

Field-testing IMPACT EPIC research results in Itanium 2

**John W. Sias, Sain-zee Ueng,
Geoff A. Kent, Ian M. Steiner,
Erik M. Nystrom, Wen-mei W. Hwu**

**Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign**

31st Annual International Symposium on Computer Architecture

IMPACT's ILP heritage: 1987–2004

- Key “proto-EPIC” compiler technologies, architectural elements, microarchitectural features
 - control speculation [ISCA-91] [ASPLOS-92] [MICRO-96]
 - data (dependence) speculation [ICS-92] [ASPLOS-94]
 - predicated execution [MICRO-92][ISCA-95] [MICRO-97]
 - integrated architecture and inline recovery [ISCA-98]
 - logic transformation with predication [ISCA-99][IEEE-Proc-01]
 - run-time code adaptation [ISCA-99][ISCA00][IEEE-TC-01]
 - High-frequency, non-stalling implementation [MICRO-03]
- Early Itanium evaluation – and more development
 - Itanium performance insights [HotChips-01][MPForum-01]
- Today IMPACT is producing record code for Itanium 2

Itanium 2 provides an unprecedented opportunity for “real world” evaluation of our concerted approach to ILP.

What's in this talk?

- EPIC philosophy in a nutshell
- “Structural” control transformation for EPIC
 - Code example
 - Implications for compiler design
- *In situ* evaluation highlights: how IMPACT beats *gcc* by 1.55x (and up to 2.3x)
- Lessons learned / future work

Read the paper for the whole story!

EPIC philosophy and the compilation problem

- EPIC shifts performance planning responsibility onto the compiler
- Reduces μ arch complexity vs. out-of-order design → greater width, efficiency – but the hardware’s performance “safety net” is removed
- Compiler’s focus: maximizing overall performance in a comprehensive plan
 - Paradigm change for compiler writers – what is “optimization”?
 - How well can the compiler handle the complex decisions required?
 - How well can the μ arch capitalize on the compiler’s plan?
- Architecture defines features to improve ILP formation: explicit control, data speculation; predication [August98]

Tailings at the compile-time ILP mine

- **Control** (branches/subroutine calls) breaks up instruction optimization and issue
 - Static (plan-time): positional encoding and segmentation
 - Dynamic (execute-time): bubbles and mispredictions
- Need to discriminate between true and **false dependences**
- May need to mitigate “**occasionally true**” dependences
- May need to build in tolerance for **variable latencies** (e.g. data cache)

Details are in the paper!

Structural transformation in context

- Inline hot call sites to expose code to transformation
- Apply suite of collaborative techniques to expose control-bound operations to ILP optimization
 - Superblock formation
 - Hyperblock formation
 - Tail duplication, loop peeling ...
- Err on the side of aggressiveness to enable best eventual transformation, but limit per-stage costs
 - Potential for over/under-shoot
 - Need for increased aggressiveness strains this working model
- Later, optimize code thoroughly to reduce overhead
 - Opportunistic specialization, promotion, demotion, instruction merging, height reduction, ...
 - Schedule taking advantage of purchased freedom

Structural transformation in *crafty*



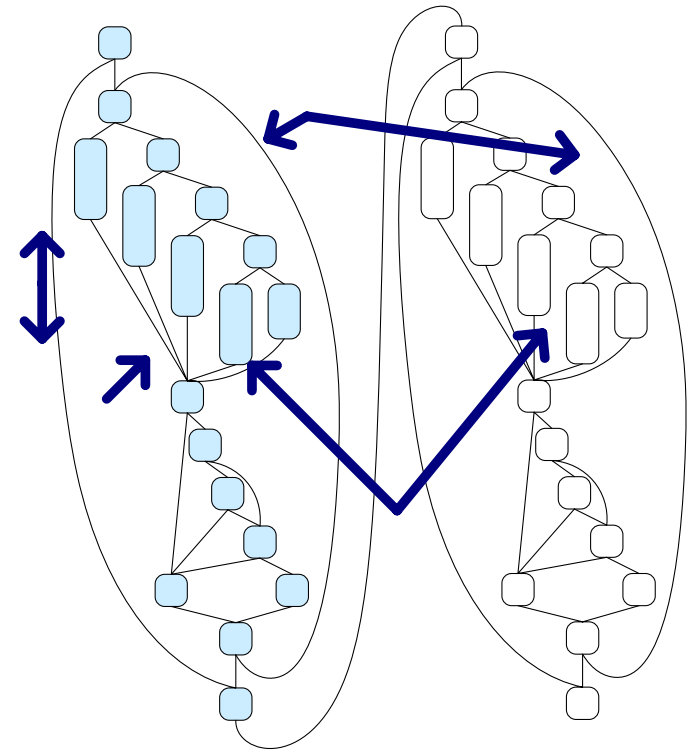
Inlining gathers code into transformable regions

- programs generally not written for ILP
- “hot”, small, internally serial functions common



Branching control is a serious obstacle to ILP

- misprediction and bubbles (9% of bench. CPU cycles)
- worse, lacks issue parallelism



Source: *crafty* Evaluate()

Structural transformation in *crafty* (2)



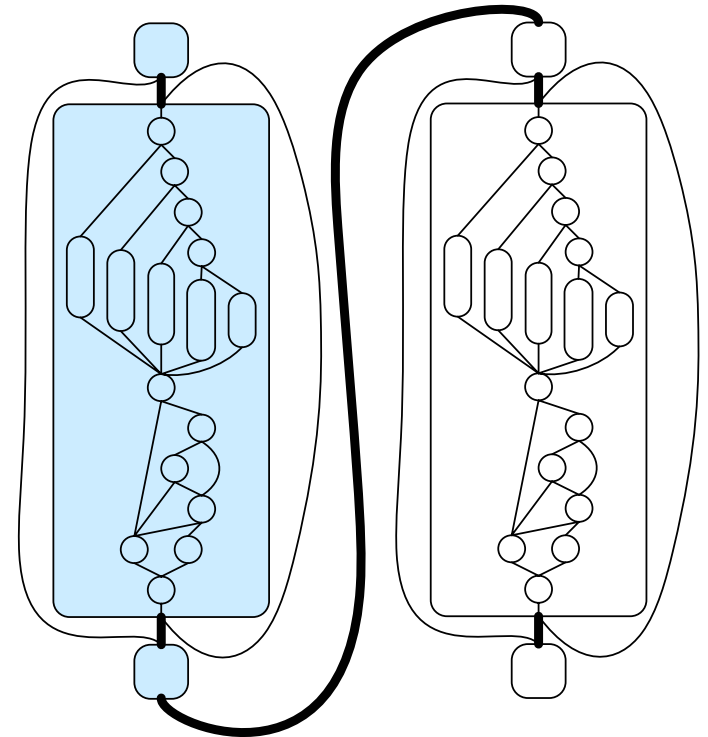
Basic if-conversion

- eliminates mispredictions within loops, increases scheduling scope
- only minimally impacts plan height
- fails to exploit parallelism across loops (SWP no good here)



