Dynamic Binary Optimization: The Dynamo Case

Outline

- Overview of the Dynamo System
- Dynamic Optimizations in Dynamo
- Performance of Dynamo
- Conclusion and Opinion

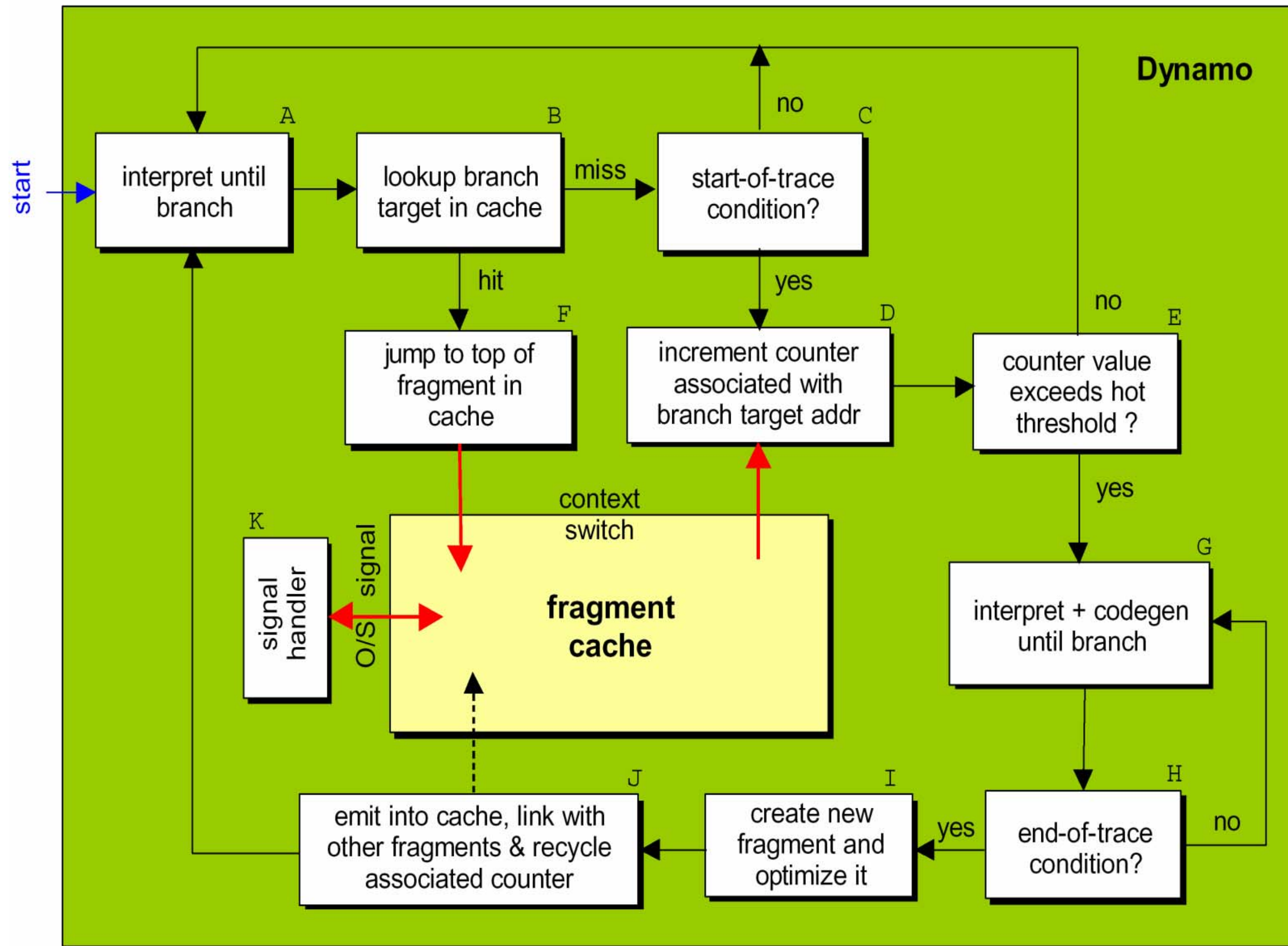
Goals and Motivations

- A dynamic software optimization system
  - Based on HP PA-RISC
- Complement static compiler optimizations
  - Input a binary compiled by –O2 and enhance its performance to the one of more like –O4
- A transparent system
- One point to note
  - Are there enough “dynamic” optimizations that would not be possible in static optimizations?

Overview of Dynamo

- Dynamo first interprets the application
- Watches for a "hot" trace
  - A stream of consecutively executed instructions
  - Can cross static program boundaries (functions or shared libraries)
- Optimize the "hot" trace
- Produce optimized code fragments
- Cache the fragment and jump to it during interpretation when it is met
- Watches the overhead and bails when necessary

