Dynamic Branch Resolution based on Combined Static Analyses

Wei-Tsun SUN, Hugues CASSE
IRIT, Université Paul Sabatier

WCET 2016, Toulouse, France
5th July 2016

This work is supported by the French research foundation (ANR) as part of the W-SEPT project (ANR-12-INSE-0001)

Content of the talk

- Goal: find the target of “dynamic branches”
- Introduction: CLP and k-set analyses
- Improvement using “program slicing”
- Experiments
- Conclusions

What are the “branches”

- We talk about branches as:
  - In the assembly manner
  - Implement if-else, function calls, switch cases, etc.
  - Have target addresses

Lets talk about the “static branches” first

- Target address is evaluated at compile-time

- PC calculation: constant value or a constant shift to the current PC

- if-else and normal function calls
  - e.g.1. BL 0x8AE0 ; calling a function
  - e.g.2. CMP R2, 3 ; a if-else construct
BEQ 0x8A9C

Dynamic branches

- Target addresses are computed at run-time
  - i.e. switch-cases, calls on function pointers

- ldrls pc, [pc, r3, lsl #2]
  - used by GCC for implementing switch-case with jump tables
  - ldrls pc: load a value from memory to PC, when condition code is LS
  - the address is calculated with registers pc and r3
  - the value of r3 varies during run-time

Overall flow of discovering target address

- To resolve dynamic branches

- We use the combination of analyses:
  - Circular-Linear Progression (CLP) + k-set + DynamicBranch
  - Program slicing + CLP + k-set + DynamicBranch

- We are going to use short names in the slides:
  - CLP: the representation of CLP or the analysis uses CLP
  - k-set
  - DB: dynamic branching
  - PS: program slicing

CLP: Circular-Linear Progression

- A way to capture a set of values

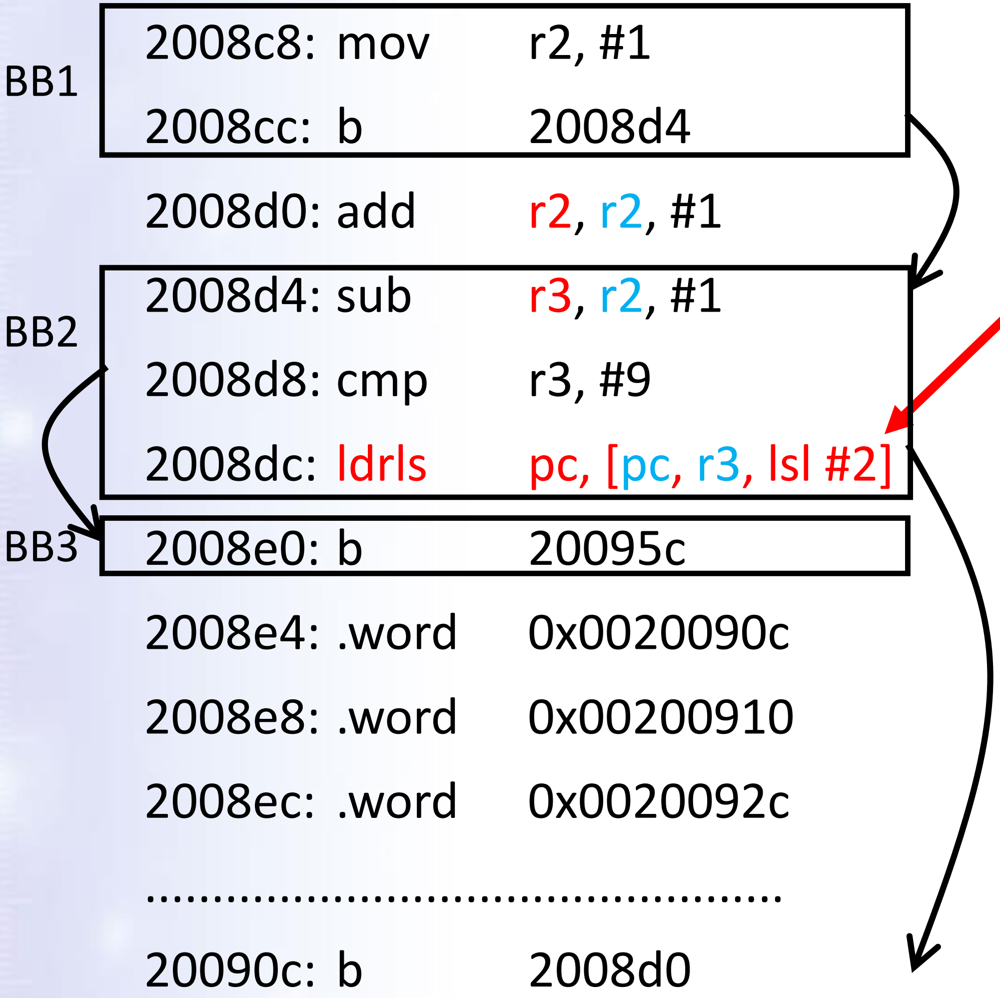
- Given a set: {2, 4, 10}
  - Pattern: difference of 2, starting from 2
  - Create: {2, 4, 6, 8, 10} // 6 and 8 are redundant
  - (base, delta, mtimes) = (2, 2, 4)

- Use abstract interpretation (AI)
- Advantage: compact in space (3 integers)
- Disadvantage: introduce imprecision

k-set

- A set of size k
- The domain capture the actual values
  - i.e. {2, 4, 10}
- More precise
- Faster to converge on AI. Widen to top when current and next sets are different
- More expensive (scalable ?)
  - When analysing the whole program, it definitely needs more memory then CLP

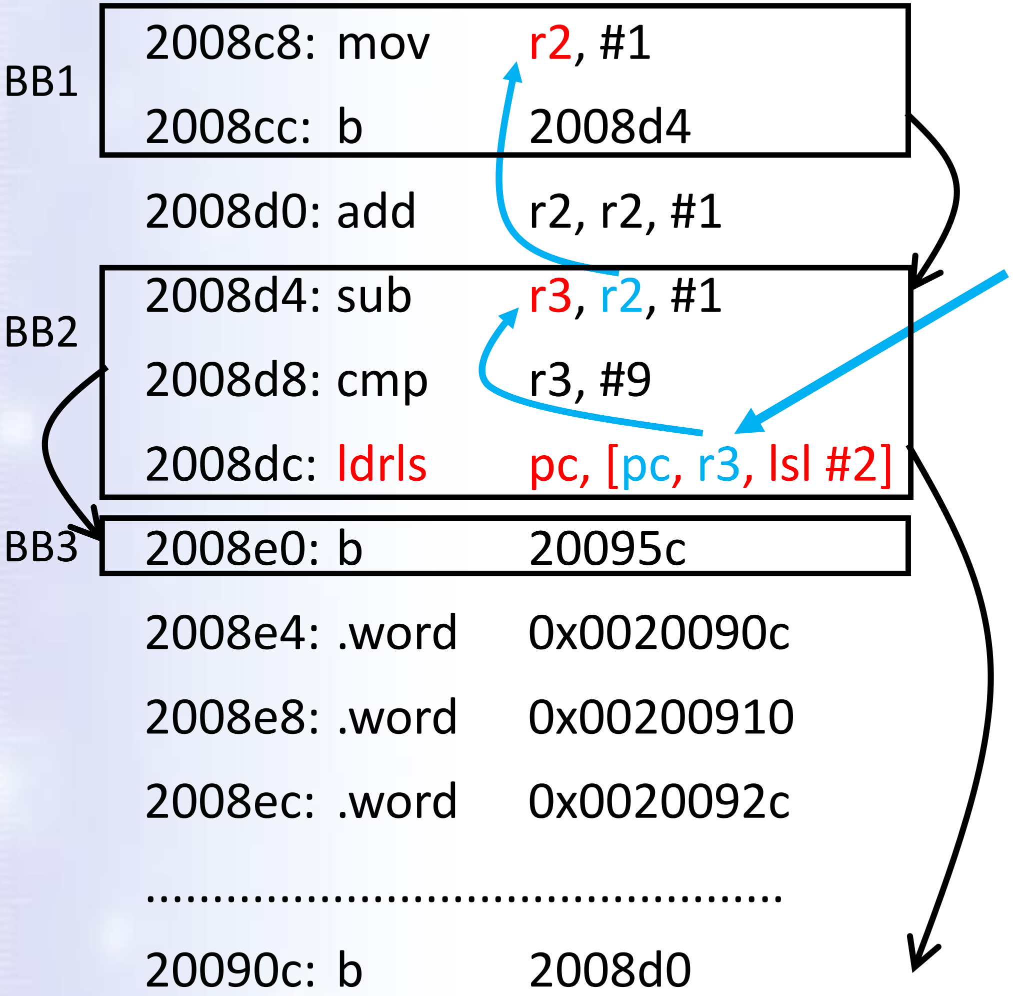
Dynamic branch analysis



```
BB1  2008c8: mov     r2, #1
     2008cc: b       2008d4
     2008d0: add     r2, r2, #1
BB2  2008d4: sub     r3, r2, #1
     2008d8: cmp     r3, #9
     2008dc: ldrls   pc, [pc, r3, lsl #2]
BB3  2008e0: b       20095c
     2008e4: .word   0x0020090c
     2008e8: .word   0x00200910
     2008ec: .word   0x0020092c
     ..............................................
     20090c: b       2008d0
```

