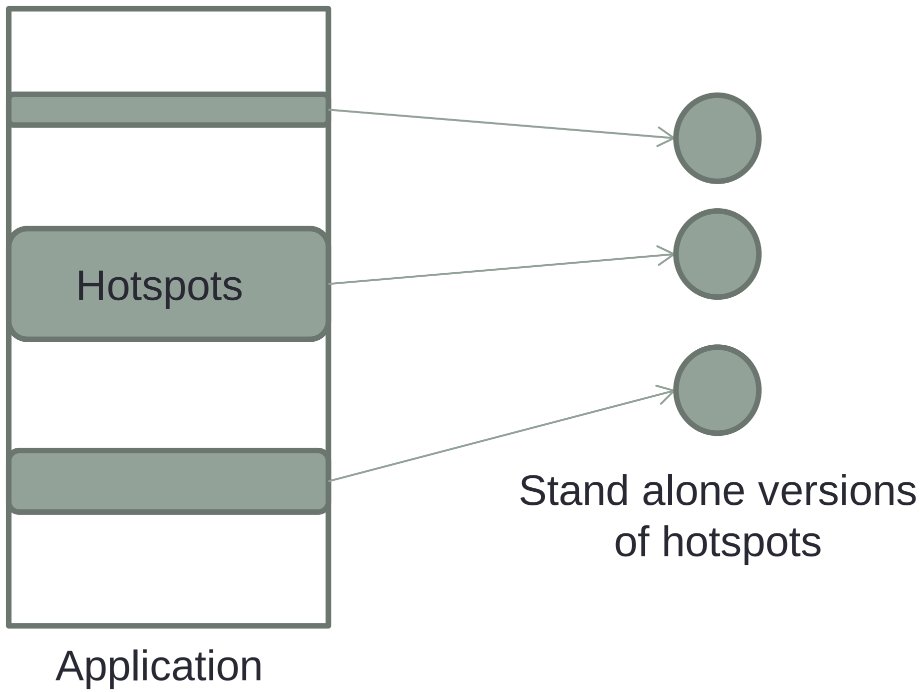
IS SOURCE-CODE ISOLATION VIABLE FOR PERFORMANCE CHARACTERIZATION?

C. Akel, Y. Kashnikov, P. de Oliveira Castro, W. Jalby
University of Versailles
Exascale Computing Research

Why extracting code?

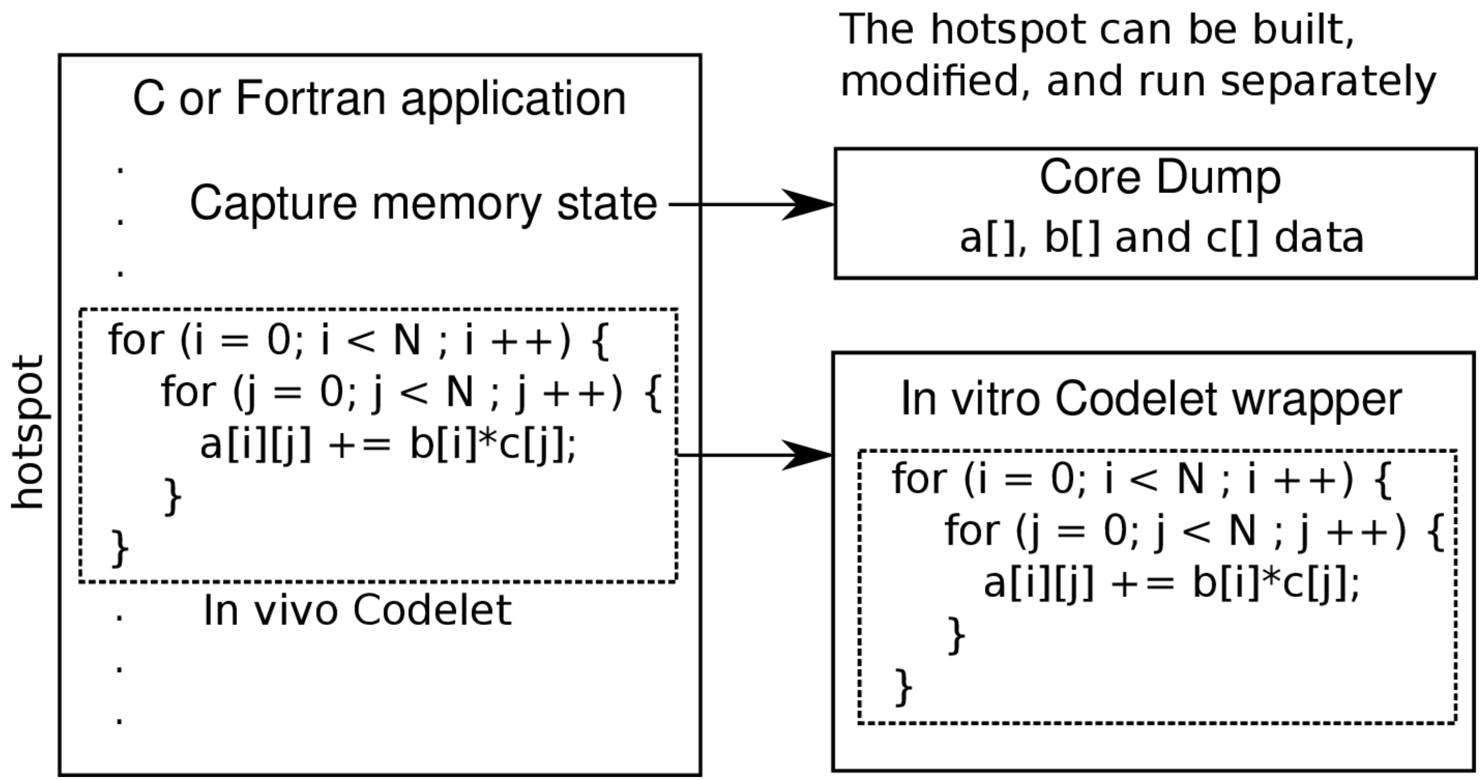
- Problem
  - Benchmarking applications is costly.
- Break applications into stand alone programs
  - « Piece-wise » benchmarking and optimisations.



