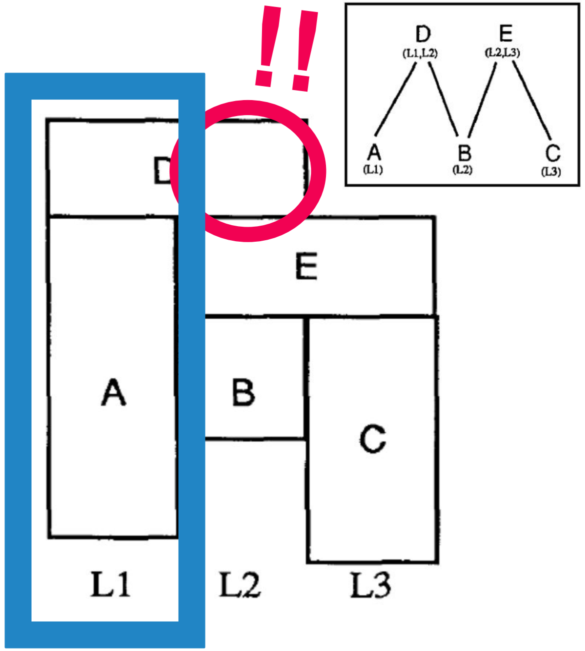
- ...
- From the previous procedure-relation graph, a procedure placement configuration emerges!
- During the execution of loop L1, procedure A is present at all times in cache, while certain sections of procedure D are marked as “special”...



Repositioning of instructions to avoid cache interferences

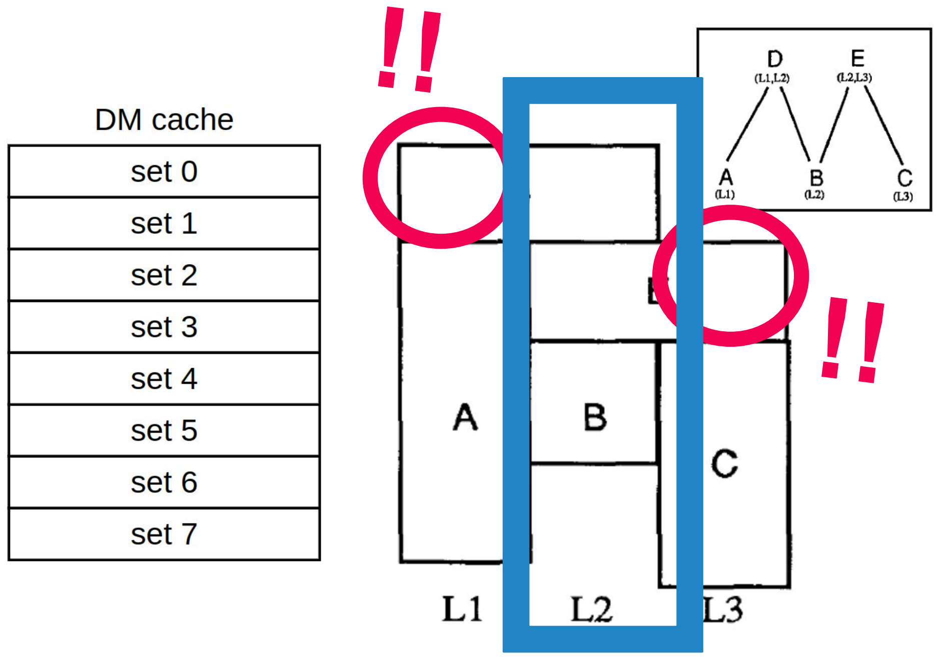






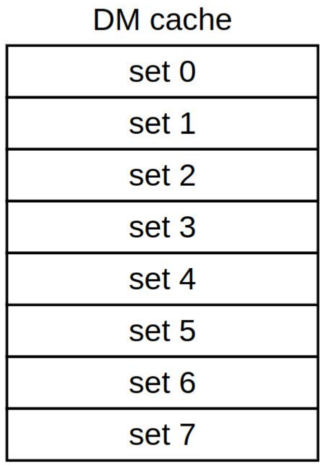
Repositioning of instructions to avoid cache interferences

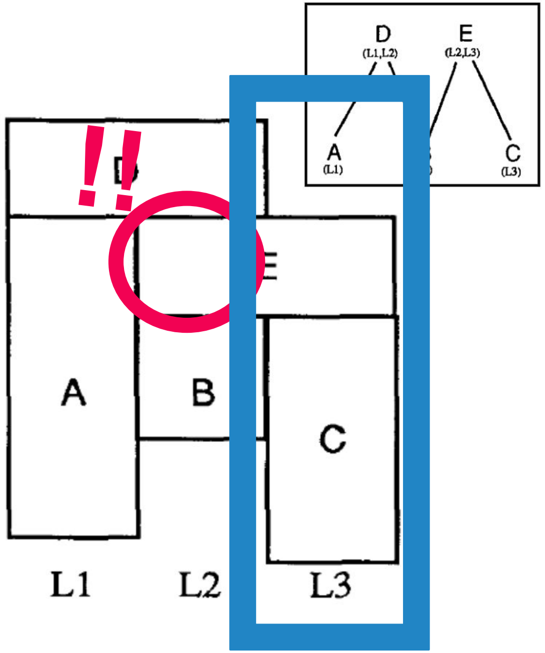
- ...
- From the previous procedure-relation graph, a procedure placement configuration emerges!
- During the execution of loop L1, procedure A is present at all times in cache, while certain sections of procedure D are marked as “special”...
- During the execution of loop L2, procedure B is present at all times in cache, procedures D and E are “special”...



Repositioning of instructions to avoid cache interferences

- ...
- From the previous procedure-relation graph, a procedure placement configuration emerges!
- During the execution of loop L1, procedure A is present at all times in cache, while certain sections of procedure D are marked as "special"...
- During the execution of loop L2, procedure B is present at all times in cache, procedures D and E are "special"...
- During the execution of loop L3, procedure C is present at all times in cache, procedure E is "special"...

