Profiling Data-Dependence to Assist Parallelization: Framework, Scope, and Optimization

Alain Ketterlin **Philippe Clauss**

Motivation

- Data dependence is central for:
  - parallelization
  - locality optimization
  - ...
- Compilers have limited capabilities
  - aliasing
  - fine grain
- Parwiz: an empirical approach
  - uses dynamic information
  - targeting fine or coarse-grain parallelism
  - includes several decision/parallelization algorithms
  - leaves final validation to the programmer

Framework > Core notions

Data-dependence

- For every access to address a
  - What was the previous access to a?
- A *shadow memory* tracks last accesses

Program structures

- Program execution is a hierarchy of calls and loops
- Correlate accesses (and dependencies) with calls and loops
- An *execution point* uniquely locates every access

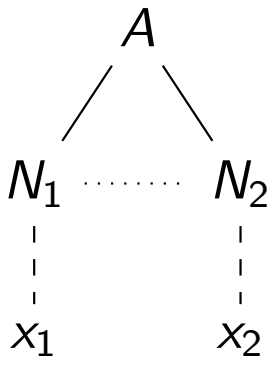
$$\texttt{call} \xleftarrow[p_0]{} \texttt{loop} \xleftarrow{i_0} \texttt{iter} \xleftarrow[p_1]{} \texttt{call} \xleftarrow[p_3]{} \texttt{loop} \xleftarrow{i_1} \texttt{iter} \xleftarrow[p_2]{} \texttt{access}$$

(carries a generalized iteration vector)

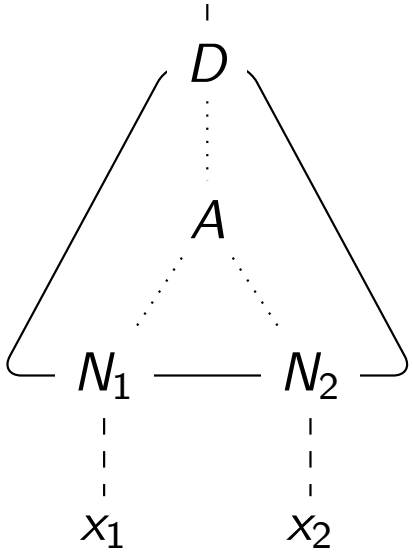
Framework > Dependence domains (1)

- An *execution tree* keeps "all" execution points (a dynamic call tree, plus nodes for loops and iterations)

