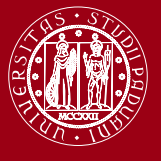
# A Rapid Cache-aware Procedure Positioning Optimization to Favor Incremental Development

**Enrico Mezzetti**, **Tullio Vardanega**

RTAS 2013
19th IEEE Real-Time and Embedded Technology and Applications Symposium
Philadelphia, USA
April 9 - 11, 2013


Università degli Studi di Padova

# Outline





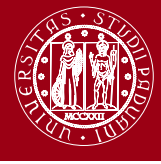
1 The case for incremental development

2 Incremental procedure positioning

3 Evaluation

4 Conclusion

# Caches and incremental development



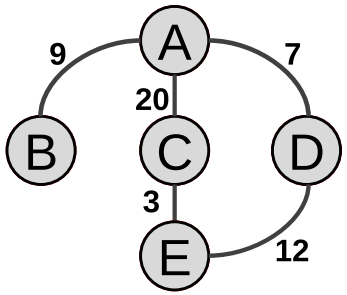


- **Holy grail of verification-intensive software industry**
  - Natural incarnation of the *divide-et-impera* approach into hardware and software development
  - To better master complexity and costs of industrial process
- **Is incremental WCET analysis even feasible?**
  - Relies on composability and early availability of timing bounds
    - The later those are determined the worse!
  - Hindered by context-dependent hardware resources
- **Caches inherently wreck incrementality**
  - Intra-task timing behaviour determined by **memory layout**
  - Not robust to software increments
    - Relatively small changes may cause significant jitter
  - Only available on the final executable
    - Too late to afford costly feedback cycles!

# Focusing on instruction cache



- **Cache-aware procedure positioning**
  - Improves both performance and predictability
    - Conflict misses avoidance or reduction
  - Granularity of procedures is industrially appealing
    - Methods on basic blocks too fine-grained and require specialized tool support
  - Reduces the potential jitter by pinpointing a memory layout
- **Graph-based program representation**
  - Weighted Call Graph ($\mathrm{WCG_P}$) for a program P is a (undirected) weighted graph with
    $V = \{p \mid p \text{ is a procedure in P}\}$
    $E \in V \times V = \{(p, p') \mid p \text{ calls } p' \vee p' \text{ calls } p\}$
    $W_{p,p'} \rightarrow$ call frequency between $p$ and $p'$ in P.




- **Placement heuristic**
  - Nodes pairwise merged according to $\max W_{p_i,p_j}$
  - Induced procedure ordering ▶ actual memory layout

# Limitations and drawbacks

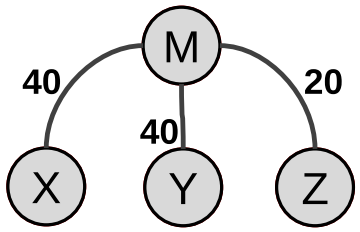


- **Weaknesses of current approaches**
  - ✗ Historically focused on **average-case** optimization
    - Build on execution traces rather than program structure
    - WCET-oriented approaches only recently proposed
  - ✗ **Poorly scalable** to large-scale industrial systems
    - Especially WCET-oriented methods as they rely on several iterations of static WCET analysis
  - ✗ Only applicable at the **tail end of development**
    - Thus failing to account for incremental nature of development
- **What we propose**
  - ✓ An alternative program representation, other than WCG
    - Improving on accuracy and scalability
  - ✓ An optimization method based on program structure
    - Holistically addressing both WCET and AVG performance
    - Incrementally applicable on subsequent software releases

# Need for an alternative representation



- **Pitfall of WCG**
  - WCG representation may be ambiguous




```
for I in 1..20 loop
  call Z;
  for J in 1..2 loop
    call X;
  end loop;
end loop;
for K in 1..40 loop
  call Y;
end loop;
```

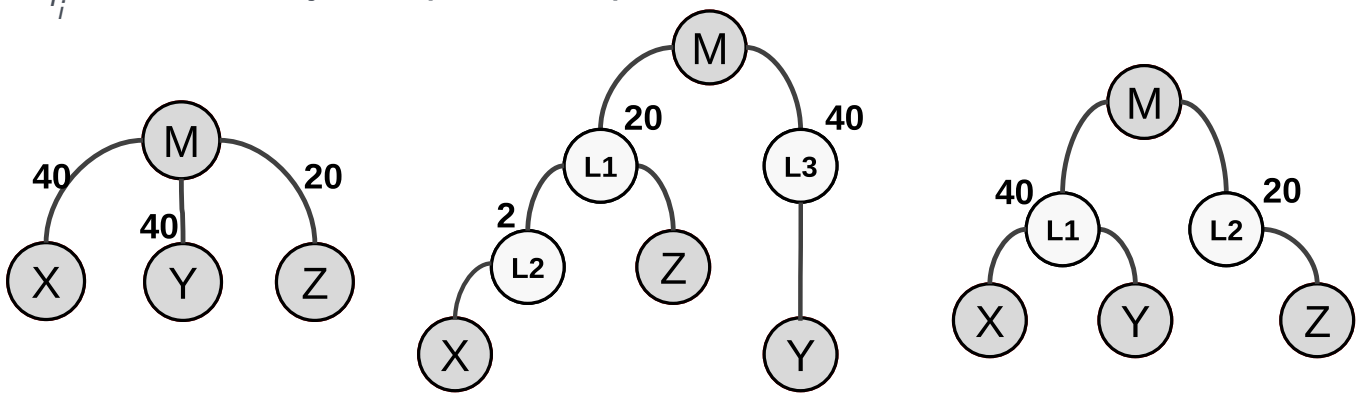
```
for I in 1..40 loop
  call X;
  call Y;
end loop;
for J in 1..20 loop
  call Z;
end loop;
```