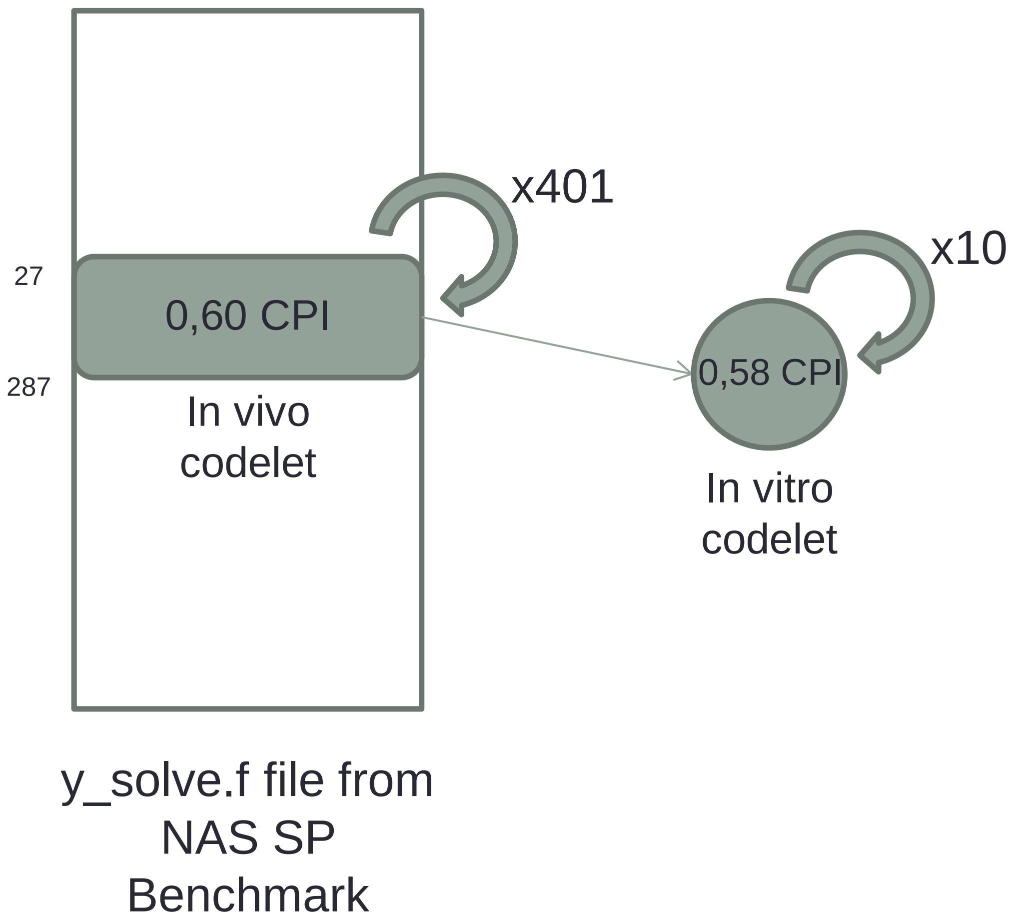
- Extractible hotspots are called Codelets

Codelets?

- **In-vivo codelets**: Hotspots inside the original application.
- **In-vitro codelets**: Stand alone hotspots extracted from the original application.



Faster Benchmarking



- Cycles Per Instructions (CPI) Error is 4,4%.
- Benchmarking the application: 215,32 seconds.
- Benchmarking the in vitro version: 0,98 second.
- Speedup of 214 in Benchmarking time.