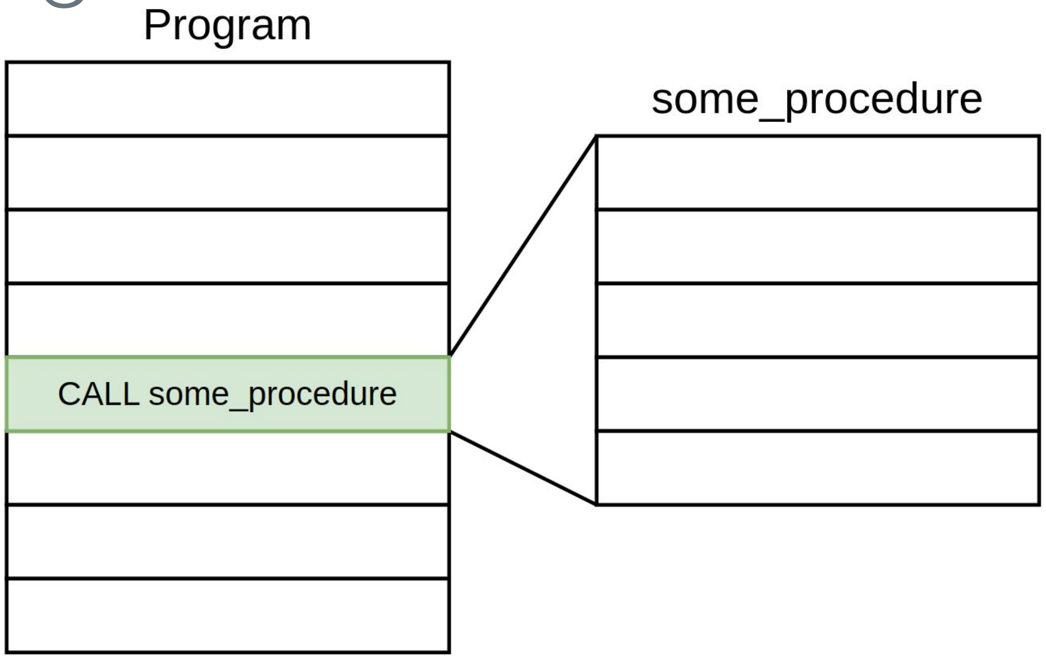




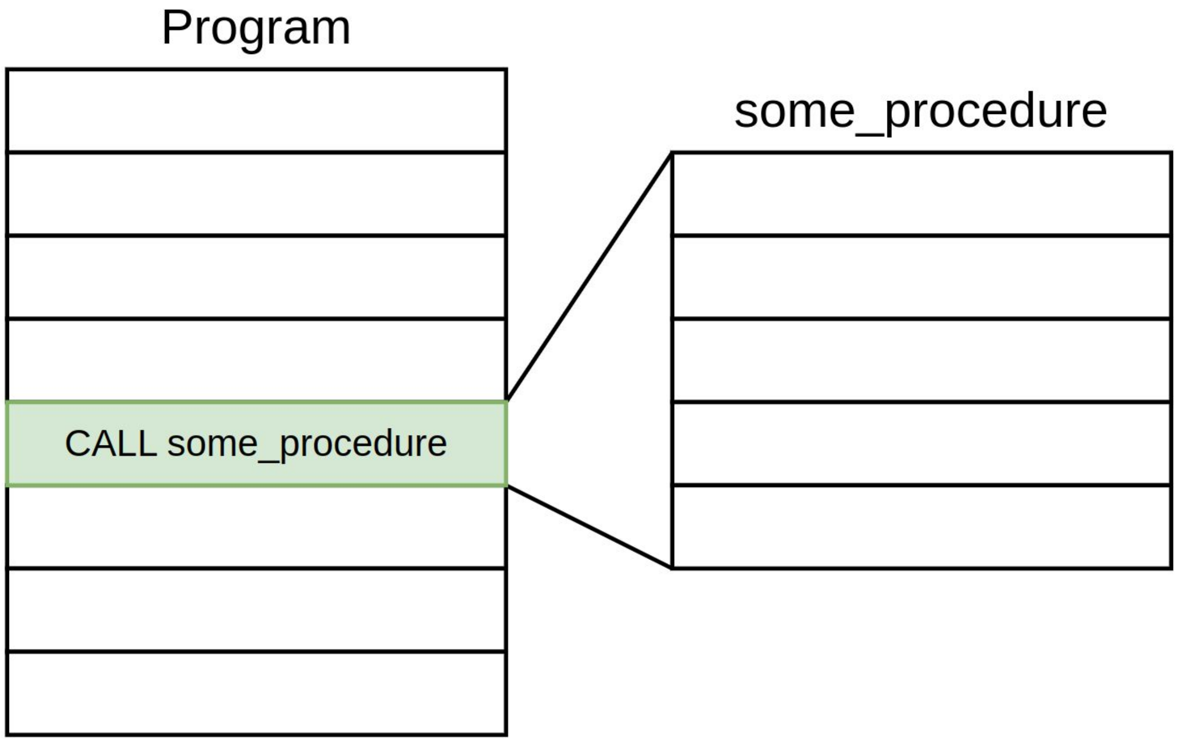
Procedure Merging

Procedure merging (inlining) refers to in merging the calls made to external programs or procedures with the instructions of the source/caller program.

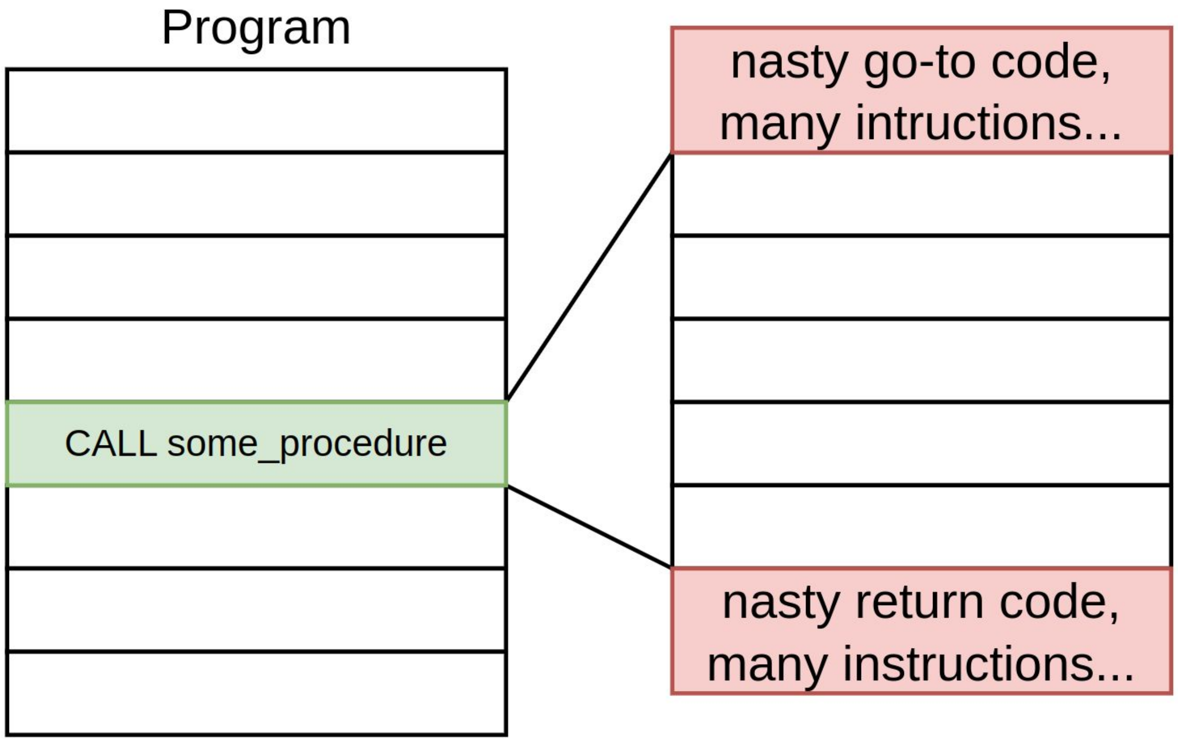
The decision of which call instances to merge with the caller program to improve performance is a complicated one...