  - With negative consequences on the computed layout
    - The sources of conflict misses are not necessarily the same
    - May lead to bad node merging (and layout)
  - Fails to account for the importance of **loop nests**
    - Call frequencies alone are not sufficient to catch all the structural information

# The Loop-Call Tree structure



- **Basic intuition**
  - Procedure involved in the same loop are the most critical source of cache conflicts
  - Need to explicitly consider loop nests
- **Loop-Call Tree**
  - $\mathrm{LCT_P}$ for a program P is an ordered directed tree with
    $V = \{p \mid p \in Proc(\mathrm{P})\} \cup \{\ l_i^p \mid l_i^p \text{ is the } i^{th} \text{ loop in } p\}$
    $E \in V \times V = \{(p,\ p') |\ p \to p'\} \cup \{(p,\ l_i^p)\} \cup \{(l_i^p, p') |\ l_i^p \to p'\} \cup$
    $\{(l_i^p, l_j^p) |\ \text{loop } l_j^p \text{ is nested inside } l_i^p\}$
    $B_{l_i^p} \to$ statically computed loop bound

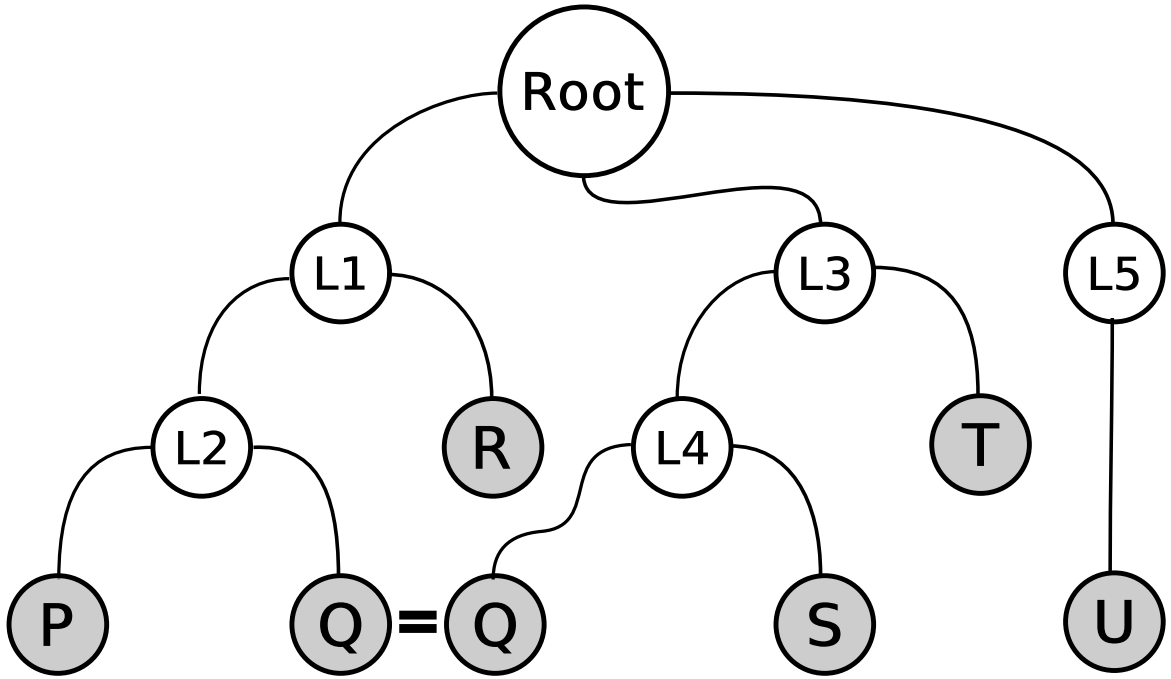


# Computing an optimal layout



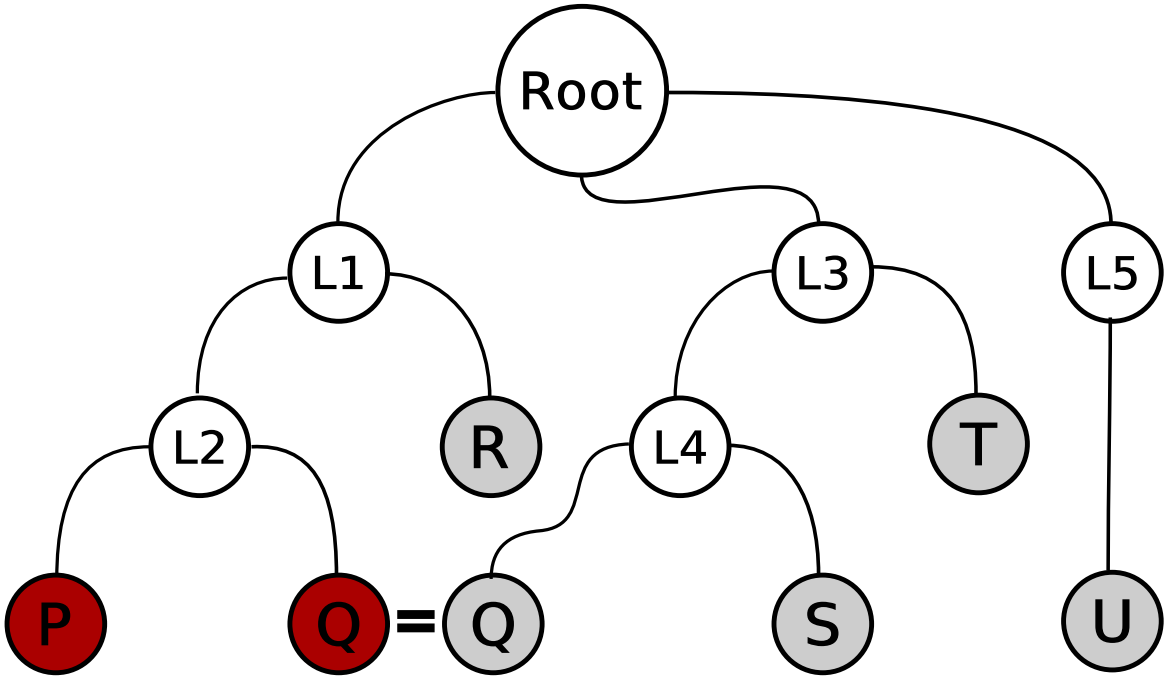
- **LCT structural properties**
  - Naturally exhibits loop-induced relation between procedures
  - Subtrees can be ordered wrt depth and execution frequency
    - Several heuristics can be defined
  - **Post-order depth-first traversal**
    - Privileges nodes belonging to the same loop nest
- **Procedure selection**
  - Procedures on the same subtree ▶ *independent pools*
  - Incrementally merged together
  - Pool independency broken by procedures appearing in different subtrees
    - *Memory displacements* introduced in the merging step
    - Fragmentation cured with *relatively independent* procedures