- Can we always use codelets for performance characterization?

Conditions

- To use in vitro benchmarking we need to guarantee that:
  - The codelet can be extracted.
  - The codelet has the same behavior in vivo and in vitro.
- To characterize an application:
  - Extracted codelets must cover most of the application’s original execution time.

Related Work

- Code Isolator [Lee2004]
- Astex [Petit2006]
- Codelet Finder, CAPS Entreprise 2010.

- No complete comparison between in vivo and in vitro codelets.

Tools and Benchmarks

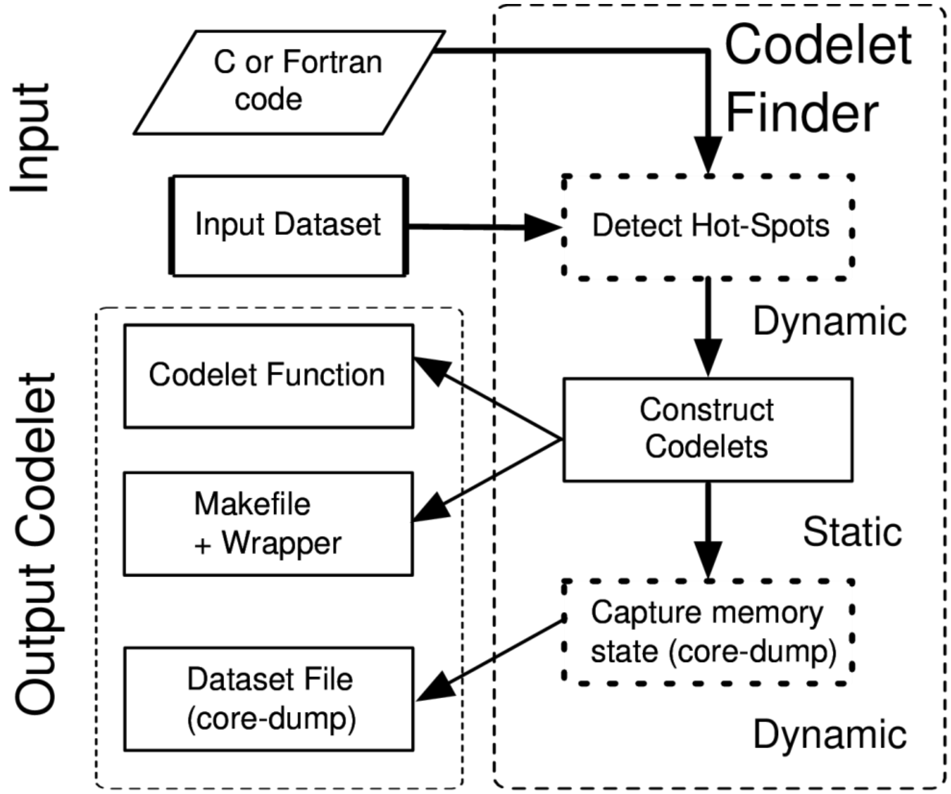
- Codelet Finder, CAPS Entreprise.
- Maqao static loop analyzer.
- Likwid.

- NAS-SER:
  - NASA Benchmarks.
  - 8 Benchmarks.
  - Class B data set size.