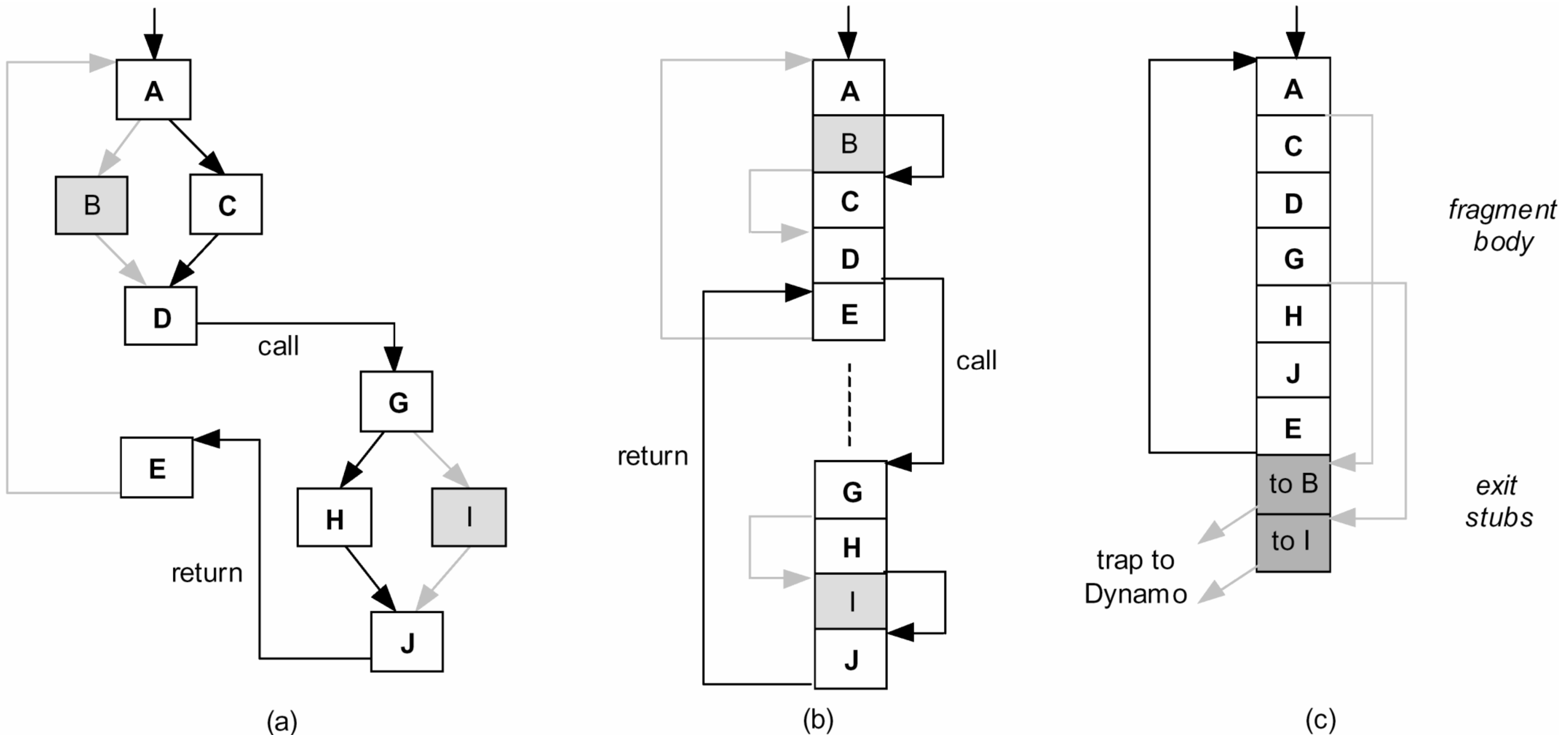
How Dynamo works



Trace Information

- A dynamic sequence of consecutively executed instructions
- Start-of-trace (SoT) candidate
  - Target of a backward-taken branch (i.e., a loop entry)
  - Target of a fragment cache exit branch
  - Dynamo interpreter associates a counter with each and increment when it is executed until determined to be hot
- End-of-trace (EoT) candidate
  - Backward taken branch
  - Taken branch to a fragment entry point

