Power Awareness through Selective Dynamically-Optimized Traces

Roni Rosner, Yoav Almog, Naftali Schwartz, Avi Mendelson, Micha Moffie, Ari Schmorak and Ronny Ronen

Intel Labs, Haifa

Compiler & Architecture Seminar at IBM Haifa
November 2003

Agenda

PARROT =
Power-aware ARchitecture Running
Optimized Traces

- Concept and micro-architecture
- Performance and energy results
- Dynamic optimizations

Trends in a Nut-Shell

☺ Ideally, each process technology generation provides

- Reduction in transistor switching energy by 3X
- Reduction in transistor delay time by 1.5X
- Reduction in transistor switching power by 2X.

😐 More instructions per cycle, more cycles per second
➔ gains performance but consumes power...

☹ We pay more (energy) than we get (performance)

- More work per instruction (transistor switches)
- More instructions per task (deep speculations)
- More leakage power

➔ Higher energy consumption per task!

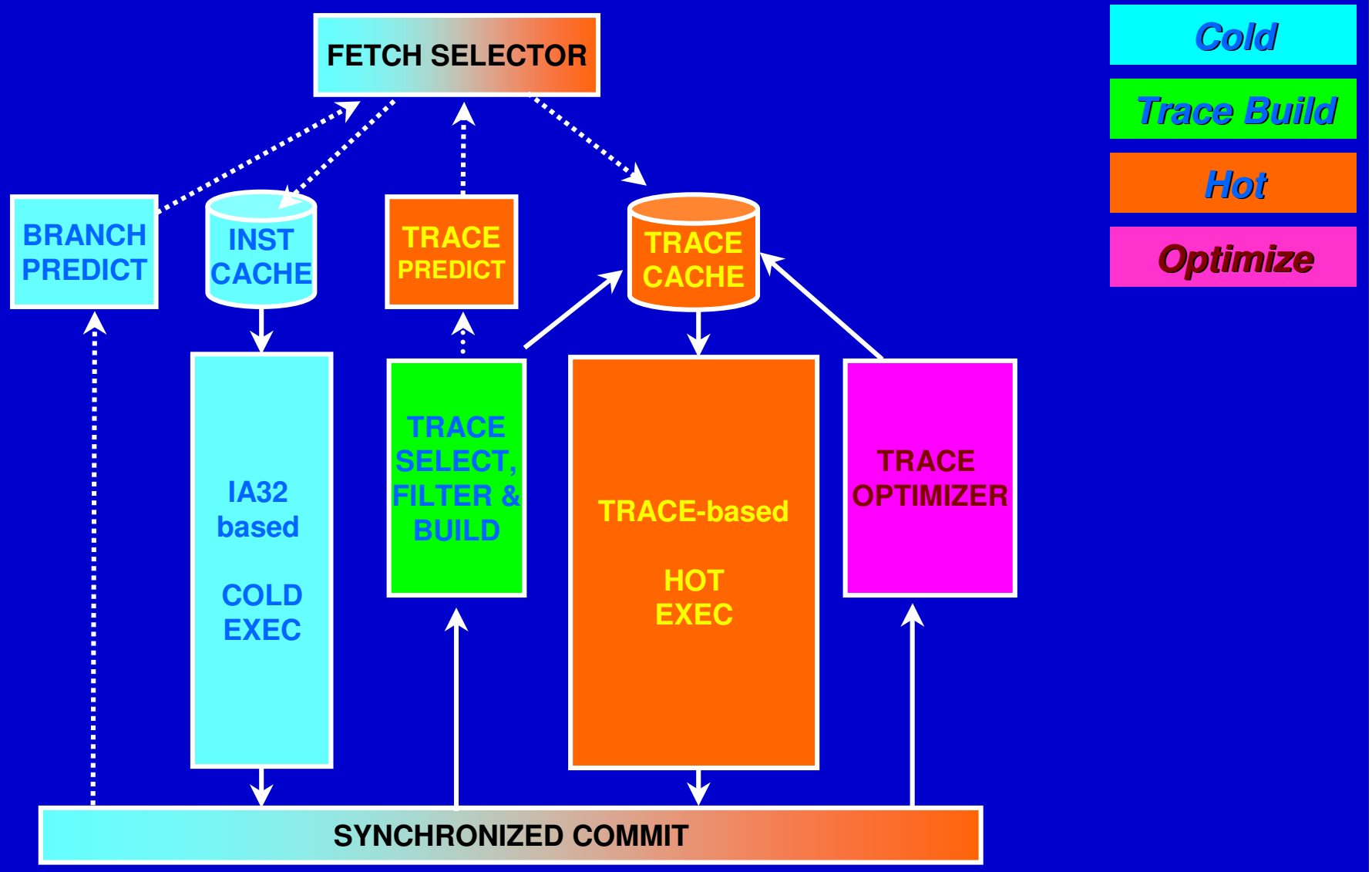
■ Parrot tries to change the balance!