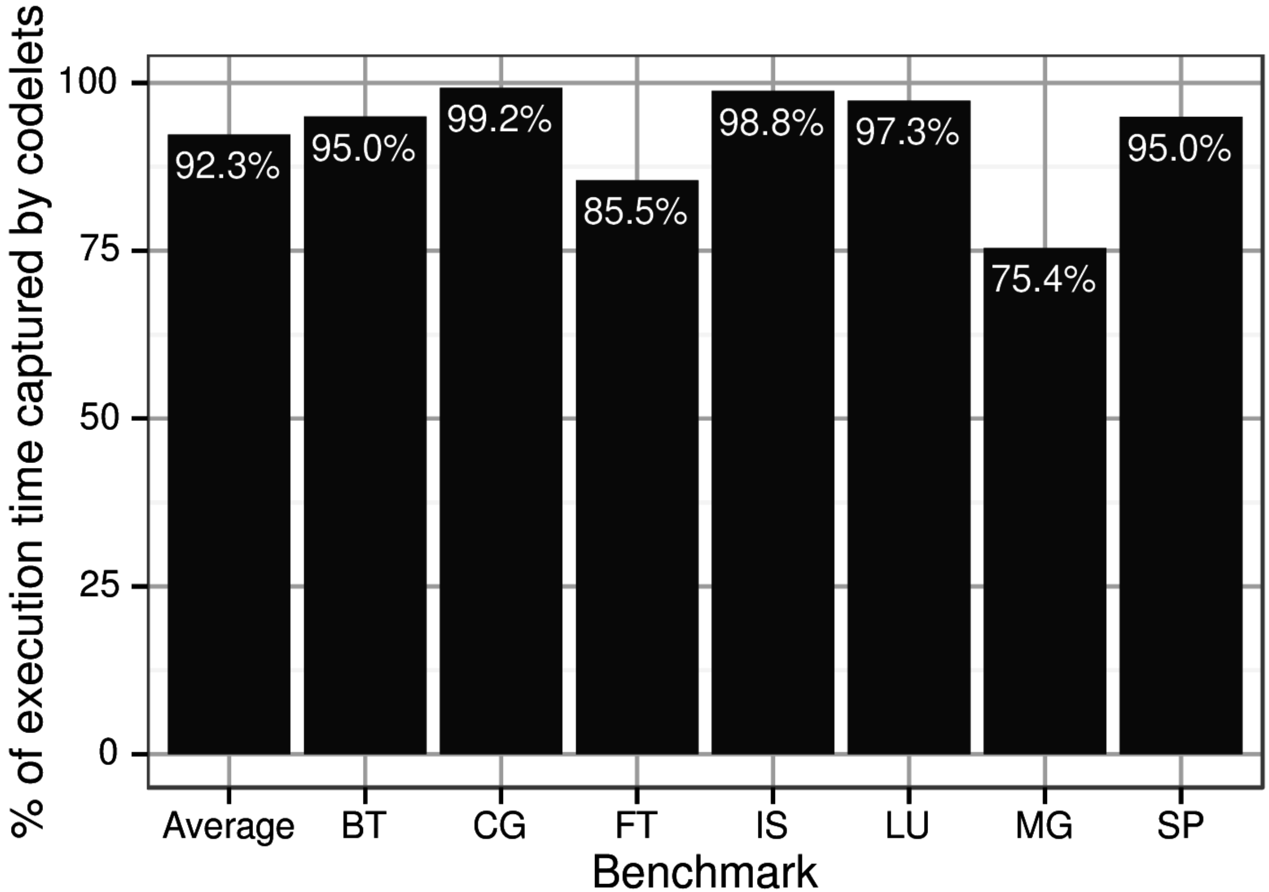
- Platform:
  - Intel(R) Xeon(R) L5609 @ 1.87GHz with 12MB L3 cache
  - 8 GB of RAM

Codelet Finder

1. Detects loops at source level.
2. Extracts each loop as a separate codelet.
3. Runs the original application to capture the memory state.



Coverage



- We extended Codelet Finder to support extraction of codelets calling functions in other files.
- Coverage ratio is sufficient to use codelets.

Discrepancies

- Behavior must be the same between in vivo and in vitro versions of a codelet.

- To verify this condition we need to:
  - Analyse the causes of discrepancies.
  - Improve matching.
  - Quantifying discrepancies.

- Two types of discrepancies:
  - Assembly discrepancies.
  - Runtime discrepancies.

Assembly Discrepancies

- Codelets are extracted at the source level.

- Drawback:
  - Assembly code may differ between in vitro an in vivo codelets.

- Three causes of assembly discrepancies:
  - Dereferencing.
  - Interference with Loop Variables.
  - Compiler Heuristics.

Assembly Discrepancies
Interference with Loop Variables

- Function parameters can prevent some optimizations.
- Fix:
  - Apply variable cloning to the loop and loop bound variables.

