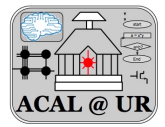
Accelerating Decoupled Look-ahead via Weak Dependence Removal: A Metaheuristic Approach

Raj Parihar, Michael C. Huang

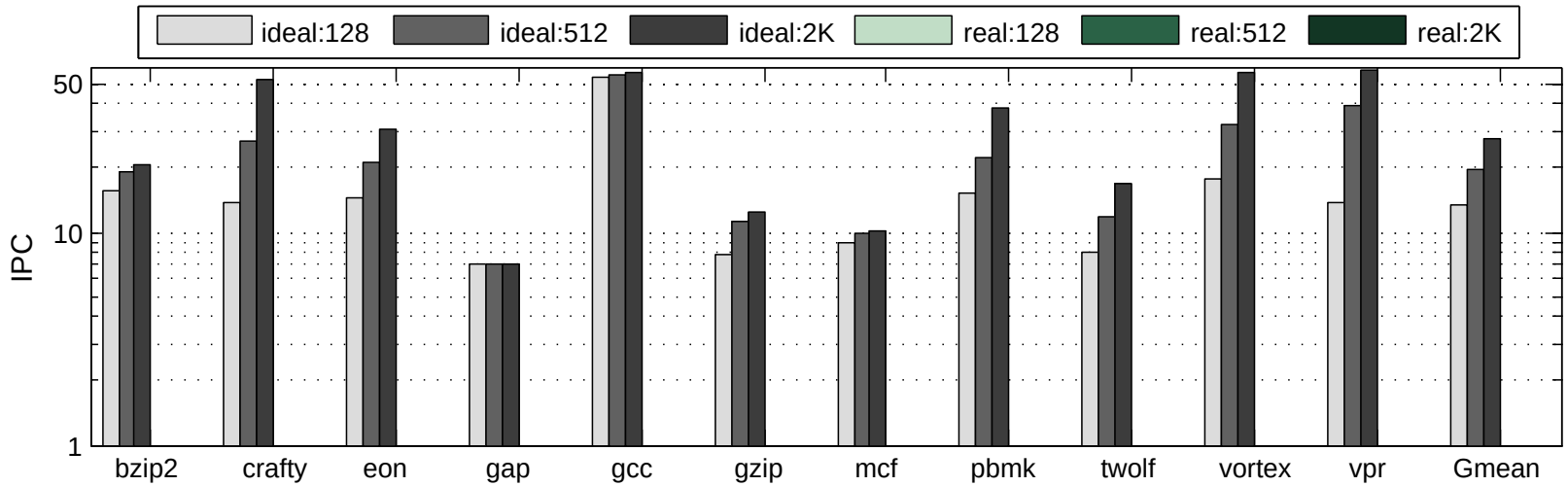
Department of Electrical & Computer Engineering

University of Rochester, Rochester, NY



Motivation

- Despite the proliferation of multi-core, multi-threaded systems
  - High single-thread performance is still an important design goal
- Modern programs do not lack instruction level parallelism
- Real challenge: exploit implicit parallelism without undue costs
- One effective approach: Decoupled look-ahead architecture

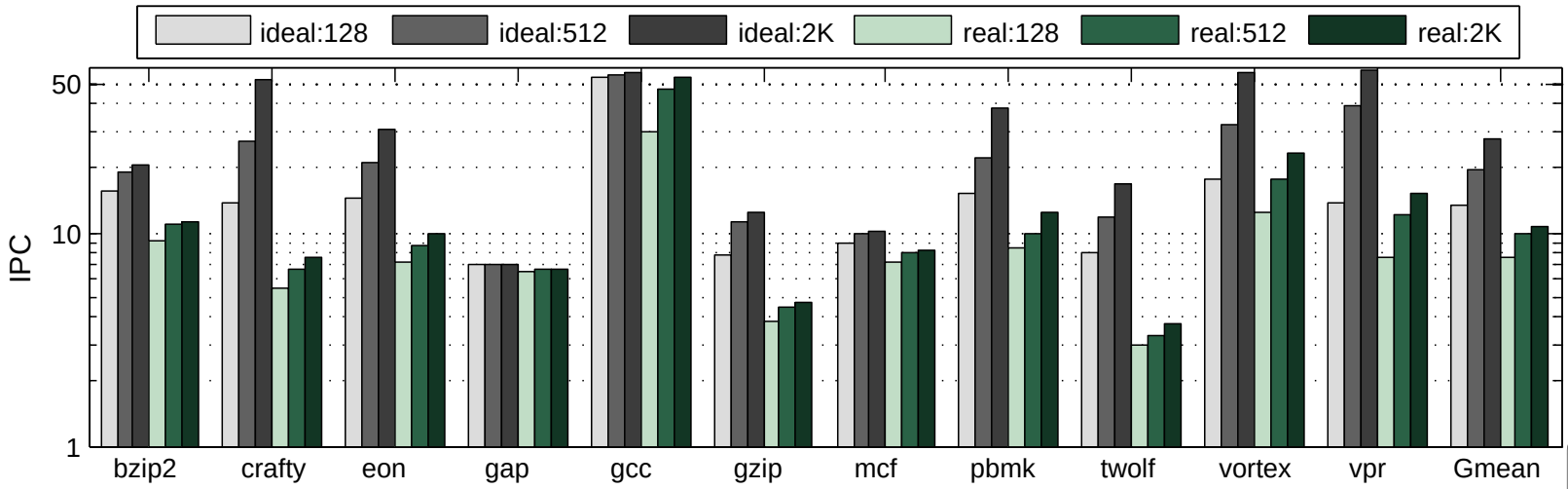


Motivation

- Decoupled look-ahead architecture targets
  - Performance hurdles: branch mispredictions, cache misses, etc.
  - Exploration of parallelization opportunities, dependence information
  - Microarchitectural complexity, energy inefficiency through decoupling

Motivation

- Decoupled look-ahead architecture targets
  - Performance hurdles: branch mispredictions, cache misses, etc.
  - Exploration of parallelization opportunities, dependence information
  - Microarchitectural complexity, energy inefficiency through decoupling



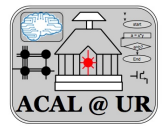
Motivation

- Decoupled look-ahead architecture targets
  - Performance hurdles: branch mispredictions, cache misses, etc.
  - Exploration of parallelization opportunities, dependence information
  - Microarchitectural complexity, energy inefficiency through decoupling
- The look-ahead thread can often become a ***new*** bottleneck

Motivation

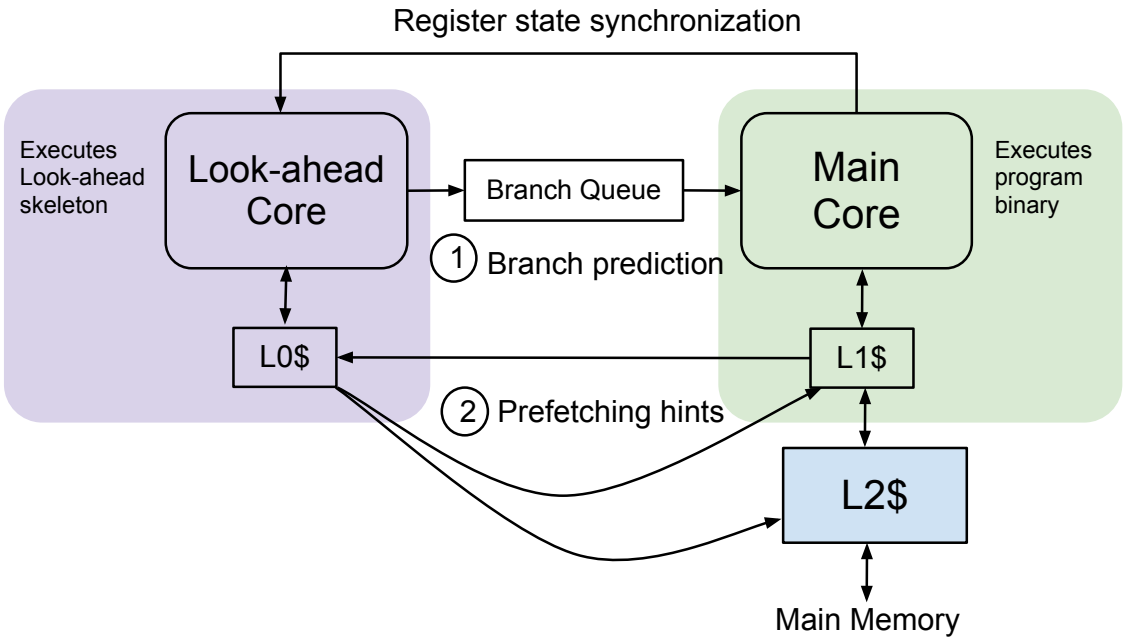
- Decoupled look-ahead architecture targets
  - Performance hurdles: branch mispredictions, cache misses, etc.
  - Exploration of parallelization opportunities, dependence information
  - Microarchitectural complexity, energy inefficiency through decoupling
- The look-ahead thread can often become a ***new*** bottleneck
- Lack of correctness constraint allows many optimizations
  - **Weak dependence**: Instructions that contribute marginally to the outcome can be removed w/o affecting the quality of look-ahead

Outline



Baseline Decoupled Look-ahead Architecture

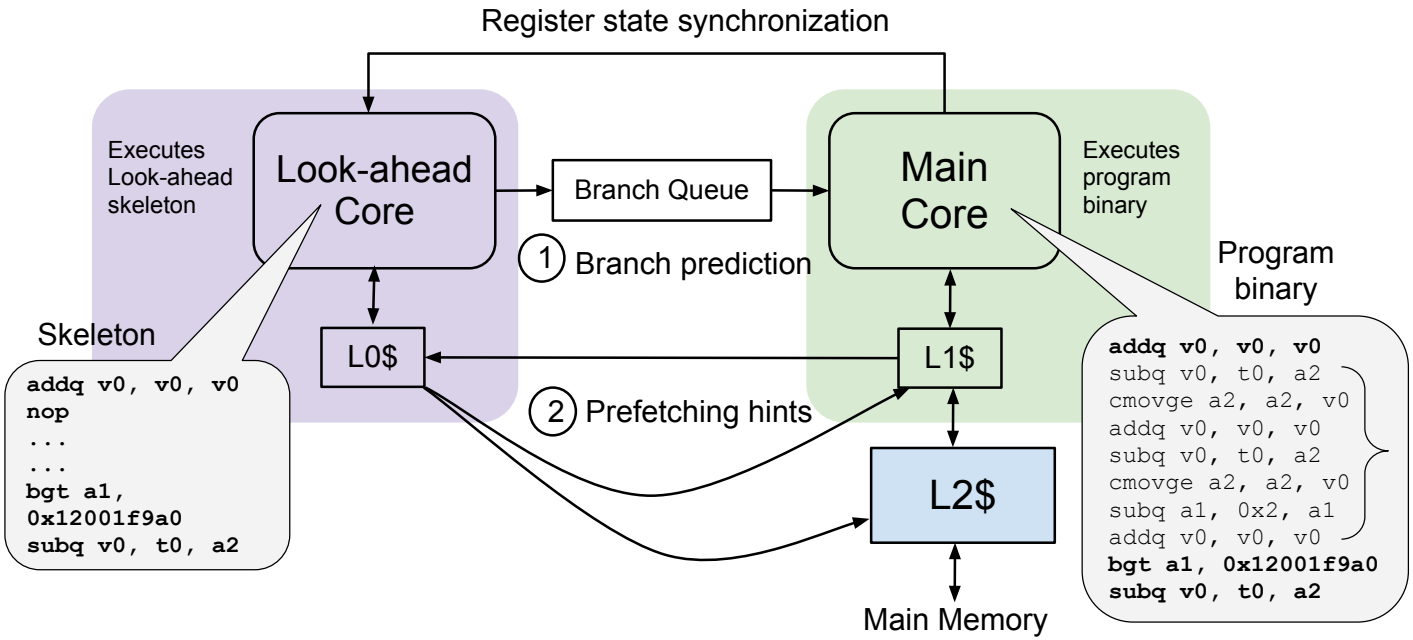
- Skeleton generated just for the look-ahead purposes
- The skeleton runs on a separate core and
  - Speculative state is completely contained within look-ahead context
  - Sends branch outcomes through FIFO queue; also helps prefetching