Parrot Concept and Microarchitecture

Microarchitecture to identify hot traces, optimize them, execute efficiently – awareness of both performance and power

- *Extend the well known cold/hot (10 code / 90 execution) paradigm*
  → frequently used code behaves differently
  - More regular, predictable
  - Coarser granularity (longer sequences of instructions)
- Parrot Principles
  - ***Reuse*** → trace-cache centric
  - ***Dynamic optimizations*** → more performance with less energy, hw-oriented
  - ***Focus*** → invest where it pays
  - ***Asymmetric decoupling*** → hybrid front-end, cold and hot execution pipelines
  - ***Transparency*** → immune to s/w compatibility, overcome legacy

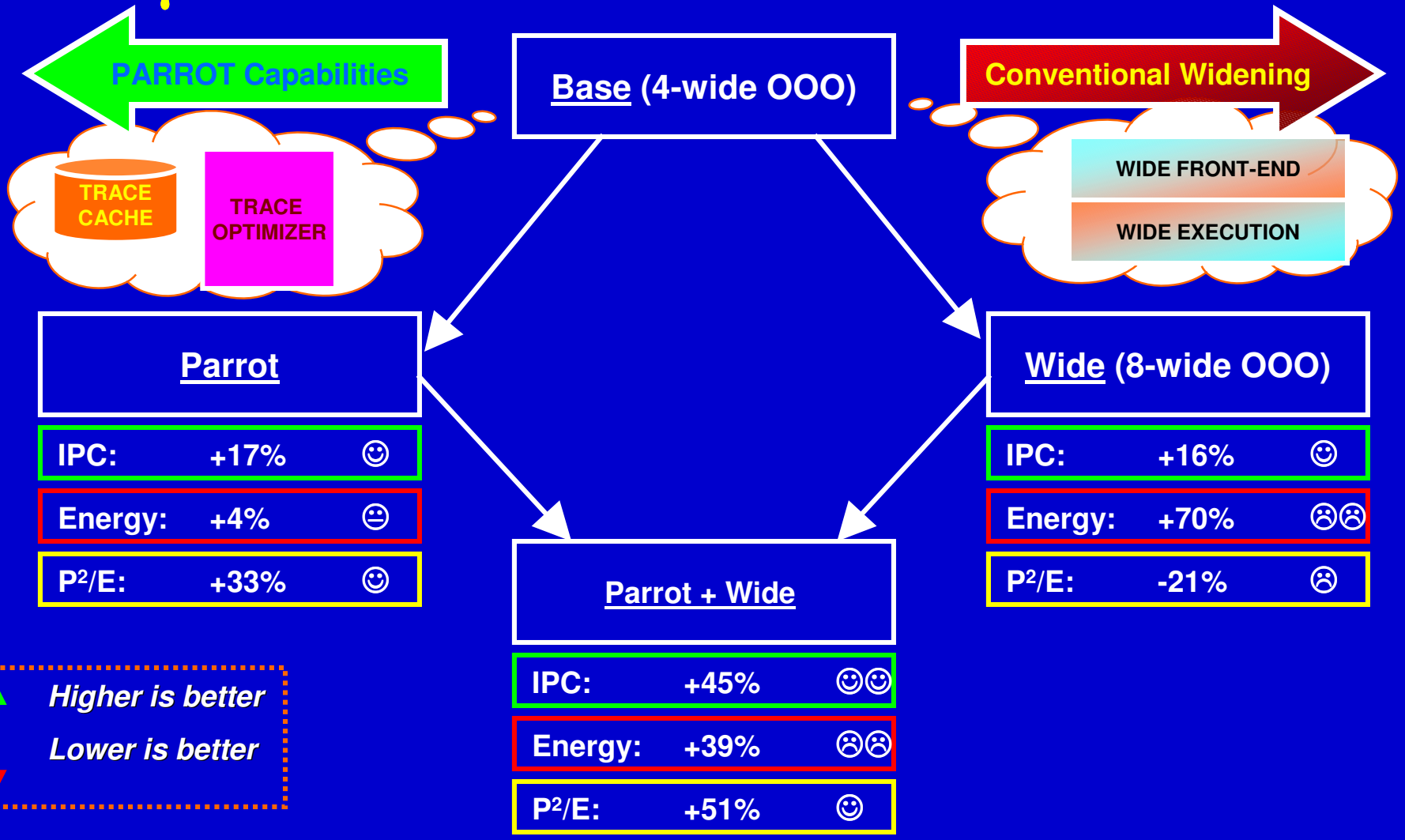
Parrot 4-Phase Scheme



Parrot in Context

- Software techniques:
  - JIT, profile-based compilation, binary-translators (Dynamo, Daisy, FX32!, IA32 Execution Layer for IA64), code morphing (Transmeta)
- Micro architectural techniques
  - Front end optimizations
    - Pentium® 4, UOP cache, RePlay (U of Illinois)
  - Asymmetric execution
    - Turbo-Scalar [Shen], DIVA [Austin]
- Our uniqueness
  - Filtering [PACT'01]
  - Smart trace selection [ICS'03]
  - Full IA32 context, including X87 floating-point model
  - Optimization for specialized execution [paper submitted]
  - Putting it *all* together [paper submitted]

μArchitecture Trade-Offs



Higher is better

Lower is better

PARROT exhibits comparable performance and better power-awareness (P^2/E)

Dynamic Optimizations

Optimization Classes

- **Trace selection**
  - Loop unrolling
  - Procedure inlining
- **Basic transformation**
  - Virtual renaming, *SSA*
  - FP-stack flattening
  - Handle partial regs
  - Color regs: *LI, tmp, LO*
  - Replace *CTI* by *ASSERT*
- **Generic optimizations**
  - Logic, arithmetic simplifications
  - Propagation of values, registers, conditions
  - Dead code elimination
- **Core-specific**
  - Uop fusion
  - *SIMD*ification
  - Semi-dynamic scheduling

Benefits of Optimizations

- **Less uops**
  - Higher performance
  - Reduced energy consumption
- **Reduced dependencies**
  - Higher ILP → higher performance

Efficiency of Optimizations

- **Utilization**
  - High rate of ***execute / optimize***
- **Sensitivity**
  - High benefits at low frequency of optimization