# Example



# Example

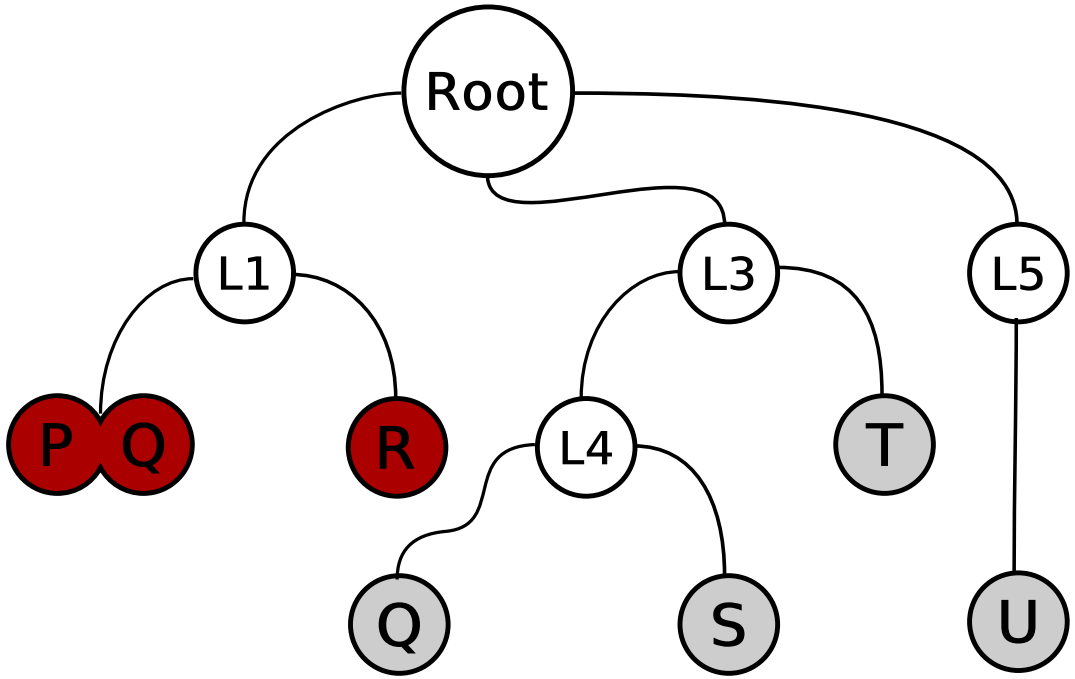





Select first nodes

# Example

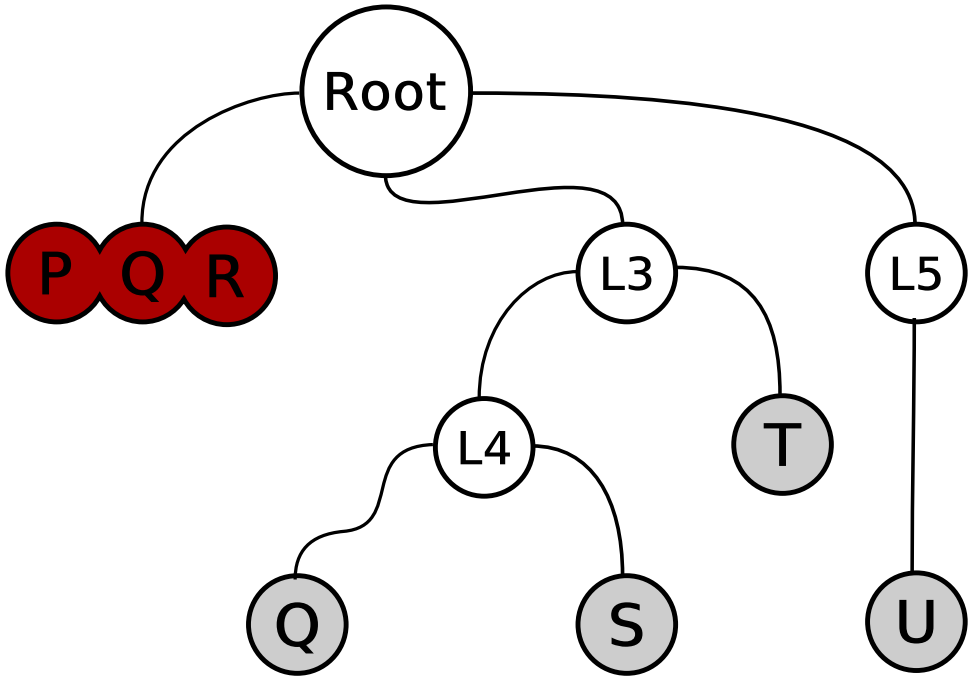





Merge P and Q