More work needed to get positive outcome

- consider frequent path through enclosing region

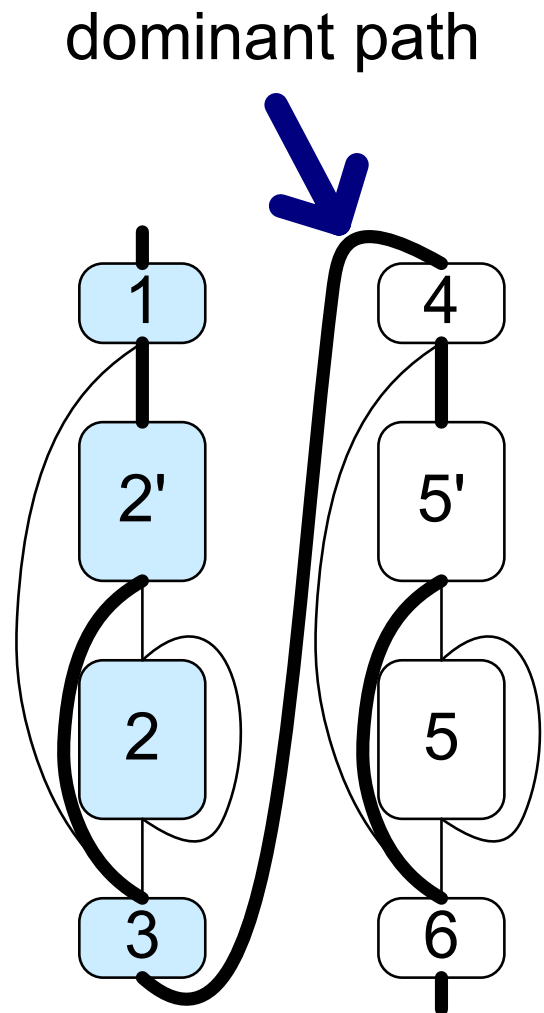


Structural transformation in *crafty* (3)



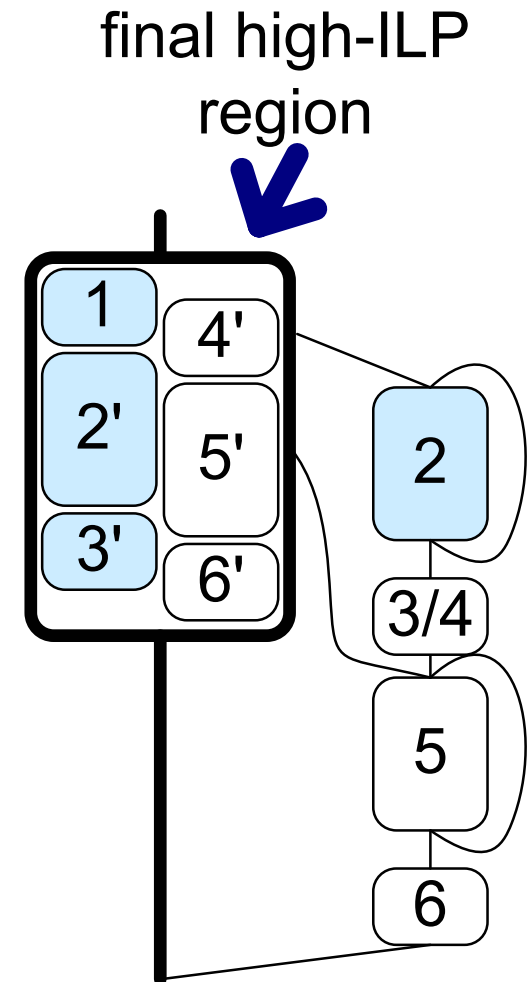
Need to expose larger regions for parallelization

- Make common case accessible for parallelization
- Loop peeling disentangles common case path out of loops
- Again, little independent impact



Structural transformation in *crafty* (4)

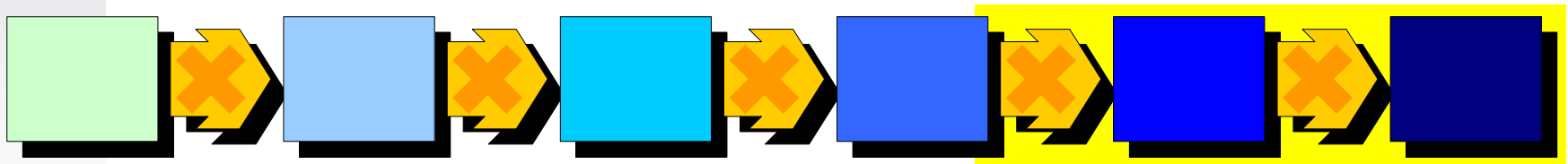
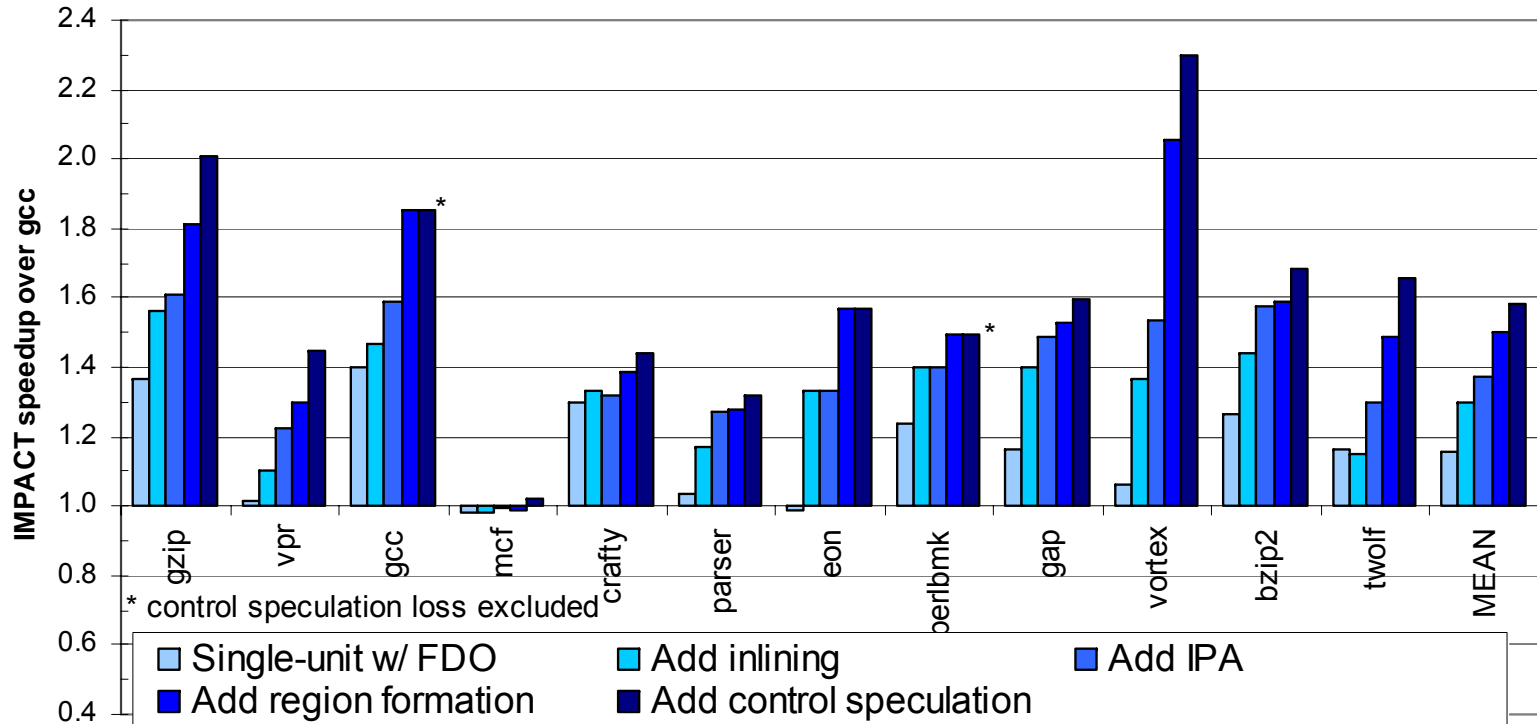
- Region formation pulls it all together
- Significant post-formation optimization improves results [August99, Sias00]
- Predication enables simultaneous processing of independent paths of control
- Control speculation reduces predication overhead and enables general code motion
- Transformation trajectory and costs
 - Compounding of individually unprofitable steps yields profit in the end
 - Some “wasteful” expansion (inlined copies in potentially-lukewarm remainder loops)