- Firstly identify the dynamic branches

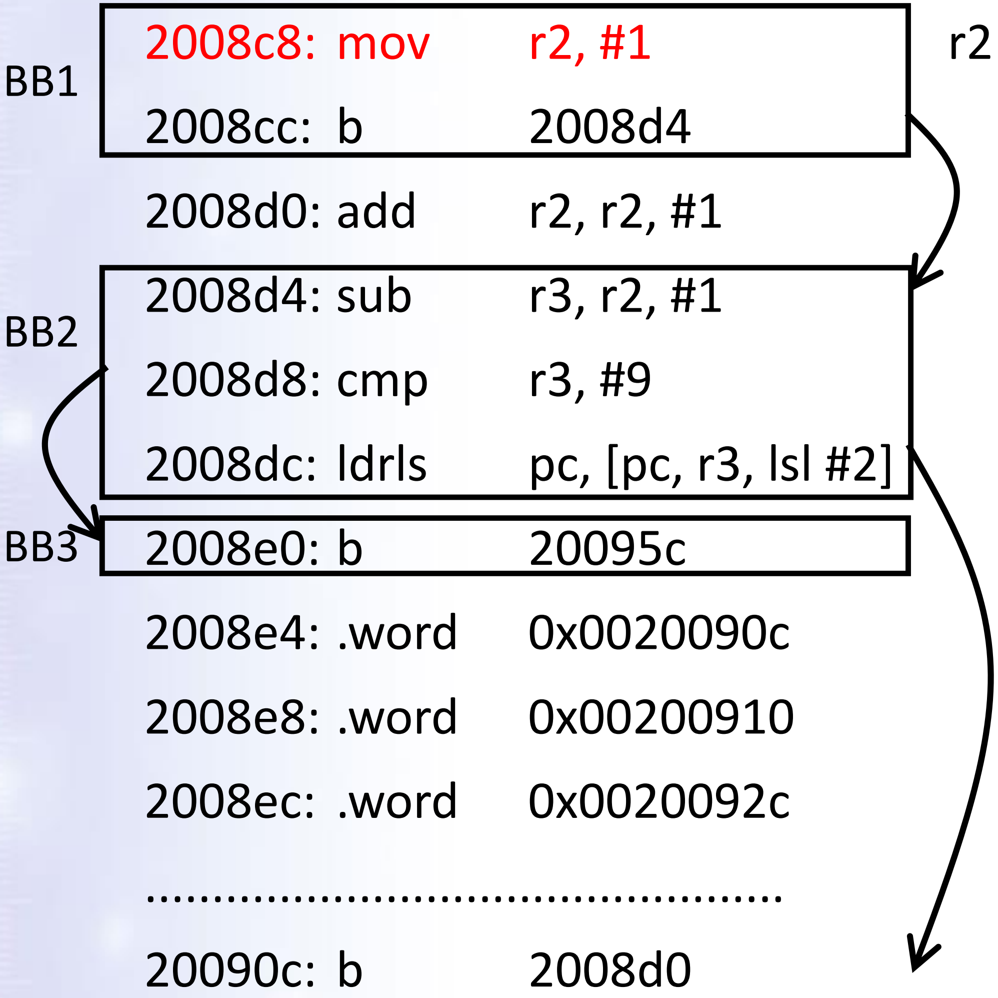
Dynamic branch analysis



```
BB1  2008c8: mov     r2, #1
     2008cc: b       2008d4
     2008d0: add     r2, r2, #1
BB2  2008d4: sub     r3, r2, #1
     2008d8: cmp     r3, #9
     2008dc: ldrls   pc, [pc, r3, lsl #2]
BB3  2008e0: b       20095c
     2008e4: .word   0x0020090c
     2008e8: .word   0x00200910
     2008ec: .word   0x0020092c
     ..............................................
     20090c: b       2008d0
```

- Firstly identify the dynamic branches
- Find out the values of relevant registers and memories
  - Get values from k-set
  - If not available, get it from CLP
- Why need k-set ?

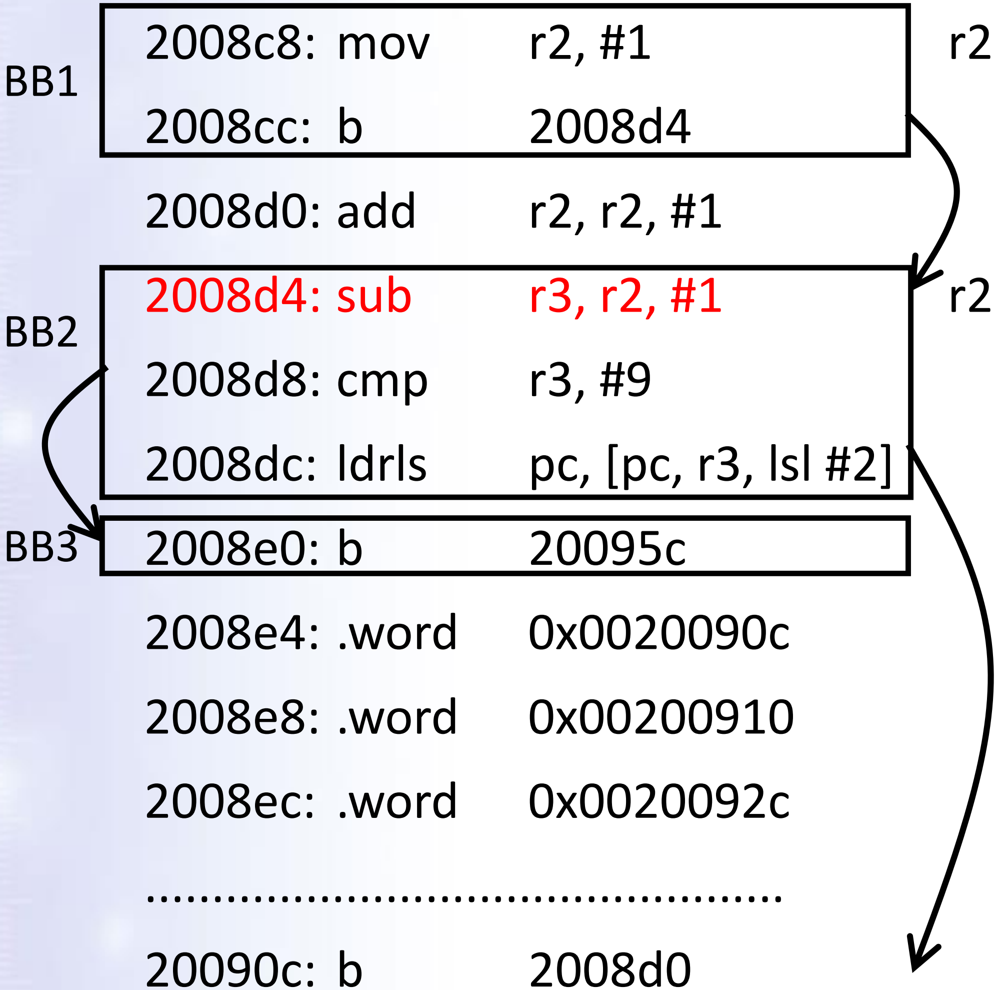
CLP analysis with abstract interpretation



```
2008c8: mov     r2, #1
2008cc: b       2008d4
2008d0: add     r2, r2, #1
2008d4: sub     r3, r2, #1
2008d8: cmp     r3, #9
2008dc: ldrls   pc, [pc, r3, lsl #2]
2008e0: b       20095c
2008e4: .word   0x0020090c
2008e8: .word   0x00200910
2008ec: .word   0x0020092c
.............................................
20090c: b       2008d0
```

r2 = (1,0,0) // a constant value of 1

CLP analysis with abstract interpretation

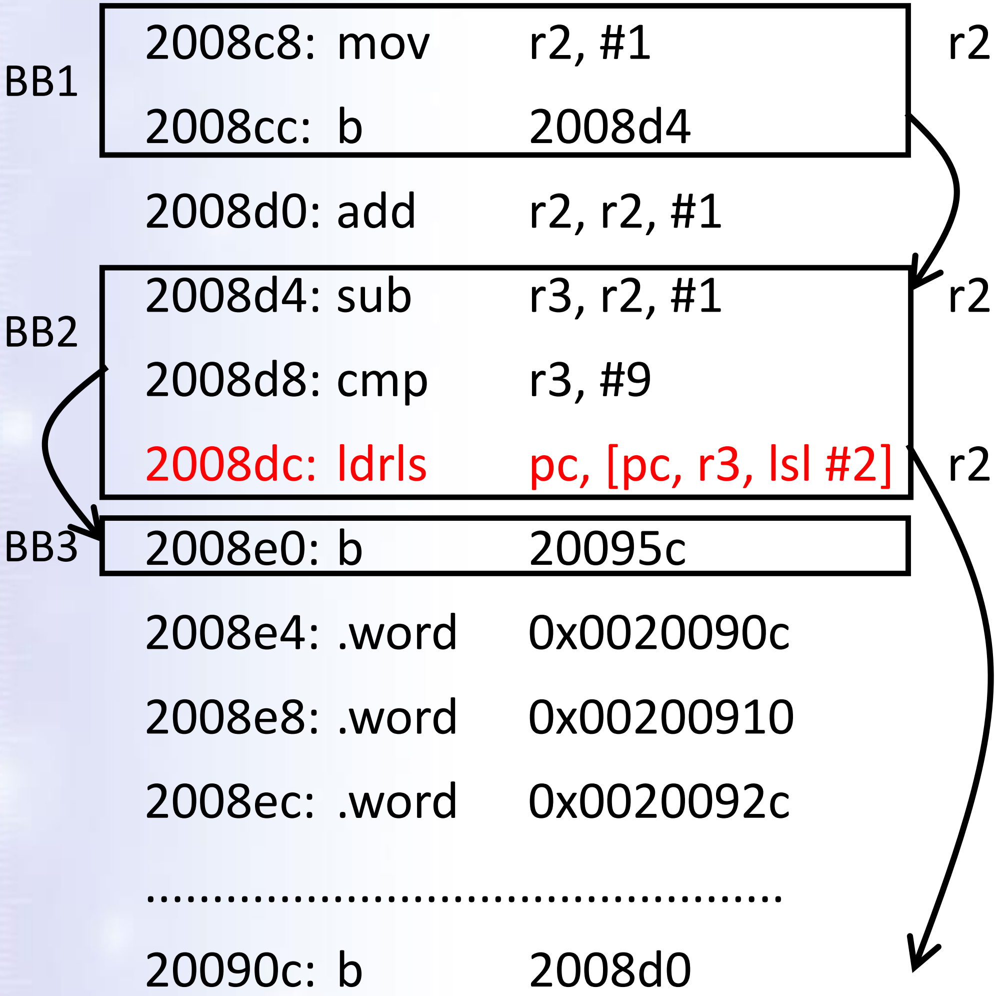


```
BB1  2008c8: mov      r2, #1
     2008cc: b        2008d4
     2008d0: add      r2, r2, #1
BB2  2008d4: sub      r3, r2, #1
     2008d8: cmp      r3, #9
     2008dc: ldrls    pc, [pc, r3, lsl #2]
BB3  2008e0: b        20095c
     2008e4: .word    0x0020090c
     2008e8: .word    0x00200910
     2008ec: .word    0x0020092c
     ..............................................
     20090c: b        2008d0
```