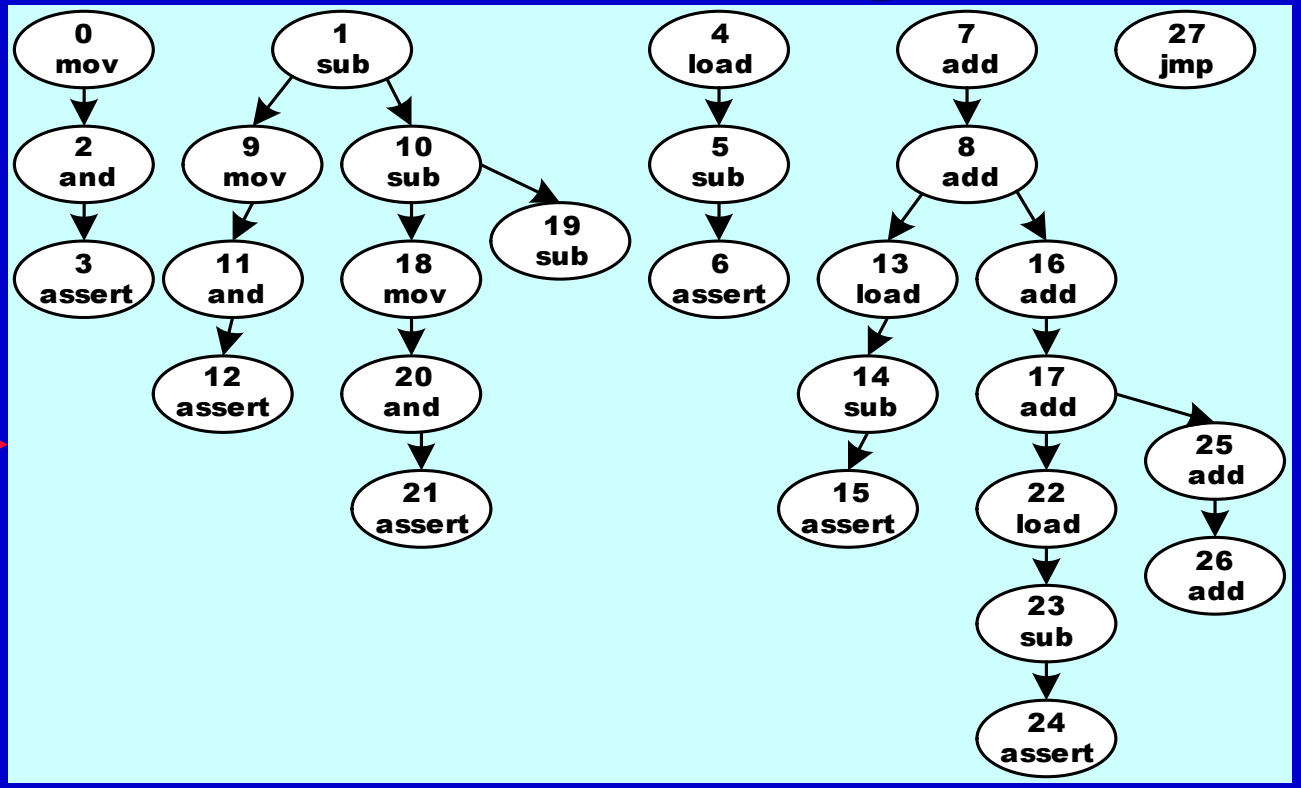
Optimization Example [SysMark, MS Word]