Trace Selection and Formation

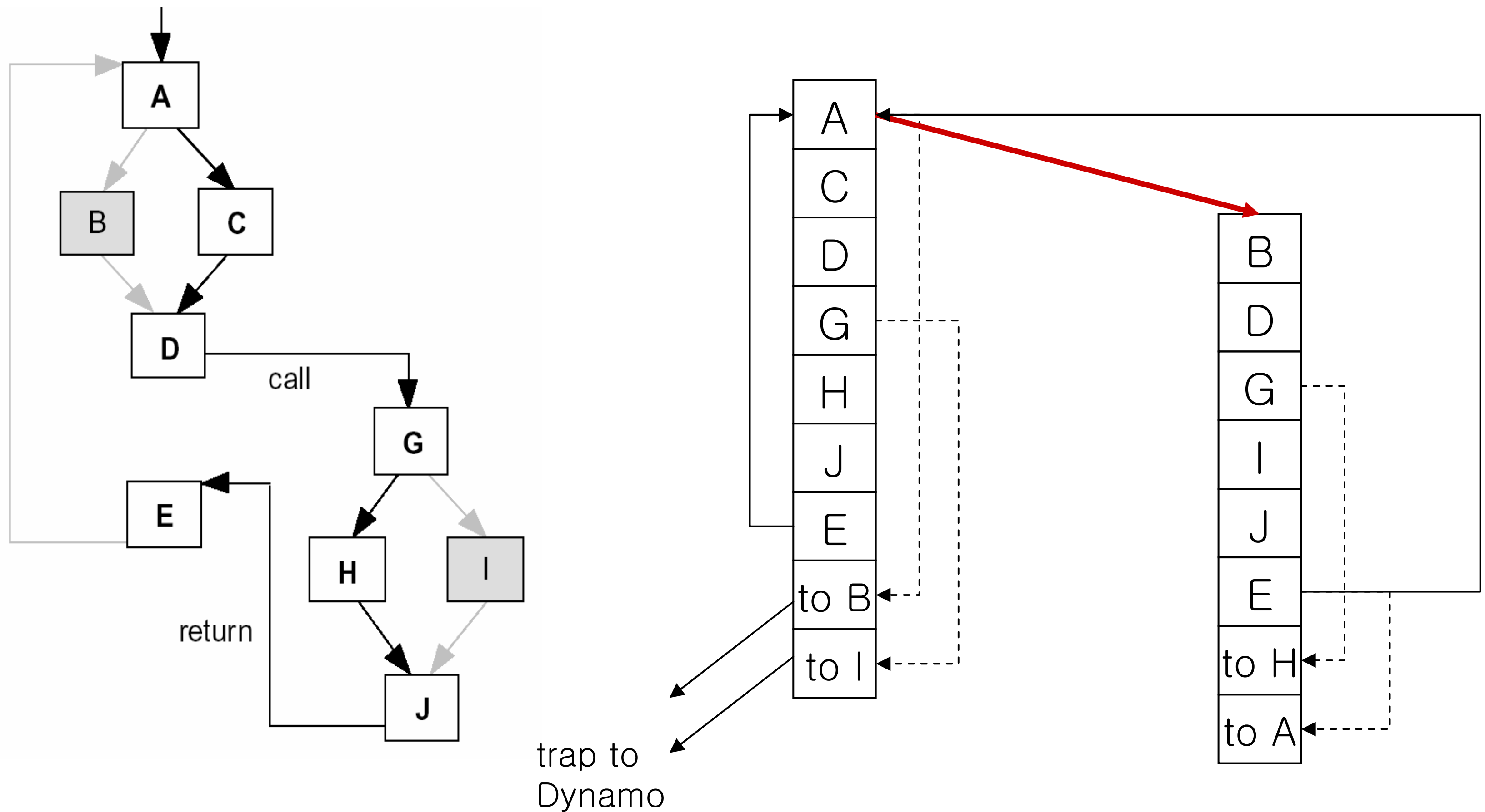


(a) Control flow snippet in an application binary,
(b) Layout of this snippet in the application program's memory.
(c) Layout of a trace through this snippet in Dynamo's fragment cache.

Trace Selection

- A strategy called Most Recently Executed Tail
- If an SoT candidate becomes hot (>50), then
  - Sequence of interpreted instructions from that point is recorded while being interpreted, until an EoT is met
  - A trace has just been found and will be added to the fragment cache after optimizations
- Simple but not always effective
  - A trace is fixed for each SoT which cannot be changed
  - What if off-trace path is hot for the SoT?
    - Well, it is just unfortunate, but duplication in off-trace and fragment linking would mitigate the damage
  - Why didn't profile execution paths to choose hotter trace?

An Example of Fragment Linking



Optimizations on the Trace

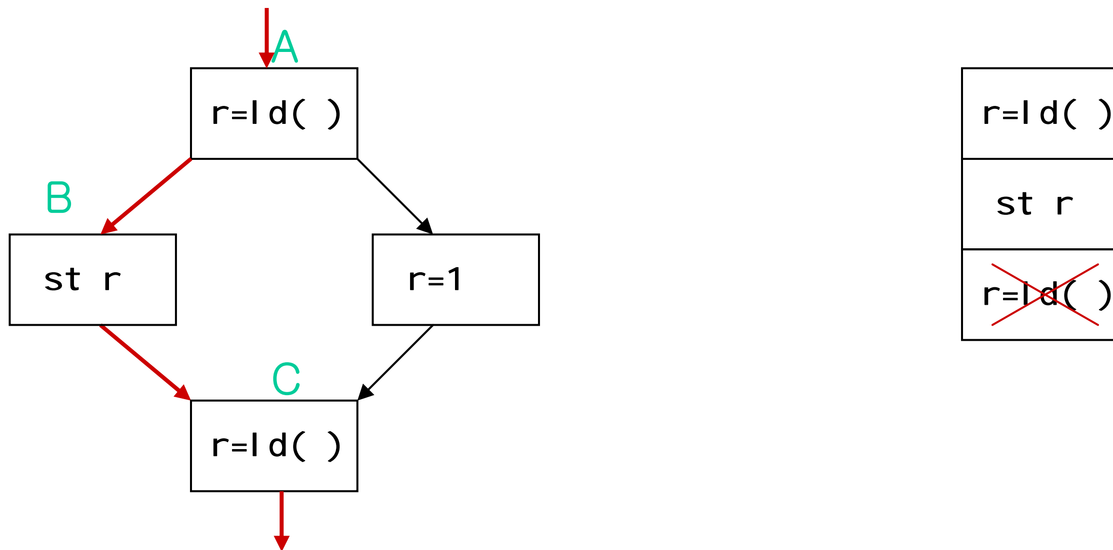
- All optimizations on the trace are "new" ones
  - Not readily available during static optimization, -O2
  - They are based on runtime information, in the sense that
    - The trace is formed based on runtime paths across calls
  - They are due to the nature of "trace", in the sense that
    - The trace allows otherwise unavailable optimizations
- Increase chances for "classical" optimizations
  - Which were impossible due to control flows in the code, or
  - Given up due to unclear benefit w/o profile information
    - Conservative optimization: copy propagation, strength reduction, constant folding/propagation, redundant branch elimination
    - Aggressive optimization: LICM, dead code elimination, redundant load elimination, loop unrolling

Branch Optimization

- Branch optimizations
  - Transform branches so the fall-through is on trace
    - Loop branch is allowed when it targets the start-of-trace
    - Otherwise a loop is a trace exit
  - Remove unconditional branch
  - Branch-and-link is converted into link-part only
  - Remove indirect branches
    - Function return is removed if the call is on the trace
    - Other indirect branch is converted into direct conditional branch
      - ✓ If fails, come to switch table for checking the cache. If there is none, return to the interpreter
- After branch optimizations, a trace is a single-entry, multi-exit stream with no control joins

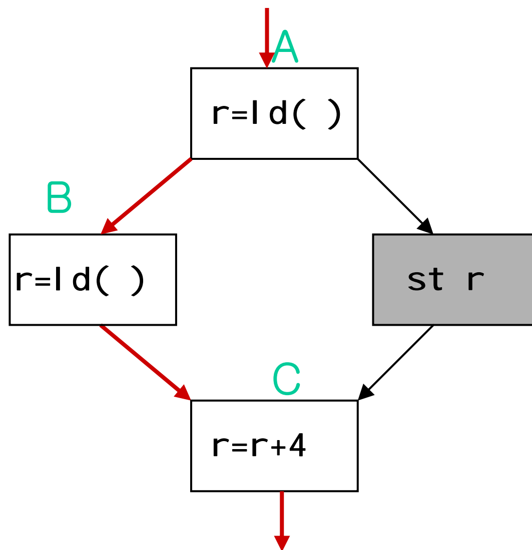
Partial Redundancy Elimination (PRE)

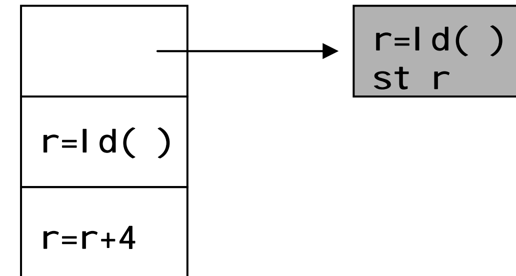
- Trace exposes otherwise unavailable PRE case