r2 = (1,0,0) // a constant value of 1

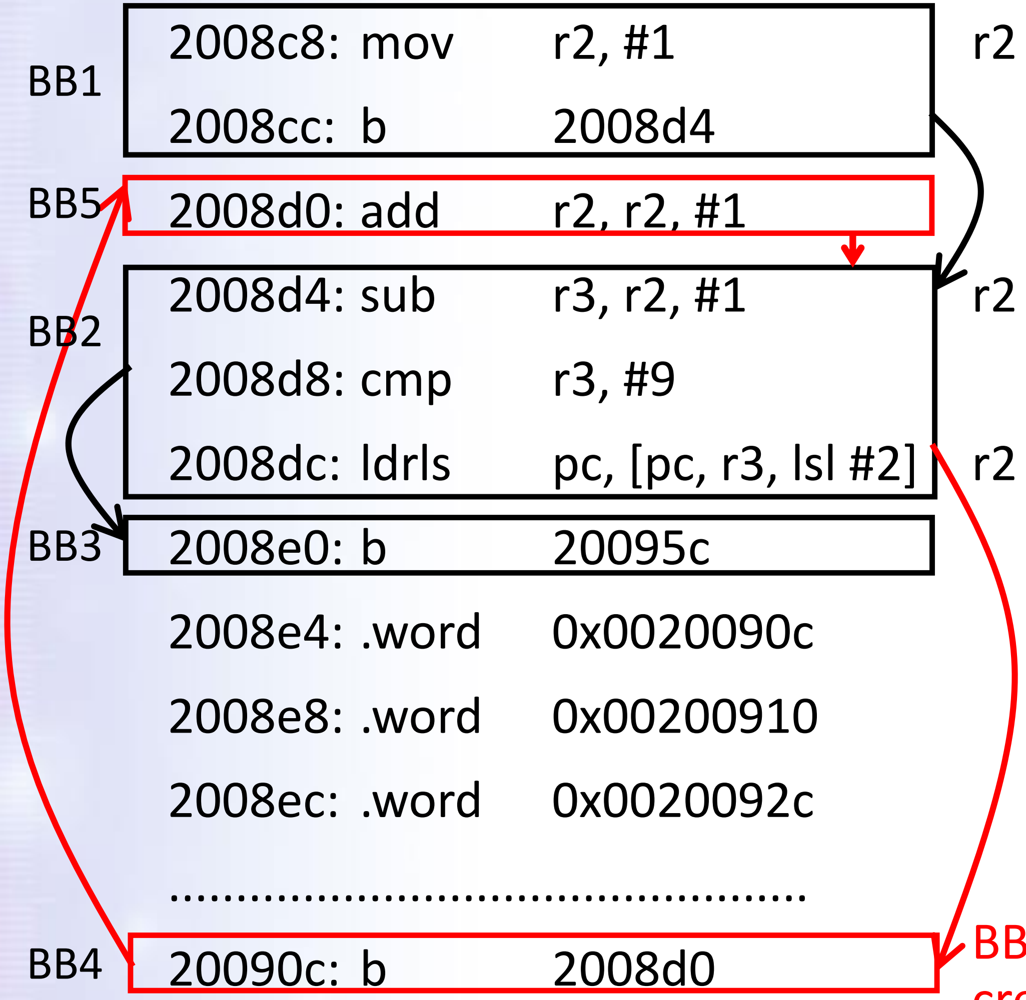
r2 = (1,0,0) r3 = (0,0,0) // introduce r3

CLP analysis with abstract interpretation



```
     2008c8: mov     r2, #1                  r2 = (1,0,0) // a constant value of 1
BB1
     2008cc: b       2008d4
     2008d0: add     r2, r2, #1
     2008d4: sub     r3, r2, #1              r2 = (1,0,0) r3 = (0,0,0) // introduce r3
BB2
     2008d8: cmp     r3, #9
     2008dc: ldrls   pc, [pc, r3, lsl #2]    r2 = (1,0,0) r3 = (0,0,0) pc = (0x2008e4,0,0)
BB3  2008e0: b       20095c
     2008e4: .word   0x0020090c
     2008e8: .word   0x00200910
     2008ec: .word   0x0020092c
     ..............................................
     20090c: b       2008d0
```

Now lets come back to DB



```
2008c8: mov      r2, #1
2008cc: b        2008d4
2008d0: add      r2, r2, #1
2008d4: sub      r3, r2, #1
2008d8: cmp      r3, #9
2008dc: ldrls    pc, [pc, r3, lsl #2]
2008e0: b        20095c
2008e4: .word    0x0020090c
2008e8: .word    0x00200910
2008ec: .word    0x0020092c
..............................................
20090c: b        2008d0
```

r2 = (1,0,0) // a constant value of 1

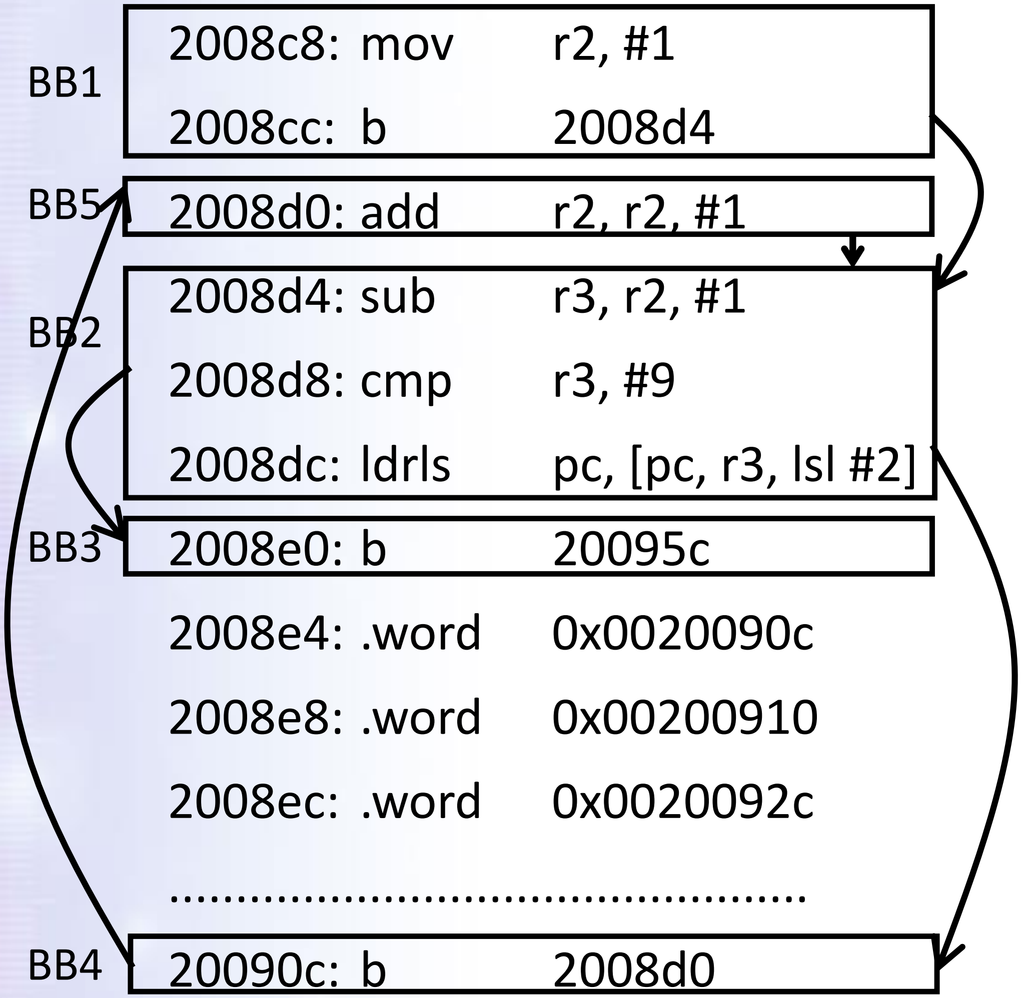
r2 = (1,0,0) r3 = (0,0,0) // introduce r3

r2 = (1,0,0) r3 = (0,0,0) pc = (0x2008e4,0,0)

=> PC will be load from [0x2008e4], which is 0x20090c

BB4 and BB5 are created, edges are created: (BB5, BB2), (BB2, BB4), and (BB4, BB5)