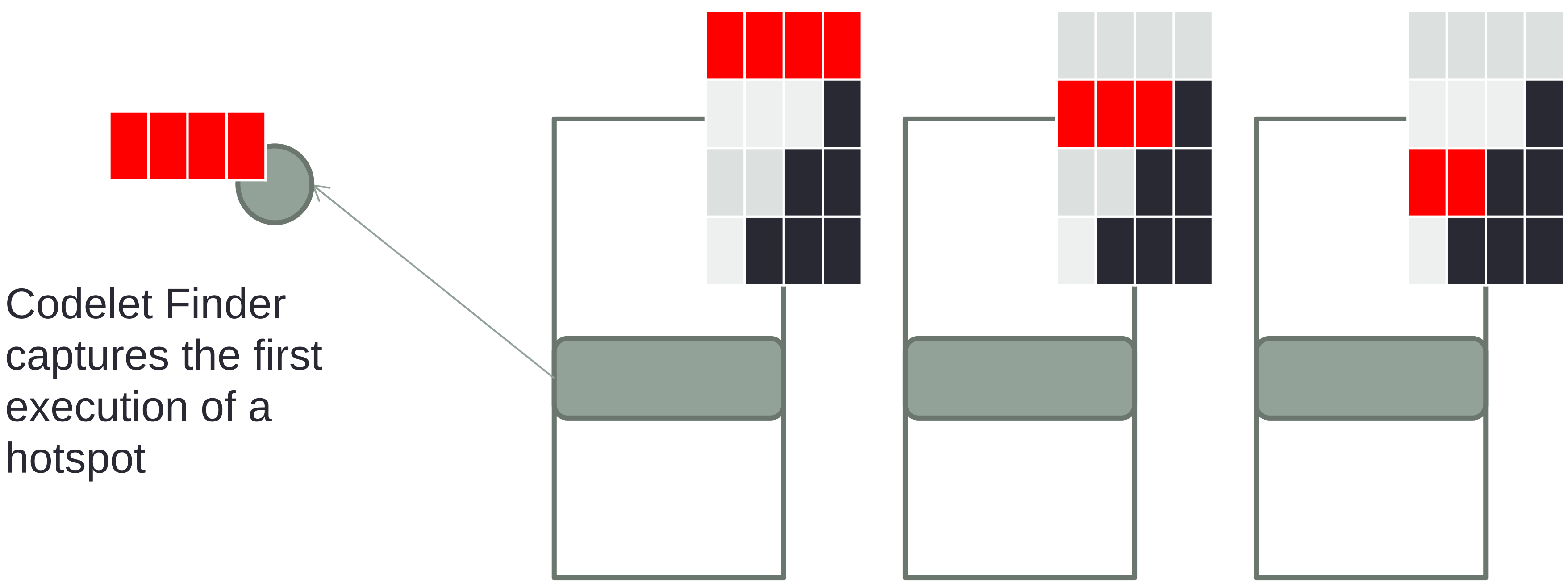
```
subroutine codelet(rhs,i,j,k,m,
                   nz2,ny2,nz2,dt)
 INTEGER :: i, j, k, m, nz2, ny2, nz2, dt
 do k=1, nz2
   do j=1, ny2
     do i=1, nx2
        do m=1, 5
          rhs(m, i, j, k) =
             rhs(m, i, j, k) * dt
 end do; end do; end do; end do


!    ...
```

Codelet extracted from NAS SP (scalar Pentadiagonal solver)

Runtime Discrepancies

- Most of the time, same assembly equals same runtime behavior.
- Runtime behavior may be different:
  - Different data per invocation.