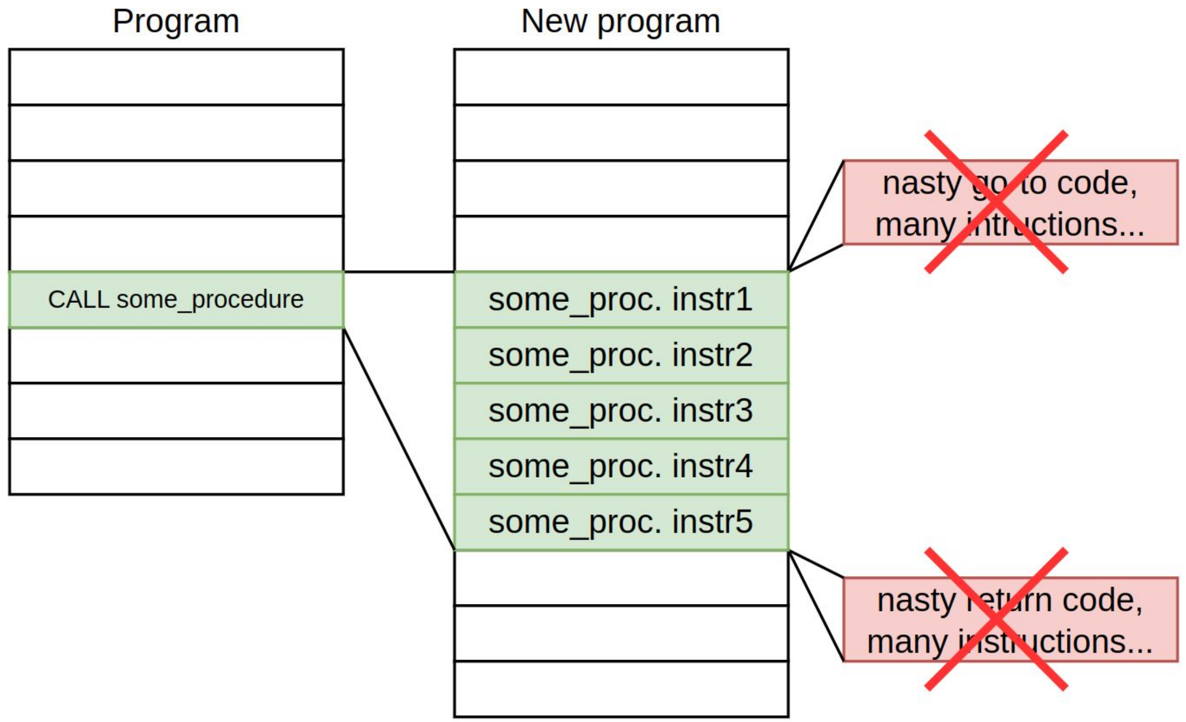
Procedure Merging



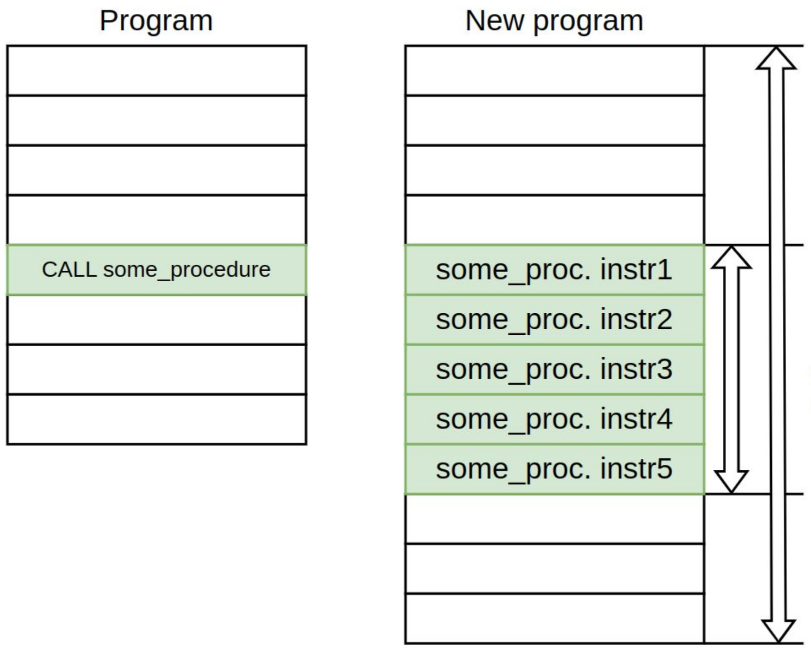
Procedure Merging



Procedure Merging

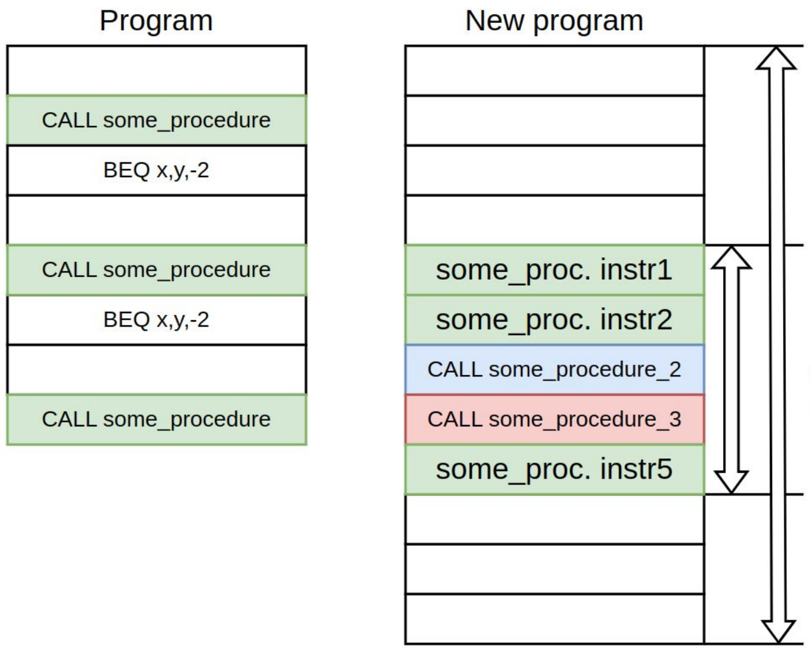


Procedure Merging



New total program size, different miss rate characteristics, maybe even more nested calls in inlined procedure...

Procedure Merging



New total program size, different miss rate characteristics, maybe even more nested calls in inlined procedure...