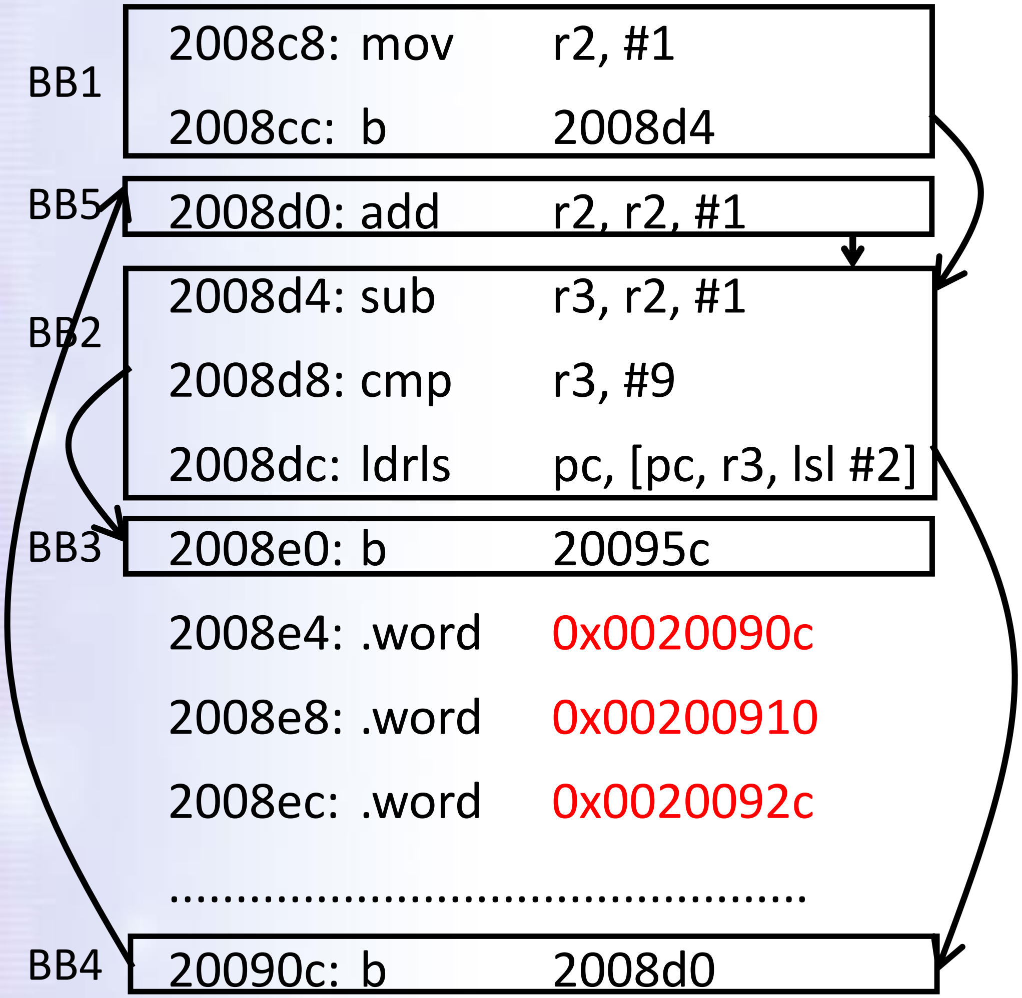
Re-new CLP because CFG changed



```
BB1  2008c8: mov     r2, #1
     2008cc: b       2008d4
BB5  2008d0: add     r2, r2, #1
BB2  2008d4: sub     r3, r2, #1
     2008d8: cmp     r3, #9
     2008dc: ldrls   pc, [pc, r3, lsl #2]
BB3  2008e0: b       20095c
     2008e4: .word   0x0020090c
     2008e8: .word   0x00200910
     2008ec: .word   0x0020092c
     ...............................................
BB4  20090c: b       2008d0
```

- Note that we have a loop
  - BB5->BB2->BB4->BB5
  - Widening is performed
    - r2 and r3 covers a lot of values
  - More targets are explored

The problem of CLP

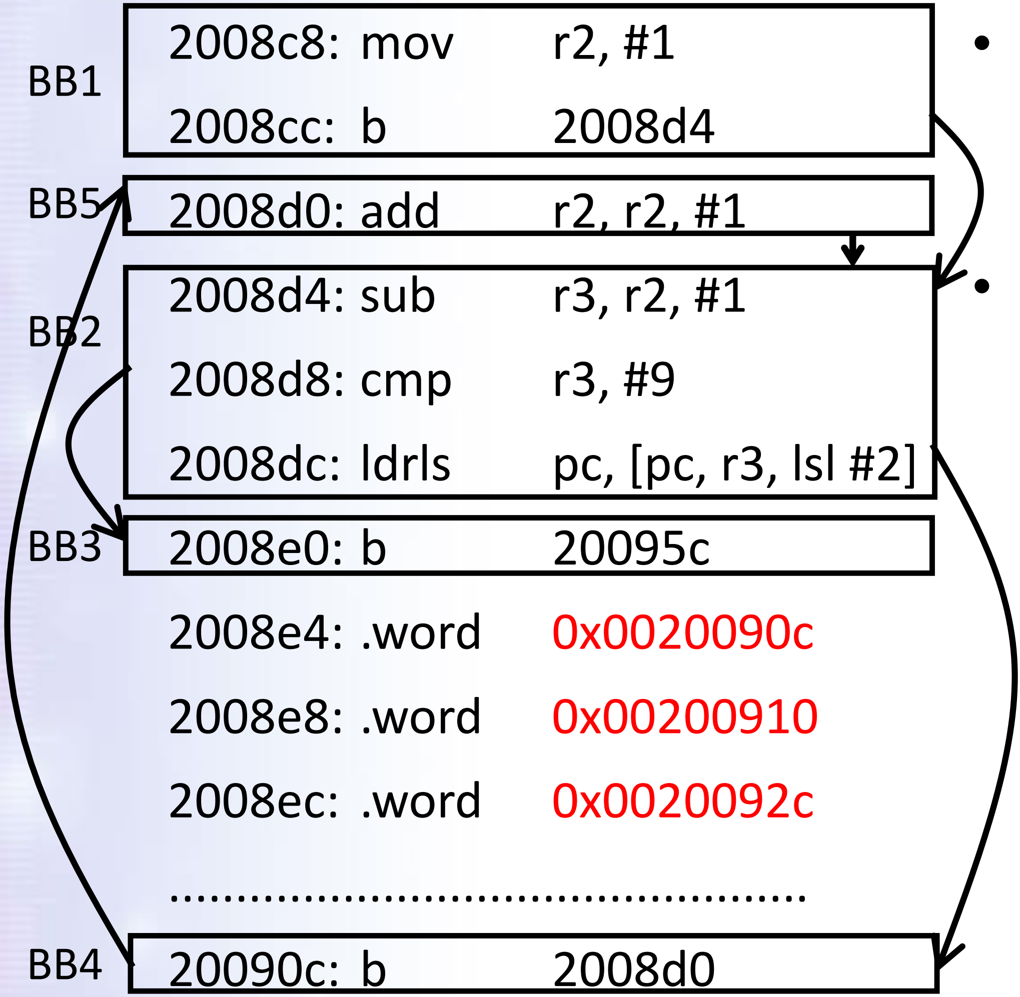


```
BB1  2008c8: mov     r2, #1
     2008cc: b       2008d4
BB5  2008d0: add     r2, r2, #1
     2008d4: sub     r3, r2, #1
BB2  2008d8: cmp     r3, #9
     2008dc: ldrls   pc, [pc, r3, lsl #2]
BB3  2008e0: b       20095c
     2008e4: .word   0x0020090c
     2008e8: .word   0x00200910
     2008ec: .word   0x0020092c
     ………………………………………….
BB4  20090c: b       2008d0
```

- Note that we have a loop
  - BB5->BB2->BB4->BB5
  - Widening is performed
    - r2 and r3 covers a lot of values
  - More targets are explored

- Because we are in CLP, the value for address [pc, r3, lsl #2] will be:
  - (0x20090c, 4, 8)
  - covers 0x20090c, 0x200910, 0x200914, 0x200918, 0x20091c……, 0x20092c ….
  - Leads to create non-existent BBs!