A. Garg and M. Huang, "A Performance-Correctness Explicitly Decoupled Architecture", *MICRO'08*

Baseline Decoupled Look-ahead Architecture

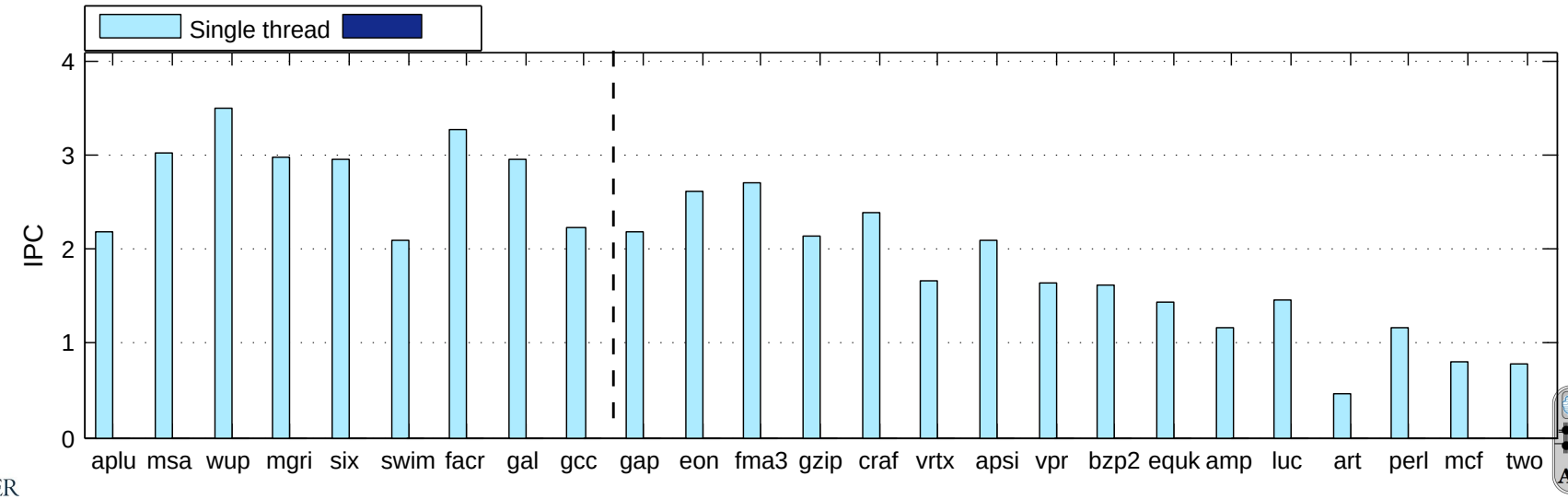
- **Skeleton** generated just for the look-ahead purposes
- The skeleton runs on a separate core and
  - Speculative state is completely contained within look-ahead context
  - Sends branch outcomes through FIFO queue; also helps prefetching



A. Garg and M. Huang, "A Performance-Correctness Explicitly Decoupled Architecture", *MICRO-08*

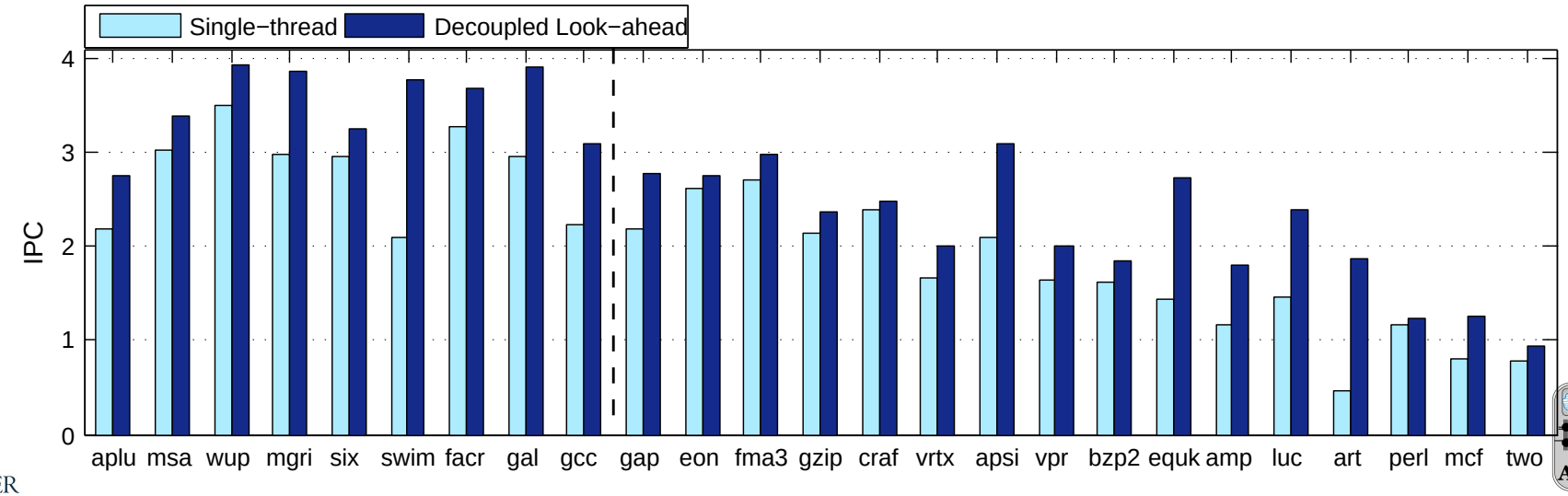
Look-ahead: A New Bottleneck

- Comparing four systems to discover new bottlenecks
  - **Single-thread**, decoupled look-ahead, ideal, and look-ahead limit
- Application categories:
  - Bottleneck removed or speed of look-ahead is not an issue
  - Look-ahead thread is the new bottleneck



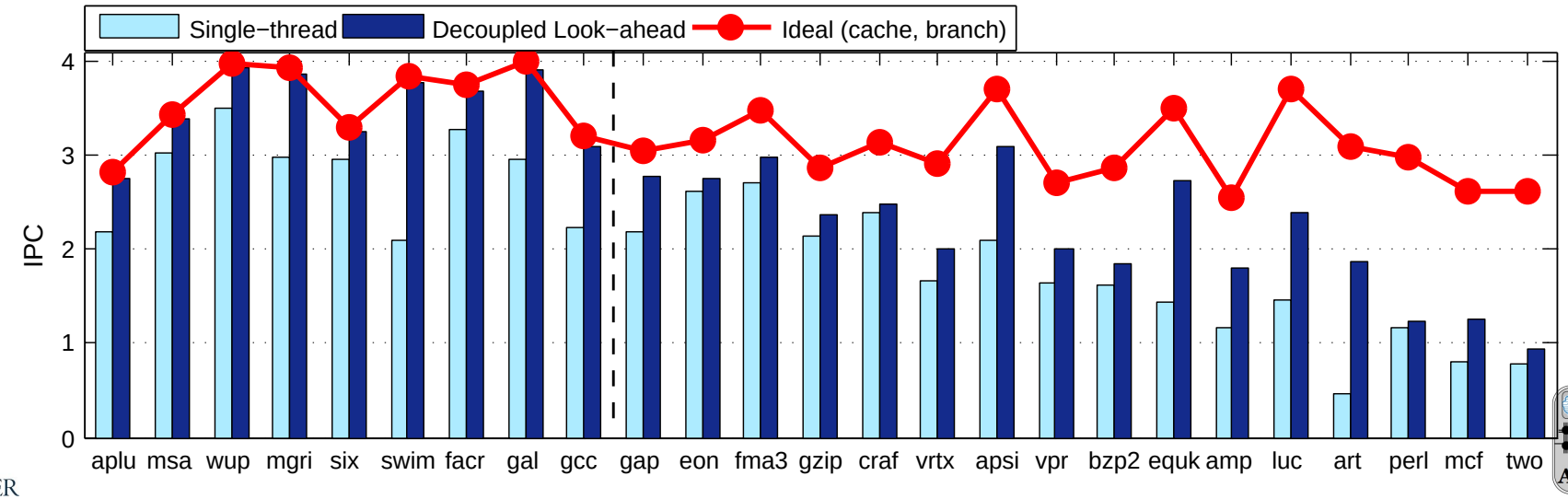
Look-ahead: A New Bottleneck

- Comparing four systems to discover new bottlenecks
  - Single-thread, **decoupled look-ahead**, ideal, and look-ahead limit
- Application categories:
  - Bottleneck removed; speed of look-ahead is not an issue (left half)
  - Look-ahead thread is the new bottleneck