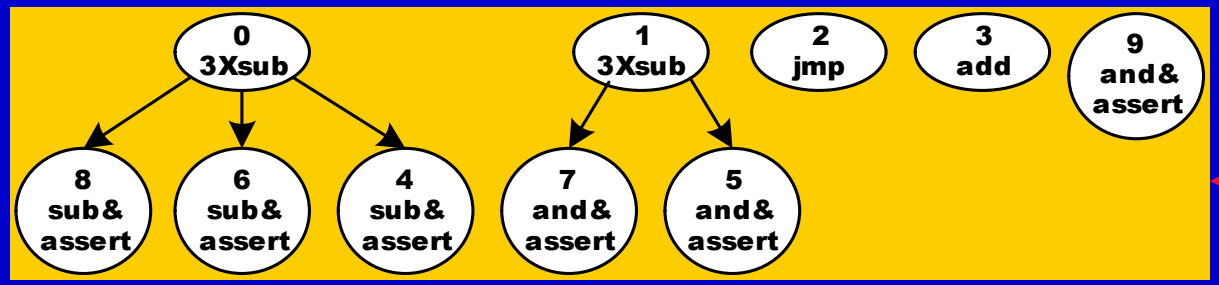
IA32 Instructions	Original PARROT uops	uops after basic-transformations	uops after optimizations
0. *mov* eax ← edi	0. eax ← *mov* (edi)	0. v_a(0) ← *move* (li_edi)	0. v_c(1),v_c(3), lo_cx ←*3Xload*(li_ds,li_esi)
1. *dec* edi	1. edi, eflags ← *sub* (edi, 1)	1. v_b(0), v_flags(0) ← *sub* (li_edi, 1)	1. v_b(0),v_a(2),v_b(3),lo_eax, lo_edi ←*3Xsub*(li_edi,1, li_edi,2, li_edi, 3)
2. *testl* eax, eax	2. eflags ←*and* (eax, eax)	2. v_flags(1) ← *and* (v_a(0), v_a(0))	2. *jmp* (0x3004d8a5)
3. *je* +0x14	3. *cond_jmp*(e,eflags)	3. *assert_cond* (v_flags.z(1), e, ntaken)	3. lo_esi ←*add*(li_esi, 0x6)
4. *mov* cx ← [esi]	4. cx ← *load* (ds, esi)	4. v_c(1) ← *load*(li_ds,li_esi)	4. lo_eflags ←*sub&assert*(lo_cx,256,nbe,ntaken)
5. *cmp* cx, 256	5. eflags ← *sub* (cx, 256)	5. v_flags(2) ← *sub* (v_c(1), 256)	5. *and&assert*(v_b(3),v_b(3), e, ntaken)
6. *ja* 0x2c645e	6. *cond_jmp* (nbe, eflags)	6. *assert_cond*(v_flags.cz(2),nbe, ntaken)	6. *sub&assert*(v_c(3), 256, nbe, ntaken)
7. *inc* esi	7. esi, eflags ← *add* (esi, 1)	7. v_b(1), v_flags(3) ← *add* (li_esi, 1)	7. *and&assert*(v_b(0), v_b(0),e, ntaken)
8. *inc* esi	8. esi, eflags ← *add* (esi, 1)	8. v_b(2), v_flags(4) ← *add* (v_b(1), 1)	8. *sub&assert*(v_c(1), 256, nbe, ntaken)
9. *jmp* -23	9. *jmp*(0x3004d8a5)	9. v_a(2) ← *move* (v_b(0))	9. *and&assert*(li_EDI, li_EDI, e, ntaken)
10. *mov* eax ← edi	10. eax ← *mov* (edi)	10. v_b(3), v_flags(5) ← *sub* (v_b(0), 1)	
11. *dec* edi	11. edi, eflags ←*sub* (edi, 1)	11. v_flags(6) ← *and* (v_a(2), v_a(2))	
12. *testl* eax, eax	12. eflags ←*and* (eax, eax)	12. *assert_cond* (v_flags.z(6),e, ntaken)	
13. *je* +0x14	13. *cond_jmp*(e,eflags)	13. v_c(3) ← *load*(li_ds,v_b(2))	
14. *mov* cx ← [esi]	14. cx ← *load* (ds, esi)	14. v_flags(7) ← *sub* (v_c(3), 256)	
15. *cmp* cx, 256	15. eflags ← *sub* (cx, 256)	15. *assert_cond*(v_flags.cz(7),nbe, ntaken)	
16. *ja* 0x2c645e	16. *cond_jmp* (nbe, eflags)	16. v_b(4), v_flags(8) ←*add* (v_b(2), 1)	
17. *inc* esi	17. esi, eflags ← *add* (esi, 1)	17. v_b(5), v_flags(9) ← *add* (v_b(4), 1)	
18. *inc* esi	18. esi, eflags ← *add* (esi, 1)	18. lo_eax ← *move* (v_b(3))	
19. *jmp* -23	19. *jmp*(0x3004d8a5)	19. lo_edi,v_flags(10)←*sub*(v_b(3), 1)	
20. *mov* eax ← edi	20. eax ← *mov* (edi)	20. v_flags(11)←*and*(lo_eax,lo_eax)	
21. *dec* edi	21. edi, eflags ←*sub* (edi, 1)	21. *assert_cond*(v_flags.z(11), e, ntaken)	
22. *testl* eax, eax	22. eflags ←*and* (eax, eax)	22. lo_cx ←*load*(li_ds,v_b(5))	
23. *je* +0x14	23. *cond_jmp*(e,eflags)	23. lo_eflags ←*sub*(lo_cx, 256)	
24. *mov* cx ← [esi]	24. cx ← *load* (ds, esi)	24. *assert_cond*(lo_eflags(65),nbe, ntaken)	
25. *cmp* cx, 256	25. eflags ← *sub* (cx, 256)	25. v_b(6),v_flags(12) ←*add* (v_b(5), 1)	
26. *ja* 0x2c645e	26. *cond_jmp* (nbe, eflags)	26. lo_esi,v_flags(13)←*add*(v_b(6), 1)	
27. *inc* esi	27. esi, eflags ← *add* (esi, 1)	27. *jmp* (0x3004d8a5)	
28. *inc* esi	28. esi, eflags ← *add* (esi, 1)		
29. *jmp* -23	29. *jmp*(0x3004d8a5)		