- A dependence is carried by the lowest common ancestor on both execution points

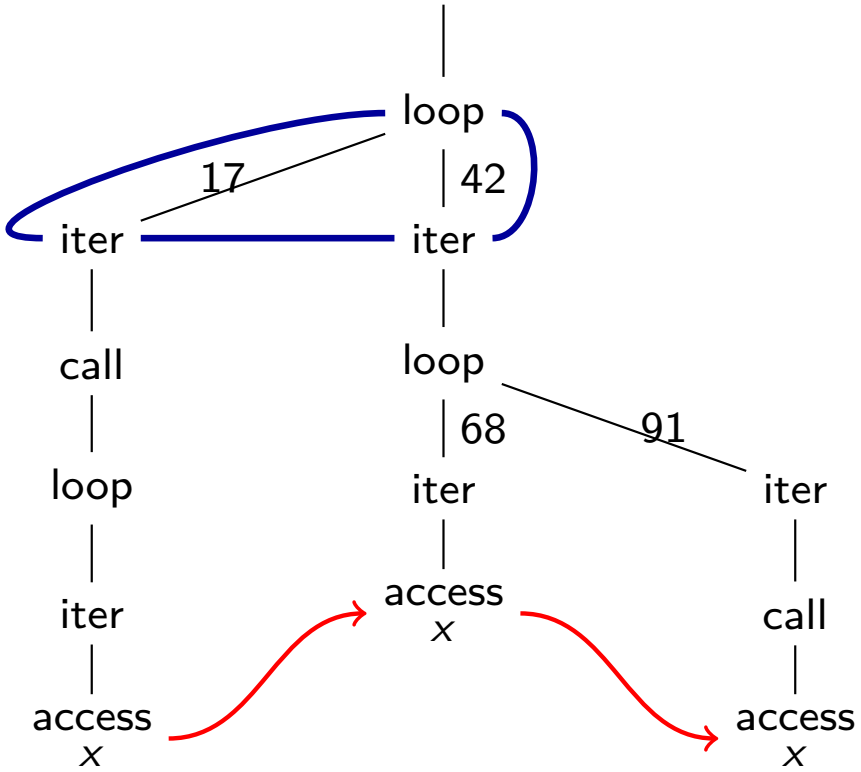


- A *dependence domain* may span several levels of the tree



Framework > Dependence domains (2)

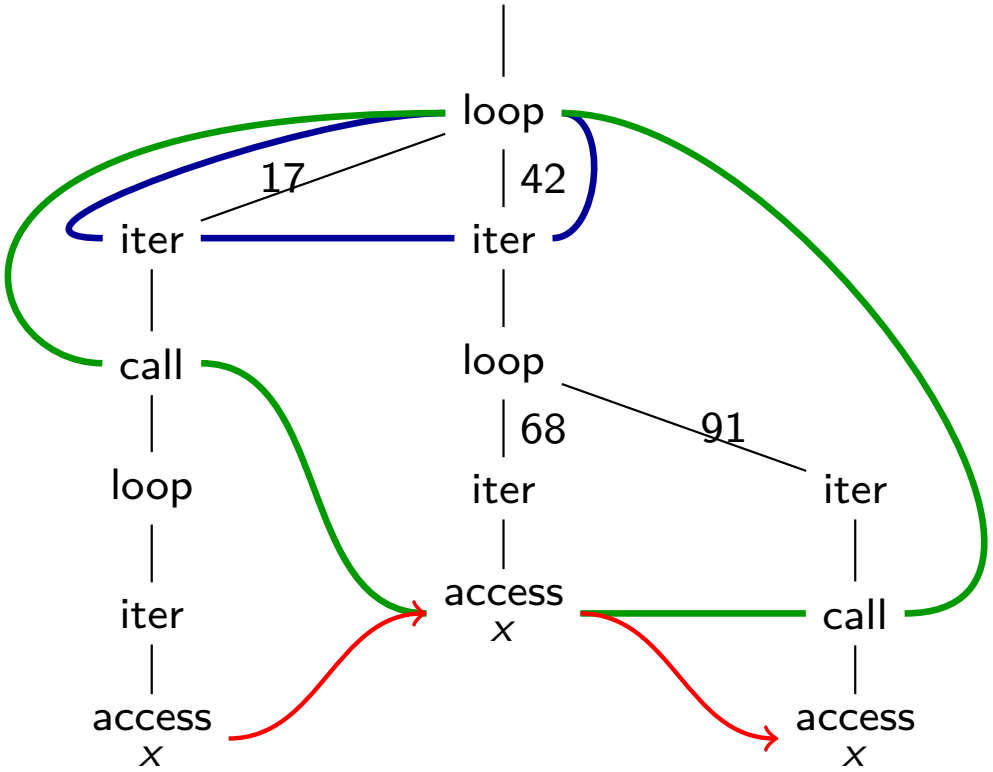
- Example:



- (17)–(42)