# Example

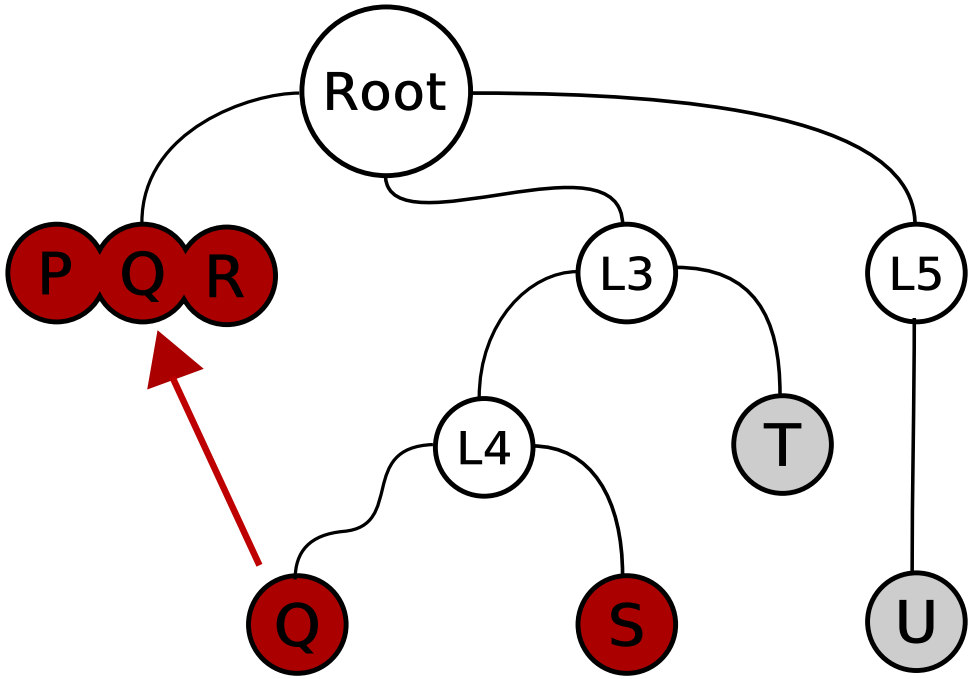





Keep on merging

# Example

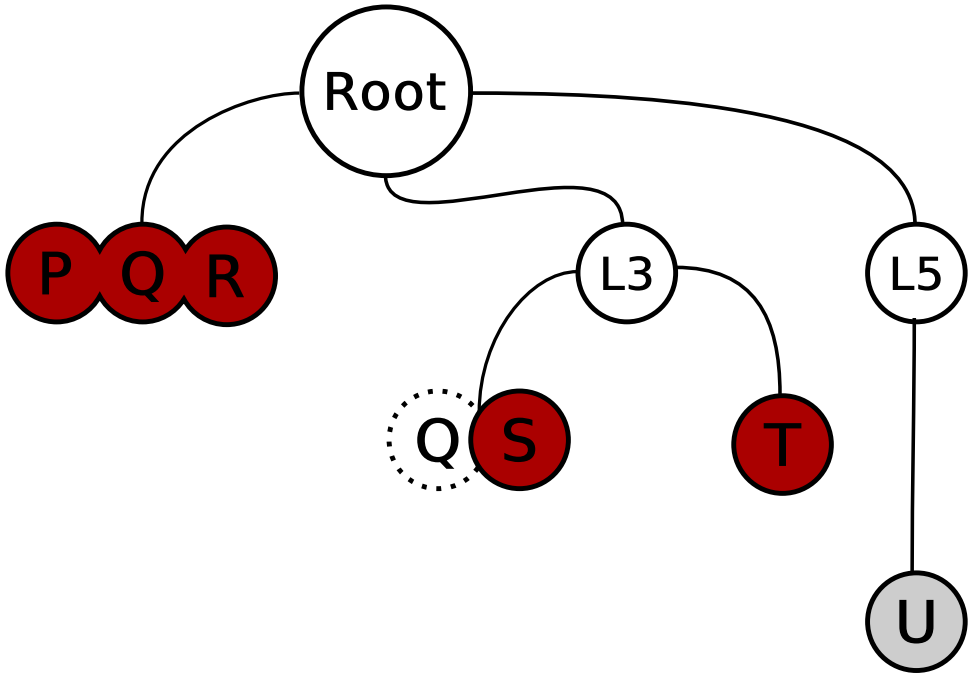





Q already in the pool...