Procedure Merging

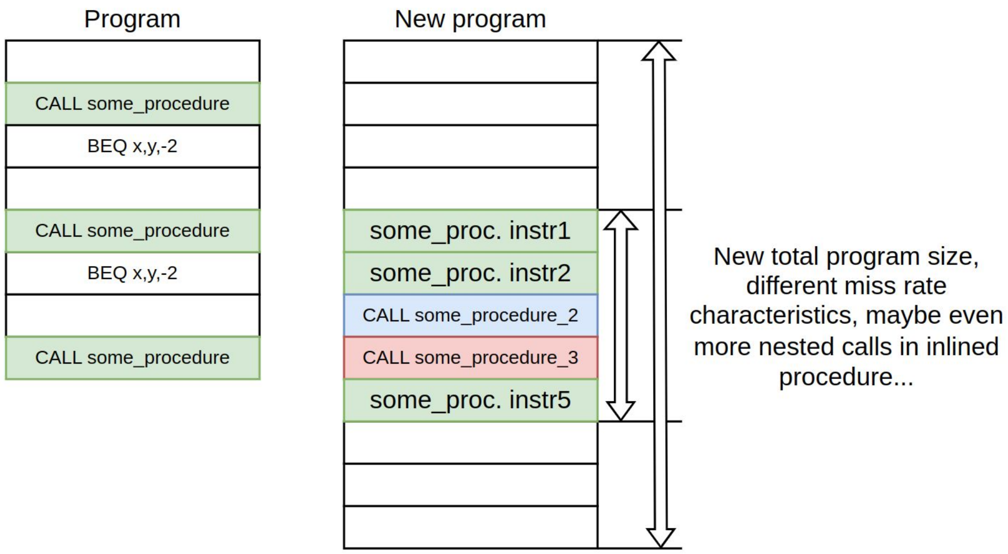
Must consider call-frequency of procedure within loop, procedure may not be executed in all iterations...

Must consider if loop will fit in cache after merging...

What if the same procedure is called from multiple sites?

What if there are other calls to procedures within the just-merged procedure?

Deciding to merge a certain procedure affects decisions regarding other procedures after the fact...

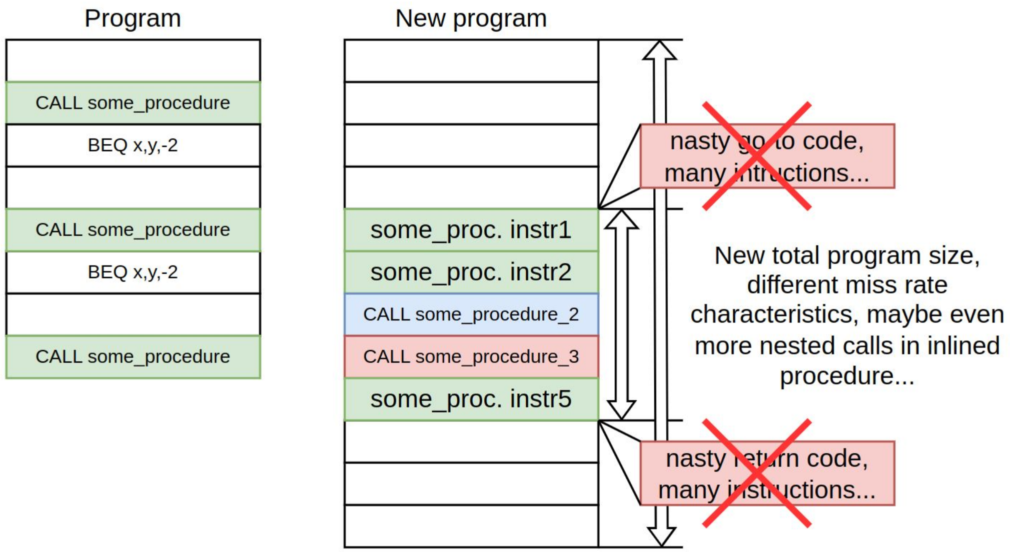


Procedure Merging

Solution: greedy algorithm, again!

Calls to external procedures are considered in decreasing order of a ratio = (call frequency) / (average size of the called procedure)

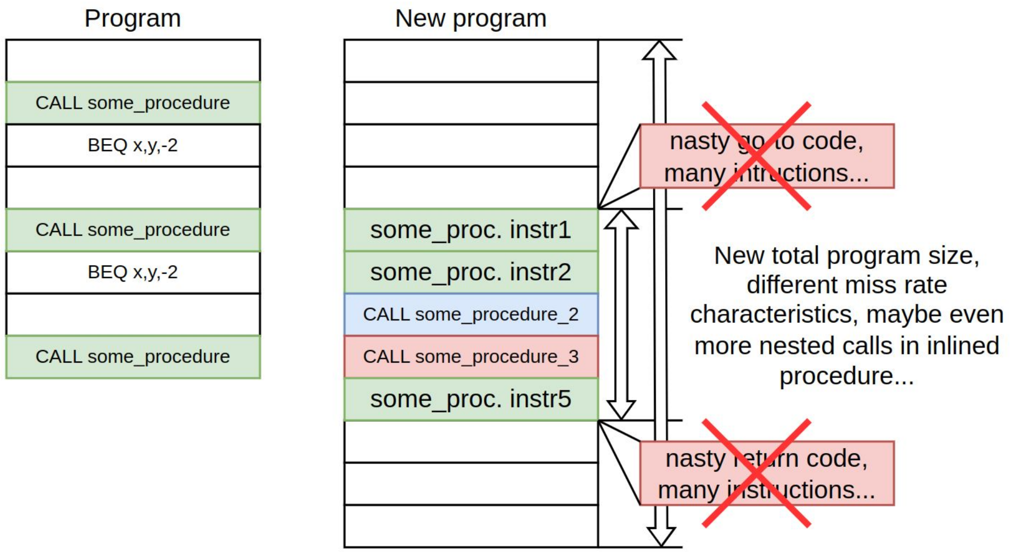
Merging-decision algorithm decides whether to merge/not a procedure based on (1) estimates on the average size of a loop's iterations and (2) the miss-rate behaviour of a modeled cache (optimal...)


Program
CALL some_procedure
BEQ x,y,-2
CALL some_procedure
BEQ x,y,-2
CALL some_procedure
New program
nasty go to code, many intructions...
some_proc. instr1
some_proc. instr2
CALL some_procedure_2
CALL some_procedure_3
some_proc. instr5
New total program size, different miss rate characteristics, maybe even more nested calls in inlined procedure...
nasty return code, many instructions...

Procedure Merging

If, after merging, the average size of the new loop's iterations results in a **BETTER** *miss-rate behaviour* **THAN** *the cost of having the explicit call to the procedure*, then a procedure is merged

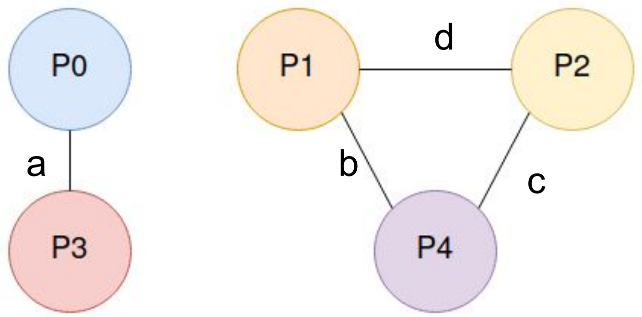
If, after merging, the average size of the new loop's iterations results in a **WORSE** *miss-rate behaviour* **THAN** *the cost of having the explicit call to the procedure*, then a procedure is not merged

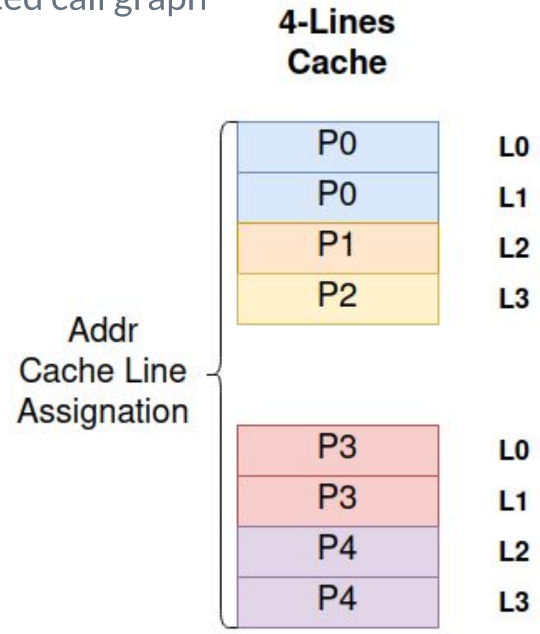

Program
CALL some_procedure
BEQ x,y,-2
CALL some_procedure
BEQ x,y,-2
CALL some_procedure
New program
some_proc. instr1
some_proc. instr2
CALL some_procedure_2
CALL some_procedure_3
some_proc. instr5
nasty go to code, many intructions...
New total program size, different miss rate characteristics, maybe even more nested calls in inlined procedure...
nasty return code, many instructions...