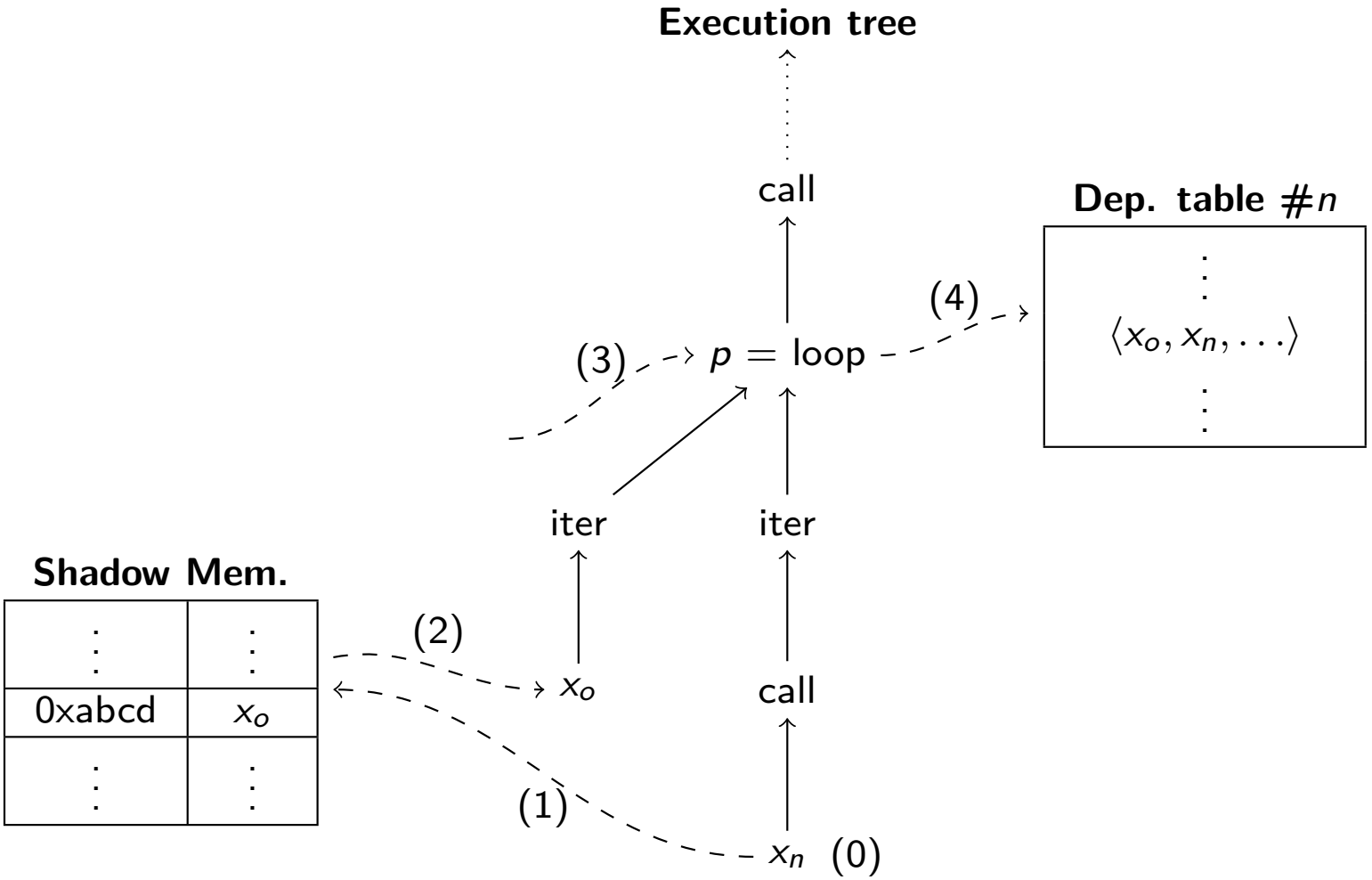
Framework > Dependence domains (2)

- Example:



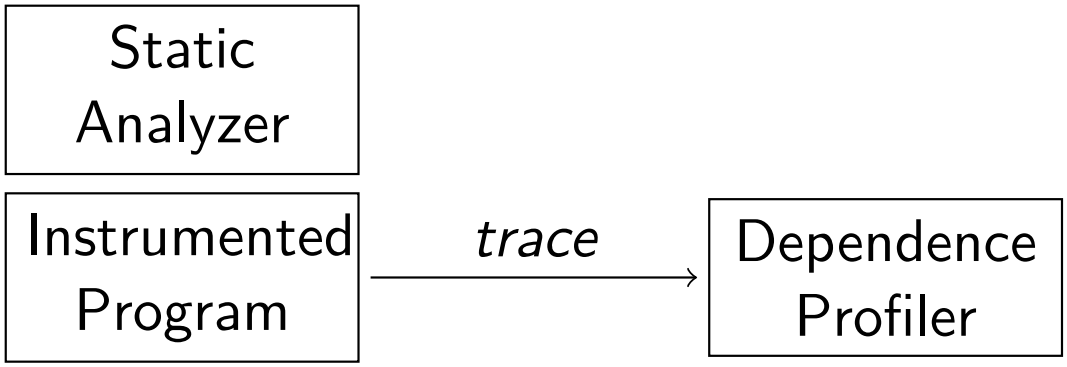
- (17)–(42)
- $(17, 0) - (42, 68)$ and $(42, 68) - (42, 91)$

Framework > Algorithm: Parwiz



Framework > Implementation

- Tool architecture



- Static analyzer: computes CFG and loop hierarchies
- Instrumentation
  - function call/return
  - loop entry/iteration/exit
  - memory accesses
- Works from x86_64 code, requires no compiler support
- Instrumentation/tracing done with Pin

Applications > Loop parallelism (1)

- all loops from the SPEC OMP-2001 programs

Program	Executed	
	#Loops	#Par.
312.swim_m	26	25
314.mgrid_m	58	52
316.applu_m	168	135
318.galgel_m	541	455
320.equake_m	73	67
324.apsi_m	191	147
326.gafort_m	58	43
328.fma3d_m	233	192
330.art_m	79	65
332.ammp_m	76	48

Applications > Loop parallelism (1)

- all loops from the SPEC OMP-2001 programs

Program	Executed		Slowdown/overhead		
	#Loops	#Par.	Trace (×)	Prof. (×)	Mem. (Mb)
312.swim_m	26	25	33	118	2527
314.mgrid_m	58	52	39	147	1376
316.applu_m	168	135	48	148	1082
318.galgel_m	541	455	42	121	1394
320.equake_m	73	67	43	150	723
324.apsi_m	191	147	44	134	4798
326.gafort_m	58	43	35	93	679
328.fma3d_m	233	192	42	99	2223
330.art_m	79	65	34	92	200
332.ammp_m	76	48	37	97	504

- massive slowdown, but an unusual use case

Applications > Loop parallelism (2)

- loops with OpenMP pragmas only

Program	OpenMP-annotated loops			
	#Loops	#Par.	Main cause of failure	#Priv.
312.swim_m	8	7	reduction	7
314.mgrid_m	12	11	reduction	11
316.applu_m	30	17	priv. + reduction	25
318.galgel_m	37	30	priv. required	30
320.equake_m	11	3	priv. required	10
324.apsi_m	28	13	priv. + reduction	27
326.gafort_m	9	7	priv. + reduction	7
328.fma3d_m	29	22	reduction	22
330.art_m	5	4	(non-openmp code)	4
332.ammp_m	7	5	priv. required	7

- #Priv.: WARs ignored (accesses are collected for feedback)
- very good coverage
- recognizing reductions is hard in the general case

Applications > Vectorization (1)