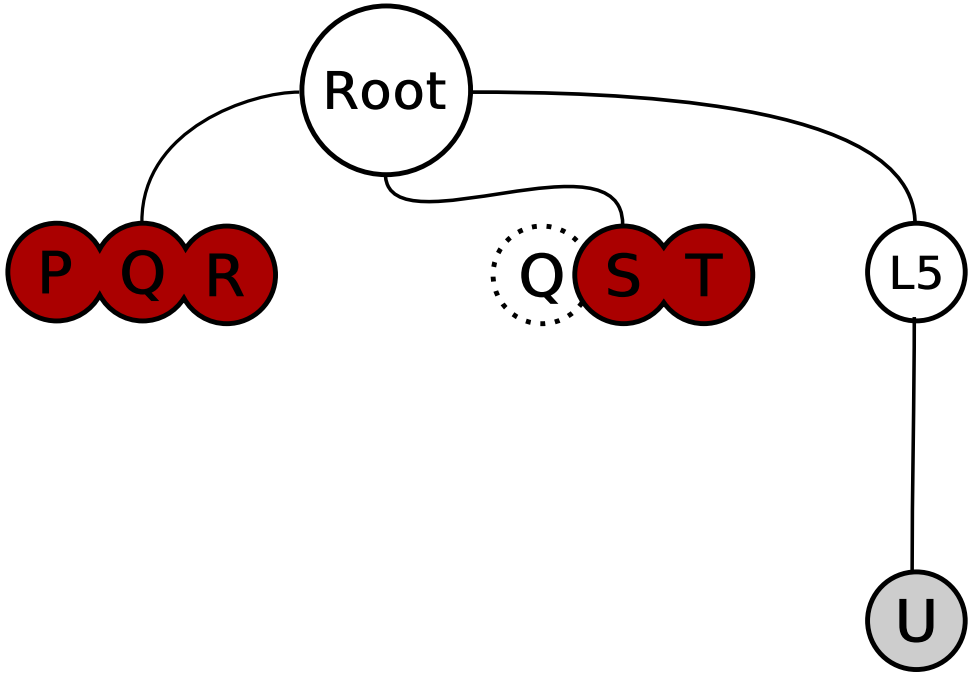
# Example





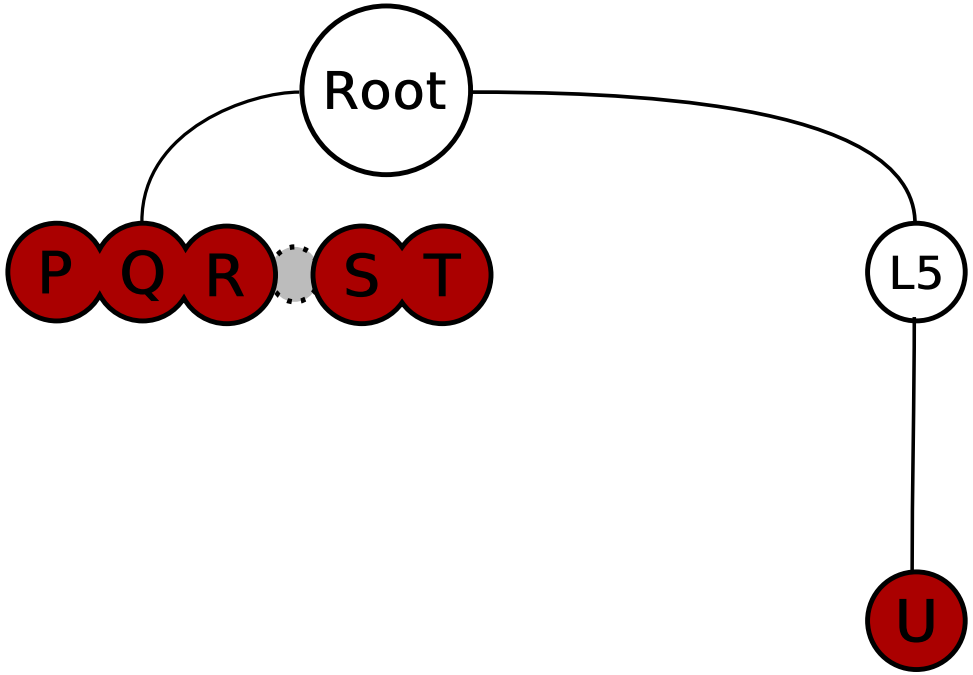

...just remind it

# Example



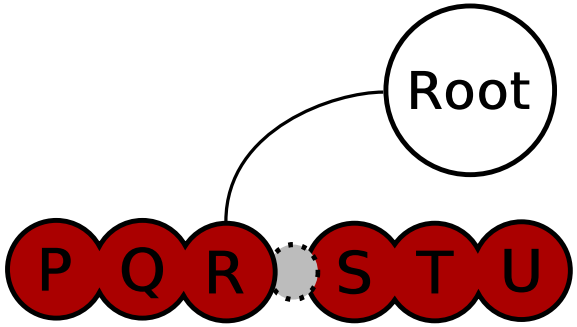

[Merge S and T]

# Example



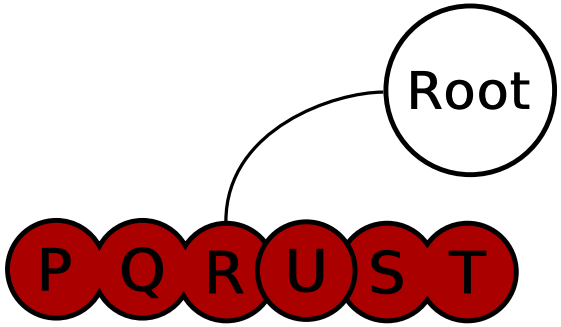

[Merge optionally with displacement]

# Example




[U does not fit in the gap]

# Example




[U fits in the gap]

# Fitting all into incremental development

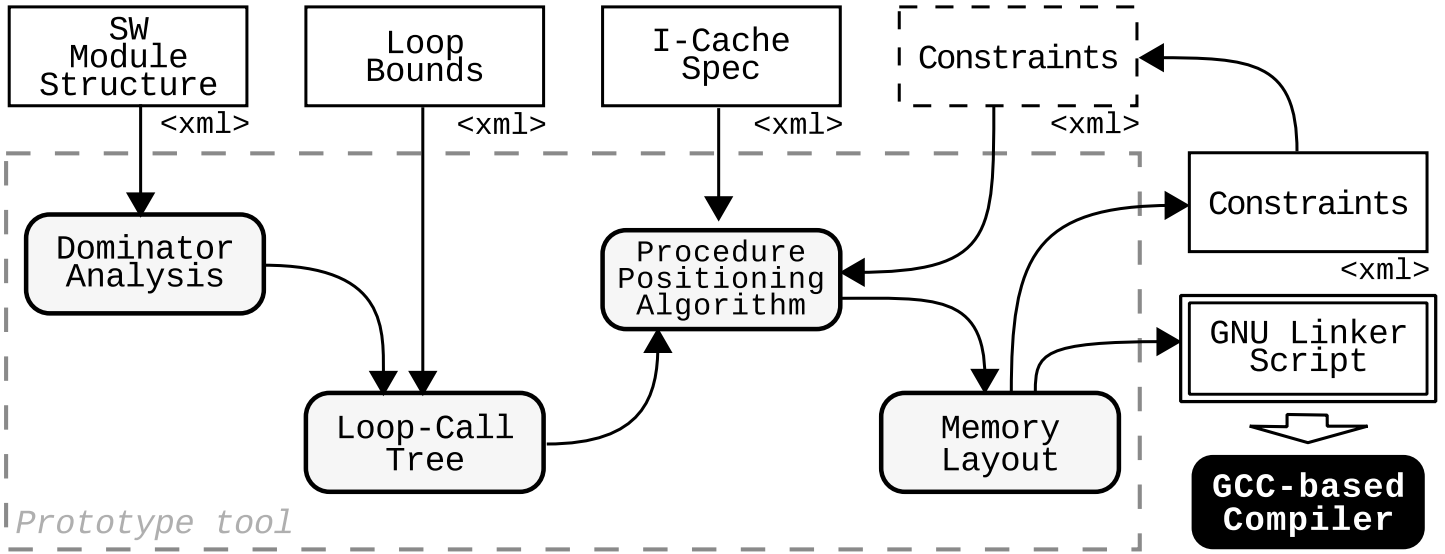


- **Development as a sequence of incremental steps**
  - Qualification status should be incrementally preserved
    - For either *additive* or *corrective* increments
    - No regression outside of the modules intentionally affected
  - When it comes to caches
    - Memory layout of pre-existent modules must be preserved
- **Incremental optimization**
  - LCT intrinsically fit to incremental addition
    - No assumptions on the pre-existing pools in the merging step
    - Keep global ordering up to the increment as set of constraints
    - Exploit them as an initial pre-existing subtree
  - Naturally absorbs changes that are local to a module
    - Changes within a subtree do not affect ordering of others
  - Problems arise with shared procedures
    - Introduce dependences (i.e., diplacements) within subtrees
    - Layout preservation may require high fragmentation