Optimization Example

Dependency graph of the trace after renaming & SSA, *before optimizations*

- *28 micro-ops*
- *tree-height 7*

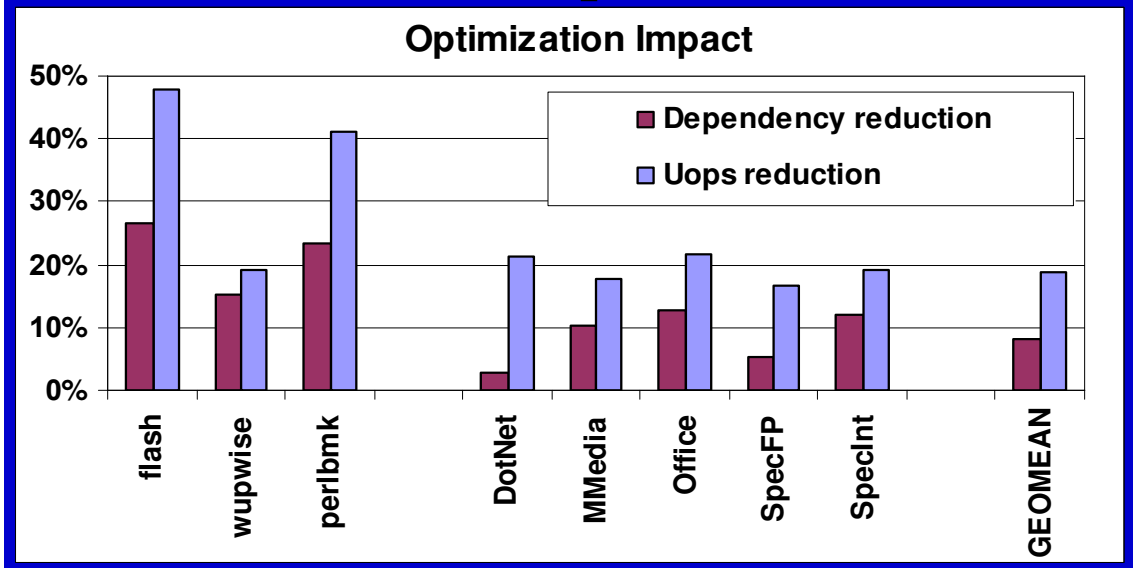




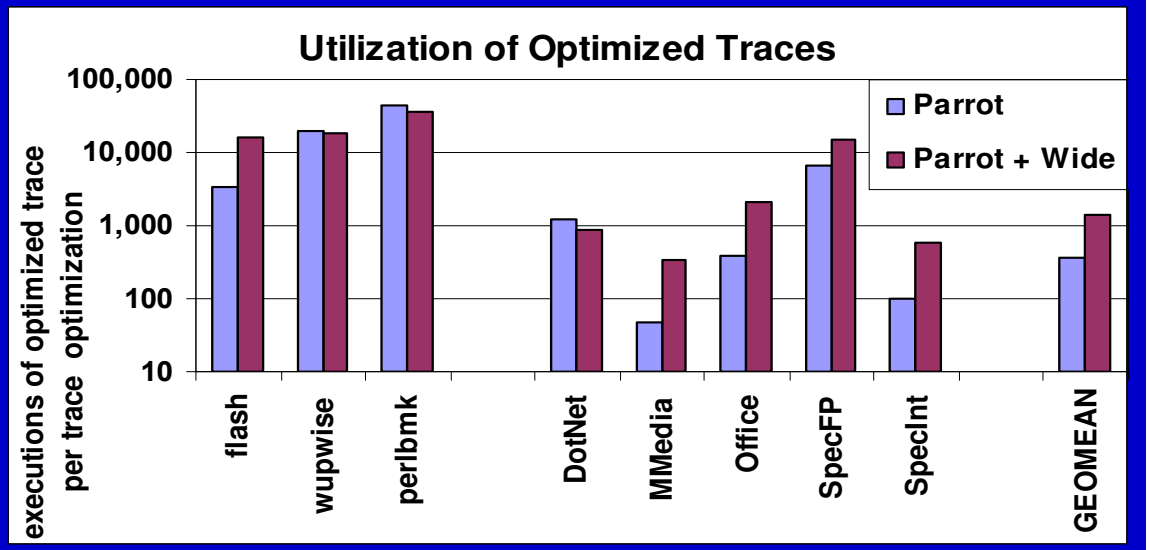
Dependency graph of the trace *after optimizations*