- Allen & Kennedy's *codegen* algorithm
- can distribute and re-order loops

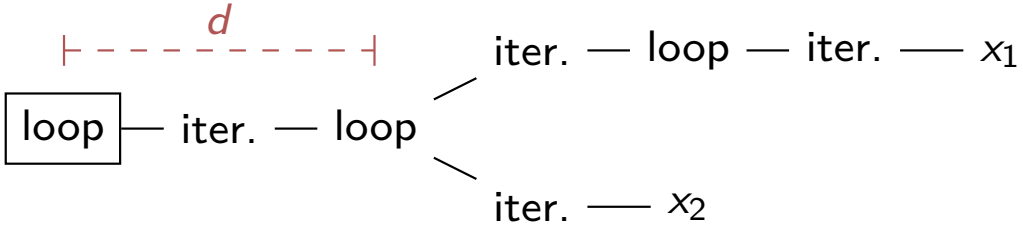
```
void ak(int * X, int * Y,
        int ** A, int * B, int ** C)
{
    for ( int i=1 ; i<=100 ; i++ ) {
S1:     X[i] = Y[i] + 10;
        for ( int j=1 ; j<=100 ; j++ ) {
S2:         B[j] = A[j][N];
            for ( int k=1 ; k<=100 ; k++ )
S3:             A[j+1][k] = B[j] + C[j][k];
S4:         Y[i+j] = A[j+1][N];
        }
    }
}
```

```
for ( i=1 ; i<=100 ; i++ ) {
    for ( j=1 ; j<=100 ; j++ ) {
        B[j] = A[j][N];
        parfor ( k=1 ; k<=100 ; k++ )
            A[j+1][k] = B[j] + C[j][k];
    }
    parfor ( j=1 ; j<=100 ; j++ )
        Y[i+j] = A[j+1][N]
}
parfor ( i=1 ; i<=100 ; i++ )
    X[i] = Y[i] + 10;
```

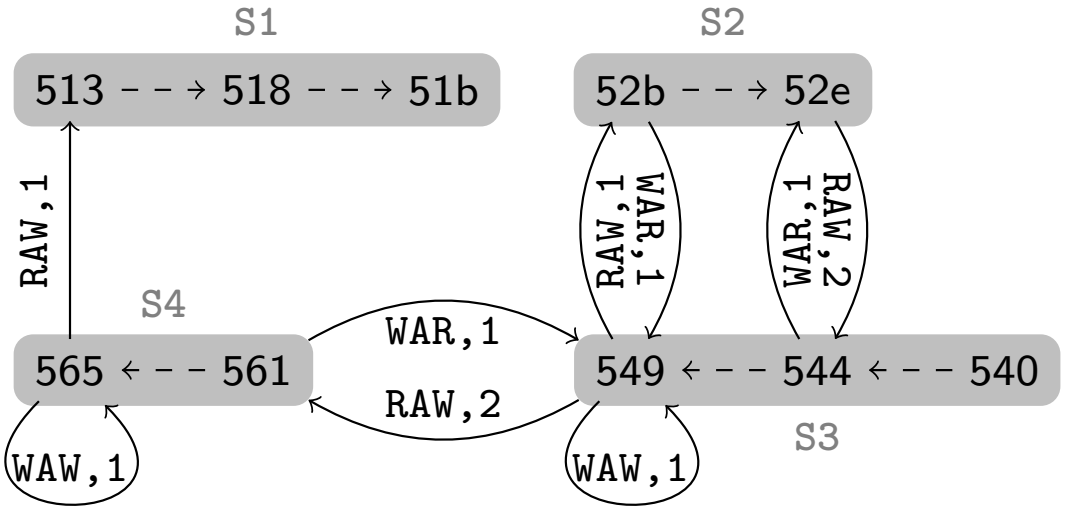
- needs a dependence graph between statements
- with dependence levels

Applications > Vectorization (2)

- Target one specific loop
- Keeps dependence type + level



- Resulting dependence graph:



- Combines memory data-dependencies and register traffic

Applications > Linked data structures

- Typically: are the links modified during the traversal of a list?
- Motivation: inspector/executor, speculative parallelization...
- Idea:
  - select a region of interest (e.g., a loop)
  - select memory loads that read an address
    (can be done conservatively by static slicing)
  - capture all RAW dependencies involving one of these loads
- Yesterday's "*Control-Flow Decoupling*" is based on such a property

+ Bags of tasks (paper), dependence polyhedra for locality optimizations, ...