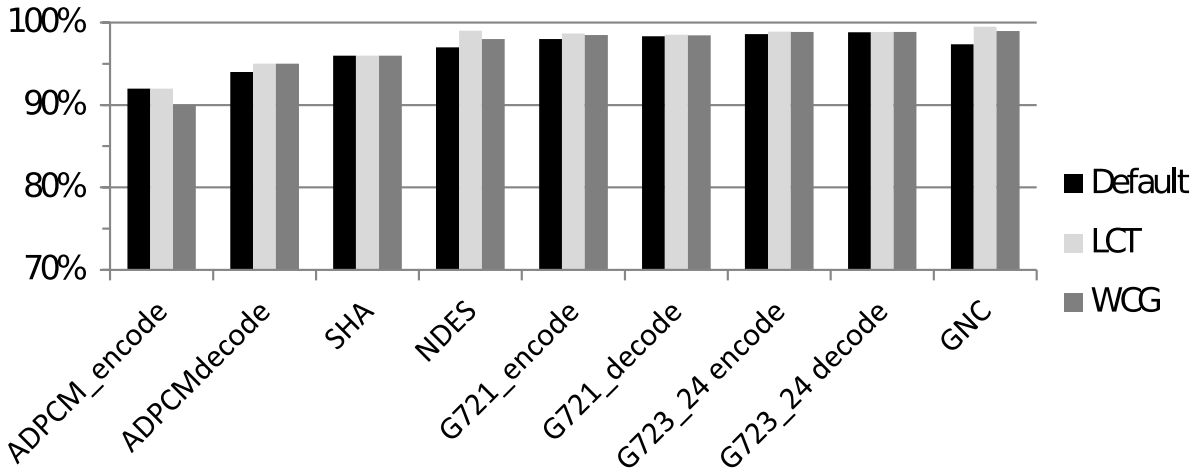
# Evaluation



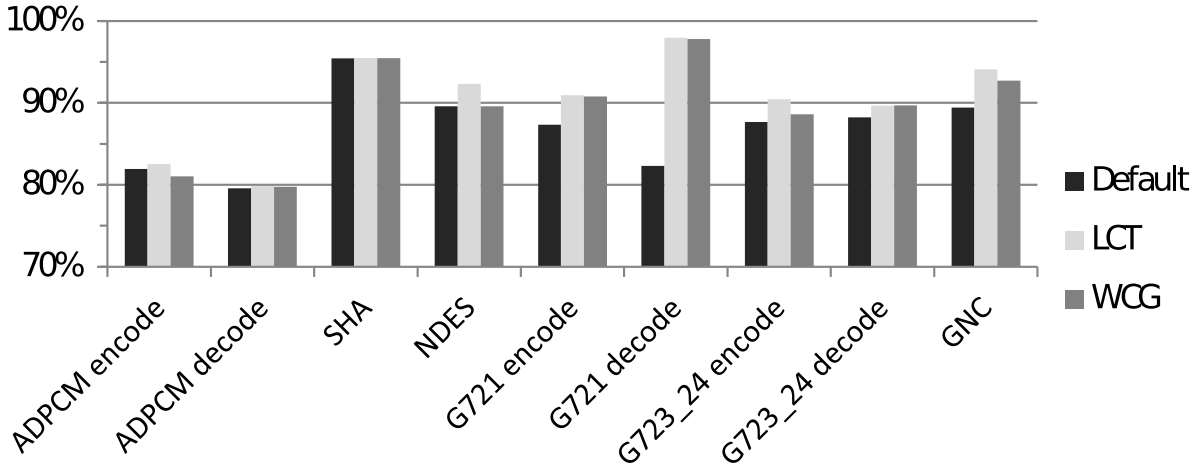
- **On AVG/WCET I-cache behaviour and WCET variation**
  - Targeting the LEON2 (SPARC V8) processor
  - Focusing on reference and domain-specific benchmarks
    - Mälardalen, Mediabench, MiBench, AOCS software
- **Prototype tool**



# Average and worst-case performance




[Average-case hit ratio]