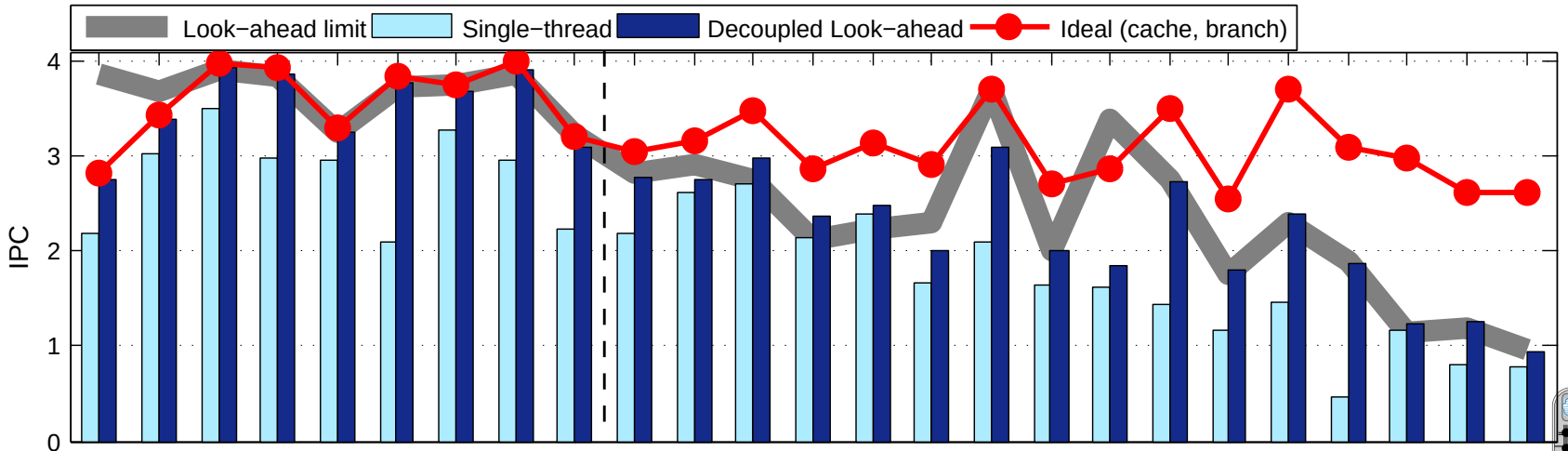
Look-ahead: A New Bottleneck

- Comparing four systems to discover new bottlenecks
  - Single-thread, decoupled look-ahead, **ideal**, and look-ahead limit
- Application categories:
  - Bottleneck removed or speed of look-ahead is not an issue
  - Look-ahead thread is the new bottleneck



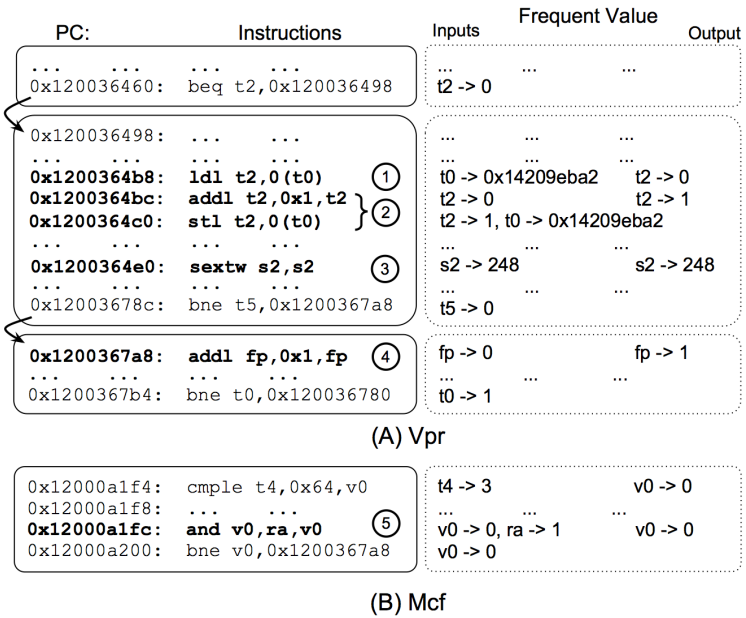
Look-ahead: A New Bottleneck

- Comparing four systems to discover new bottlenecks
  - Single-thread, decoupled look-ahead, ideal, and **look-ahead limit**
- Application categories:
  - Bottleneck removed or speed of look-ahead is not an issue
  - **Look-ahead thread is the new bottleneck (right half)**



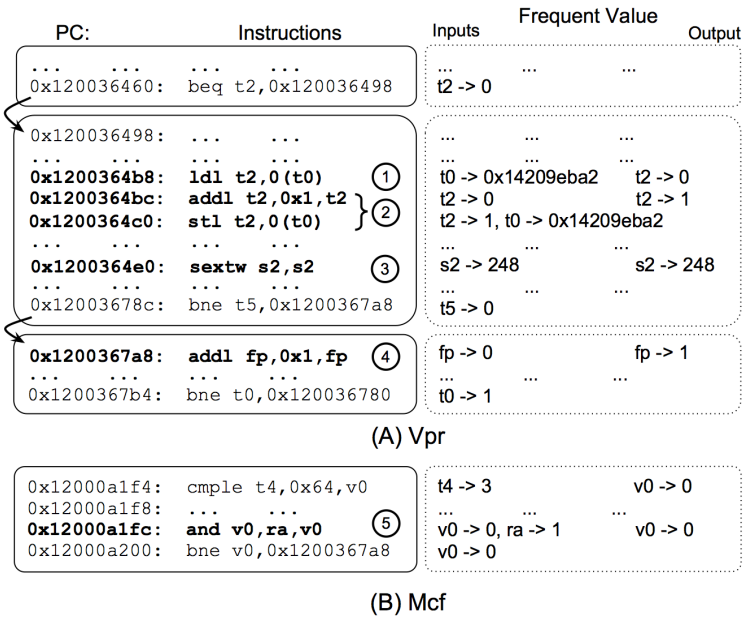
Weak Dependences/Instructions

- Not all instructions are equally important and critical
  - Example of weak instructions:
    - Inconsequential adjustments
    - Load and store instructions that are (mostly) silent
    - Dynamic NOP instructions



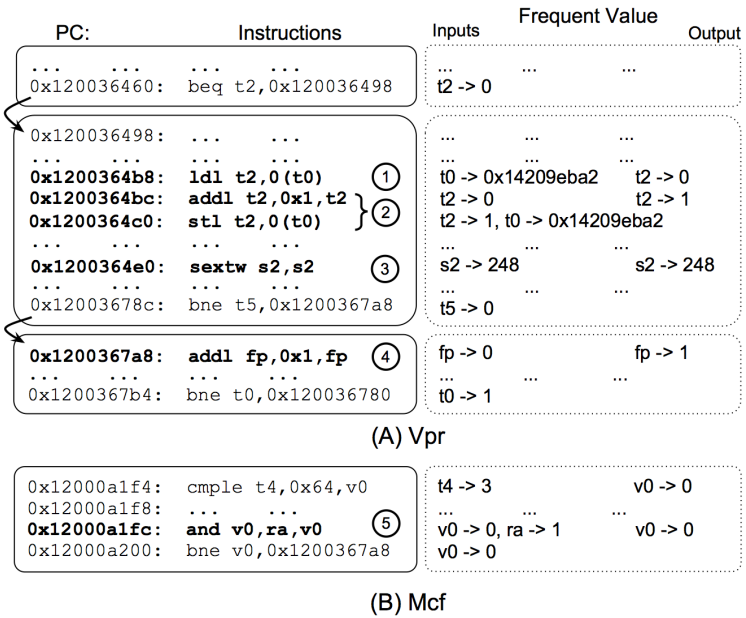
Weak Dependences/Instructions

- Not all instructions are equally important and critical
  - Example of weak instructions:
    - Inconsequential adjustments
    - Load and store instructions that are (mostly) silent
    - Dynamic NOP instructions
  - Plenty of weak instructions are present in programs (100s of)



Weak Dependences/Instructions

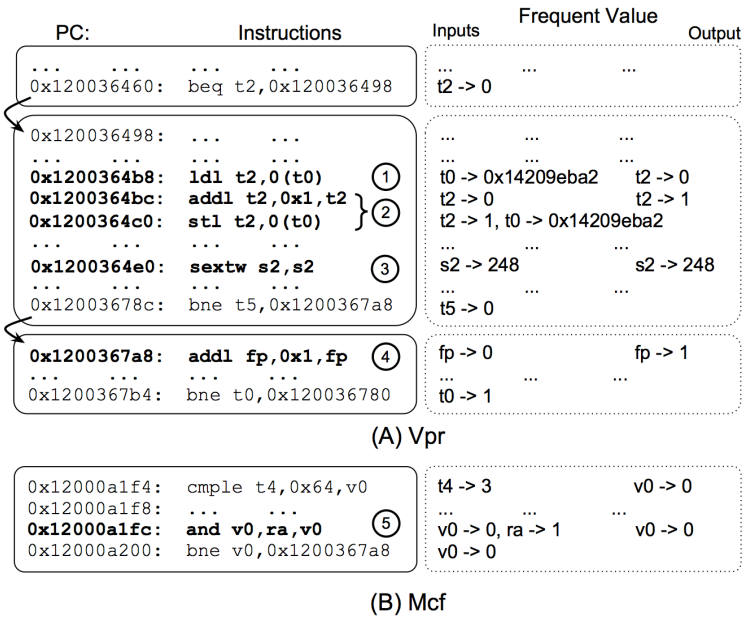
- Not all instructions are equally important and critical
  - Example of weak instructions:
    - Inconsequential adjustments
    - Load and store instructions that are (mostly) silent
    - Dynamic NOP instructions
  - Plenty of weak instructions are present in programs (100s of)



- Single weak instruction: their impact is non-trivial

Weak Dependences/Instructions

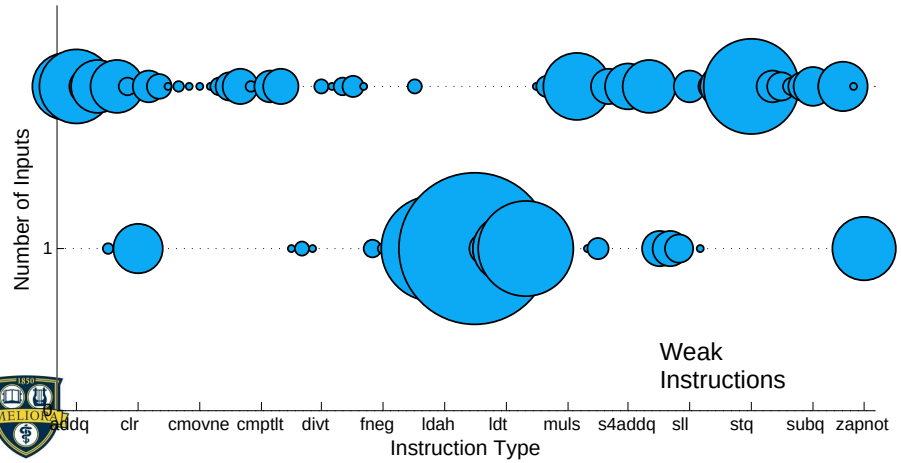
- Not all instructions are equally important and critical
  - Example of weak instructions:
    - Inconsequential adjustments
    - Load and store instructions that are (mostly) silent
    - Dynamic NOP instructions
  - Plenty of weak instructions are present in programs (100s of)