Optimization > Motivation

- Memory (+ control flow) tracing is expensive
  - instrumentation causes code bloat
  - large volume of data
- Impacts both tracing and profiling
- Sampling does not apply (well)
  - sample memory accesses
  - miss dependencies
  - produces wrong dependencies
- Use static analysis

Optimization > Static analysis of binary code (1)

- Goal: reconstruct address computations
- Static single assignment form (slicing for free)

```
mov eax, 0x603140                rax.8 ⇐
...
sub r13, 0xedb                   r13.7 ⇐ r13.6
...
———                              rsi.9 = φ(rsi.8, rsi.10)
...
lea r11d, [rsi+0x1]              r11.6 ⇐ rsi.9
movsxd r10, r11d                 r10.9 ⇐ r11.6
lea rdx, [r10+r13*1]             rdx.15 ⇐ (r10.9,r13.7)
...
lea r9, [rdx+0x...]              r9.9 ⇐ rdx.15
...
movsd xmm0, [rax+r9*8]           xmm0.6 ⇐ (M.22,rax.8,r9.9)
...                              0xe28d4b0 + 8*rsi.9 + ....
```

→ derive symbolic expressions

Optimization > Static analysis of binary code (2)

- Scalar evolution (introduces normalized loop counters I, ...)

```
0x406ad2 mov r13.8, qword ptr[...] ; value unknown
         ...
0x406afd r11.93 = phi(...)         ; value unknown
         ...
0x406b05 mov rdi.97, r11.93        ; = r11.93
         ...
0x406b10 rdi.98  = phi(rdi.97,rdi.99)
               ; = r11.93 + I*r13.8
         ...
0x406b41 add rdi.99/.98, r13.8     ; = rdi.98 + r13.8
         ...
0x406b4a j... 0x406b10
```

- Branch conditions are also parsed (when possible)
  - loop trip-counts

Optimization > Memory access coalescing (1)

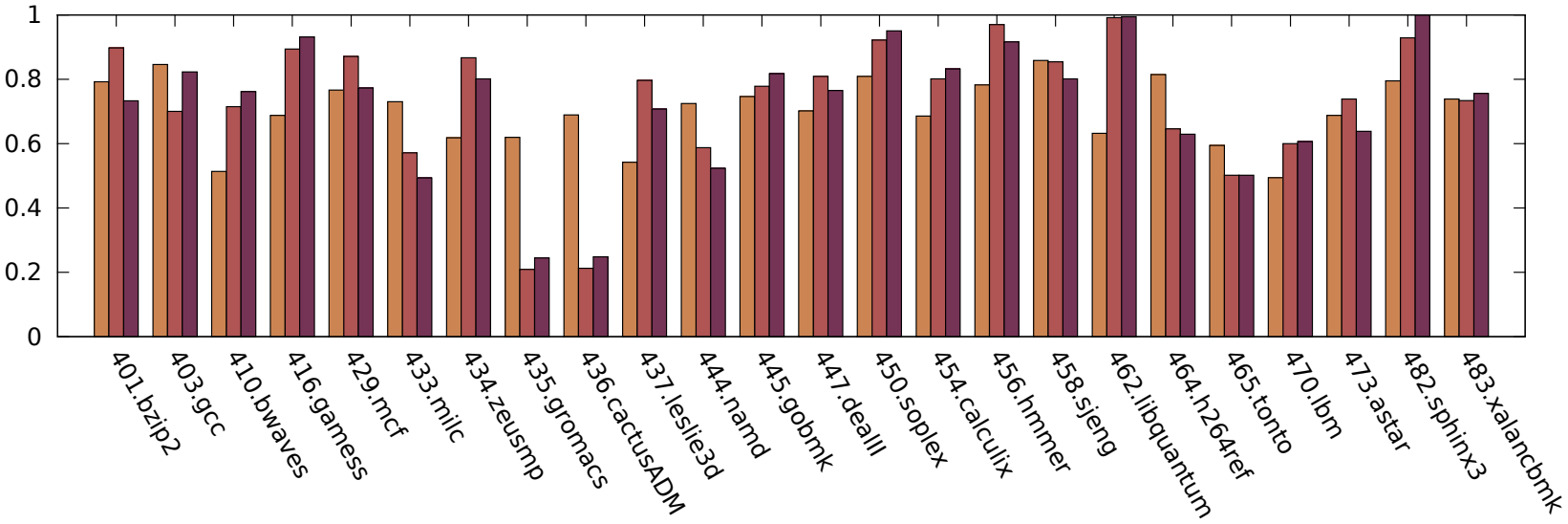
- Look for accesses to contiguous addresses
  - structure fields
  - unrolling
  - ...
- Inside a basic block only
- Use address expressions

```
mov rdx, qword ptr [r13+rdx*8]
; → [-0x10 + r13_7 + 8*rax_29 - 8*I]
...
mov rax, qword ptr [r13+rax*8]
; → [-0x8 + r13_7 + 8*rax_29 - 8*I]
```