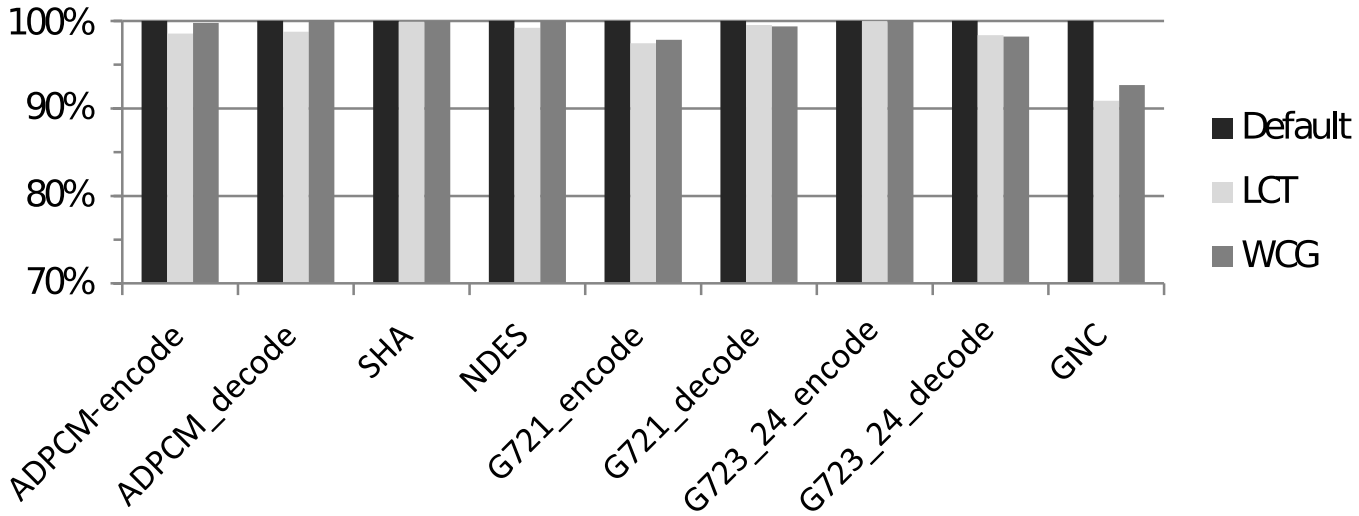



[Worst-case hit ratio]

# Corresponding global WCET performance



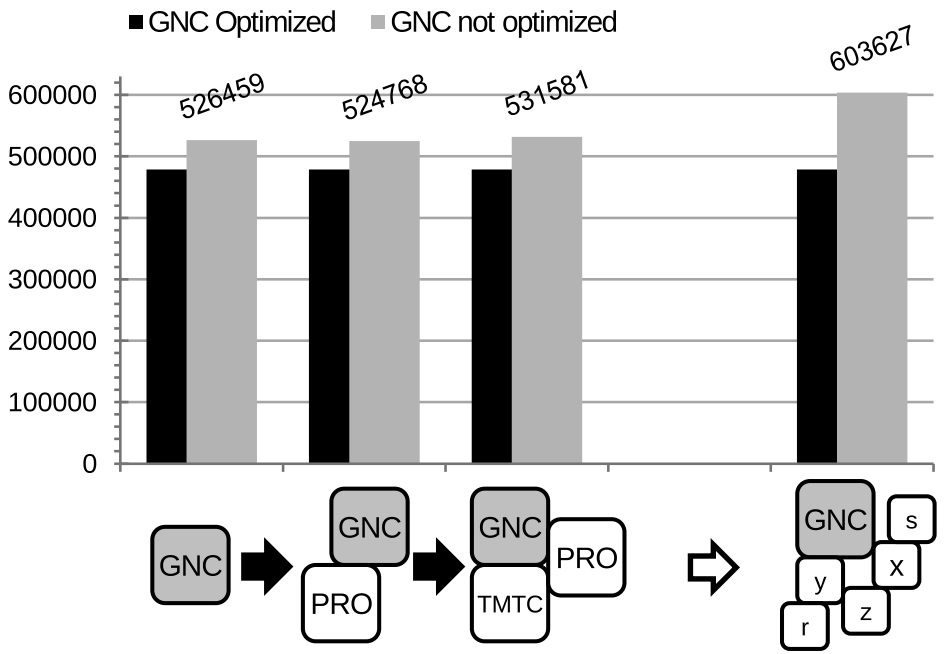
- **Assessing the overall WCET improvement**
  - Fairly proportional due to the relatively simple HW platform and setting (e.g., D-cache disabled)




[Global WCET reduction]

# Robustness to incremental release

- **Simulated incremental steps**
  - Modules from the AOCS benchmark (GNC, PRO, TMTC)
  - Confirms constant WCET behavior for GNC
    - Against an up to +26% potential variation if no countermeasure is taken
    - Low fragmentation: less than 2% increase in executable size




[WCET variation across releases]

# Conclusion



- **Novel procedure positioning approach**
  - More accurate program representation
  - Improves both avg and wc performance
  - Robust against incremental development
- **Limitations**
  - Still need a better solution to handle regression in the presence of shared procedures
  - Iterative (but costly) WCET-oriented approaches may provide better WCET performance
- **Future work**
  - Implement our approach as a plugin to standard GCC compiler
  - Undergo an extensive evaluation of different ordering heuristics