Structural transformation highlights

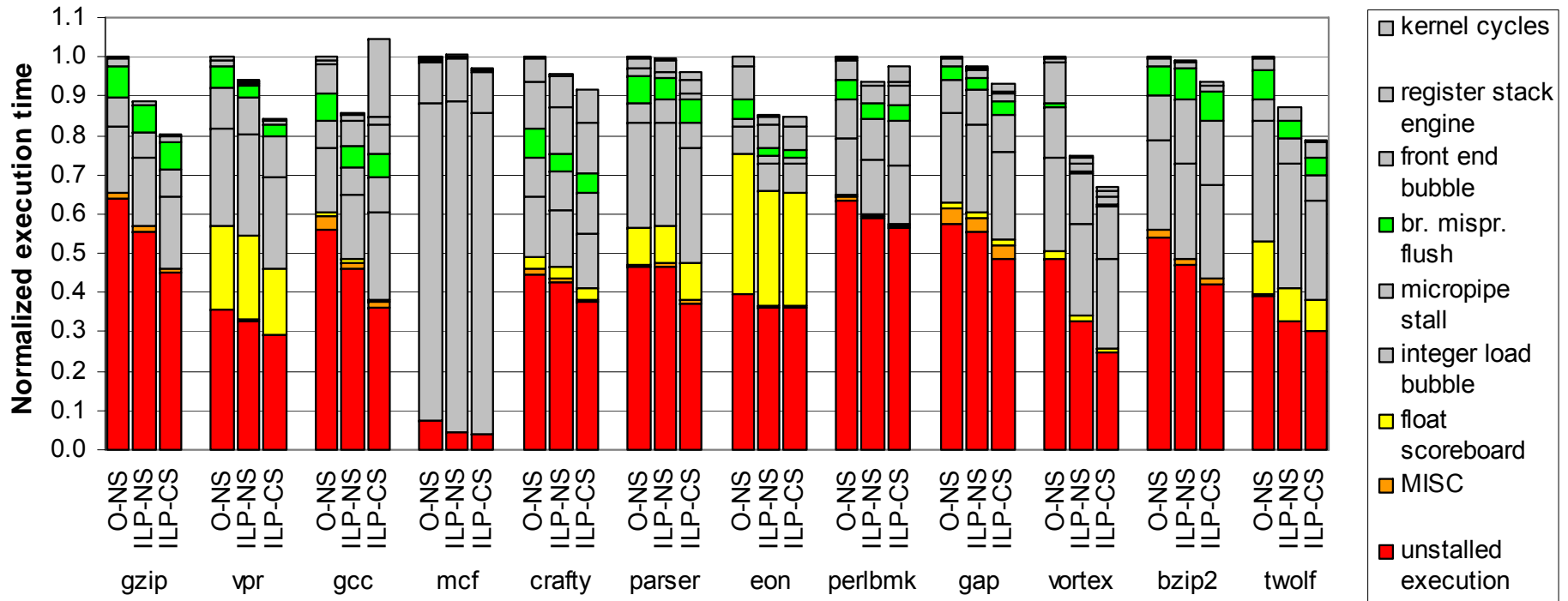
- Structural techniques form large execution regions for efficient ILP exploitation and optimization [Park98, Mahlke92, August98]
 - 27% of branches removed (*cf.* 7% in [Choi01])
- Techniques come at a cost
 - Inlining: up to 60% static code size growth
 - ILP transformations: 23% static code size growth
- Costs can become significant
 - Need to limit losses, but “safe” steps don’t get over “energy barriers” – some risk is unavoidable
 - Models had to evolve with apps and μ arch: difficult to balance SB heuristic system for Itanium 2 and SPECint2000 (I-cache effects, larger active footprints, and more pervasive control)
- Our working framework will help develop more strategic approaches: more selective, comprehensive transforms

In situ overview: speedup vs. gcc



- Combination of techniques deliver 1.55x (up to 2.3x) speedup over gcc

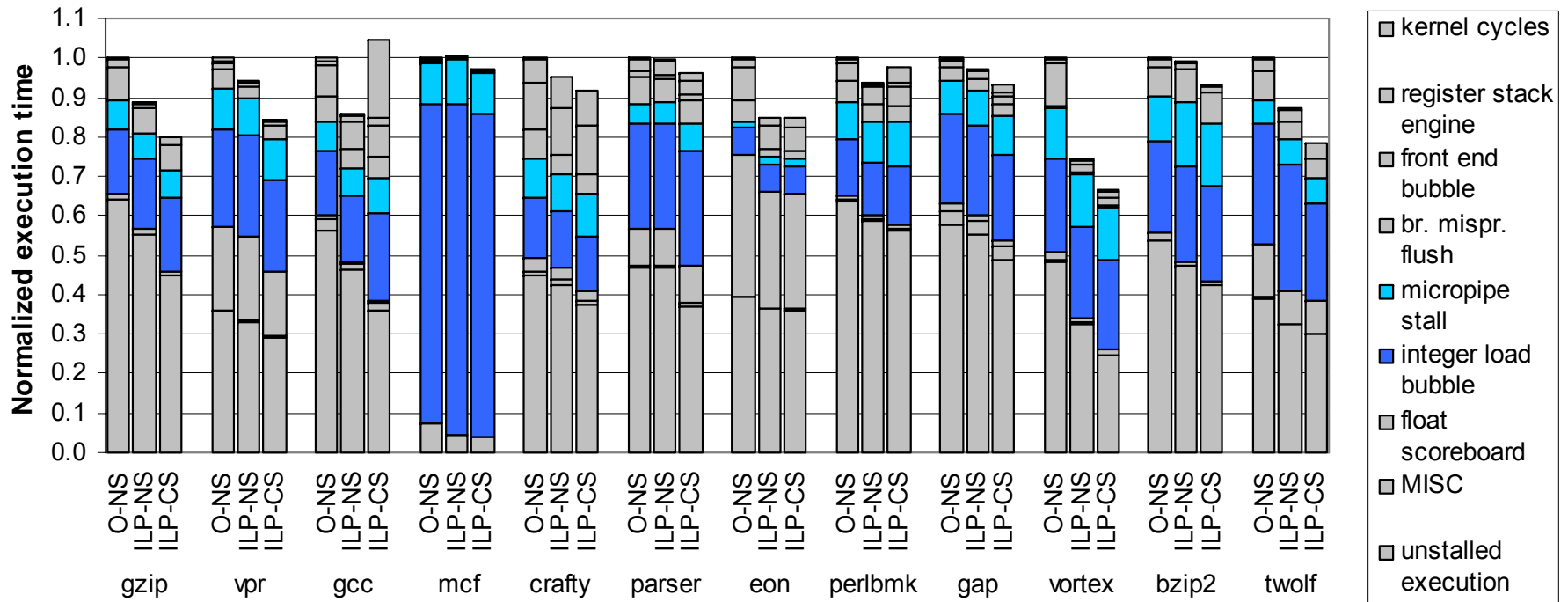
ILP compilation performance benefits



ILP techniques express planned execution in more parallel form and reduce runtime control overhead