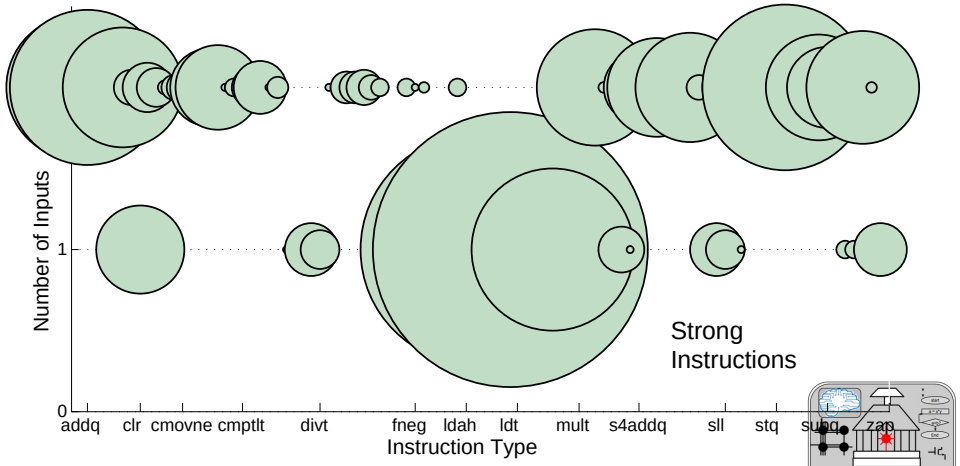


- Single weak instruction: their impact is non-trivial
- Weak instruction can be experimentally defined and their impact quantified in *isolation*

Challenge # 1: Weak insts do not look different

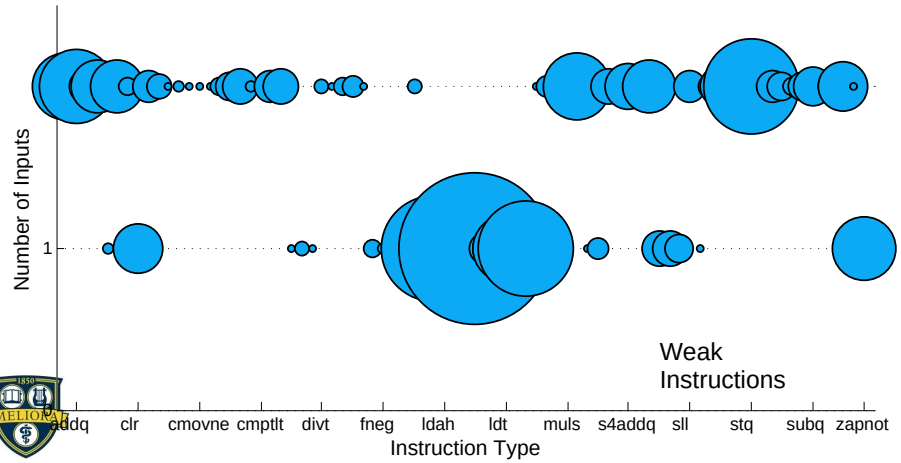
- After the fact analysis: based on static attributes of insts reveals
  - Static attributes of weak and regular insts are ***remarkably similar***
  - Correlation coefficient of the two distributions is very high (**0.96**)

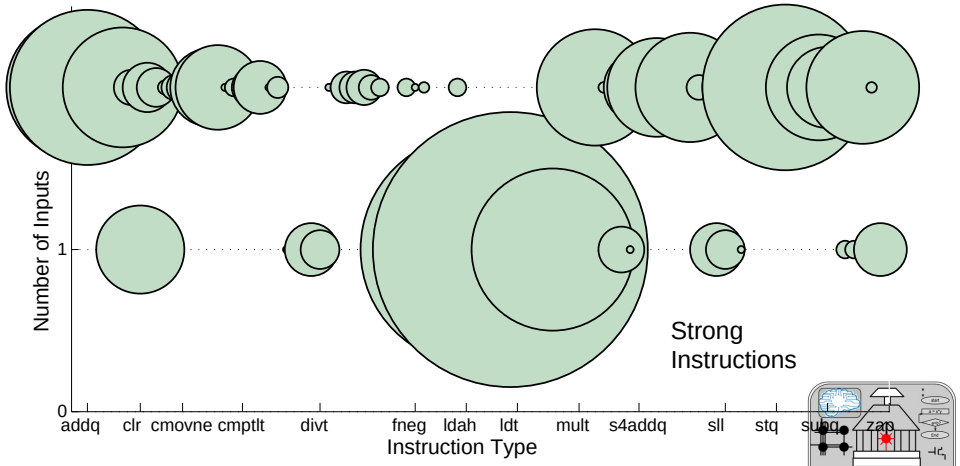




Challenge # 1: Weak insts do not look different

- After the fact analysis: based on static attributes of insts reveals
  - Static attributes of weak and regular insts are ***remarkably similar***
  - Correlation coefficient of the two distributions is very high (**0.96**)
- Weakness has very poor correlation with static attributes
  - Hard to identify the weak instructions through static heuristics





Challenge # 2: False positives are extremely costly

- After the fact analysis and close inspection also reveals
  - Some instructions are more likely to be weak than others
  - Even then, a single false positive can negate all the gains

Challenge # 2: False positives are extremely costly

- After the fact analysis and close inspection also reveals
  - Some instructions are more likely to be weak than others
  - Even then, a single false positive can negate all the gains

- Case in point: `zapnot` in *gap*

```
zapnot Ra Rb Rc
```

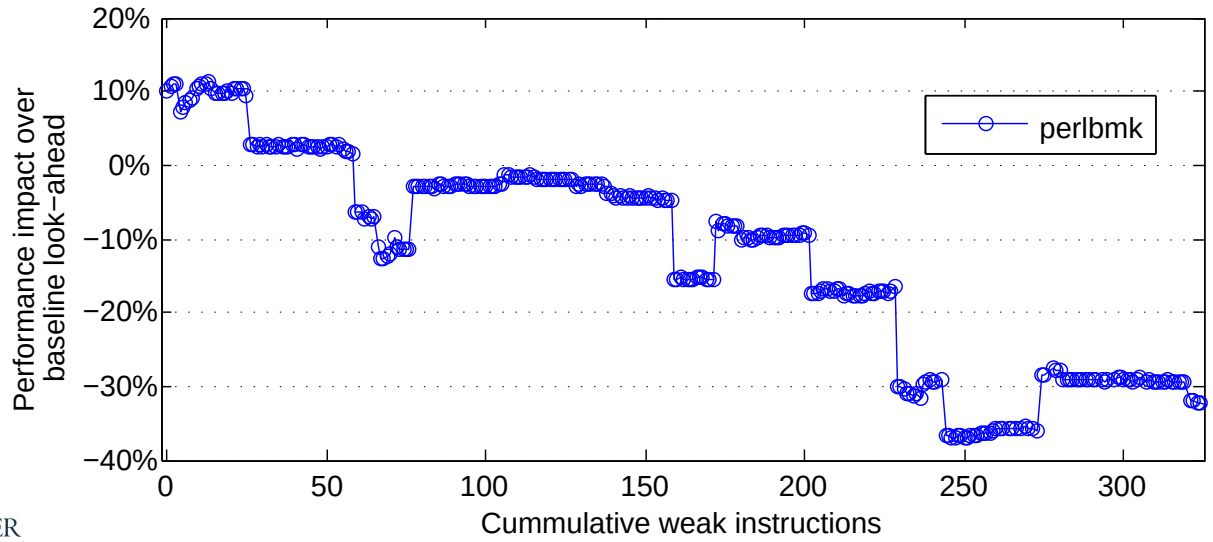
  - 84% of the `zapnot` insts are weak in isolation: 3.4% speedup
  - Single false positive `zapnot` instruction: 6% slowdown
  - More than 1 false positive instructions can slowdown upto 13%

Challenge # 3: Neither absolute nor additive

- Weakness is context dependent, non-linear – much like Jenga
  - All weak instructions combined together are not weak!

Challenge # 3: Neither absolute nor additive

- Weakness is context dependent, non-linear – much like Jenga
  - All weak instructions combined together are not weak!
- Example: weak instruction combining in *perlbmk*
  - About 300 weak instructions when tested in isolation
  - All combined together can result in up to 40% slowdown



Metaheuristic Based Trail-and-Error Approach

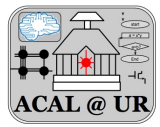
- Recap: **Challenges** in identifying weak instructions
  - Weak instructions look very similar to regular instructions
  - False positives are extremely costly and can negate all the gain
  - Weakness is context dependent: neither absolute nor additive

Metaheuristic Based Trail-and-Error Approach

- Recap: **Challenges** in identifying weak instructions
  - Weak instructions look very similar to regular instructions
  - False positives are extremely costly and can negate all the gain
  - Weakness is context dependent: neither absolute nor additive
- Our approach: **Metaheuristic based self-tuning**
  - Experimentally identify/verify weakness
  - Search for profitable combination via metaheuristic

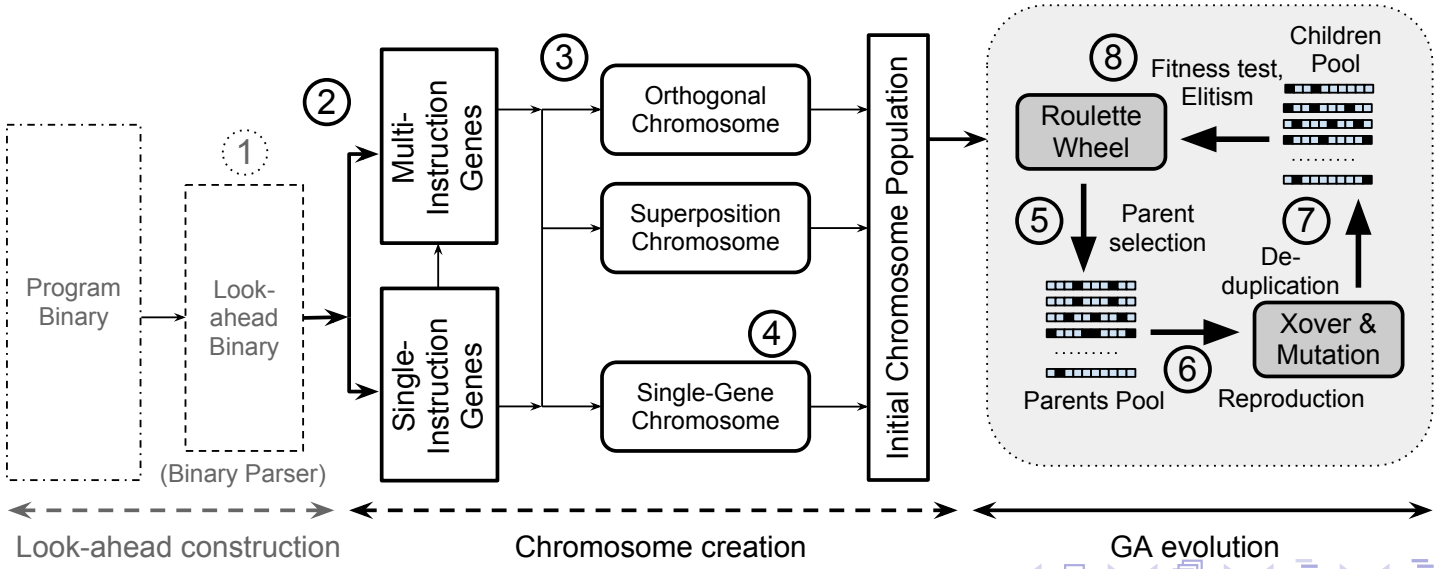
Metaheuristic Based Trail-and-Error Approach

- Recap: **Challenges** in identifying weak instructions
  - Weak instructions look very similar to regular instructions
  - False positives are extremely costly and can negate all the gain
  - Weakness is context dependent: neither absolute nor additive
- Our approach: **Metaheuristic based self-tuning**
  - Experimentally identify/verify weakness
  - Search for profitable combination via metaheuristic
- Metaheuristic: Completely agnostic of meaning of solution
  - Derive new solutions from current solutions through modifications
  - Example: genetic algorithm, simulated annealing, etc.