- *10 micro-ops*
- *Tree-height 2*

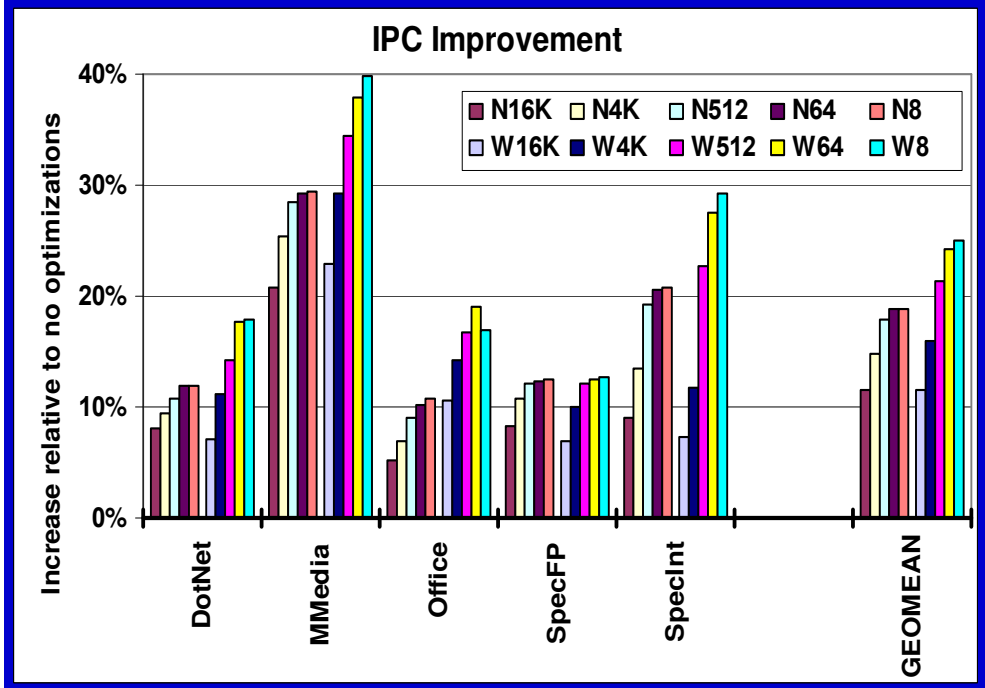
Optimizer Quality

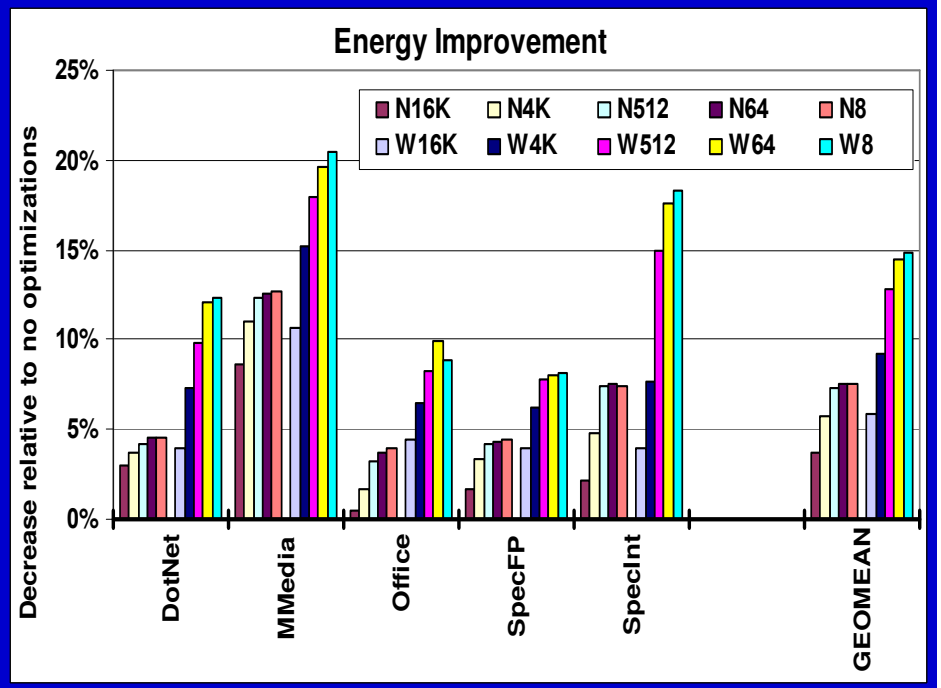


Higher is better



Optimizer Sensitivity: Impact of Filter's Threshold on IPC,Energy





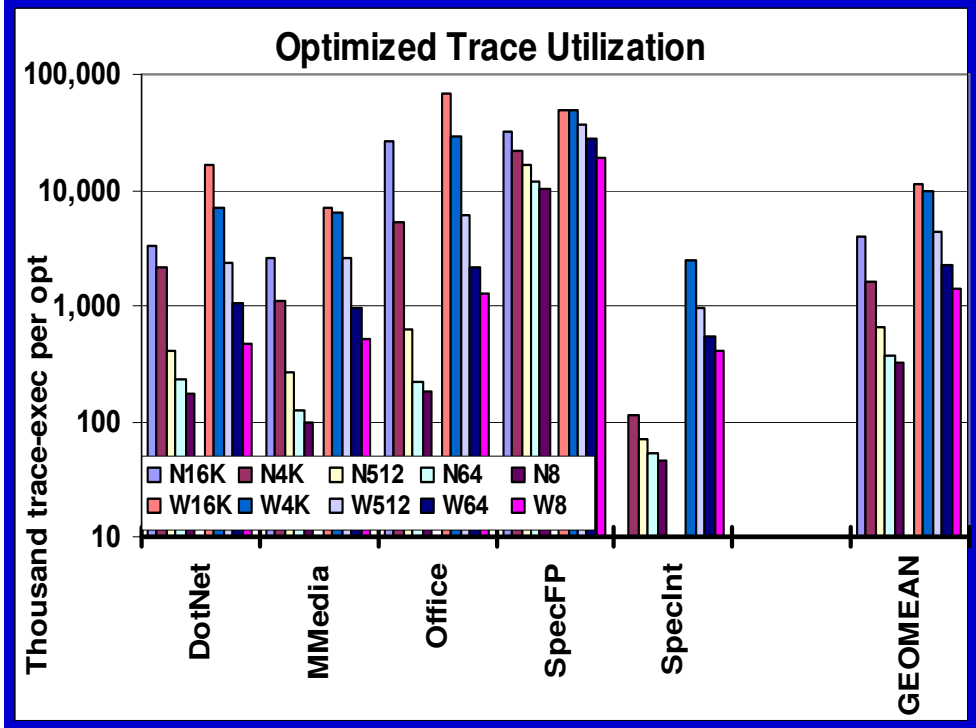
- **Higher bars are better for overall performance, energy consumption**
- **Higher threshold *j* represent less sensitivity to optimizer cost, latency**
- **Tradeoff point: "knee" where smaller threshold has a low marginal contribution to performance ~512**