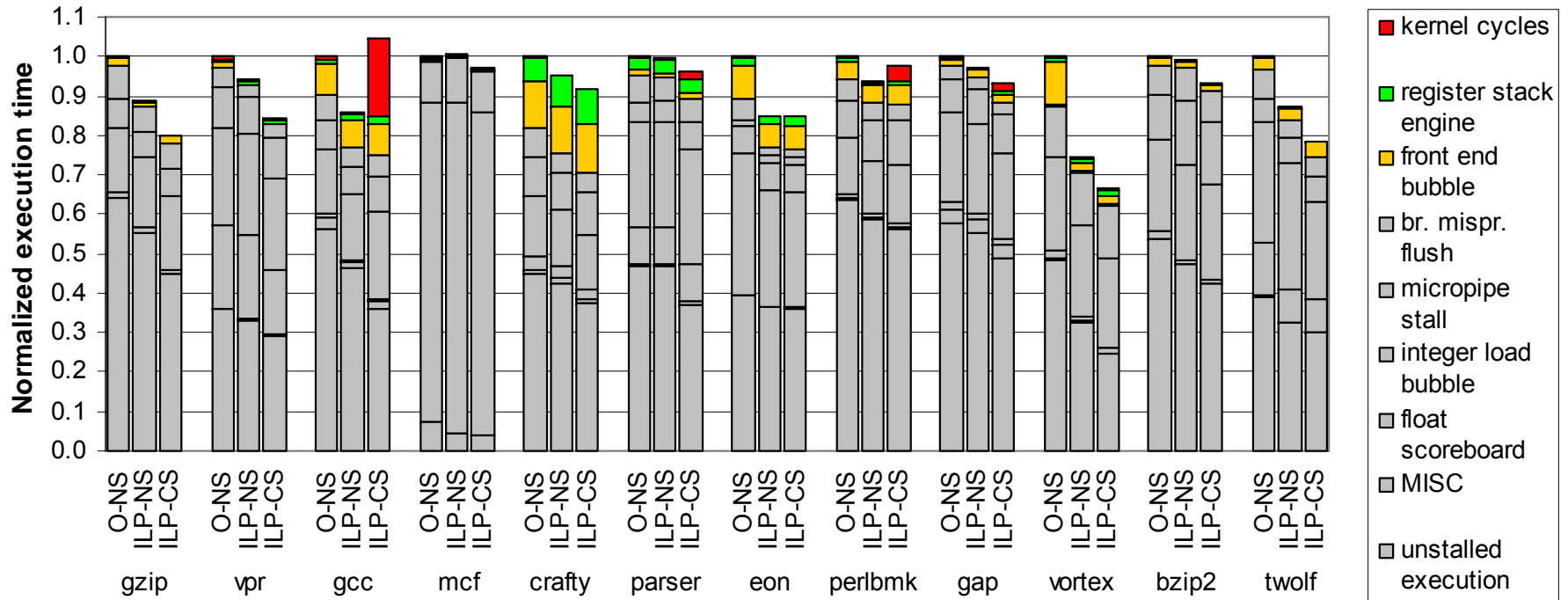
- *primarily* planned execution time (26% reduction)
- *secondarily* branch misprediction flush (22% reduction)

Little effect on data stall time



- Impact of off-path speculative accesses very minor
- Limited success hiding unknown latencies w/ slack (*twolf*)
- Data delivery anomalies dilute ILP benefits
 - “Micropipe” = TLB, bank and fill conflicts, etc.