Quantifying Discrepancies
Methodology

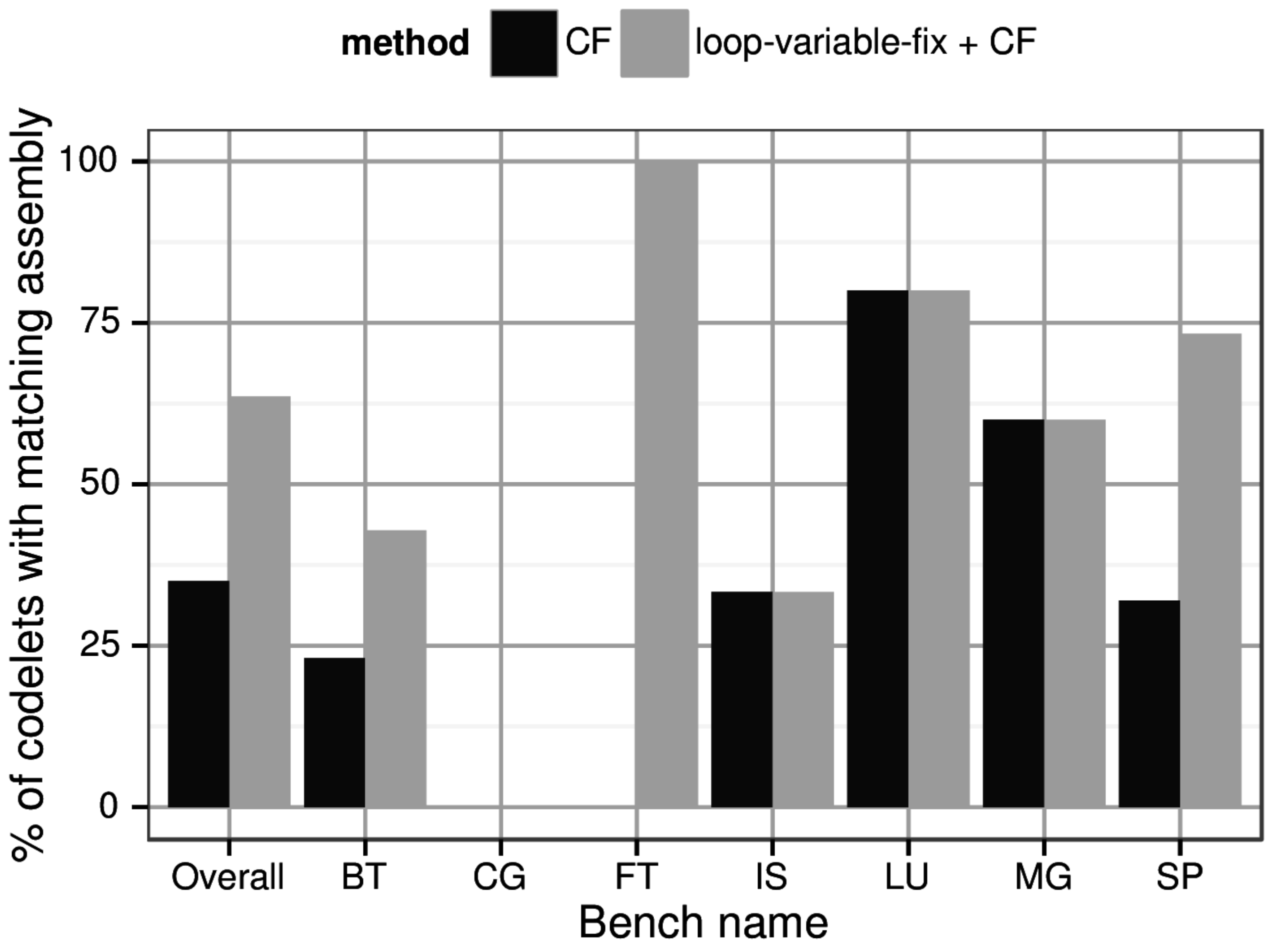
Assembly comparison

- Conducted using the MAQAO static loop analyser.
- For unroll Factor, Nb FLOP mul, Vec. ratio etc…
- Difference between those characteristics must be under 15%.

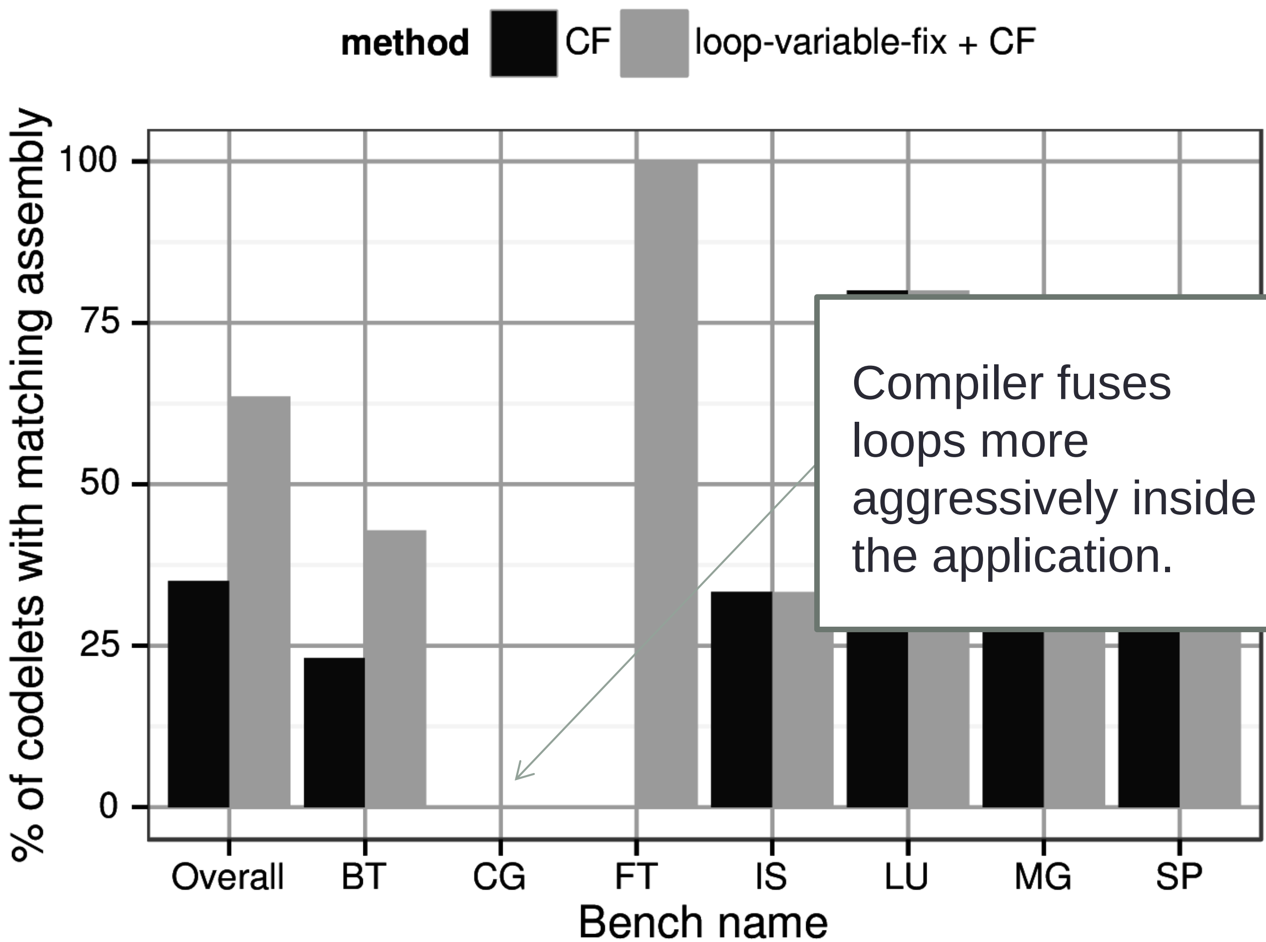
Runtime comparison

- Conducted using Likwid.
- For Intructions retired and CPI.
- Codelets runtime match if difference is above 15%.