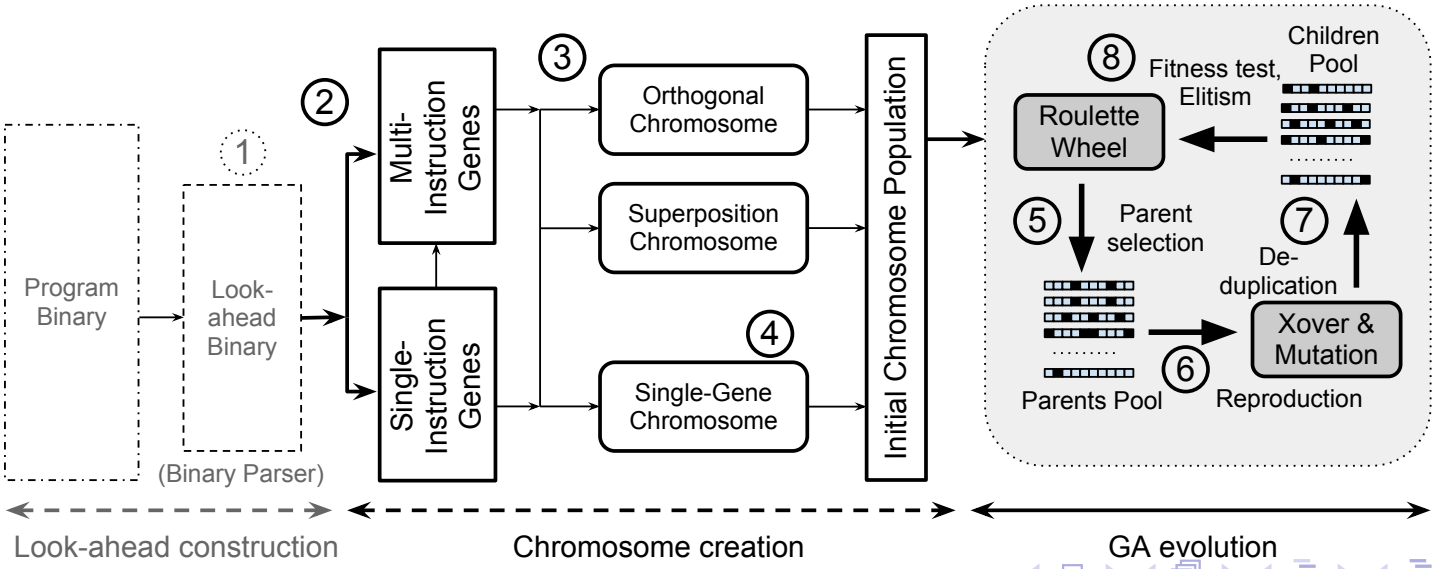
Genetic Algorithm based Framework

- The problem naturally maps to genetic algorithm



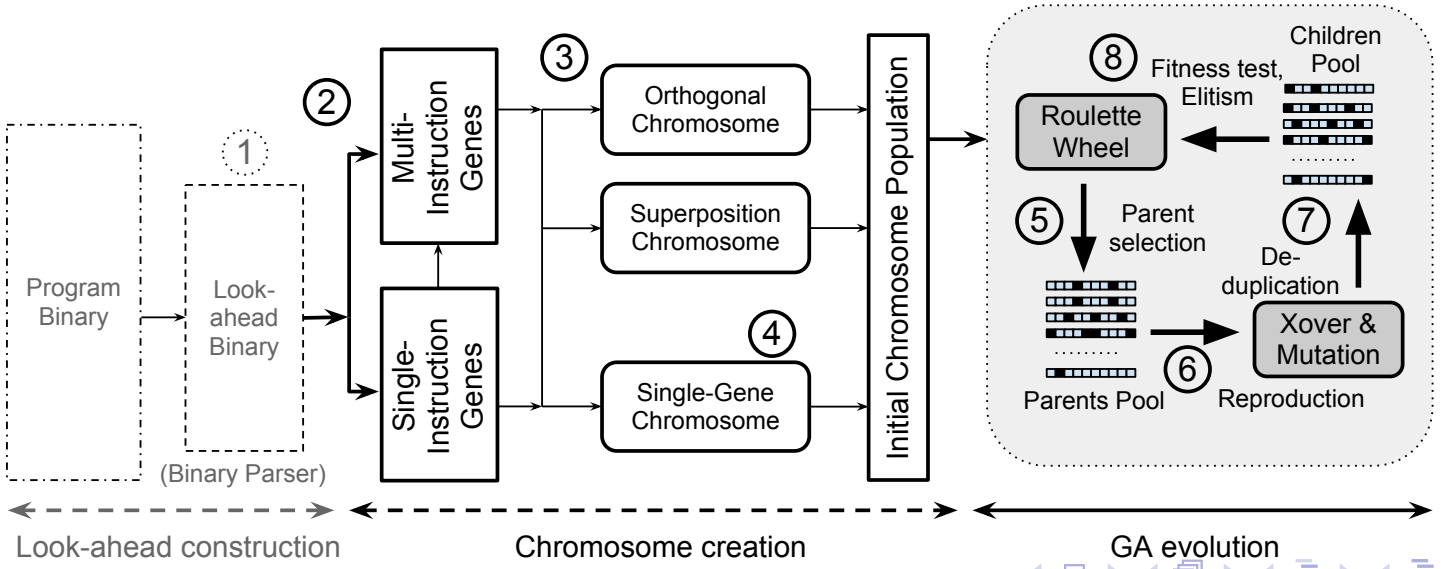
Genetic Algorithm based Framework

- The problem naturally maps to genetic algorithm
  - Skeleton is represented by a bit vector
  - Natural mapping: weak inst → gene, collection → chromosome
  - Objective: find optimal combination (chromosome)



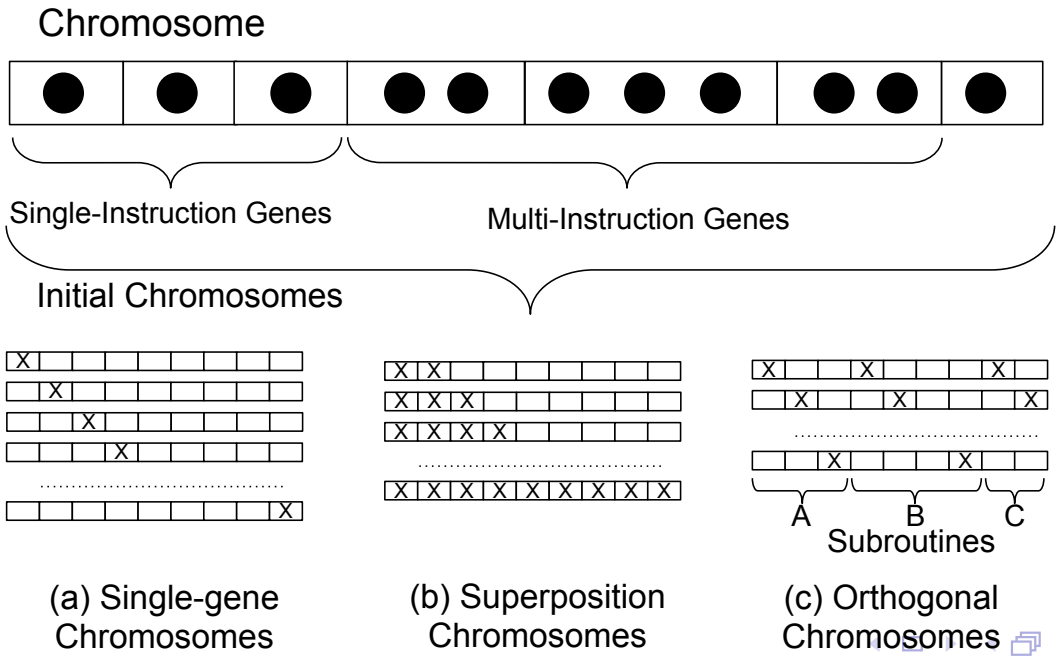
Genetic Algorithm based Framework

- The problem naturally maps to genetic algorithm
  - Skeleton is represented by a bit vector
  - Natural mapping: weak inst → gene, collection → chromosome
  - Objective: find optimal combination (chromosome)
- Genetic evolution: Procreation, mutation, fitness-based selection



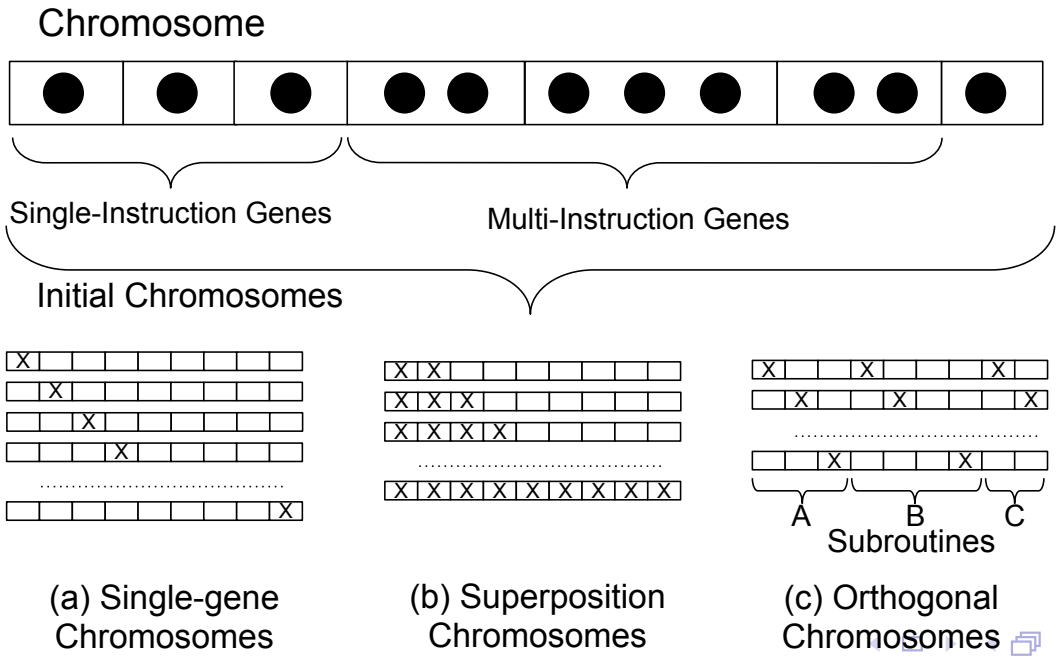
Hybridization: Heuristically Designed Initial Solutions