ILP compilation potential overheads



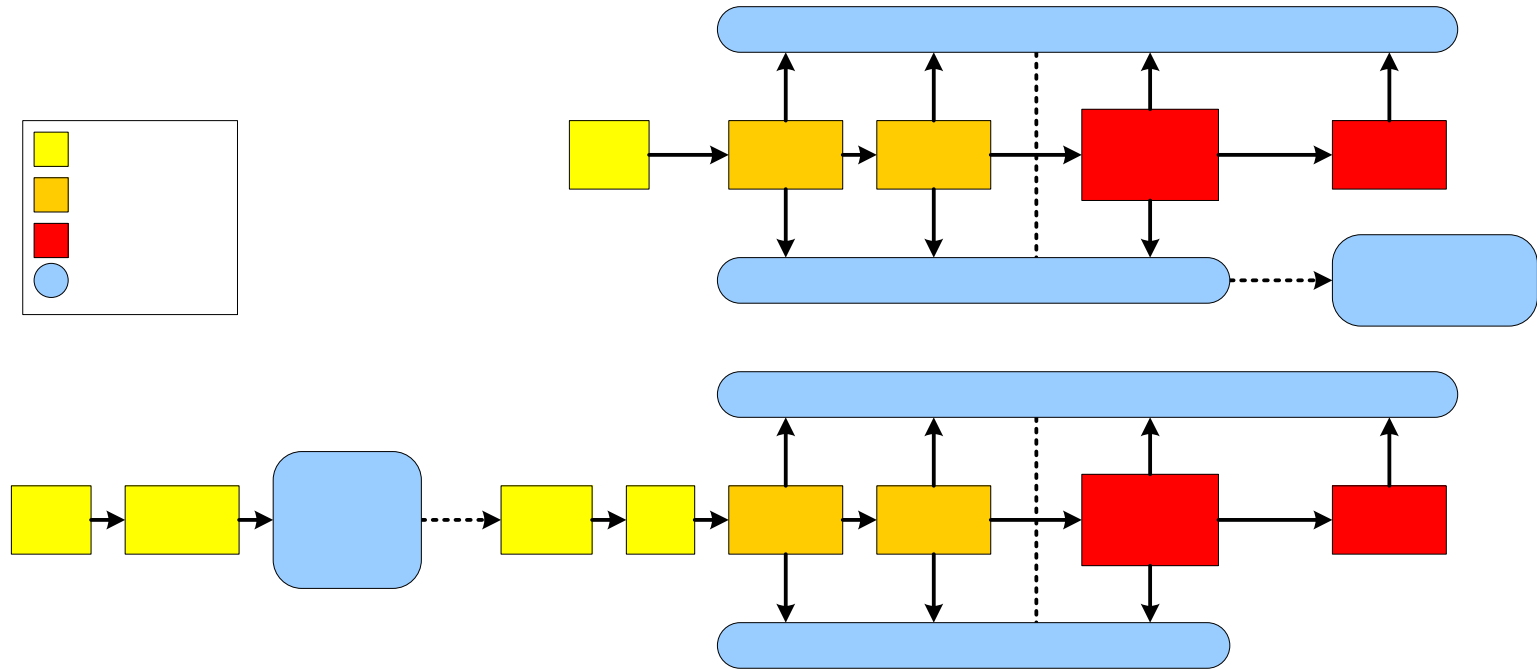
Instruction cache pressure managed

– expansion-countering effects and limits on growth



Handful of off-path “wild” loads raise spurious page faults

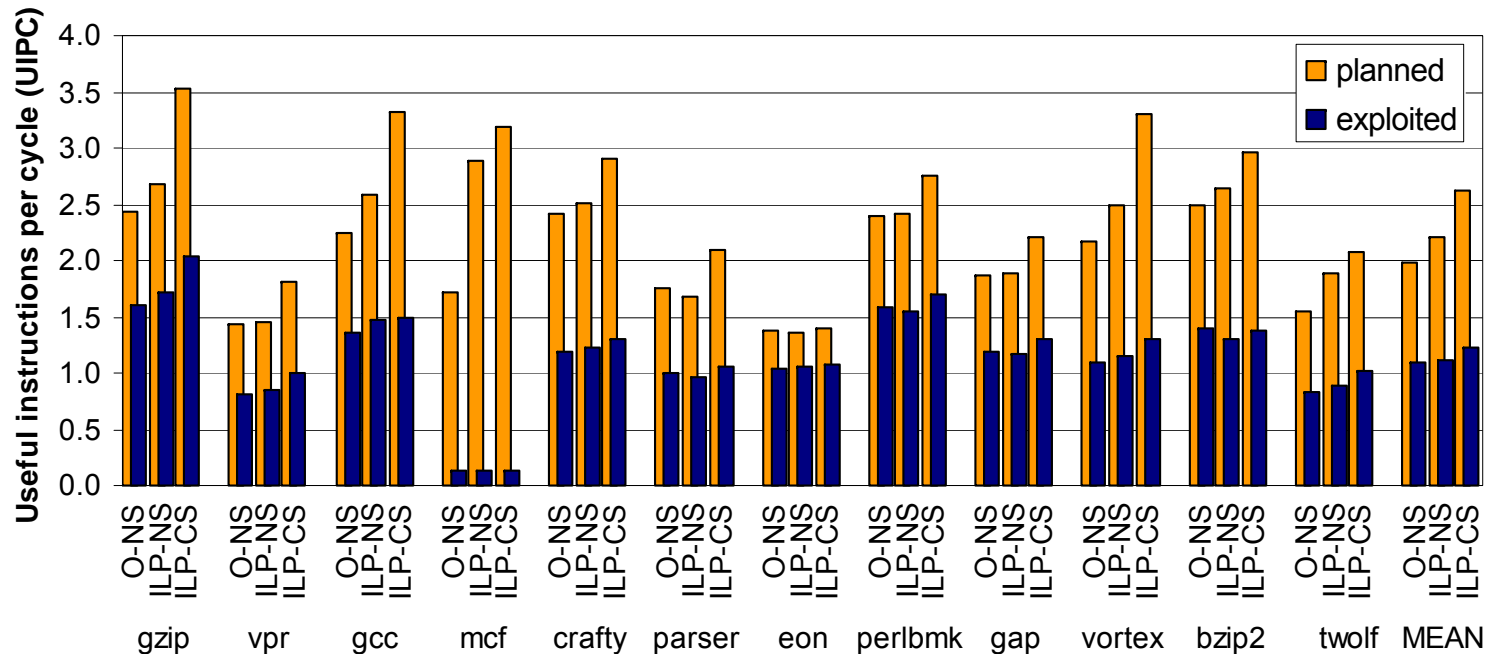
Detail: control speculation and “wild loads”



- IPF provides Sentinel w/ recovery code and general speculation
- Speculated “mixed type” loads can incur repeated, costly translation failures in general speculation model
 - work in progress on avoiding “dangerous” loads via analysis

General Sp

Taking stock of where we are



- Compiler effectively optimizes height it understands: 1.32x speedup
 - Consistent with previous “plan evaluating” results—after a lot of hard work!
- Net speedup is reduced by unanticipated dynamic effects to 1.13x
- Useful, but there exists potential for improvement
 - In the compiler (increasing planned ILP or understanding more effects)
 - In the micro-architecture (making good on the compiler’s plan, e.g. by providing some load-latency tolerance)

In conclusion...



Lessons learned

- How ILP-oriented analyses and transformations collaborate to deliver 1.55x average (and up to 2.3x) speedup over *gcc* on Itanium 2
- How research compiler techniques adapt to new benchmarks and “real world” system constraints



Future directions

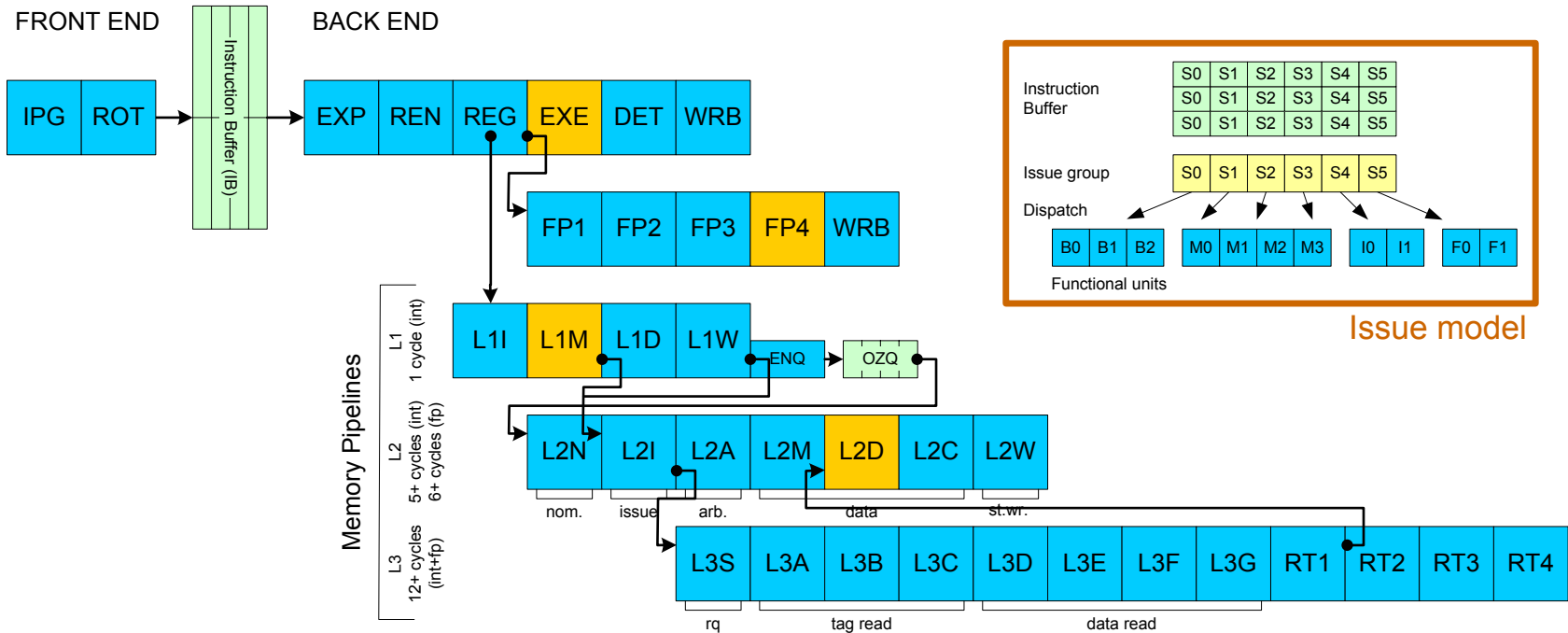
- Reducing general speculation “wild load” overhead with analysis
- Increasing memory tolerance – compiler and μ arch [Barnes03]
- Development of strategic approaches to optimization – more comprehensive transformations with better managed costs
 - Profile stability / independence