Quantifying Assembly Discrepancies
Results

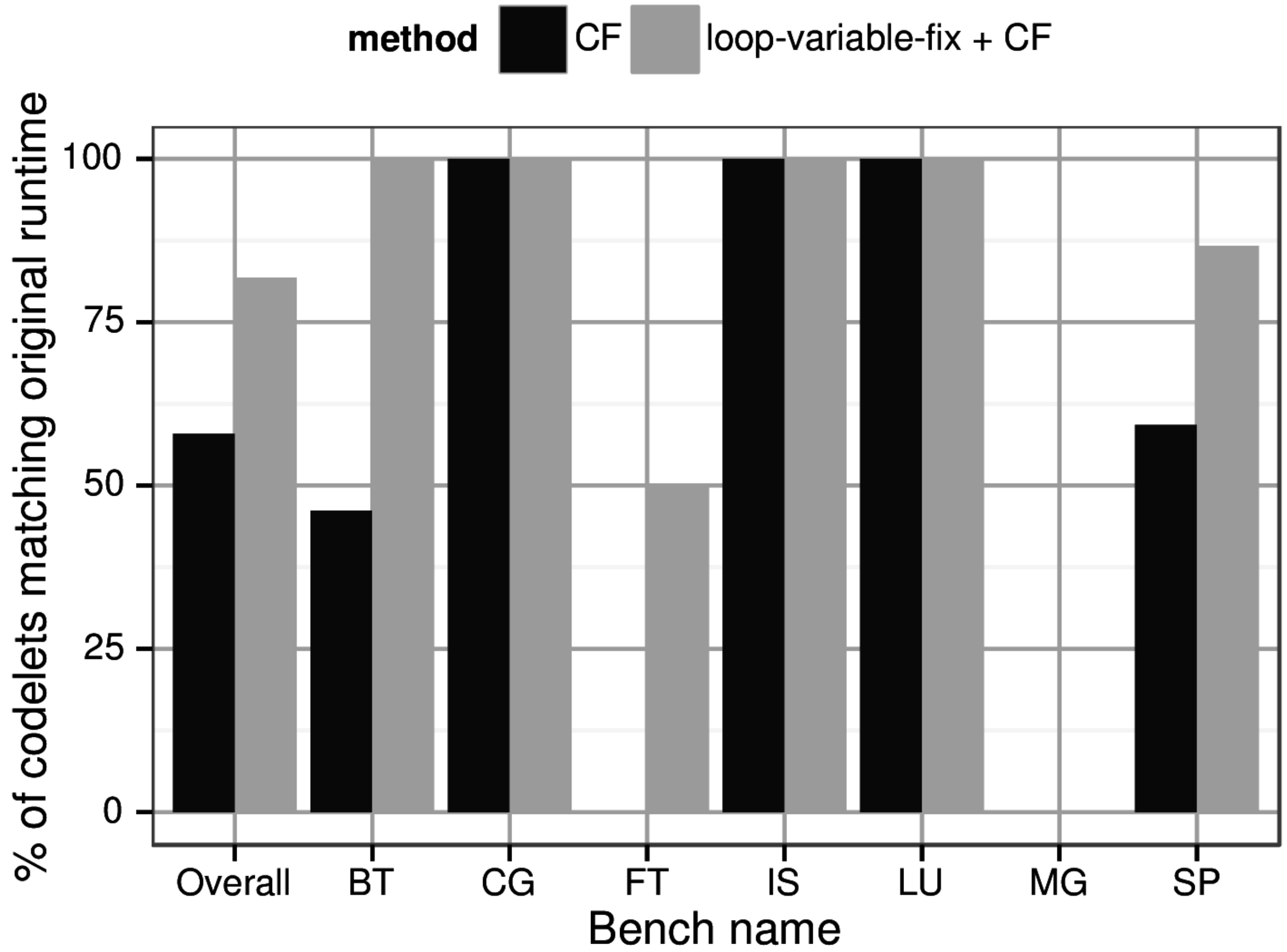


Quantifying Assembly Discrepancies
Results

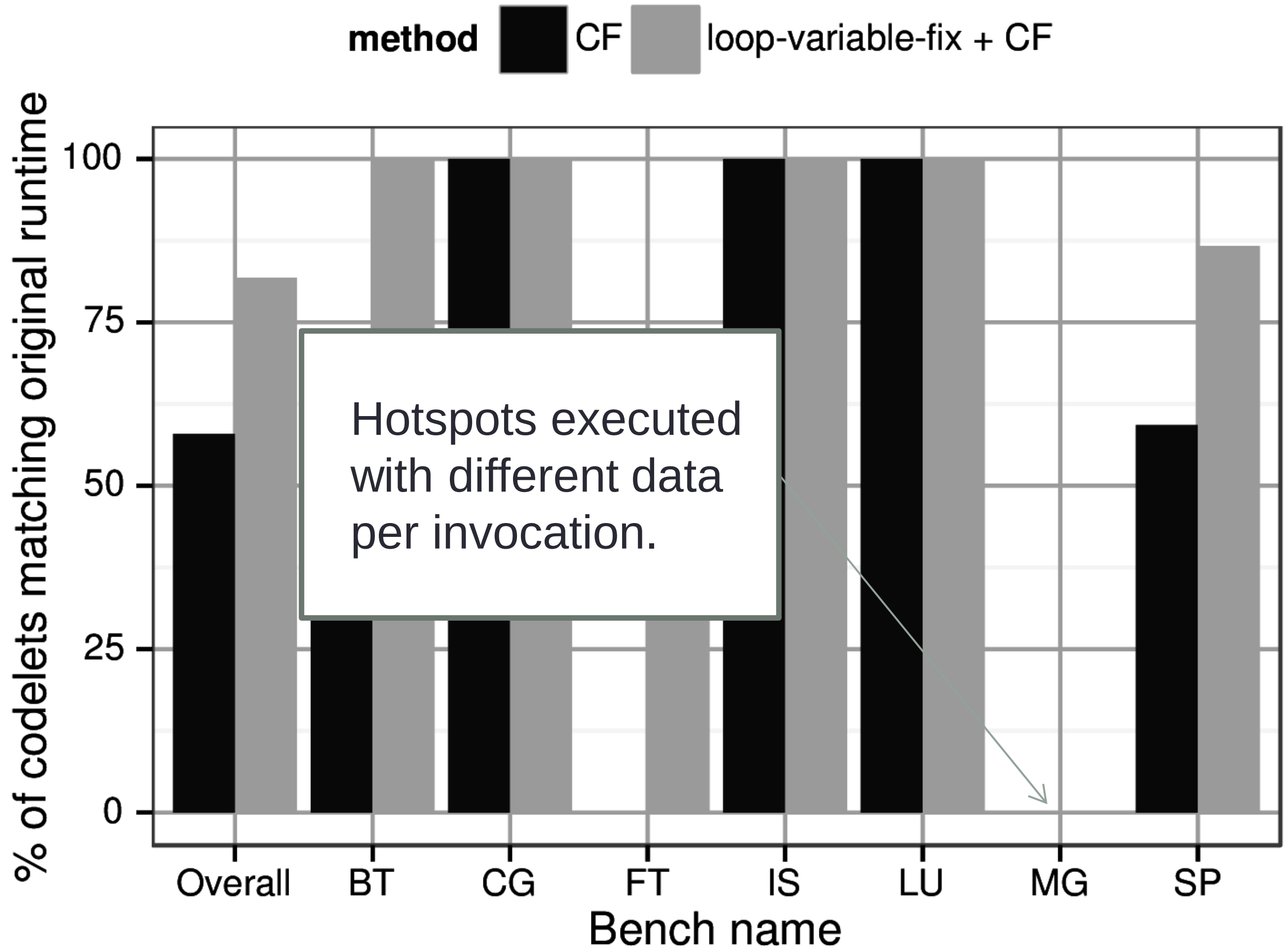


Compiler fuses loops more aggressively inside the application.

Quantifying Runtime Discrepancies
Results



Quantifying Runtime Discrepancies
Results



Results analysis

- Four scenarios:
  - **Assembly and Runtime matches**: 52.1%
  - **Nothing matches**: 6.9%
  - **Only Assembly matches**: 11.5%
    - In vivo codelets invoked with different data.
  - **Only Runtime matches**: 29.5%
    - Different compiler optimizations.
    - But did not impact the performance.

Future Work

- Manage different dataset per invocation.
- Extend this study to include more architectures and benchmarks.
- Evaluate in what measure codelets can be used for piece-wise optimization of programs.
- Predict application performance using codelets.

Conclusion

- Code isolation captures 92.3% of the total running time of the original NAS benchmarks.

- Automated the loop-variable-fix.

- Overall for the NAS benchmarks:
  - 63.6% of the codelets match the original hotspot assembly.
  - 81.6% of the codelets match the original runtime behavior.

- Codelets can therefore be used to optimize or benchmark an application most of the time.