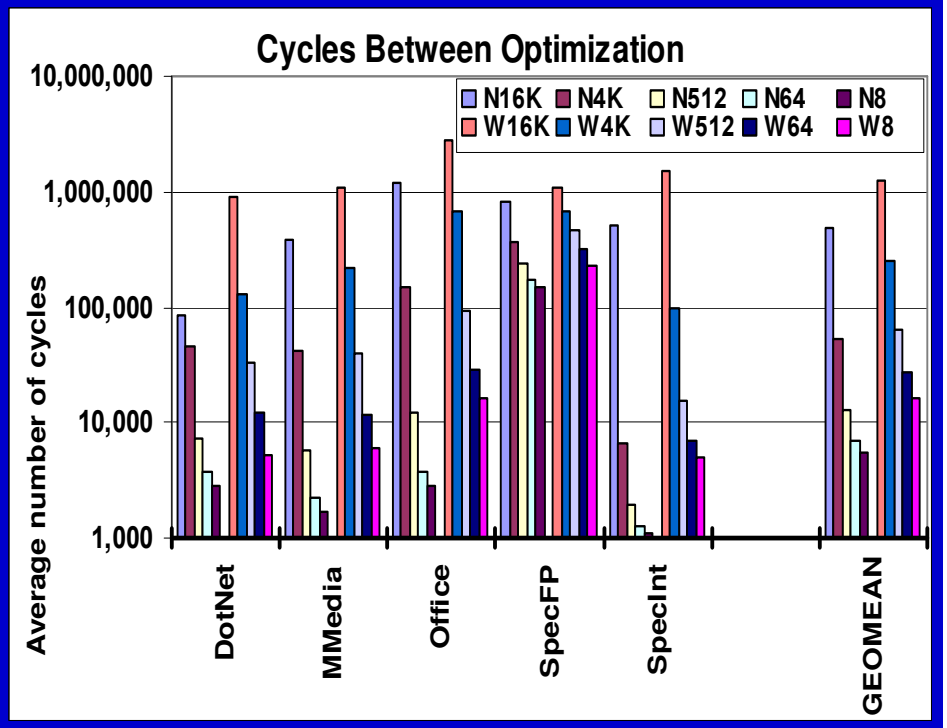
Legend

N*j* - Parrot, optimizer threshold is *j*

W*j* - Parrot+Wide, optimizer threshold is *j*

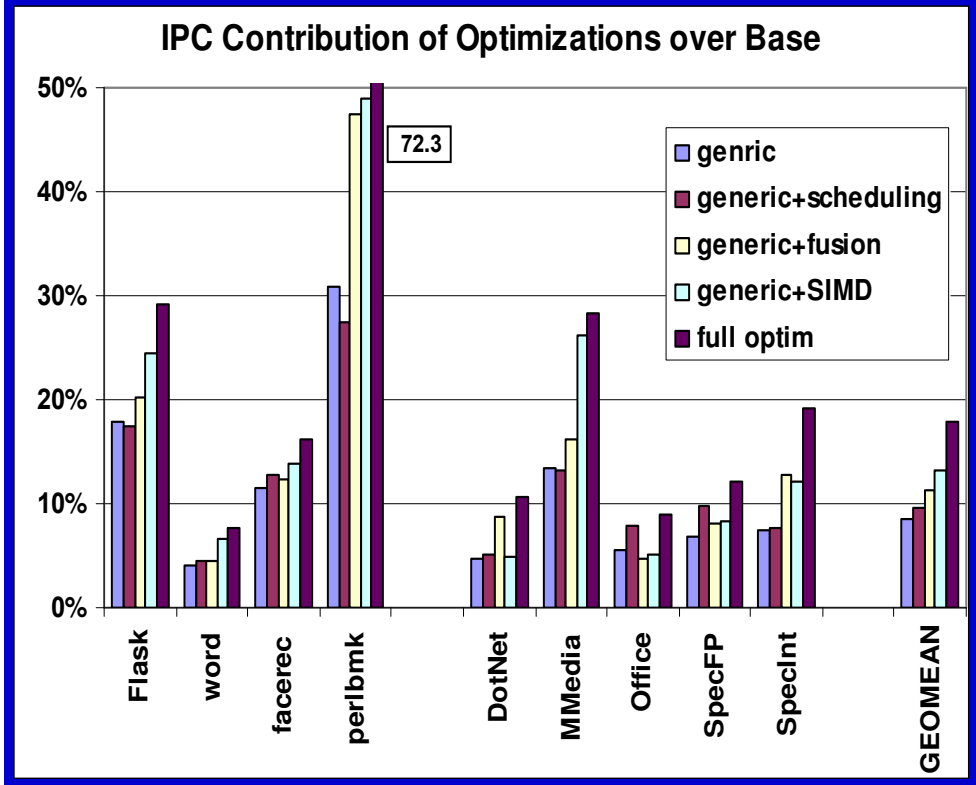
Optimizer Sensitivity: Impact of Filter's Threshold on IPC, Energy