Partial Redundancy Elimination (PRE)

- Can also sink partially redundant instructions





A Case in GO benchmark

```
<mrglist>:
    . . .
+144:    ldw 0(sr0,r5),r23
+148:    combf,=,n r23,r4,<mrglist+208>
+152:    bl 80f8 <cpylist>,rp
+156:    ldo 0(r5),r25
+160:    ldw 0(sr0,r5),r31
+164:    comb,=,n r4,r31,<mrglist+124>
+168:    ldo 0(r3),ret0
+172:    addil 800,dp
+176:    ldo 550(r1),r23
+180:    ldwx,s r31(sr0,r23),r31
+184:    ldo 1(r3),r3
+188:    combf,=,n r4,r31,<mrglist+184>
+192:    ldwx,s r31(sr0,r23),r31
+196:    b <mrglist+124>
+200:    ldo 0(r3),ret0
+204:    nop
+208:    addil 5c800,dp
+212:    ldo 290(r1),r31
+216:    ldwx,s r26(sr0,r31),r24
+220: l  dwx,s r23(sr0,r31),r25
+224:    comb,<=,n r25,r24,<mrglist+316>
    ....
```

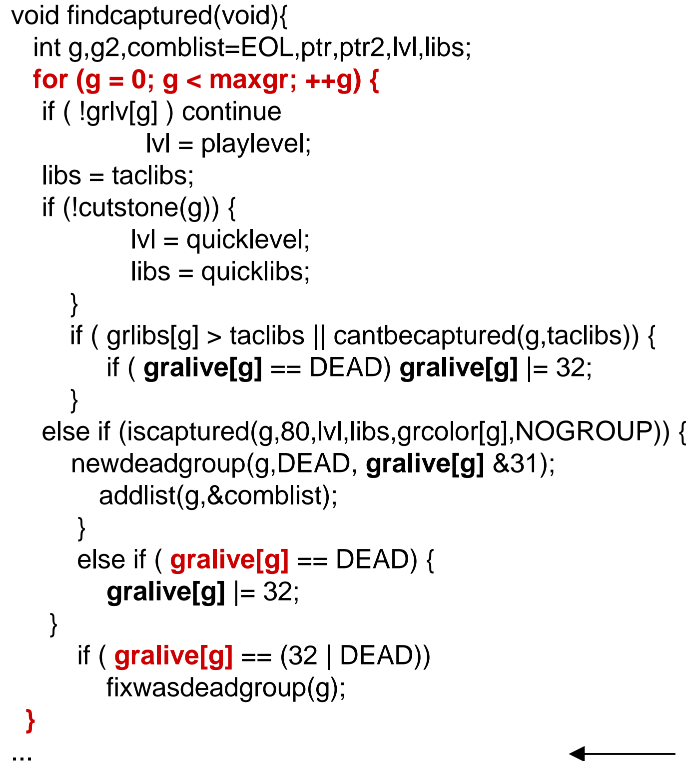
```
    ....
+316:    combf,=,n r25,r24,<mrglist+336>
+320:    addil 800,dp
+324:    ldo 550(r1),r23
+328:    b <mrglist+300>
+332:    ldwx,s r26(sr0,r23),r26
+336:    ldw 0(sr0,r5),r23
+340:    addil 35000,dp
+344:    ldw 518(sr0,r1),r25
+348:    addil 800,dp
```

Register Allocation Enhancement

- Some suboptimal register allocation results can be improved in the trace
- Example: register allocation of array element
  - Redundant load from the same element can be avoided
  - Usually done within basic blocks, not across basic blocks
  - Trace allows cross-block redundant load removal
  - See an example
    - There are multiple accesses to gralive[g] in the loop
    - Only gralive[g] is accessed
    - However, there are redundant loads across blocks

An Example in GO benchmark

Source code for function findcaptured from the go source file g25.c:



```
void findcaptured(void){
  int g,g2,comblist=EOL,ptr,ptr2,lvl,libs;
  for (g = 0; g < maxgr; ++g) {
   if ( !grlv[g] ) continue
             lvl = playlevel;
   libs = taclibs;
   if (!cutstone(g)) {
            lvl = quicklevel;
            libs = quicklibs;
      }
      if ( grlibs[g] > taclibs || cantbecaptured(g,taclibs)) {
          if ( gralive[g] == DEAD) gralive[g] |= 32;
      }
   else if (iscaptured(g,80,lvl,libs,grcolor[g],NOGROUP)) {
      newdeadgroup(g,DEAD, gralive[g] &31);
         addlist(g,&comblist);
       }
       else if ( gralive[g] == DEAD) {
          gralive[g] |= 32;
    }
       if ( gralive[g] == (32 | DEAD))
          fixwasdeadgroup(g);
 }
...
```

Static Assembly Code

```
               ...
+228:          ldo 0(r4),r26
+232:          comb,=,n ret0,r0,<findcaptured+592>
+236:          ldwx,s r4(sr0,r9),r31
+240:          combf,=,n r31,r6,<findcaptured+260>
+244:          sh2addl r4,r9,r25
+248:          ldw 0(sr0,r25),r26
+252:          depi -1,26,1,r26
+256:          stws r26,0(sr0,r25)
+260:          ldwx,s r4(sr0,r9),r18
+264:          combf,=,n r18,r15,<findcaptured+276>
+268:          bl <fixwasdeadgroup>,rp
+272:          ldo 0(r4),r26
```

Optimization of Shared Library Call

- Static optimization cannot do anything for shared library accesses
  - Since shared library is available only at runtime
- Resolution of shared library references
  - Procedure linkage table is created for every library call
  - Each call is redirected to an import stub which accesses the table, and an export stub is created for a return
- Shared library call overhead can be reduced
  - Complete inlining of the shared library call, or
  - Remove redundant call to import stub and loop-up tables

Shared Library Call Sequences

Import Stub:

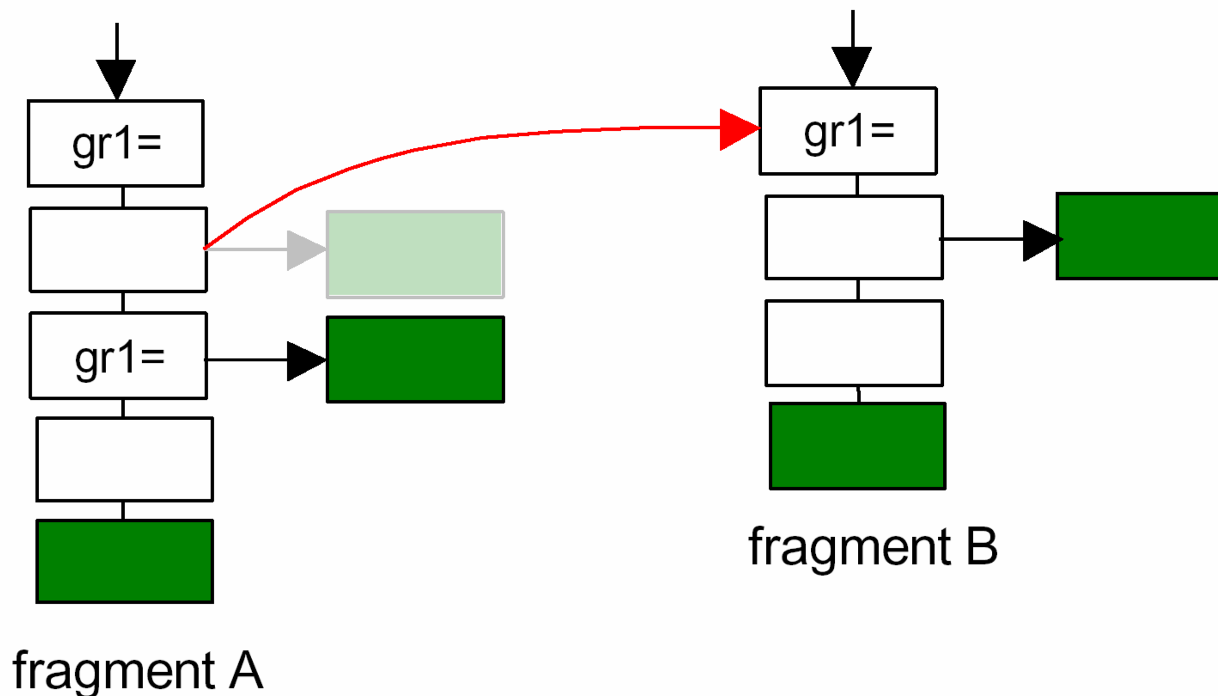
```
ADDIL L'lt_ptr+ltoff,                ; get procedure entry point
LDW  R'lt_ptr+ltoff(1),r21
LDW  R'lt_ptr+ltoff+4(1),r19         ; get new Linkage Table value for r19
LDSID      (r21),r1                  ; load new space ID
MTSP       r1,sr0                    ; set new space ID
BE         0(sr0,r21)                ; branch to procedure export stub
STW  rp,-24(sp)                      ; store return point
```

Export Stub:

```
BL,N  X,rp                           ; jump to procedure X
NOP
LDW  -24(sp),rp                      ; restore original return pointer rp
LDSID      (rp),r1                   ; restore original space ID
MTSP       r1,sr0                    ; set original space ID
BE,N 0(sr0,rp)                       ; return
```

Optimizations of Fragment Linking

- Fragment linking may allow the removal of redundant code in the compensation blocks