- A single instrumentation point

Optimization > Memory access coalescing (2)

- 3 quantities to consider
  1. static amount of instrumentation points
  2. number of dynamic events
  3. run time
- SPEC 2006, train (tracing only)
- All quantities normalized to the unoptimized case:
- static dynamic runtime



Optimization > Parametric loop nests (1)

- Extract static control loops: accesses and control involve
  - loop invariant parameters
  - counters
- Example (436.CactusADM, `bench_staggeredleapfrog`)

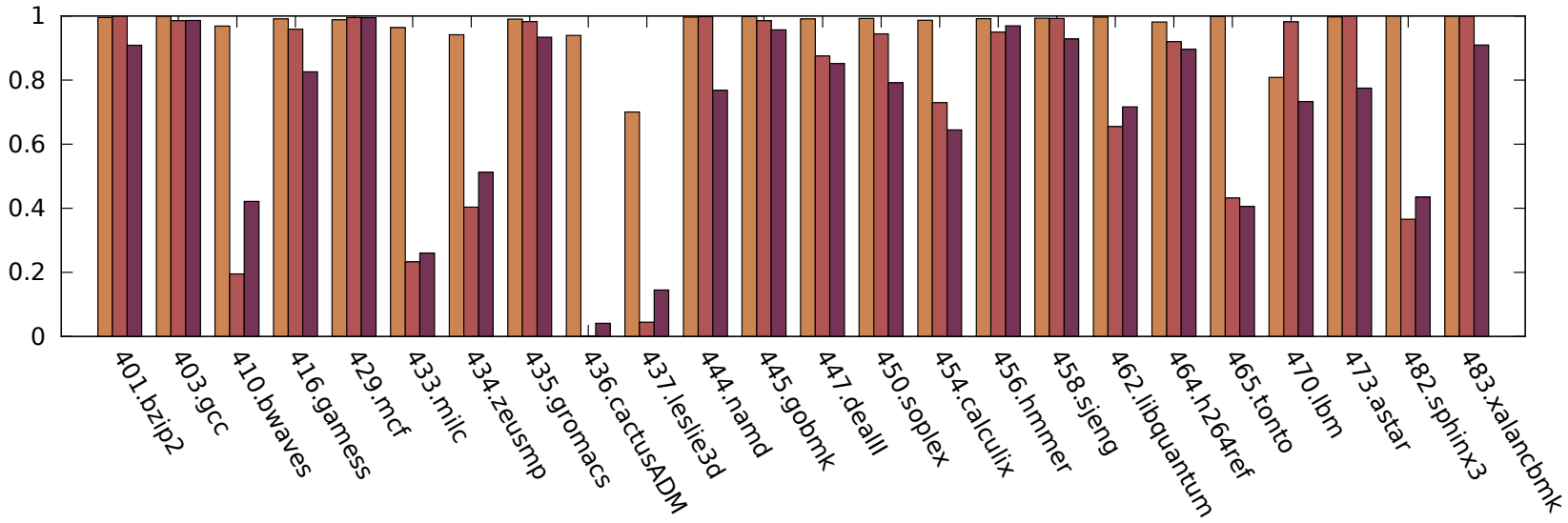
```
void 0x406b10_1(reg_t r15_58, reg_t r9_81, reg_t r11_93,  reg_t rbp_2,
                reg_t r14_7,  reg_t r13_8, reg_t rsi_214, reg_t r10_94)
{
    for ( reg_t I=0 ; (-0x1 + r9_81 + -I >= 0) ; I++ ) {
        if ( (rbp_2 > 0) ) {
            for ( reg_t J=0 ; (-0x1 + rbp_2 + -J >= 0) ; J++ ) {
                ACCESS('R', 8, r15_58 + 8*r11_93 + 8*J + 8*r13_8*I);
                ACCESS('W', 8, r14_7  + 8*r10_94 + 8*J + 8*rsi_214*I);
    }}}
}
```