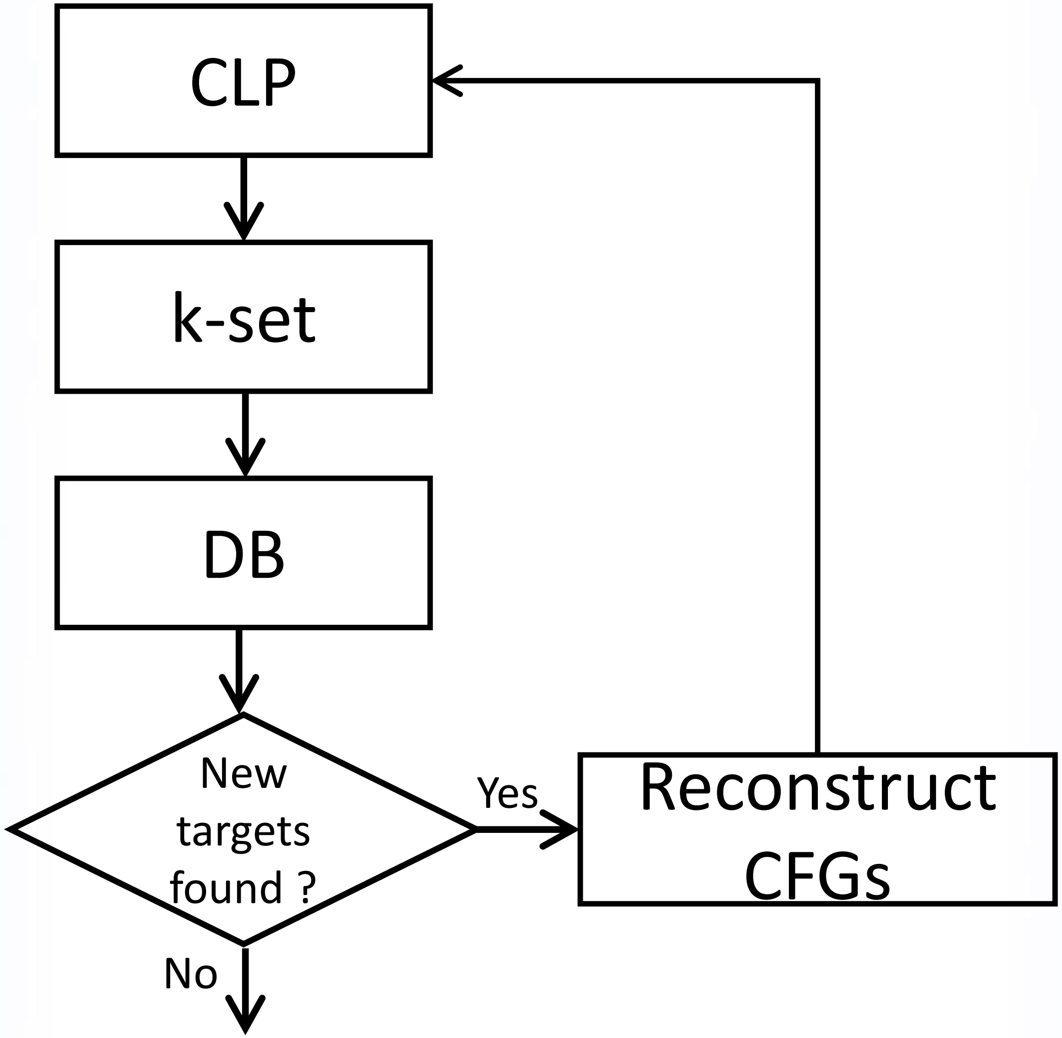
Use k-set to keep the values for DB



```
BB1  2008c8: mov     r2, #1
     2008cc: b       2008d4
BB5  2008d0: add     r2, r2, #1
BB2  2008d4: sub     r3, r2, #1
     2008d8: cmp     r3, #9
     2008dc: ldrls   pc, [pc, r3, lsl #2]
BB3  2008e0: b       20095c
     2008e4: .word   0x0020090c
     2008e8: .word   0x00200910
     2008ec: .word   0x0020092c
     ..............................................
BB4  20090c: b       2008d0
```

- The problem of CLP can propagate and influence a lot more
- We apply a simpler k-set analysis
  - With abstract interpretation too
  - Coarse grain than CLP
  - Now the address [pc, r3, lsl #2] is: {0x20090c, 0x200910, 0x20092c}

Recap: CLP + k-set + DB



CLP

k-set

DB

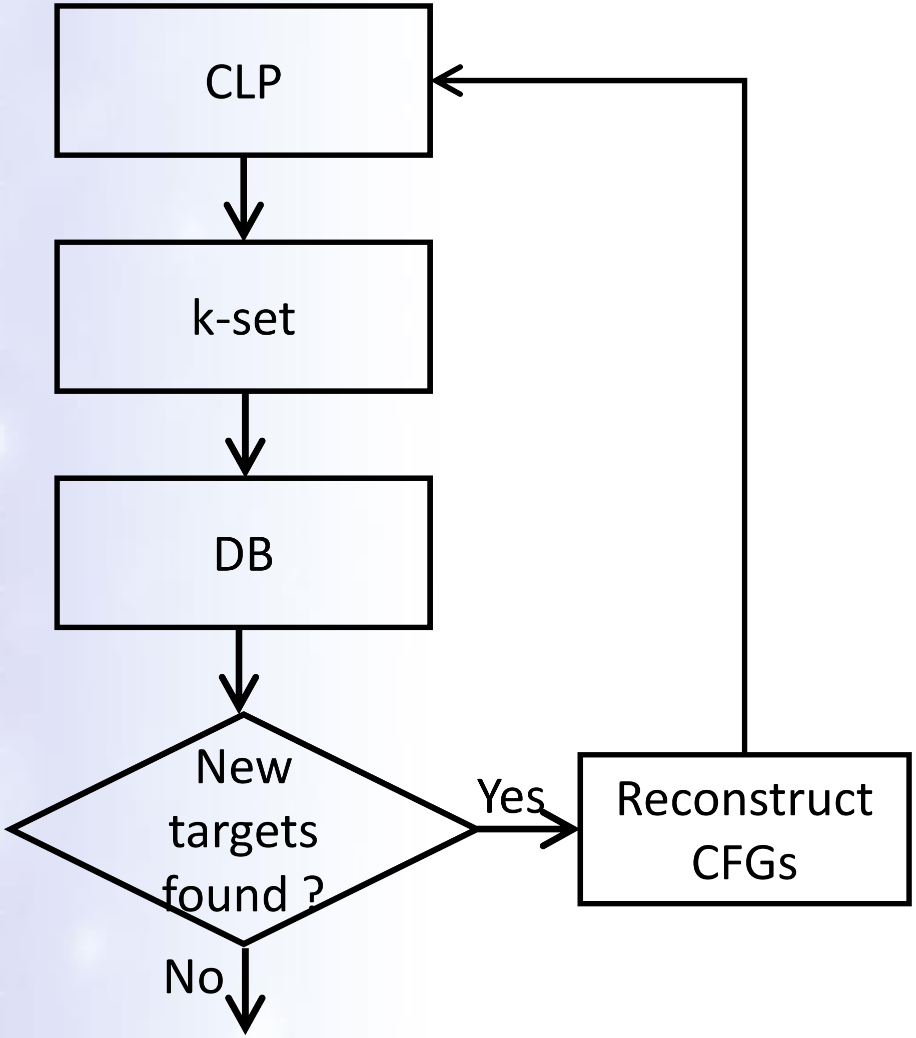
New targets found ?

Yes

Reconstruct CFGs

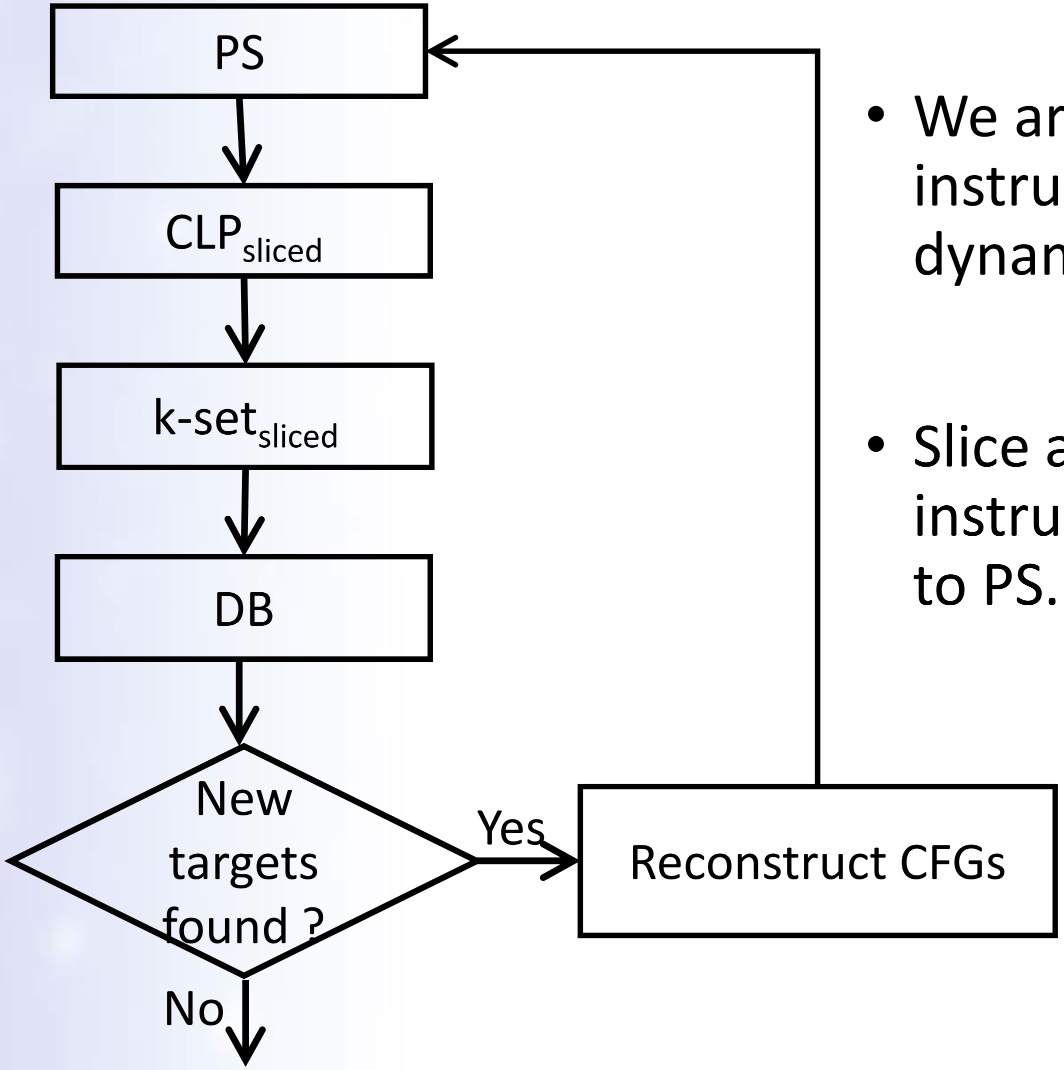
No

A lot of time are spending on CLP and k-set



- Because DB is simple, most of the are spent in CLP and k-set.
- As new paths are found, CFGs grow.
- We only care about finding the new paths, hence only need to apply CLP and k-set on necessary parts => use program slicing.