- Genetic evolution could be a slow and lengthy process
  - Heuristic based solutions are helpful to ***jump start*** the evolution



Hybridization: Heuristically Designed Initial Solutions

- Genetic evolution could be a slow and lengthy process
  - Heuristic based solutions are helpful to ***jump start*** the evolution
- Heuristically designed solutions in our system:
  - Superposition chromosome; Orthogonal subroutine chromosome



Experimental Setup

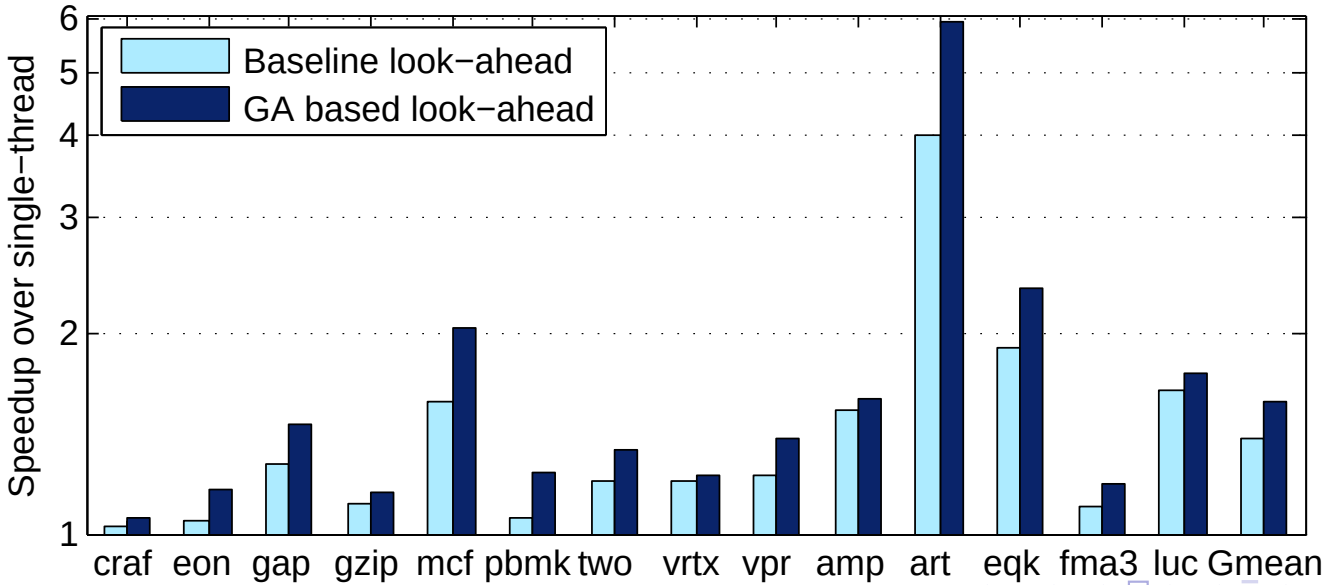
- Program/binary analysis tool: ALTO
- Simulator: detailed out-of-order, cycle-level in-house
  - SMT, look-ahead and speculative parallelization support
  - True execution-driven simulation (faithfully value modeling)
- Genetic algorithm framework
  - Modeled as offline and online extension to the simulator

Microarchitectural configurations:

Baseline core (Similar to POWER5)	
Fetch/Decode/Issue/Commit	8 / 4 / 6 / 6
ROB	128
Functional units	INT 2+1 mul +1 div, FP 2+1 mul +1 div
Fetch Q/ Issue Q / Reg. (int,fp)	(32, 32) / (32, 32) / (80, 80)
LSQ(LQ,SQ)	64 (32,32) 2 search ports
Branch predictor	Gshare – 8K entries, 13 bit history
Br. mispred. penalty	at least 7 cycles
L1 data cache (private)	32KB, 4-way, 64B line, 2 cycles, 2 ports
L1 inst cache (private)	64KB, 2-way, 128B, 2 cycles
L2 cache (shared)	1MB, 8-way, 128B, 15 cycles
Memory access latency	200 cycles
Look-ahead core:	Baseline core with only LQ, no SQ L0 cache: 32KB, 4-way, 64B line, 2 cycles Round trip latency to L1: 6 cycles
Communication:	Branch Output Queue: 512 entries Reg copy latency (recovery): 64 cycles

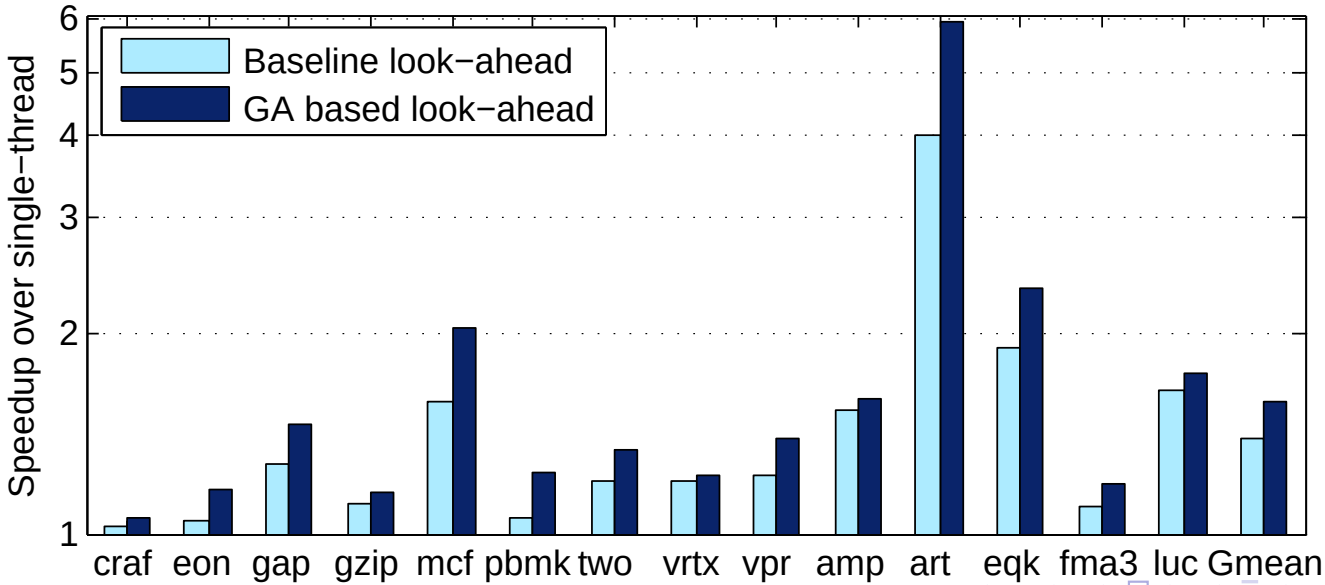
Speedup of Self-tuned Look-ahead

- Applications in which the look-ahead thread is a bottleneck



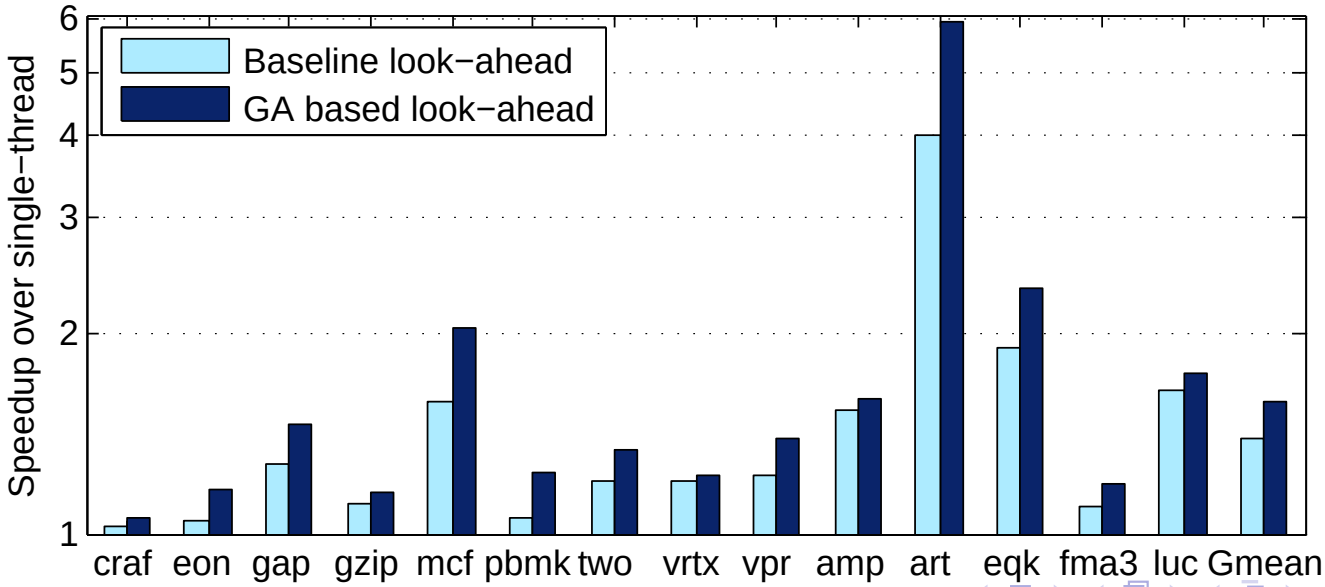
Speedup of Self-tuned Look-ahead

- Applications in which the look-ahead thread is a bottleneck
- Self-tuned, genetic algorithm based decoupled look-ahead
  - Speedup over baseline decoupled look-ahead: **1.14x** (geomean)