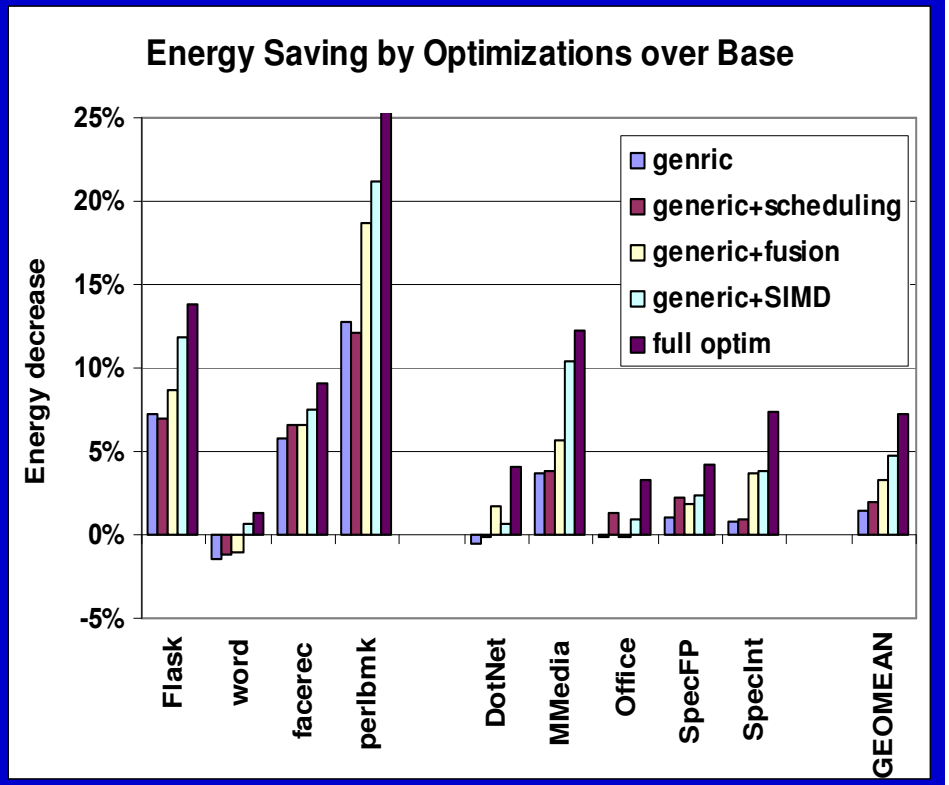




Relaxed design of optimizer is feasible!!!

Optimizations Breakdown: Generic vs. Core-Specific





Higher is better

Conclusions and Future

Major conclusions

- Parrot-style μarchitecture presents a power-aware alternative to conventional design
- Parrot μarch shows low sensitivity to optimizer latency, greediness

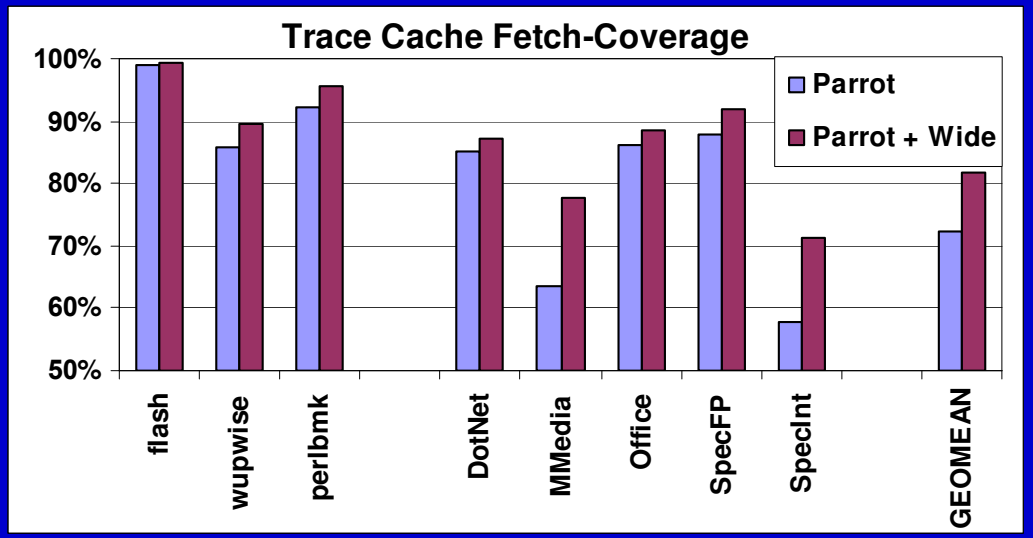
  → feasibility of relaxed optimizer
- Core specific optimizations are central for hardware-based dynamic optimizations

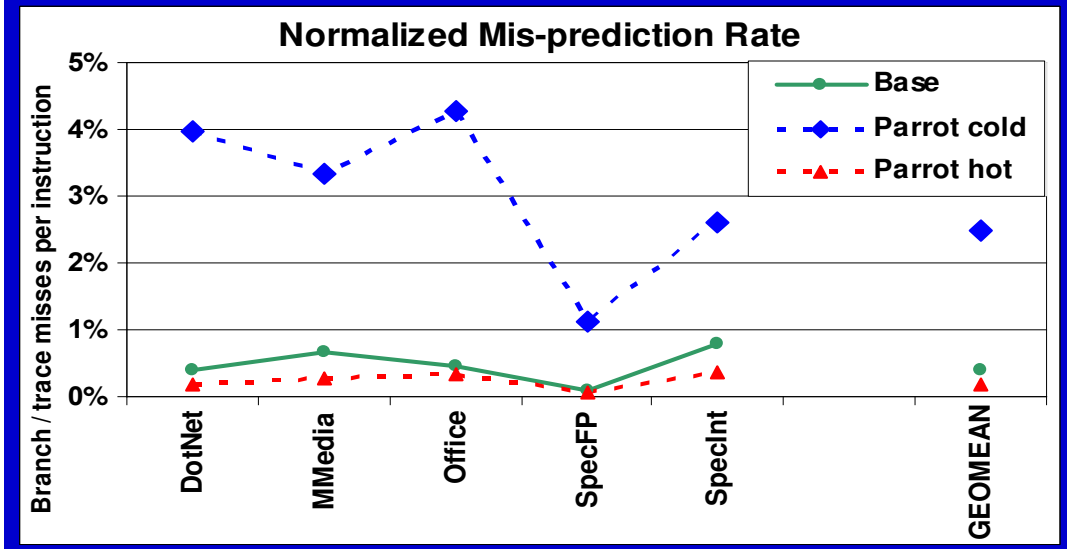
Interesting Future Directions

- Microarchitecture
  - Improved selection, prediction
  - Specialized traces, value prediction
  - Multi-threading
- Architecture and software
  - Interaction with compilers, profiling, JIT
  - Quality of service
- Improved dynamic optimizations

Backup Slides

Front-End Characteristics





Higher is better

Lower is better