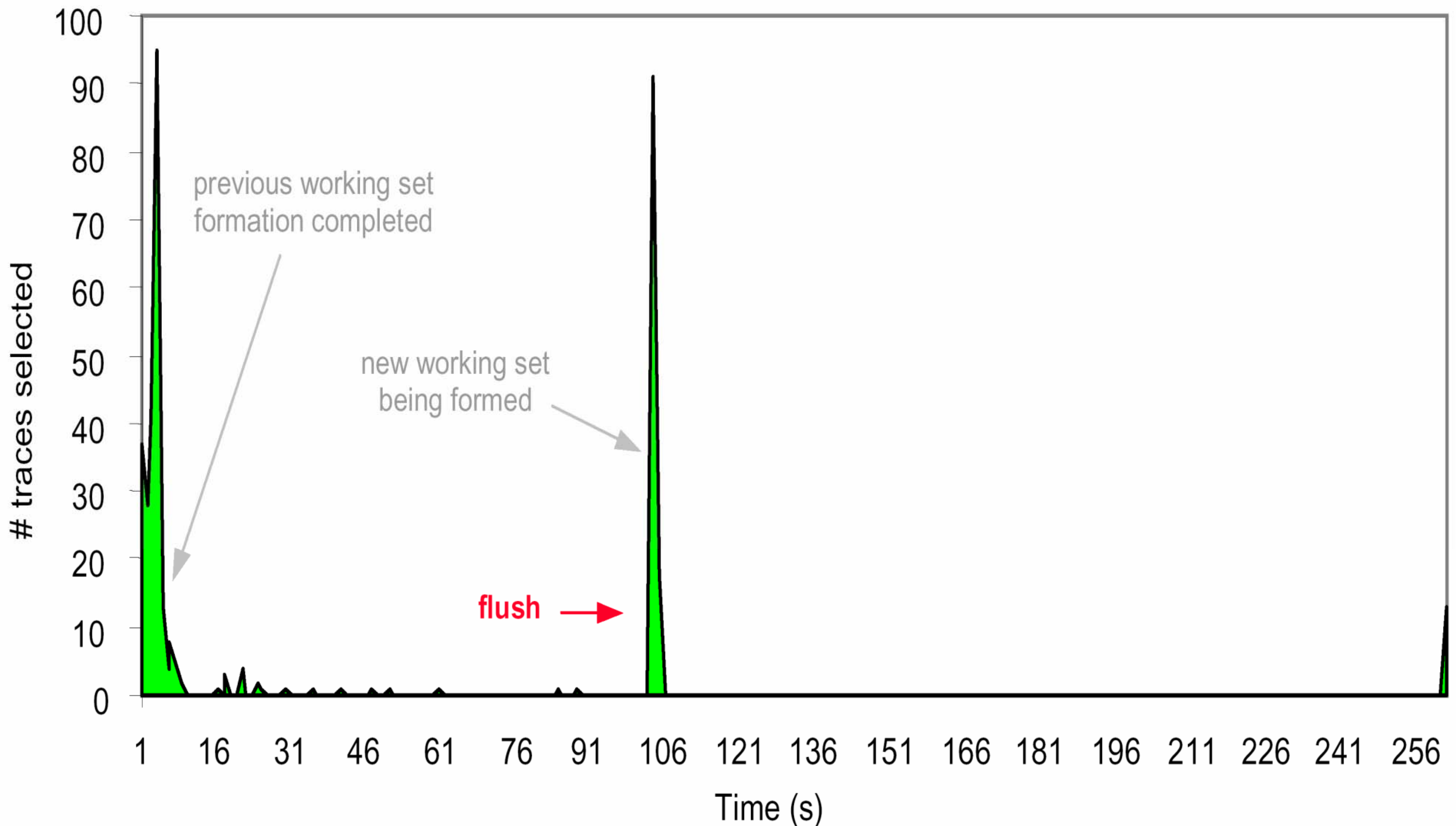
Instruction Scheduling

- No additional scheduling is done in Dynamo
  - Since PA8000 is already doing aggressive H/W scheduling
- But, aggressive scheduling would be simple
  - Scheduling on a single-path with no joins across calls

Fragment Cache Management

- Important for success of dynamic optimizers
- Issues in managing fragment cache
  - Size: can we simply have a large fragment cache?
    - Dynamo keeps the cache size small enough to preserve locality
    - Dynamo strategy (Most Recently Executed Tail) lowers # of traces
  - Flush: Need to flush fragments as working set changes
    - Hard to remove a fragment since we must unlink its predecessors
    - Dynamo flushes the entire cache at once, triggered by high fragment creation rate

Fragment Cache



Dynamic trace selection rate for m88ksim, showing a sharp change in the working set ~106 sec into its execution

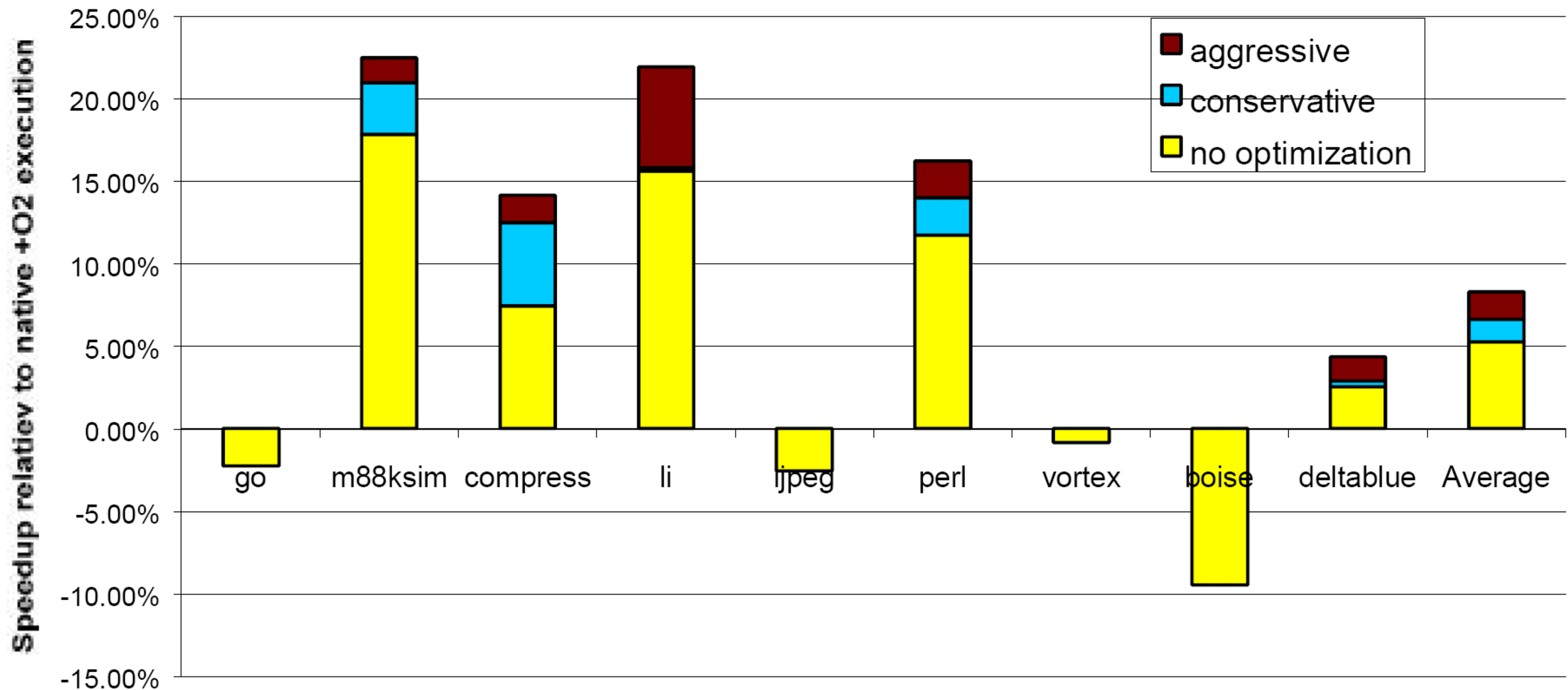
Experimental Environment

- Integer benchmarks
- HP C/C++ compiler, -O (+O2)
- HP PA-8000, HP-UX 10.20
- Fixed size 150Kbyte fragment cache
- Three cases of Dynamo:
  - No optimization: trace selection across calls only
  - Conservative trace optimization
  - Aggressive trace optimization