Procedure Positioning

Procedures frequently executed together, as close as possible in memory:

Example: We execute the code with the profiler and get this weighted call graph

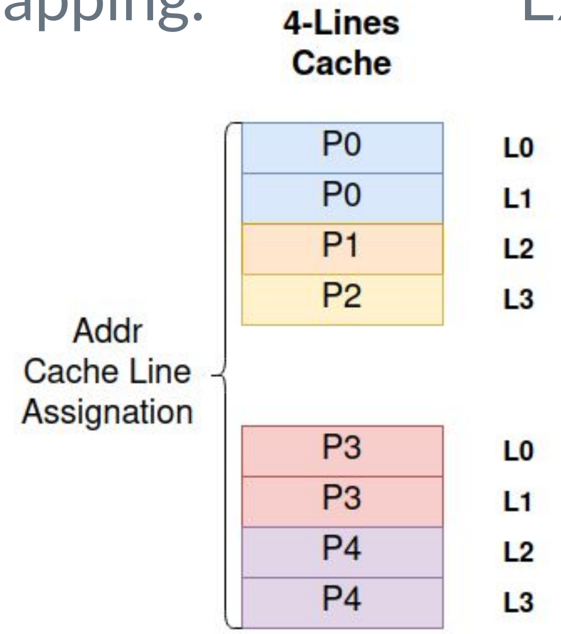




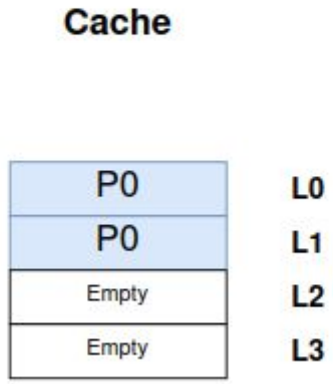
Procedure Positioning

Execution case

Mapping:



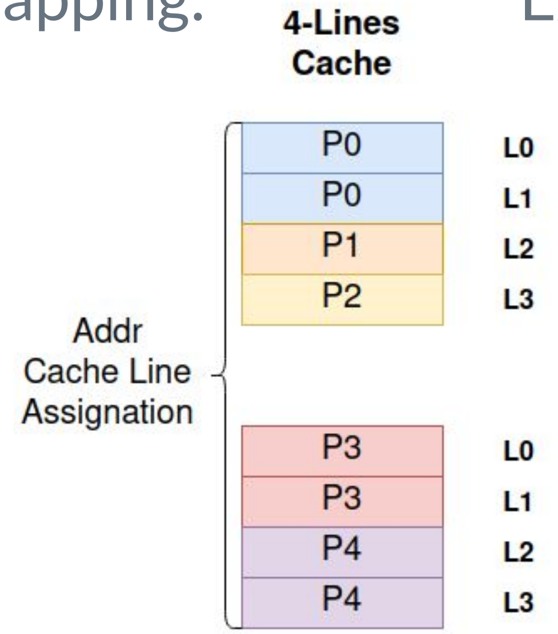
Execution:



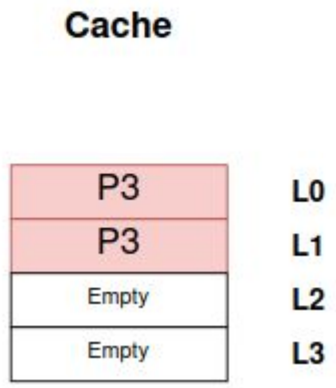
Procedure Positioning

Execution case

Mapping:

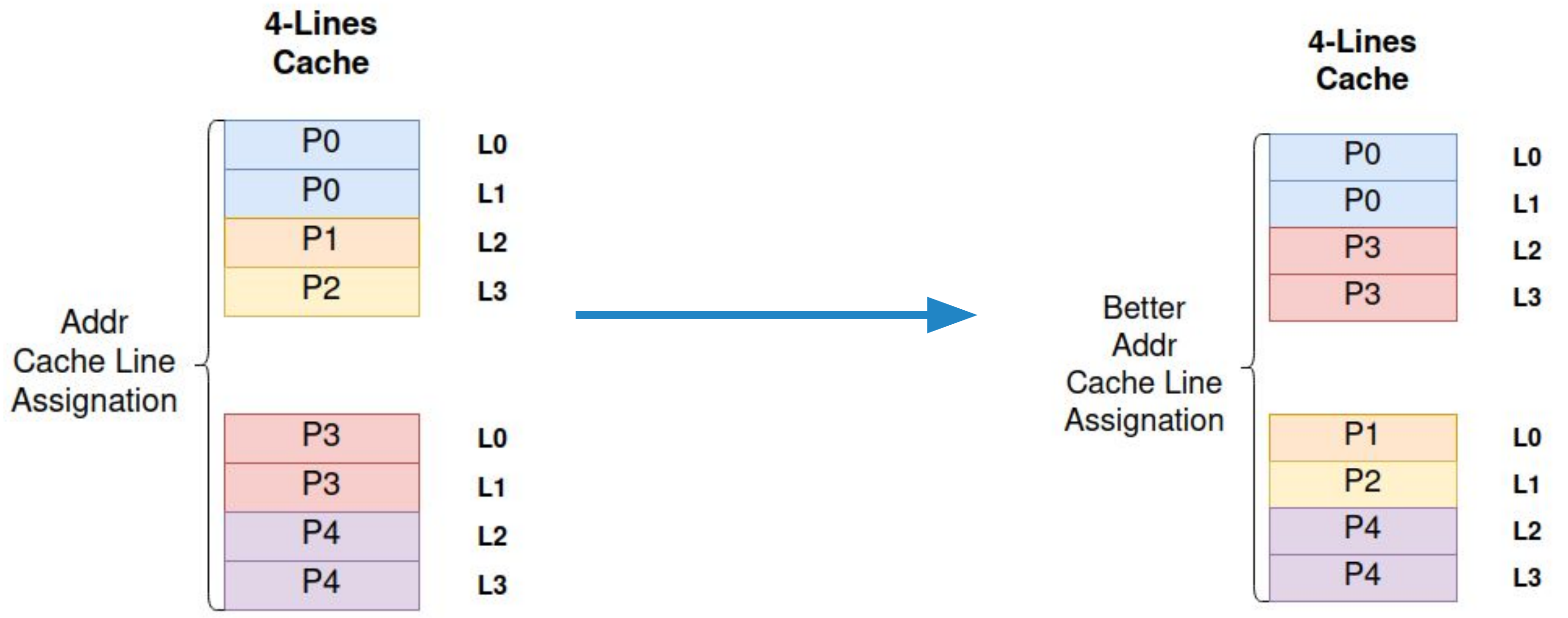


Execution:



Procedure Positioning

A better positioning is possible:



Procedure Positioning

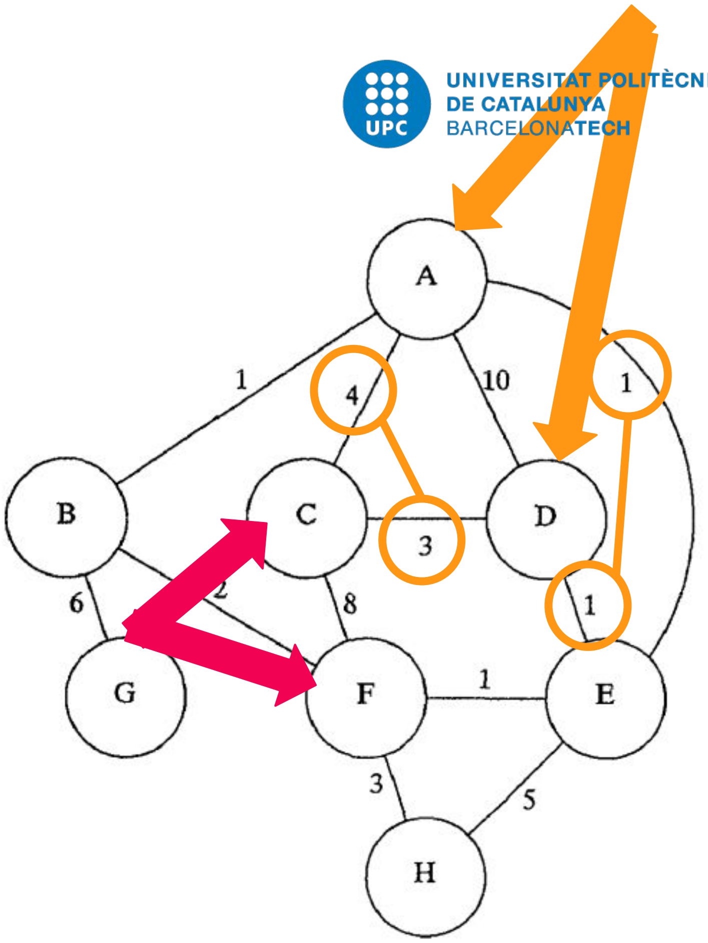
How to reorder procedures:

1. Merge nodes with highest arcs
2. Recompute arcs of affected nodes

Procedure Positioning

How to reorder procedures:

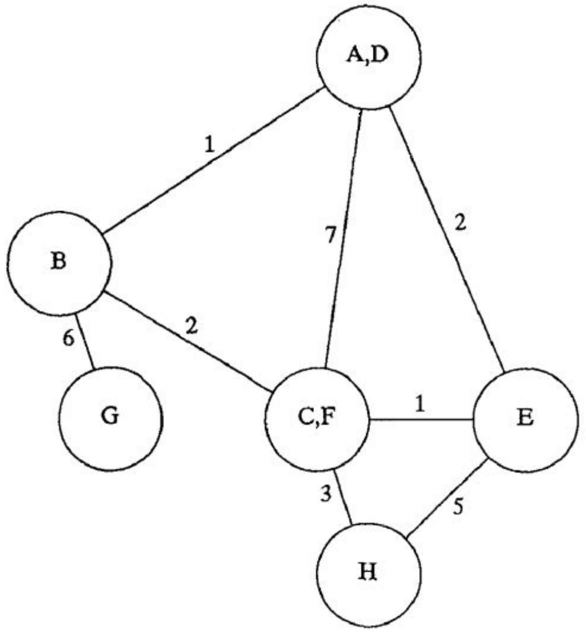
1. Merge nodes with highest arcs
2. Recompute arcs of affected nodes



Procedure Positioning

How to reorder procedures:

1. Merge nodes with highest arcs
2. Recompute arcs of affected nodes



Procedure Positioning

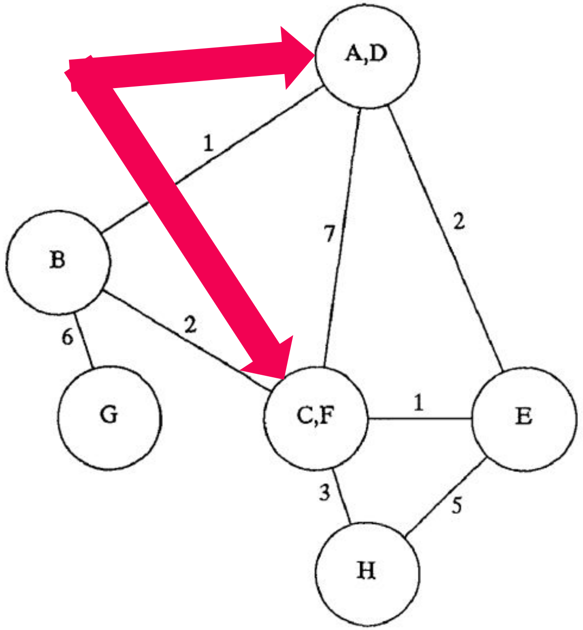
How to reorder procedures:

1. Merge nodes with highest arcs
2. Recompute arcs of affected nodes
3. Solve ambiguity (stronger connection)

- A-D-C-F or F-C-D-A,
- A-D-F-C or C-F-D-A,
- D-A-C-F or F-C-A-D,
- D-A-F-C or C-F-A-D

F not connected to A nor D!!

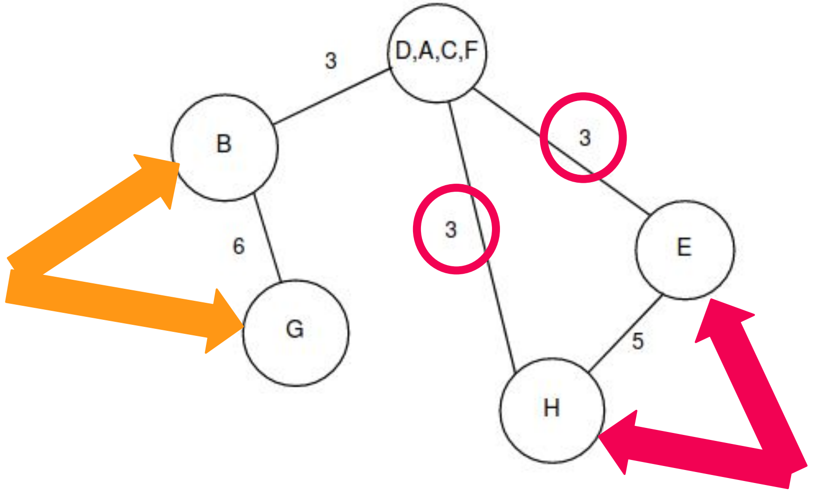
- A-D-C-F or F-C-D-A,
- D-A-C-F or F-C-A-D