CLP + k-set + DB + PS (program slicing)



- We are interested in the instructions which influence the dynamic branching
- Slice away all the other instructions also empty BBs due to PS.

Put programming slicing in place

- Many works and many flavours [10][11] ….

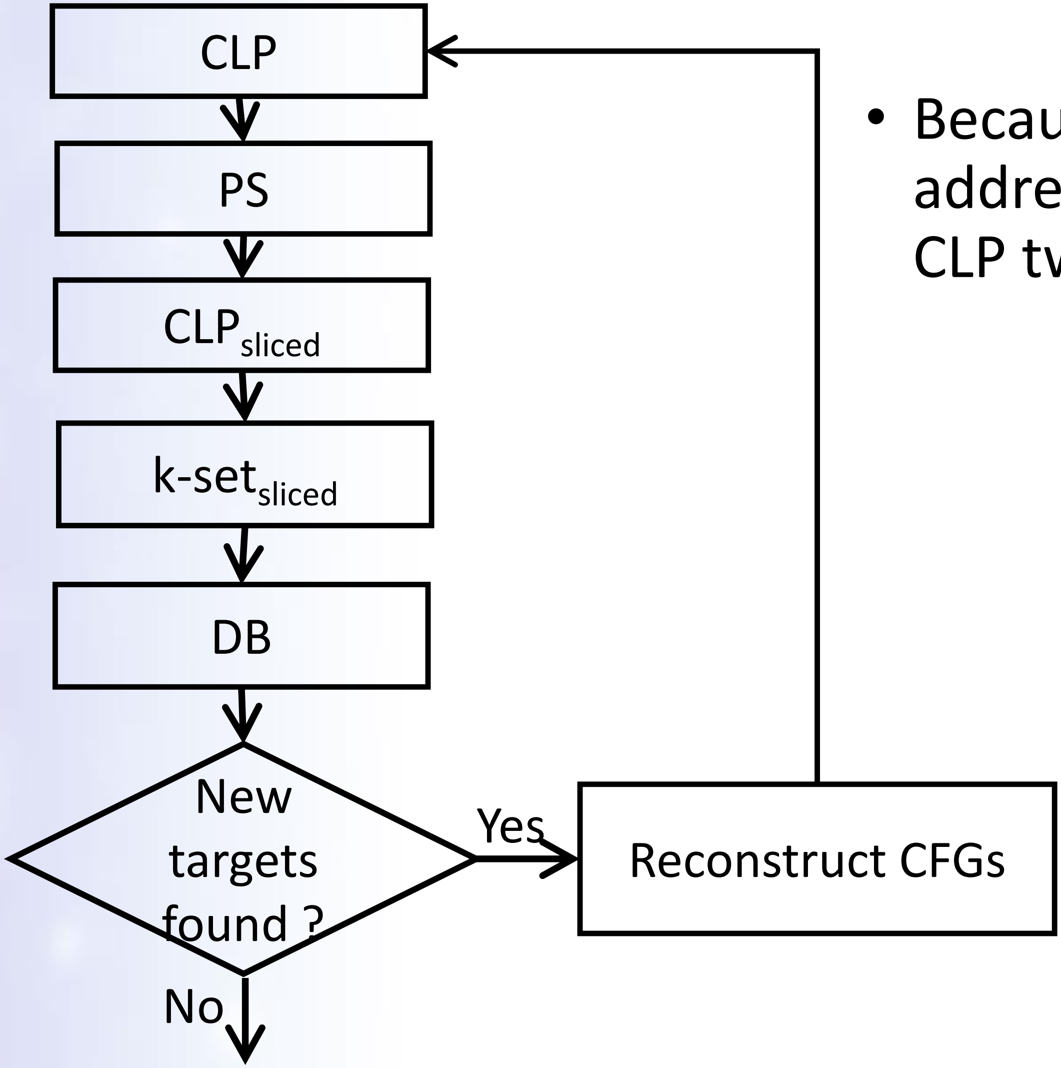
- Program slicing decision: useful memory addresses and registers
  - Register – simple, because:
    - the # is fixed.
    - encoded in the instruction
  - Memory – need address analysis
    - Needs to go through the whole program again
    - Address analysis is provided by CLP

---

[10] M. Weiser. Program slicing. In Proceedings of the 5th international conference on Software engineering, pages 439–449. IEEE Press, 1981.

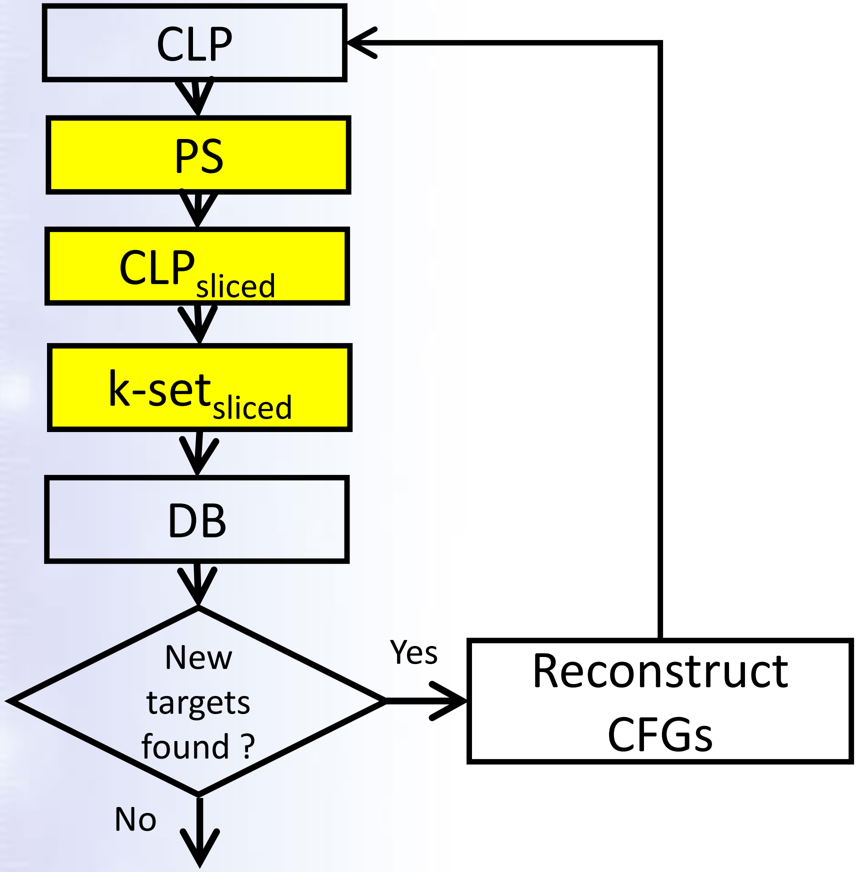
[11] C. Sandberg, A. Ermedahl, J. Gustafsson, and B. Lisper. Faster WCET flow analysis by program slicing. In Proceedings of the 2006 ACM SIGPLAN/SIGBED, pages 103–112.ACM, 2006.

What’s really happening



- Because we need CLP as the address analysis, we are applying CLP twice in the flow.

Comparing with the approaches



CLP

PS

CLP$_{\text{sliced}}$

k-set$_{\text{sliced}}$

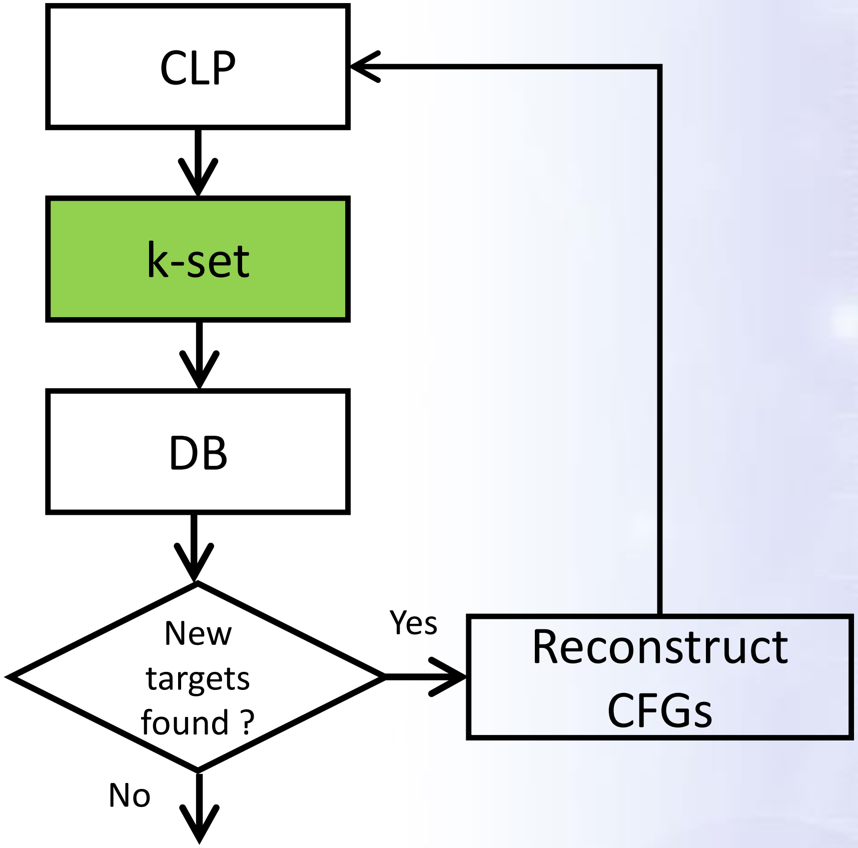
DB

New targets found ?