Dynamic Optimization Opportunities

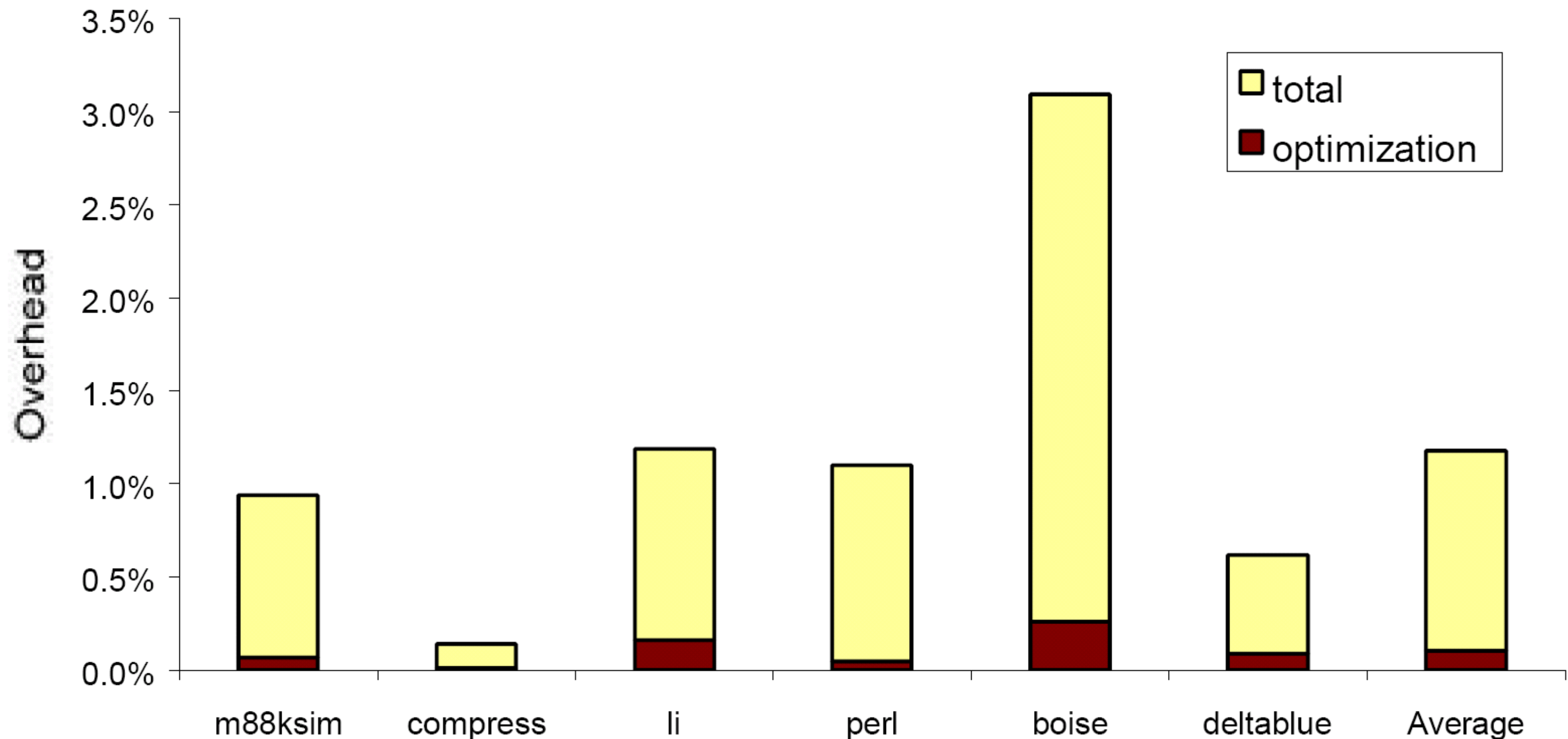
	strength reduction	redundant load	redundant branch	dead code	code sinking	guarded load	loop invariant motion
go	20620	2430	4315	2034	1172	0	21
m88ksim	2641	244	507	545	112	11	4
compress	194	9	30	28	1	0	2
li	3974	969	590	1162	122	0	3
ijpeg	5710	377	939	733	74	0	2
perl	3748	919	372	304	123	0	4
vortex	50310	3186	5548	12864	557	2	1
boise	7151	701	1128	1108	176	0	0
deltablue	2513	101	210	259	13	0	1

Dynamo Performance



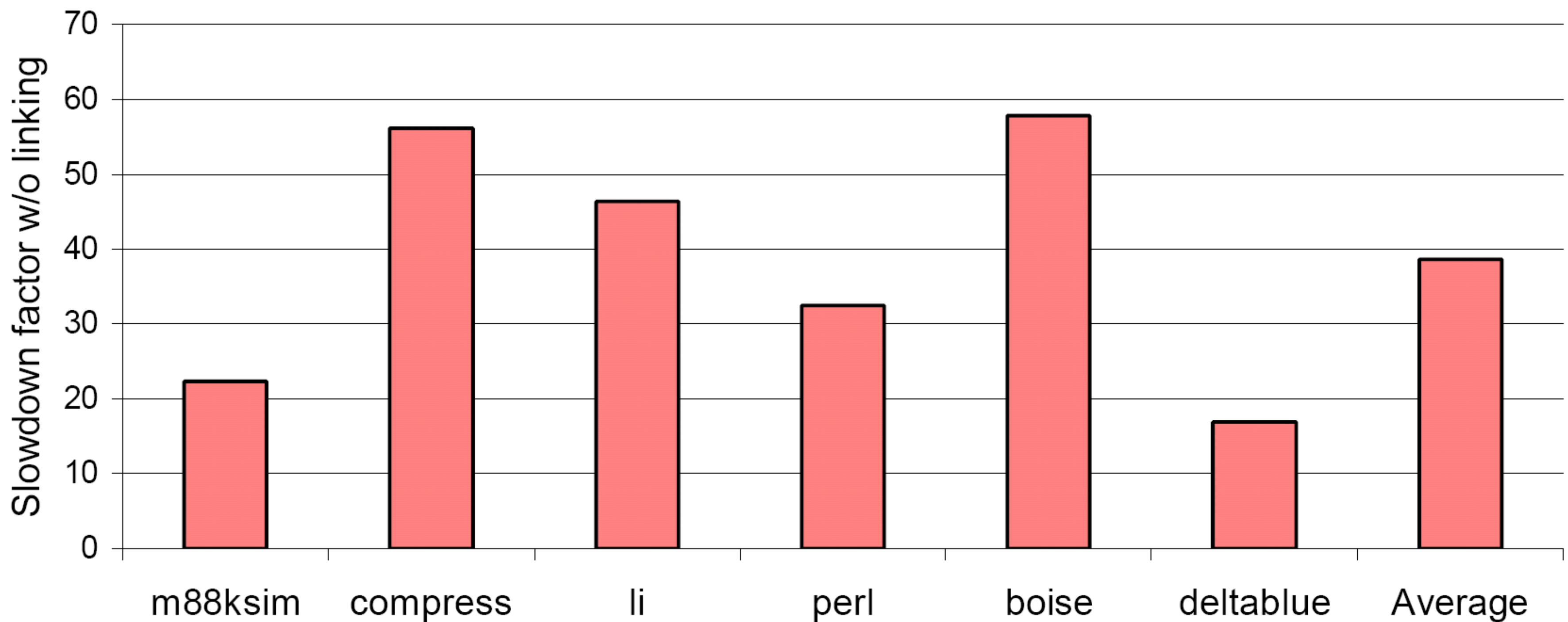
Overall Dynamo speedup with optimization contribution