Procedure Positioning

How to reorder procedures:

1. Merge nodes with highest arcs
2. Recompute arcs of affected nodes
3. Solve ambiguity (stronger connection)



Procedure Positioning

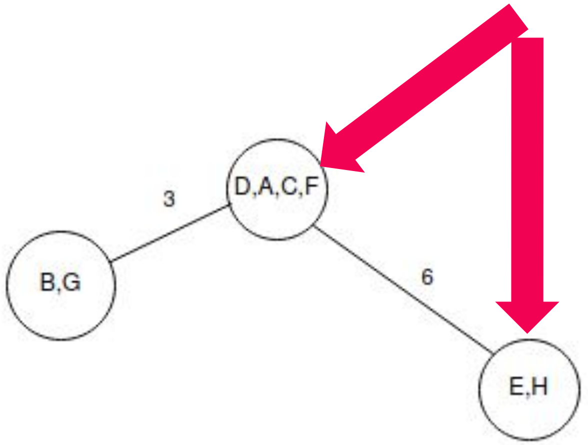
How to reorder procedures:

1. Merge nodes with highest arcs
2. Recompute arcs of affected nodes
3. Solve ambiguity (stronger connection)

- D-A-C-F-E-H, H-E-F-C-A-D,
- D-A-C-F-H-E, E-H-F-C-A-D,
- F-C-A-D-E-H, H-E-D-A-C-F,
- F-C-A-D-H-E, E-H-D-A-C-F

In the original graph, H to F connection is the stronger than any other combination

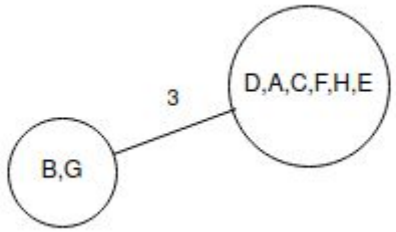
- D-A-C-F-H-E, E-H-F-C-A-D



Procedure Positioning

How to reorder procedures:

1. Merge nodes with highest arcs
2. Recompute arcs of affected nodes
3. Solve ambiguity (stronger connection)



Procedure Positioning

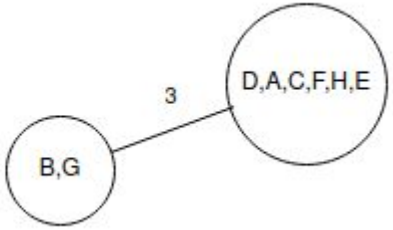
How to reorder procedures:

1. Merge nodes with highest arcs
2. Recompute arcs of affected nodes
3. Solve ambiguity (stronger connection)

- B-G-D-A-C-F-H-E, E-H-F-C-A-D-G-B,
- G-B-D-A-C-F-H-E, E-H-F-C-A-D-B-G,
- D-A-C-F-H-E-B-G, G-B-E-H-F-C-A-D
- D-A-C-F-H-E-G-B, B-G-E-H-F-C-A-D

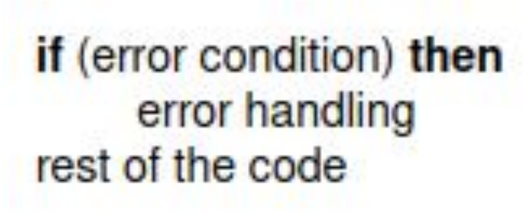
G only connected to B originally and B:

- G-B-D-A-C-F-H-E, E-H-F-C-A-D-B-G,
- D-A-C-F-H-E-B-G, G-B-E-H-F-C-A-D



Basic Block Positioning

Same idea as Procedure Positioning but at basic block granularity, BBs frequently executed, close in memory



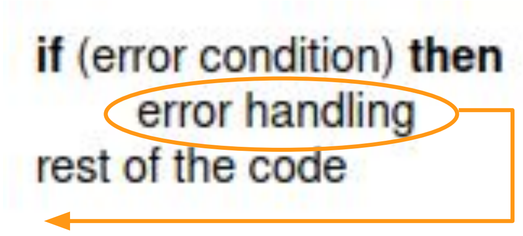
```
if (error condition) then
        error handling
rest of the code
```

Some considerations:

- ▷ Heavy amount of code with error treatment code (rarely or never executed), contiguous to executed code (bad mapping)
- ▷ Usually taken forward branches (bad static prediction!)

Basic Block Positioning

Same idea as Procedure Positioning but at basic block granularity, BBs frequently executed, close in memory

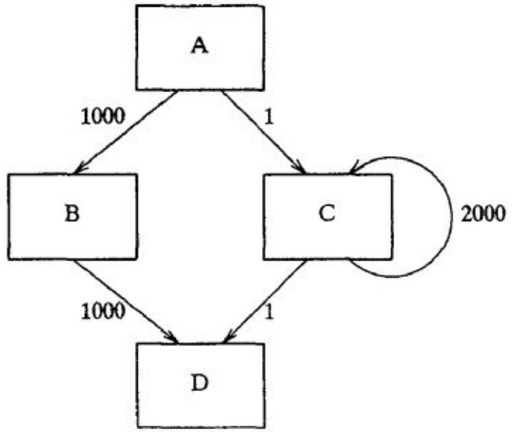


Rarely executed blocks are positioned after usually taken paths, resulting in:

- Better utilization of DM caches
- Forward branches are rarely taken, improvement in branch prediction

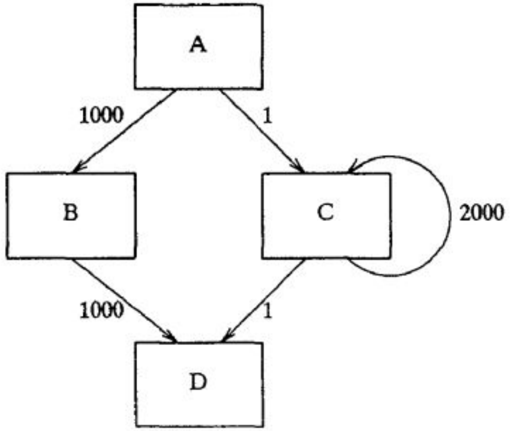
Basic Block Positioning

1. First step is to obtain a weighted directed graph of the BBs execution in a procedure:



Basic Block Positioning

1. First step is to obtain a weighted directed graph of the BBs execution in a procedure
2. To order the BBs, we have to form chains:
    a. A chain has a head and a tail BB
    b. Initially, each BBs form a chain with itself as head and tail of the chain
    c. Two chains can be merged if the arc connects the tail of one chain with the head of another
    d. Chains with the highest connecting are merged first
    e. When no more chains can be merged, order them



Basic Block Positioning

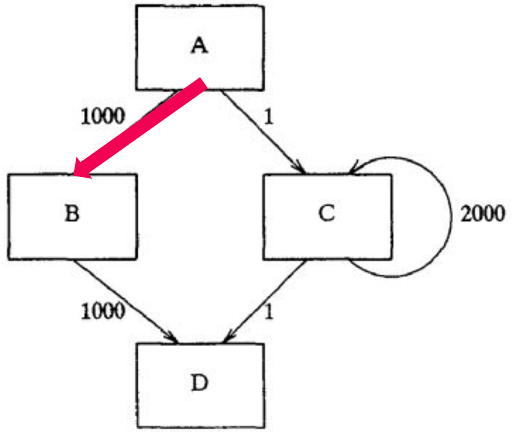
1. First step is to obtain a weighted directed graph of the BBs execution in a procedure
2. To order the BBs, we have to form chains:

Chains:

A -> B

D

C



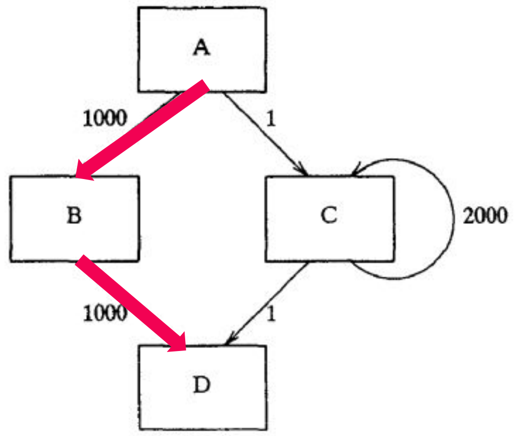
Basic Block Positioning

1. First step is to obtain a weighted directed graph of the BBs execution in a procedure
2. To order the BBs, we have to form chains:

Final Chains:

A -> B -> D

C



Basic Block Positioning

1. First step is to obtain a weighted directed graph of the BBs execution in a procedure
2. To order the BBs, we have to form chains:

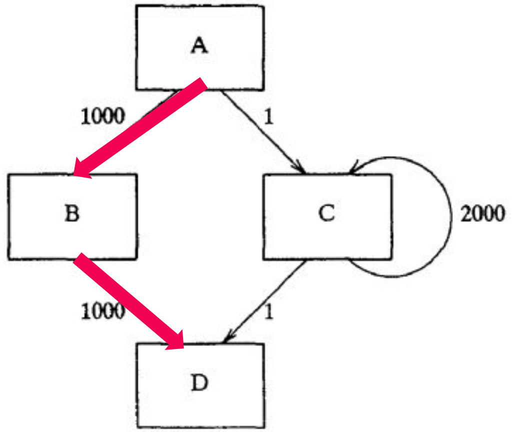
Final Chains:

A -> B -> D

C

Chain order:

1. A -> B -> D
2. C



Rarely taken path as forward branch

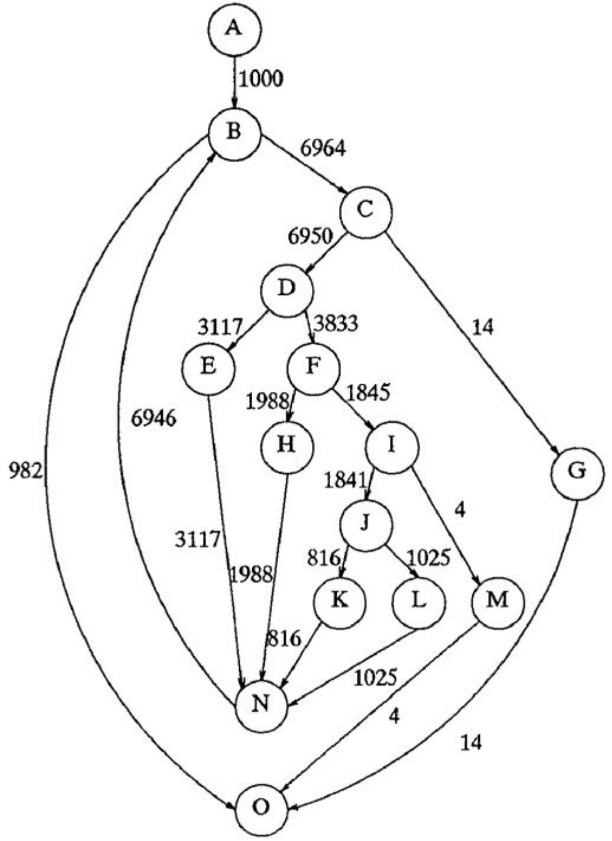
Basic Block Positioning

Sometimes, the ordering of chains is more complex…

Multiple final chains:

- A
- E-N-B-C-D-F-H
- I-J-L
- G-O
- K
- M

Sometimes, a perfect solution is not possible



Procedure Splitting

Combines Procedure and Basic Block Positioning techniques

- Procedures close in memory
- BBs close in memory

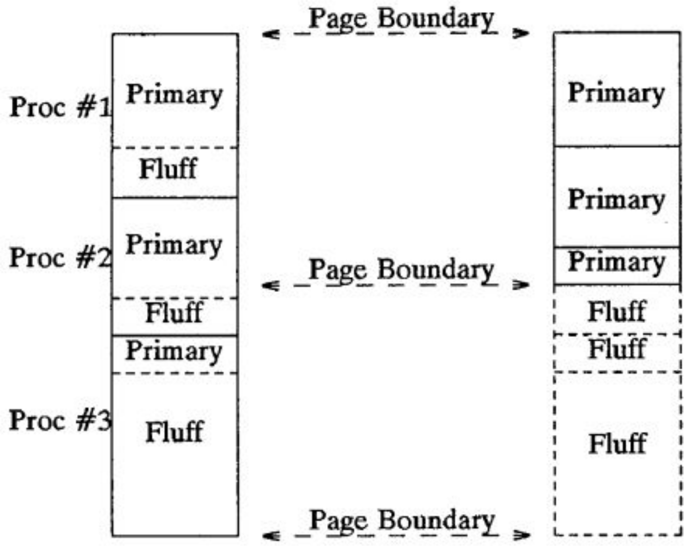
With the profiling input, error treatment BBs are commonly not even executed once (fluff BBs)

Idea: Separate fluff of procedures from useful blocks (primary BBs) and place primaries of procedures close in memory, improving cache, page and TLB efficiency.

Procedure Splitting

Combines Procedure and Basic Block Positioning techniques

Idea: Separate fluff of procedures from useful blocks (primary) and place primaries of procedures close in memory, improving cache, page and TLB efficiency.



Conclusions

- ▷ Feasibility of introducing the compilation-process modifications?

No matter how complicated a technique might seem, no mention was ever made as to the increase in time/resource utilization of the resulting compiler after integrating the proposed technique.

- ▷ Experimental setup a bit lacking in some cases

For the first two techniques, it is not mentioned what compiler framework or what architecturethe benchmarks are performed on, whereas the last two papers are much more nuanced in its presentation and detailing of their experimental setup.

- ▷ How to obtain the necessary profile information for the techniques?

Credits

- McFarling, S. - Program Optimization for Instruction Caches - Proceedings of ASPLOS III, 1989
- McFarling, S. - Procedure Merging with Instruction Caches - PLDI 1991, 71-79
- Pettis, K; C. Hansen, Robert - Profile Guided Code Positioning - SIGPLAN Not. 25, 6 (Jun. 1990), 16-27
- Hisu, WW; PPChang, PP - Achieving High Instruction Cache Performance With An Optimizing Compiler - The 16th Annual International Symposium on Computer Architecture, June 1989

Thank you for your attention,
any questions?