- 8 loop-invariant parameters → instrumented
- no instrumentation on the loop

Optimization > Parametric loop nests (2)

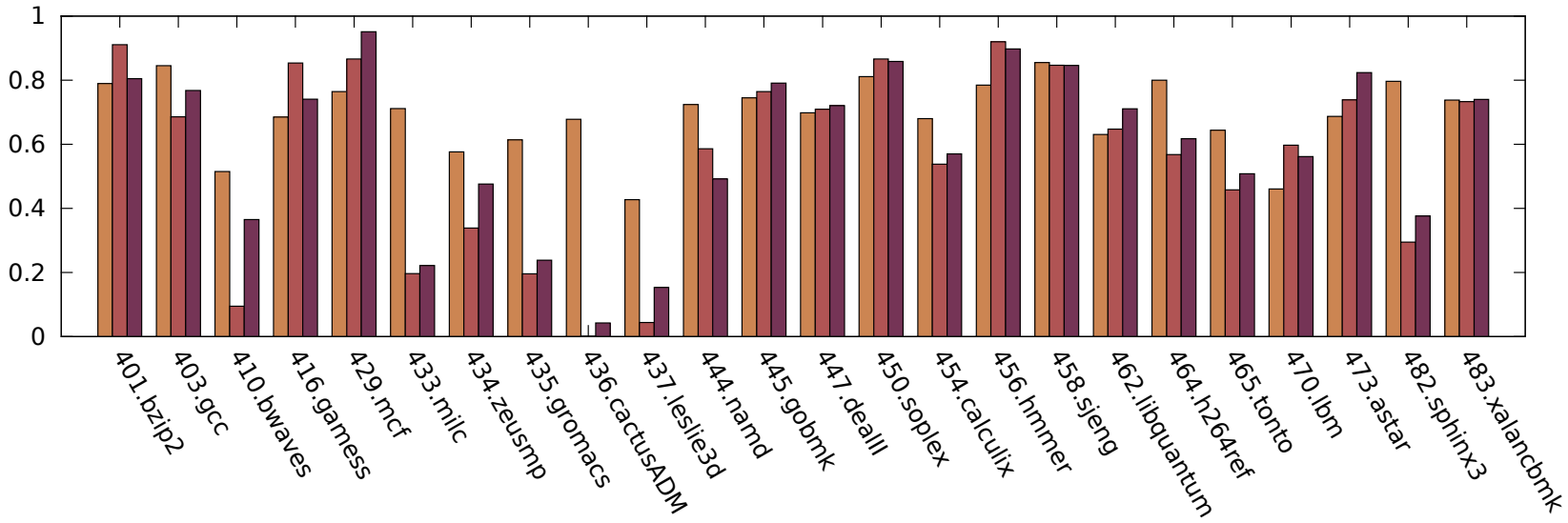
- the loop is compiled and linked to the profiler: 2 cases
  - the loop has an analytical footprint
  - the profiler is responsible for reproducing dependencies

- static dynamic runtime



Optimization > Overall

- Both optimizations accumulate nicely
- Reduce run time by ≈ 35%
- static dynamic runtime



Conclusion

- A general framework
  - user-selectable dependence domains
  - several decision/parallelization strategies
- A useful tool for (targeted) studies
- Tracing optimizations
  - rely on static analysis
  - applicable to any tracing task