As efficiency concerns motivate leaning harder on the compiler, holistic consideration of application / architecture needs and the compiler’s capabilities is crucial!

Detail slides

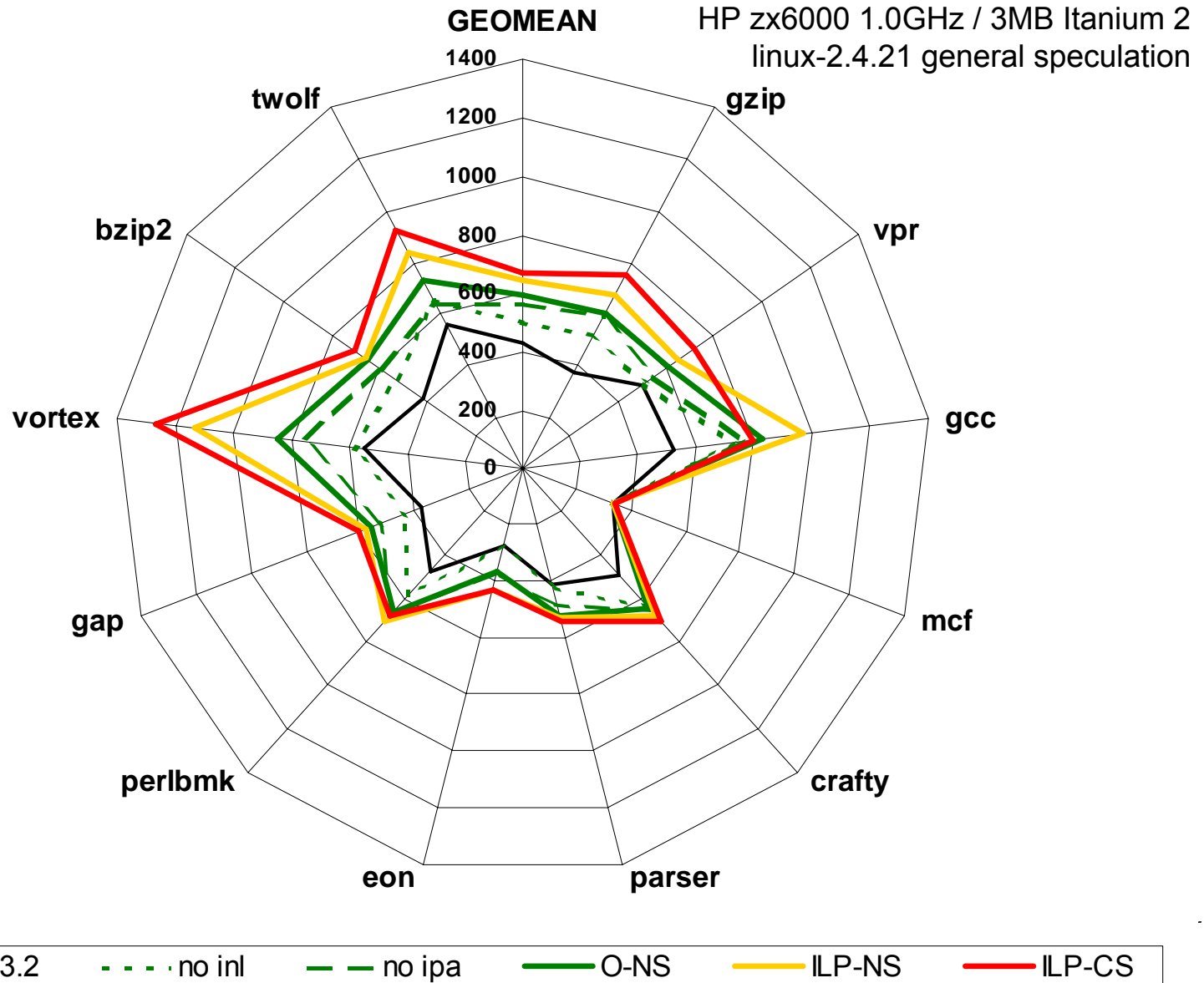
Itanium 2 considerations



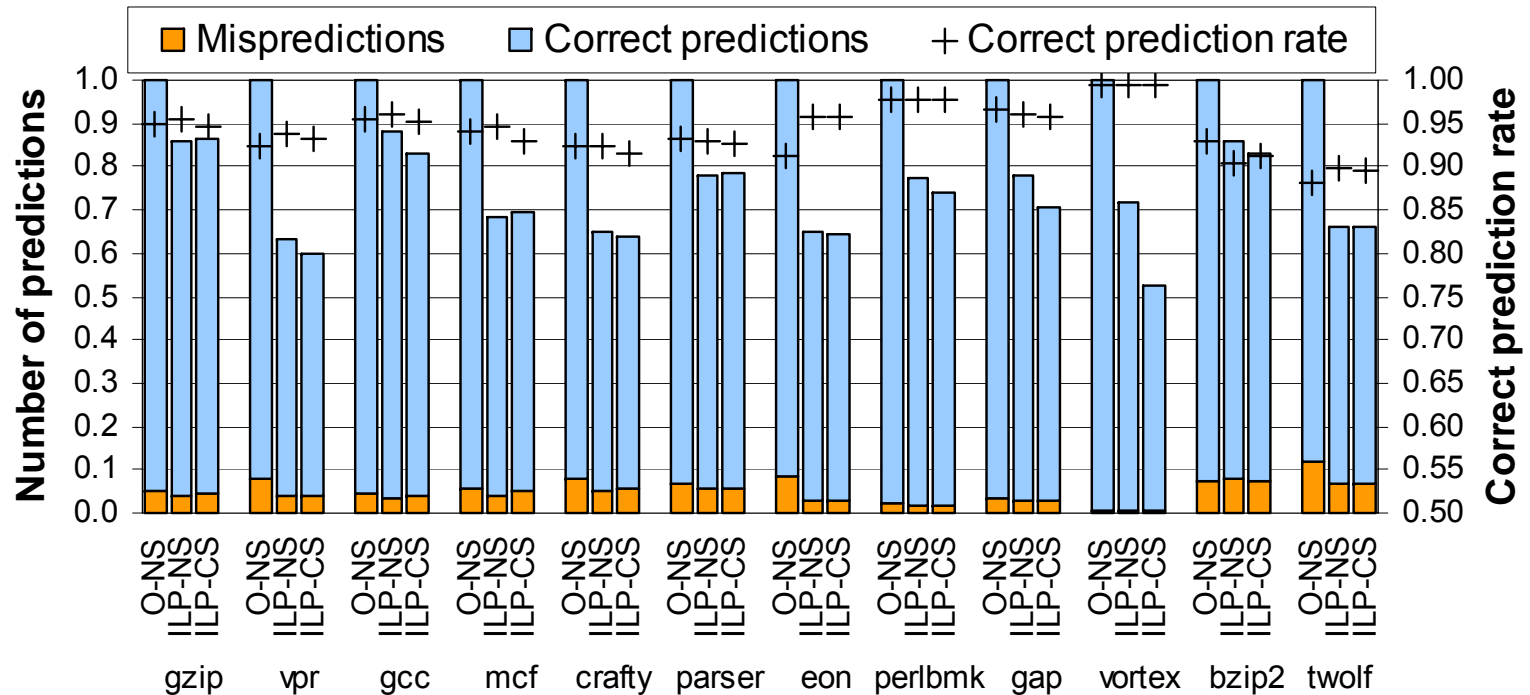
Itanium 2 Pipeline Organization (source: Intel docs and ISSCC presentations)