Yes

Reconstruct CFGs

No



CLP

k-set

DB

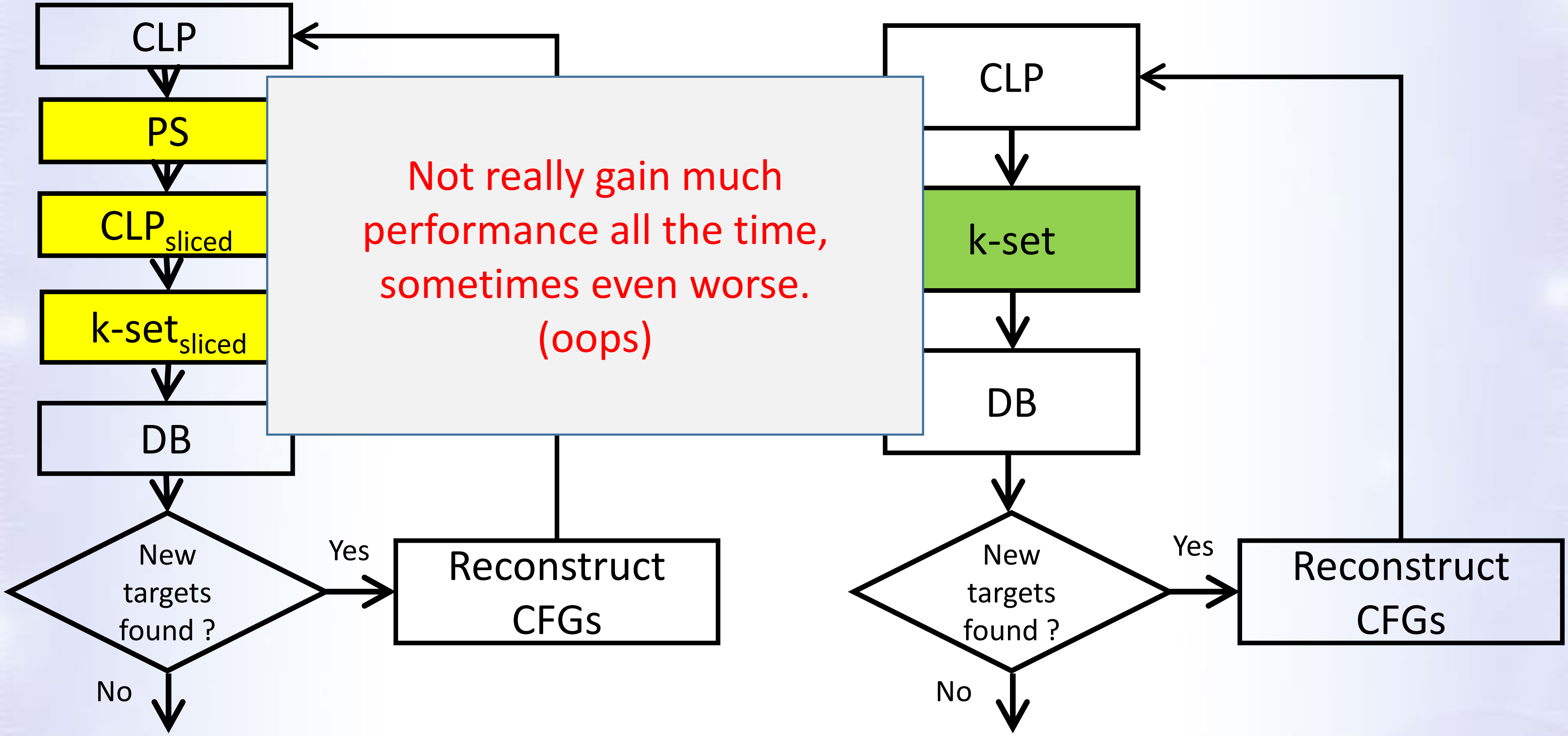
New targets found ?

Yes

Reconstruct CFGs

No

Comparing with the approaches

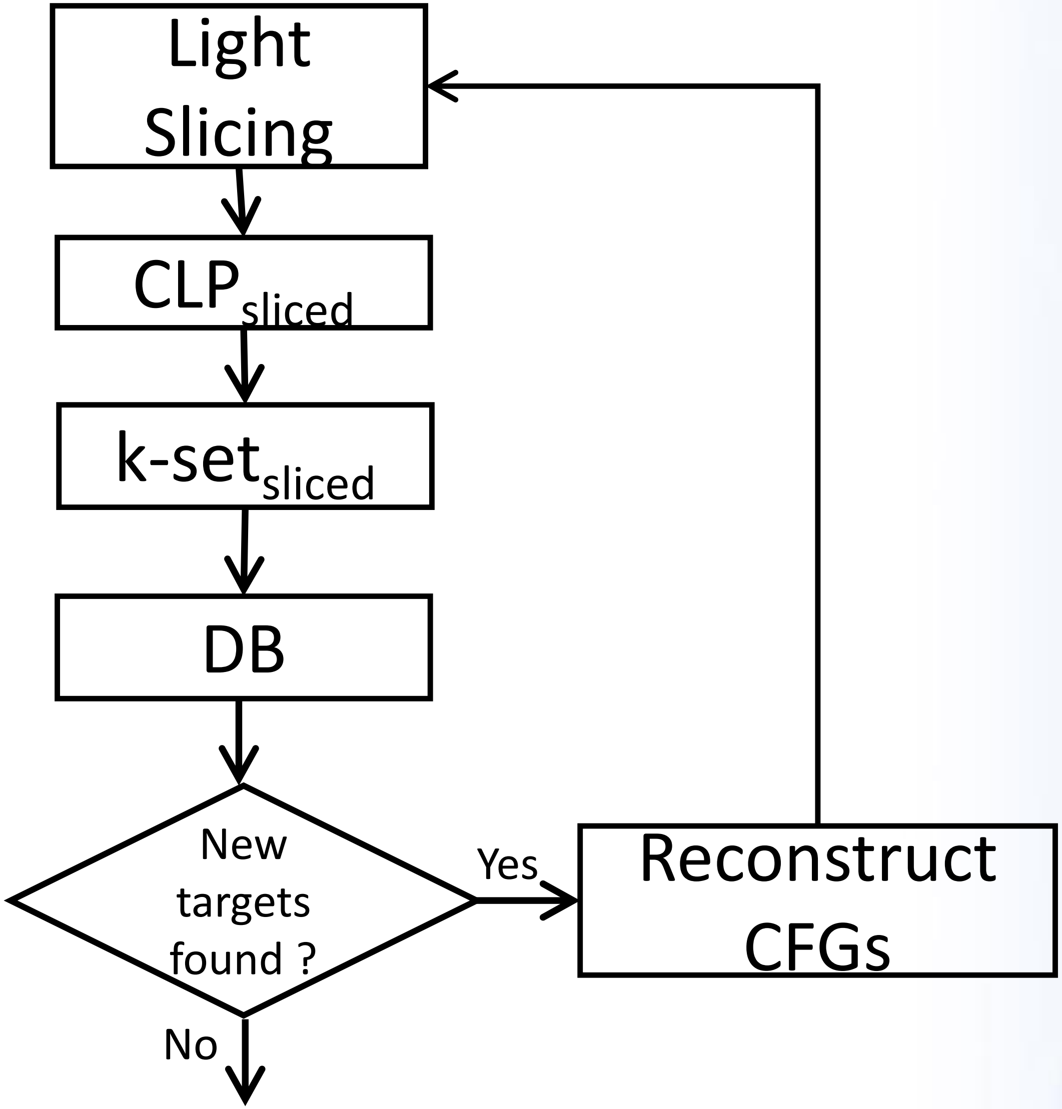


Not really gain much performance all the time, sometimes even worse. (oops)

Light slicing

- Address analysis is not used
- Consider the whole memory space as a single register
- To be safe, we keep all the instructions which write to the memory
- Only keep the memory loading instruction when the target register is of interest

CLP + k-set + DB + Light Slicing



Light Slicing

$\text{CLP}_{\text{sliced}}$

$\text{k-set}_{\text{sliced}}$

DB

New targets found ?