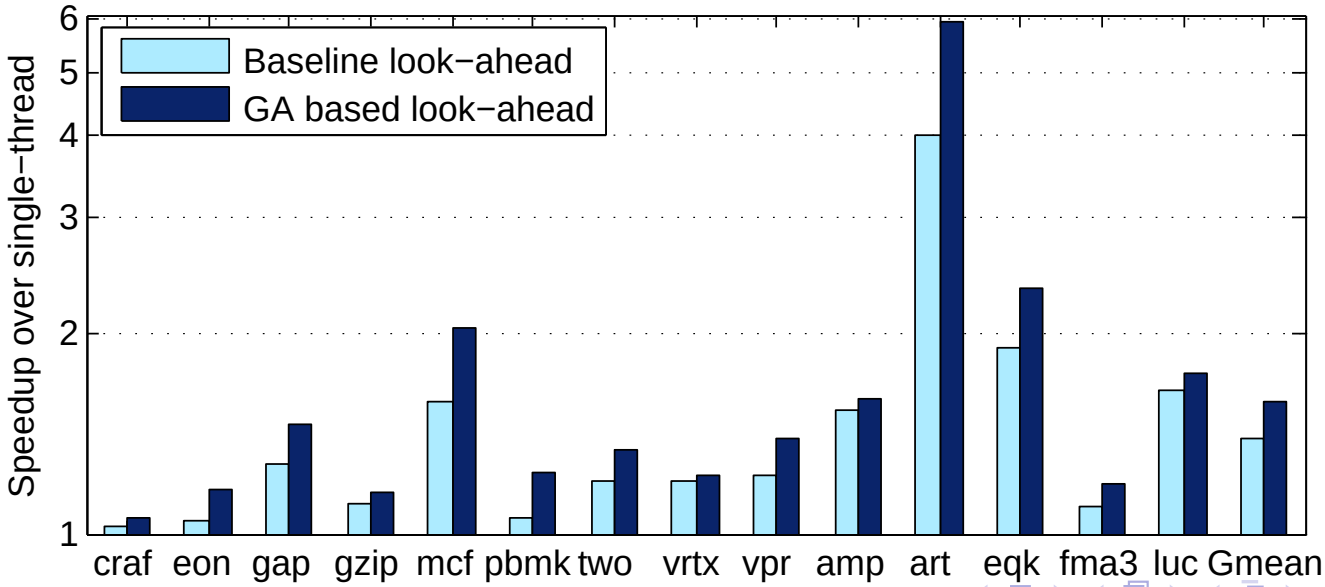
Speedup of Self-tuned Look-ahead

- Applications in which the look-ahead thread is a bottleneck
- Self-tuned, genetic algorithm based decoupled look-ahead
  - Speedup over baseline decoupled look-ahead: **1.14x** (geomean)
  - Overall speedup over single-thread baseline: **1.58x**



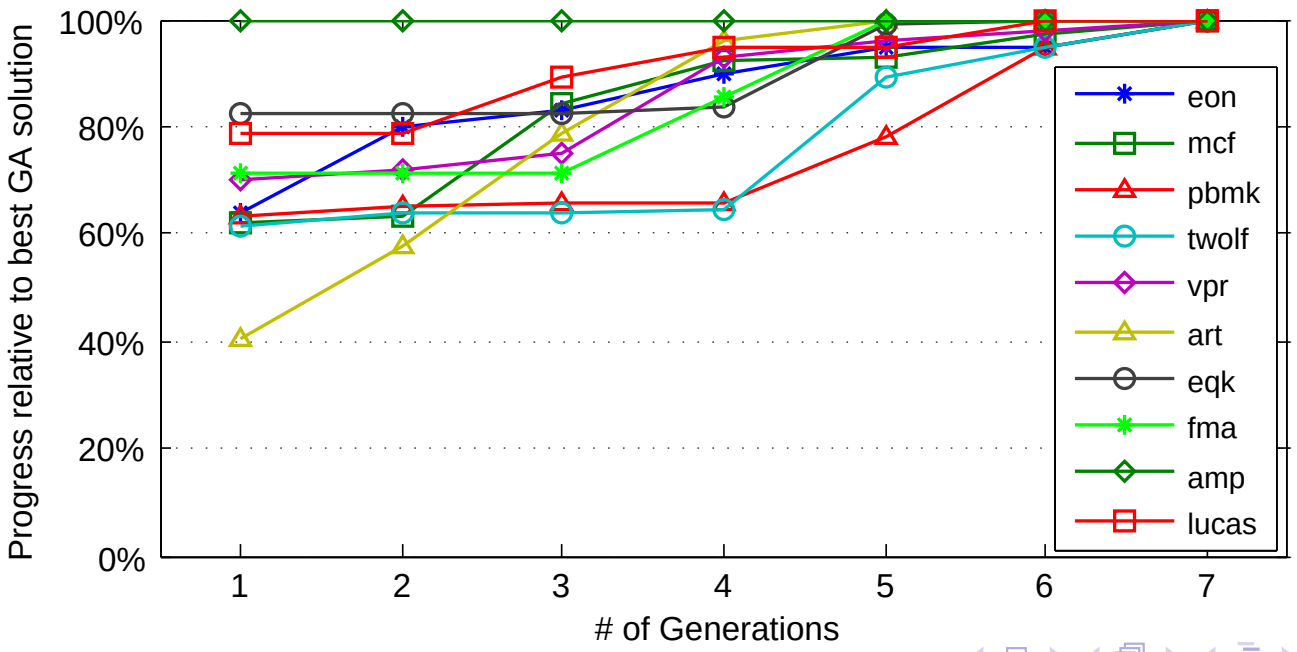
Speedup of Self-tuned Look-ahead

- Applications in which the look-ahead thread is a bottleneck
- Self-tuned, genetic algorithm based decoupled look-ahead
  - Speedup over baseline decoupled look-ahead: **1.14x** (geomean)
  - Overall speedup over single-thread baseline: **1.58x**
- 2-core speedup of 1.58x: a compelling solution for turbo boosting



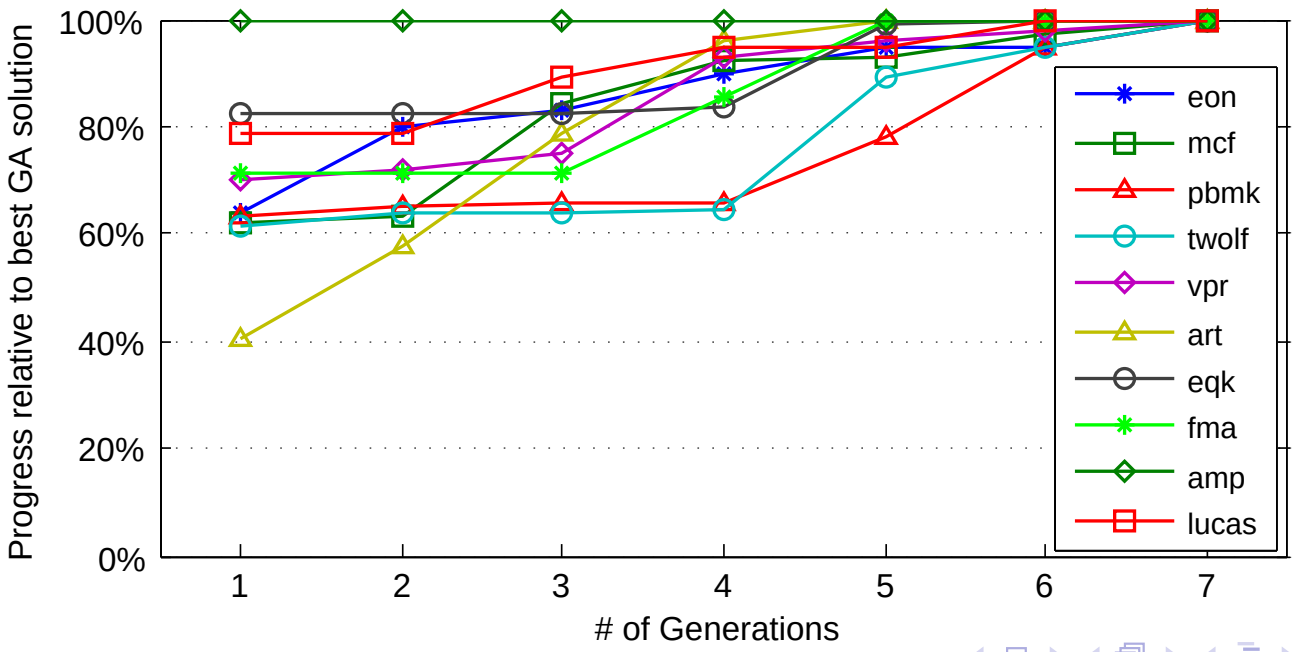
Progress of Genetic Evolution Process

- Per generation progress compared to the final best solution
  - After 2 generations, more than half of the benefits are achieved
  - After 5 generations, significant performance benefits are achieved



Progress of Genetic Evolution Process

- Per generation progress compared to the final best solution
  - After 2 generations, more than half of the benefits are achieved
  - After 5 generations, significant performance benefits are achieved
- GA evolution, helped by hybridization shows good progress

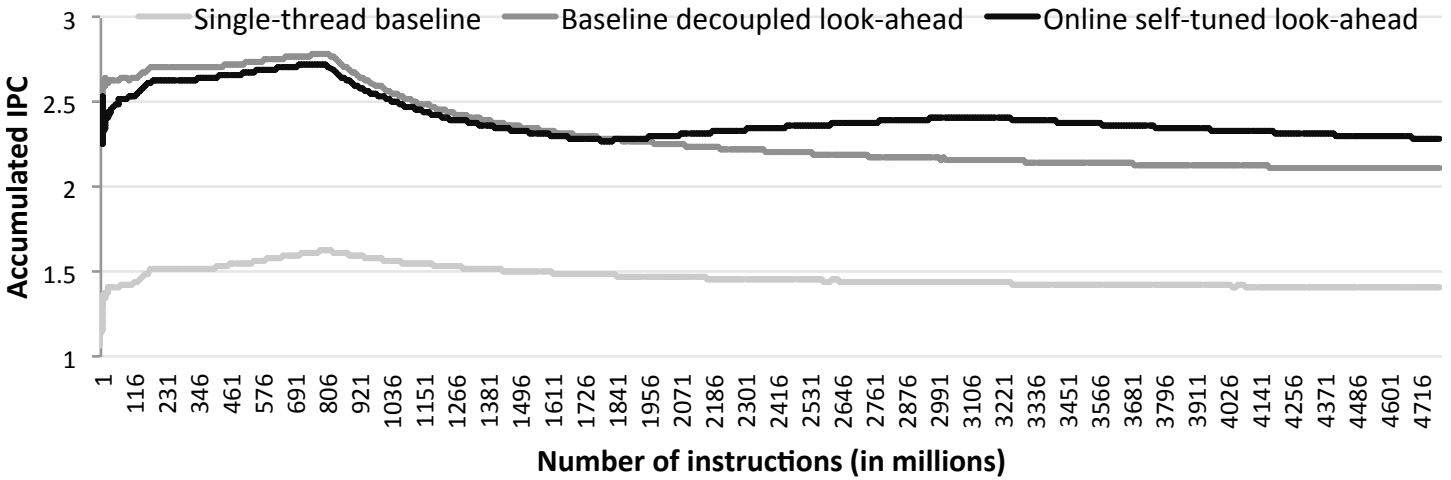


Evolution can be Online or Offline

- Offline evolution: one time tuning (e.g. install time)
  - Fitness tests need not take long (2-20s on target machine)
  - Different input and configuration do not invalidate result