Optimization and Dynamo Overhead



Optimization overhead is too small; more opportunities left

Performance Slow Down w/o Fragment Linking

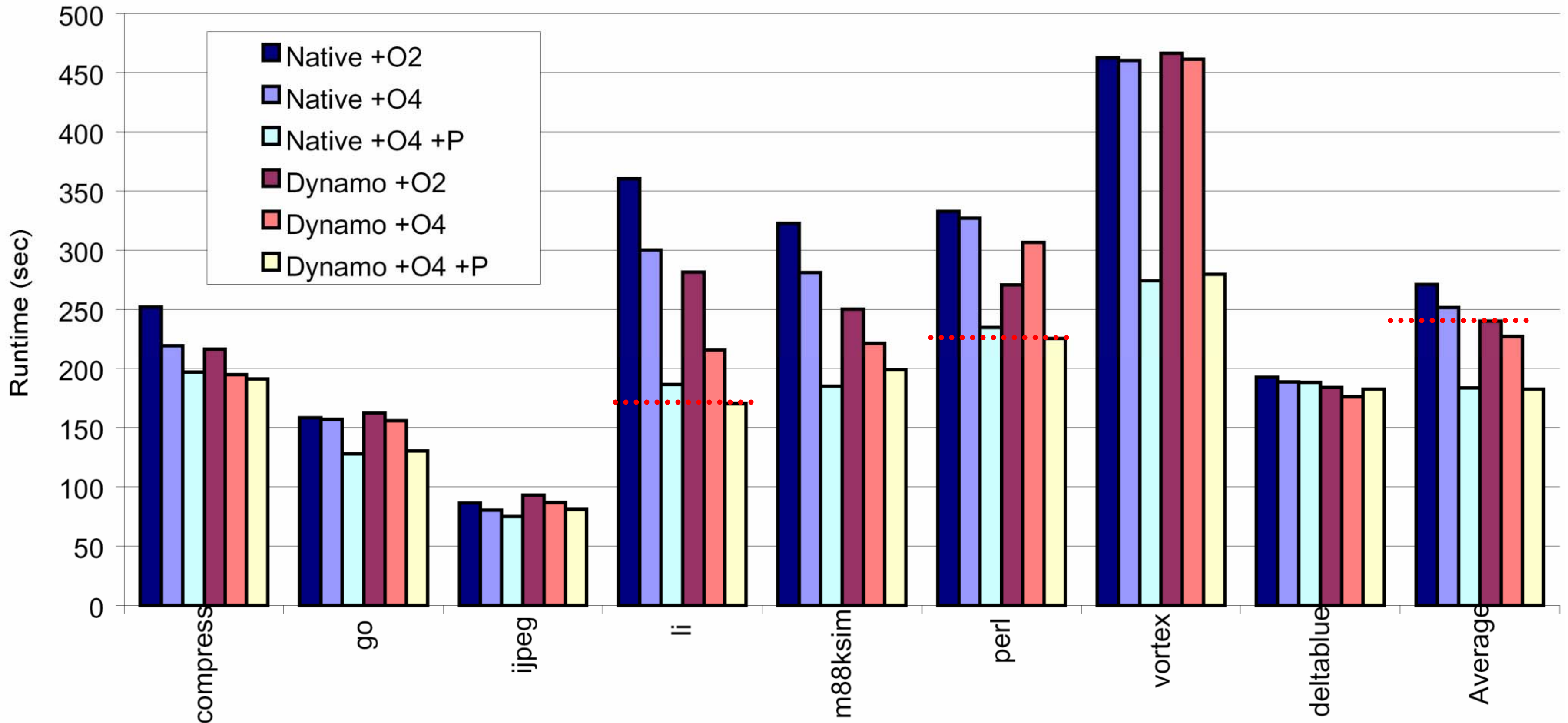


Performance slowdown when linking is disabled

Experimental Environment

- Three cases of Native and Dynamo:
  - +O2
  - +O4 (interprocedural optimization & link-time optimization)
    - Roughly corresponds to Dynamo's trace selection across calls
  - +O4 with profile feedback
    - Roughly corresponds to Dynamo's trace optimization
- Observations
  - Performance of +O2 binaries improved to +O4 levels
  - Even improves performance of +O4 binaries
  - Does not improve much on profiled code

Experimental Result



Dynamo performance on native binaries compiled at higher optimization levels (the first 3 bars for each program correspond to the native runs without Dynamo, and the next 3 bars correspond to the runs on Dynamo)