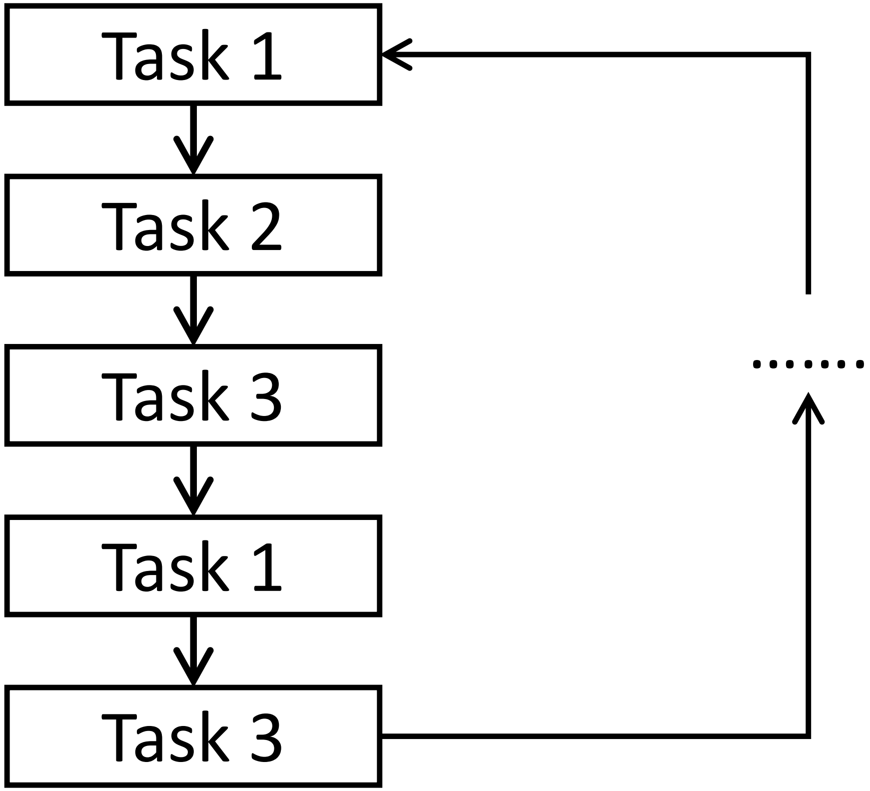
Yes

No

Reconstruct CFGs

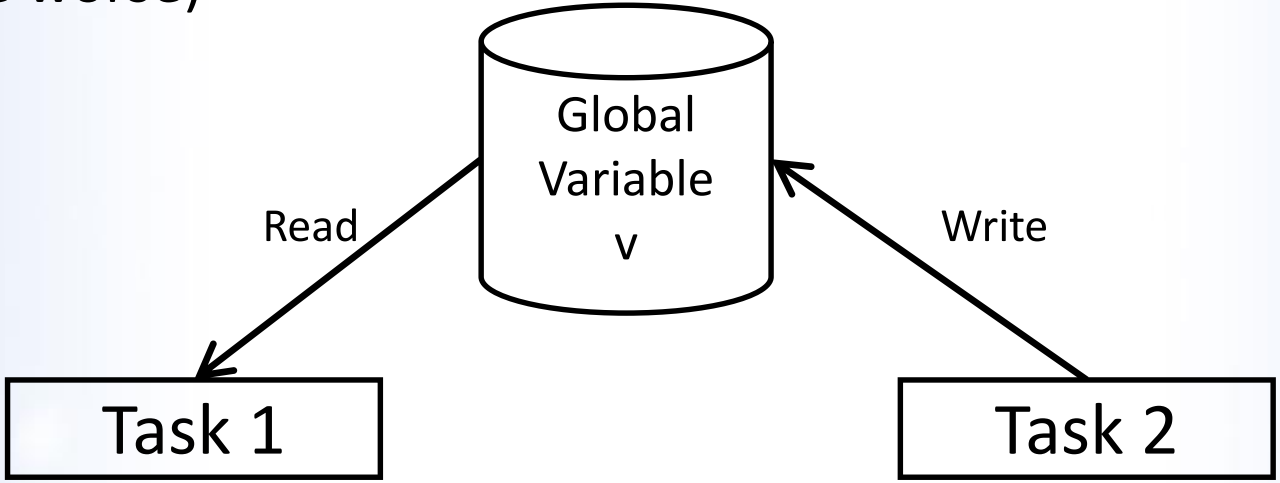
We need to have performance gain

- For large applications
  - consists of multiple tasks (functions)
  - tasks are called in loops



Why not perform analysis on individual tasks?

- Yes, only if the tasks are independent.
- But sometimes they communicate
  - Through shared variable (global variable)
  - Such variable could also be the function pointer
  - Analysis can not make assumption on these variables.
  - e.g. the value of v is depending on Task 2, making assumption on v (e.g. T) will leads to inaccuracy. (The more communication, the worse)

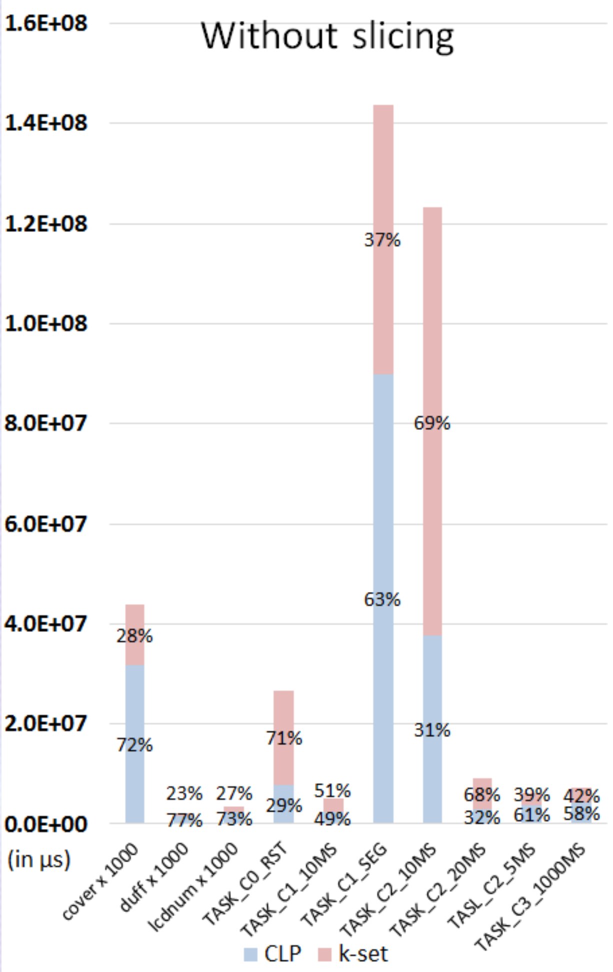


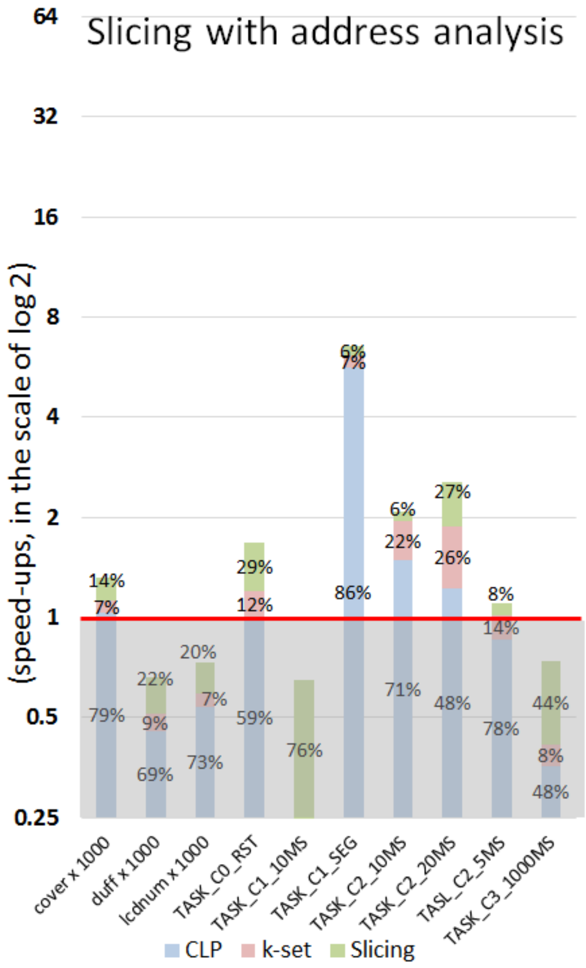
Experiments

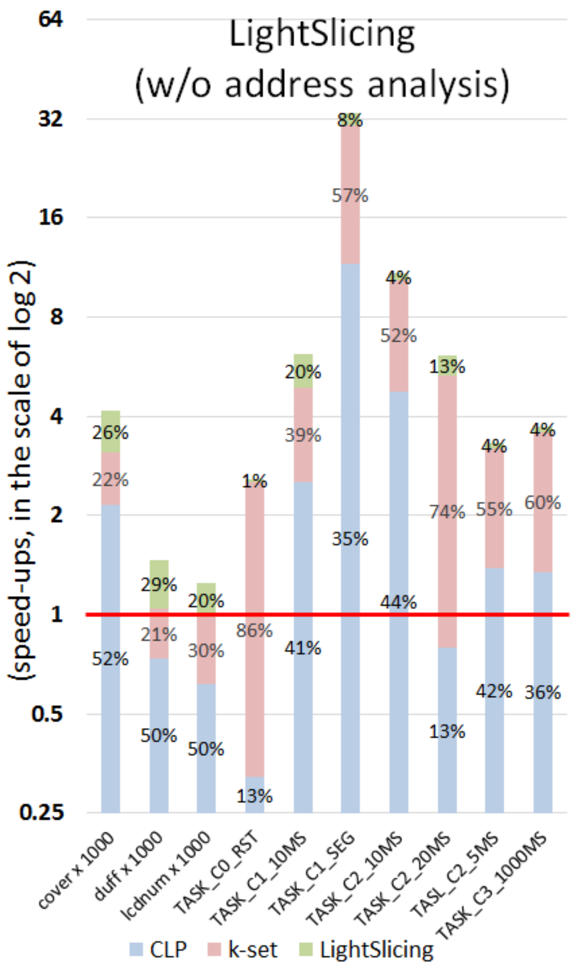
- On Mälardalen benchmark
  - duff, cover, lcdnum

- Realistic application
  - From Continental SAS France
  - Multi-task engine control software
  - 172,985 instructions, 2493 functions, 212,620 lines of C codes

Experiments







Results

- For more complex scenario, CLP takes more time
    - Conventional slicing does not save much time

- Light Slicing helps to obtain more speed-ups
    - 2 times+ faster (up to 33 times) in larger application

- All the dynamic branches from Mälardalen are solved
    - 92% for the industrial example
    - Due to irreducible loops not handled well by the framework (on-going work)

Conclusions

- Combine multiple analyses to achieve dynamic branching analysis
- Speed-ups from Light Slicing
- Works reasonable well for large and realistic applications
- Incremental computation on analysis
  - Since majority part of the program does not change
  - Re-use the state computed previously

Questions?

- Thank you ☺