- In-order issue / out-of-order retirement implications
 - Lack of renaming → stall on WAW in scoreboard
 - Stall on DTLB miss until resolved (danger for speculation!)
- Compressed VLIW encoding (some explicit NOPs)
- In-order performance monitors have direct meaning

Performance comparison



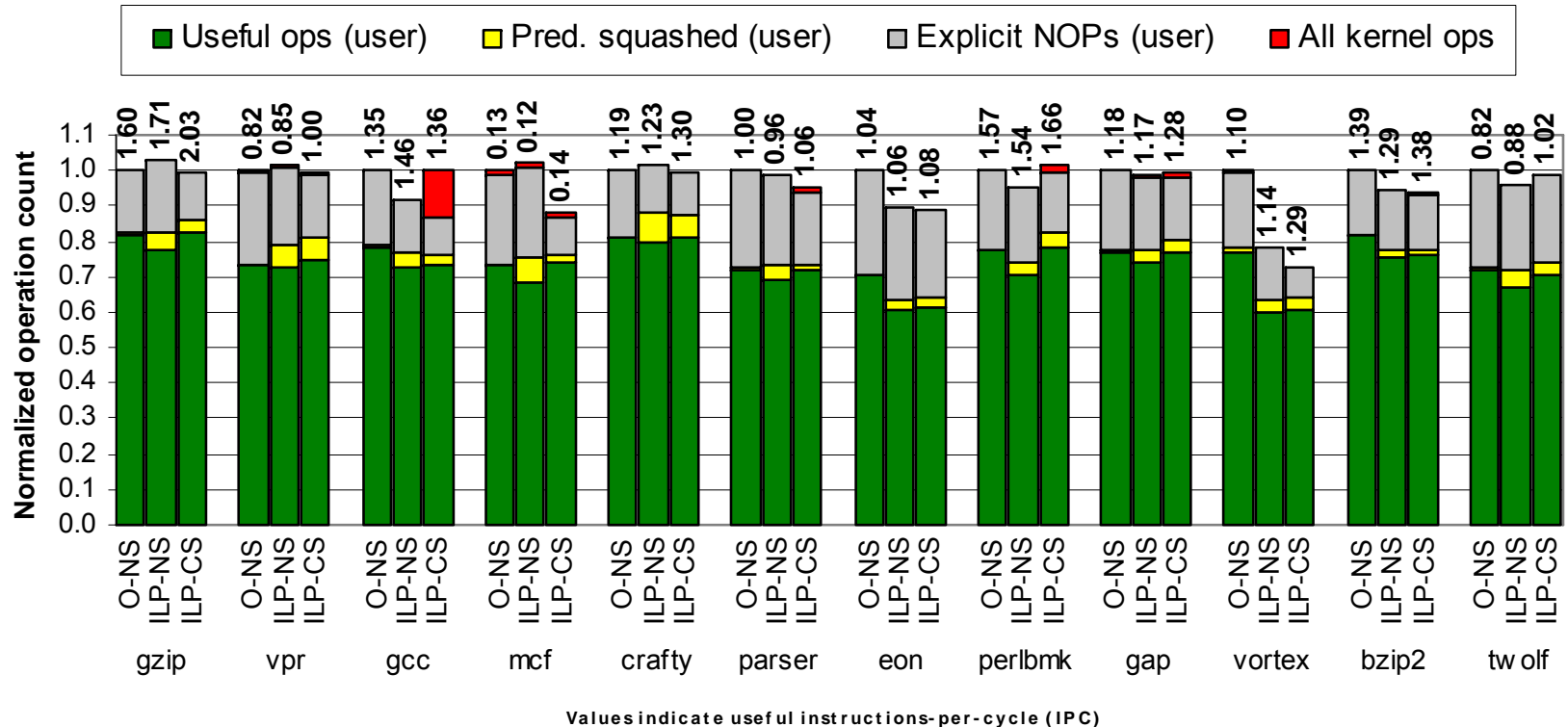
Branch prediction effects



ILP techniques restructure program control

- Branches reduced by 27%
- Misprediction stall cycles reduced by 22%

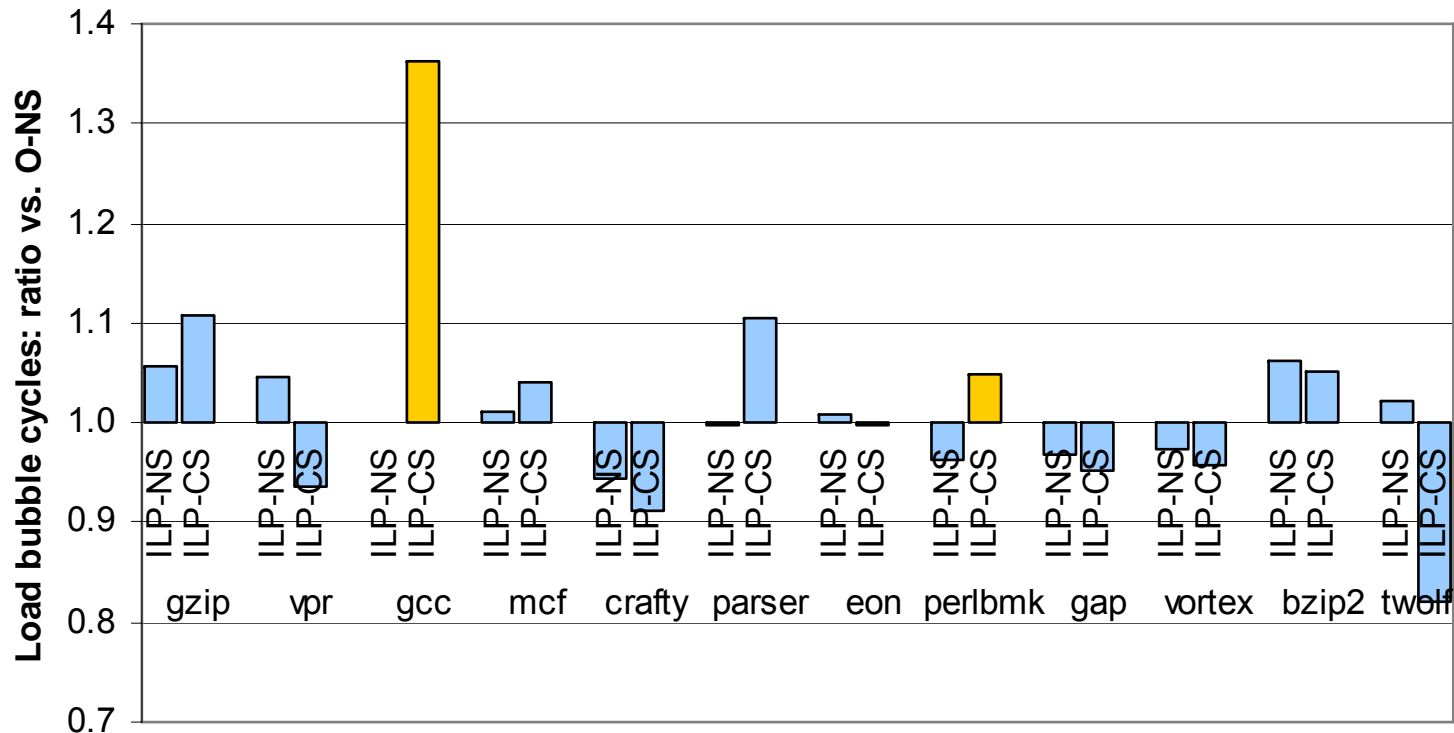
Operation accounting and IPC



General effect on # of “useful” dynamic operations

- Region formation: decrease
- Control speculation: increase

Integer data cache stall cycles

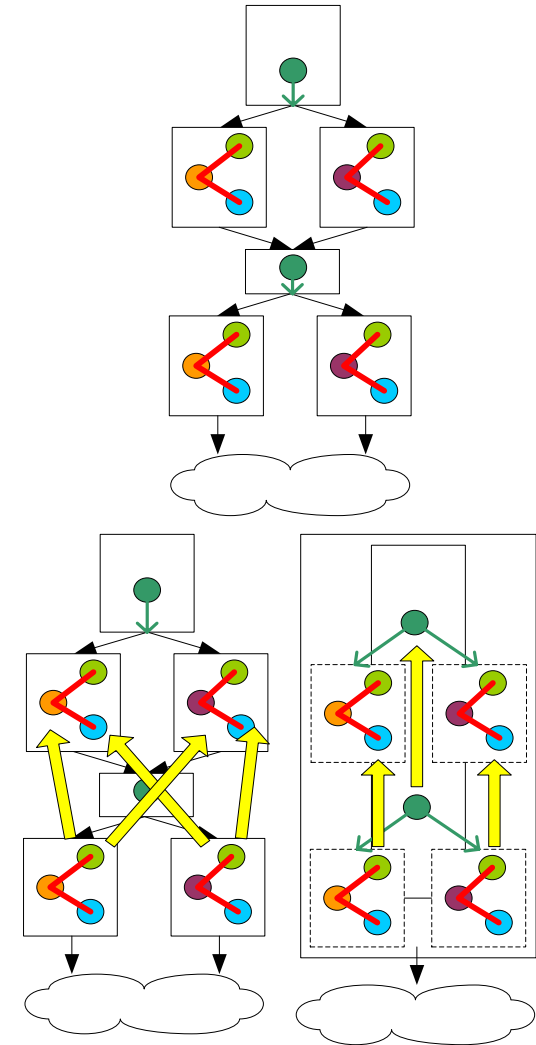


Mixed results

- Increase: speculative uses incur stalls
- Decrease: purchased scheduling slack hides latency
- Wild loads distort results in *gcc*, *perlbnk*

Using EPIC tools against control

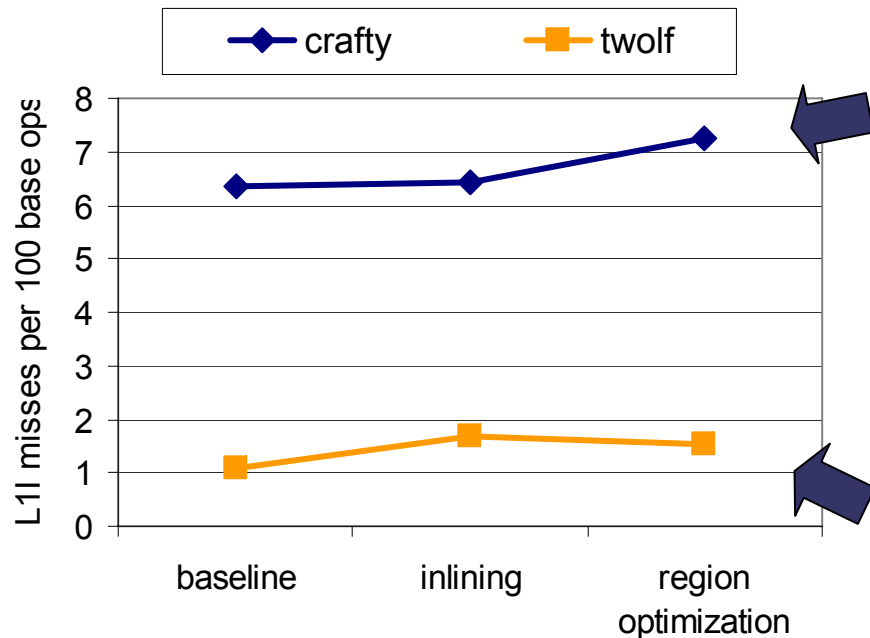
- Predication, speculation break limits of positional control
- Incremental (conservative) [Bharadwaj99]
 - EPIC tools improve global scheduling
 - Fills schedule gaps, but leaves control
 - +/- limited by short-leash constraints
- Structural (radical) [Park98, Mahkle92, August98]
 - Form large execution regions for efficient ILP exploitation – decouple position and control
 - Heavy control flow restructuring (27% of branches eliminated vs. 7% [Choi01])
- A variety of techniques are combined to achieve dramatic results
 - accumulation of costs (e.g. 23% static code size growth)
 - need to limit losses (cut off specialization)
 - “safe” steps don’t get over “energy barriers”



Why invest this effort?

- 2.60 IPC planned / 1.25 IPC achieved on a 6-issue μ P
- Some benchmarks need $\sim 1.5x$ additional speedup to compete with an out-of-order superscalar in the same technology
- Research techniques not fully implemented in production compilers
 - awaiting real-world validation: system effects, modern benchmarks, comparison against other compilers
- Future research will be more on-target with a “real world validated” starting point

Detail: context, compounding and I-cache



Specialization in low-reuse context hurts I-cache hit

Extensive inlining extracts a penalty, which subsequent optimization starts to offset

- ILP optimizations replicate, specialize
 - inlining, region optimization, loop development
 - More touched lines vs. better fetch/dispatch efficiency
 - Reuse context crucial [McFarling91]
- Optimizations interact in penalties and opportunities
 - Important to manage interactions in future work