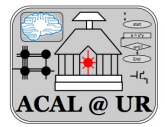
Evolution can be Online or Offline

- Offline evolution: one time tuning (e.g. install time)
  - Fitness tests need not take long (2-20s on target machine)
  - Different input and configuration do not invalidate result
- Online evolution takes longer but has little overhead
  - Additional work minimum: book keeping, bit vector manipulation
  - Main source of slowdown: testing bad configurations



Other Details in the Paper

- The evolution process is remarkably robust
  - Different inputs and configuration do not invalidate results
  - Can use sampling to accelerate fitness test w/o appreciable impact on quality of solution found
- Energy reduction $\rightarrow$ due to less activity and stalling
  - About 10% dynamic instructions removed from skeleton
  - About 11% energy saving compared to baselne decoupled look-ahead
- Impact of weak insts removal on look-ahead quality is very small
  - Similar prefetch and branch hint accuracy



Summary

- Decoupled look-ahead can uncover significant implicit parallelism
  - However, look-ahead thread often becomes a new bottleneck

Summary

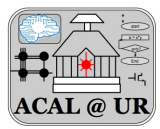
- Decoupled look-ahead can uncover significant implicit parallelism
  - However, look-ahead thread often becomes a new bottleneck
- Fortunately, look-ahead lends itself to various optimizations:
  - Weak instructions can be removed w/o affecting look-ahead quality
  - Plenty of weak instructions present across all programs (100s of)

Summary

- Decoupled look-ahead can uncover significant implicit parallelism
  - However, look-ahead thread often becomes a new bottleneck
- Fortunately, look-ahead lends itself to various optimizations:
  - Weak instructions can be removed w/o affecting look-ahead quality
  - Plenty of weak instructions present across all programs (100s of)
- Removing weak instructions through simple heuristics is hard
  - Expensive false positives; weakness interdependence

Summary

- Decoupled look-ahead can uncover significant implicit parallelism
  - However, look-ahead thread often becomes a new bottleneck
- Fortunately, look-ahead lends itself to various optimizations:
  - Weak instructions can be removed w/o affecting look-ahead quality
  - Plenty of weak instructions present across all programs (100s of)
- Removing weak instructions through simple heuristics is hard
  - Expensive false positives; weakness interdependence
- Metaheuristic based self-tuning approach is simple and robust
  - Improves decoupled look-ahead performance by 1.14x while saving energy by 10%



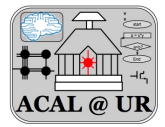
Backup Slides

Accelerating Decoupled Look-ahead via Weak Dependence Removal: A Metaheuristic Approach

Raj Parihar, Michael C. Huang

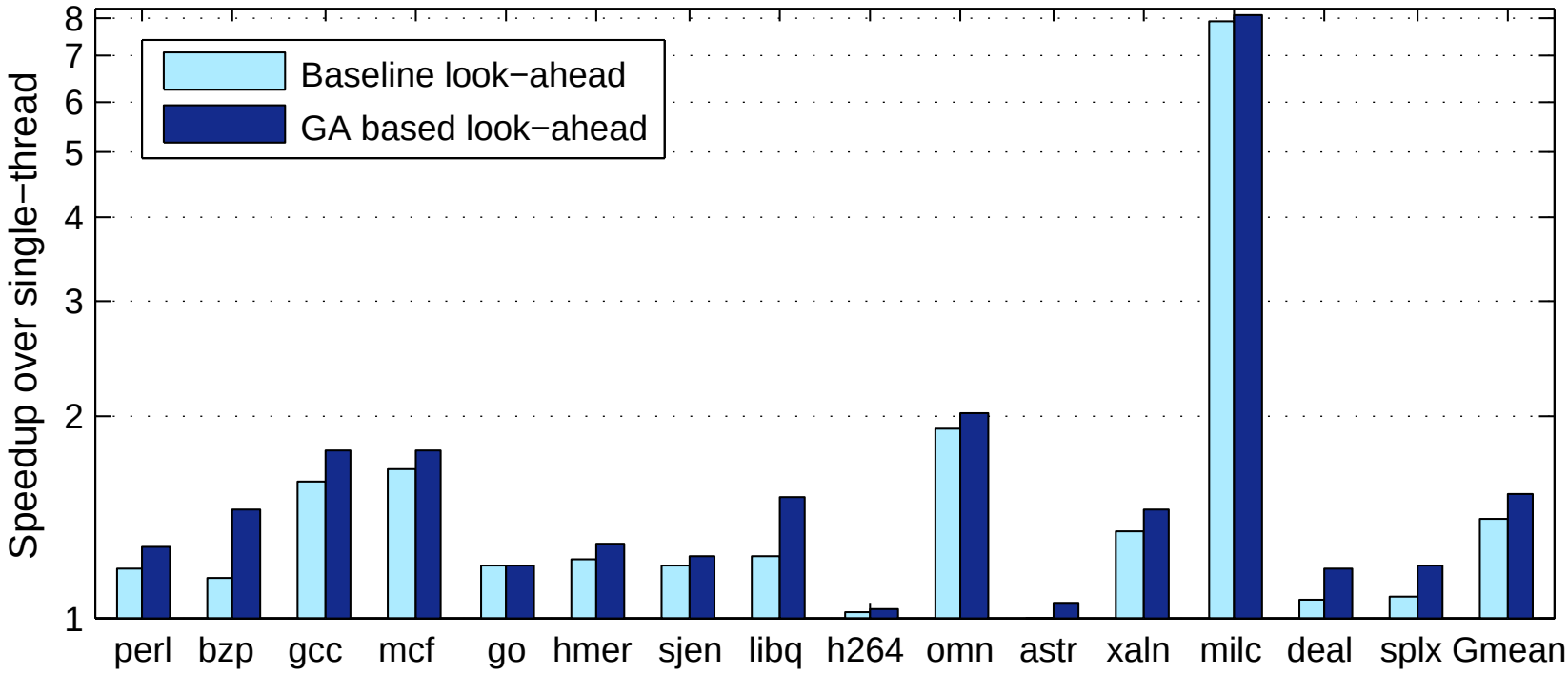
Department of Electrical & Computer Engineering

University of Rochester, Rochester, NY



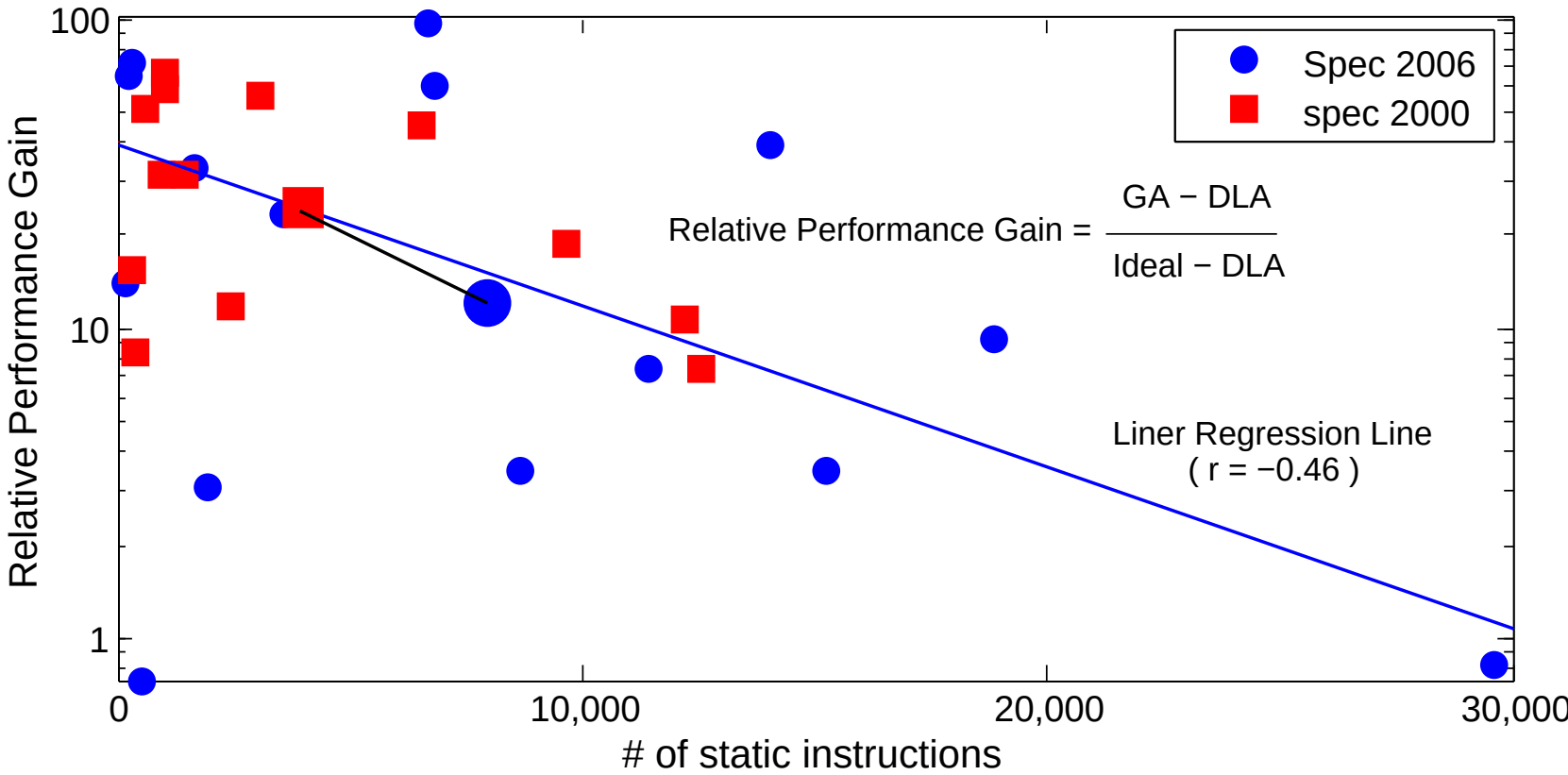
Self-tuned Look-ahead: SPEC 2006

- Self-tuned look-ahead achieves 1.10x speedup over baseline look-ahead for SPEC CPU 2006 applications



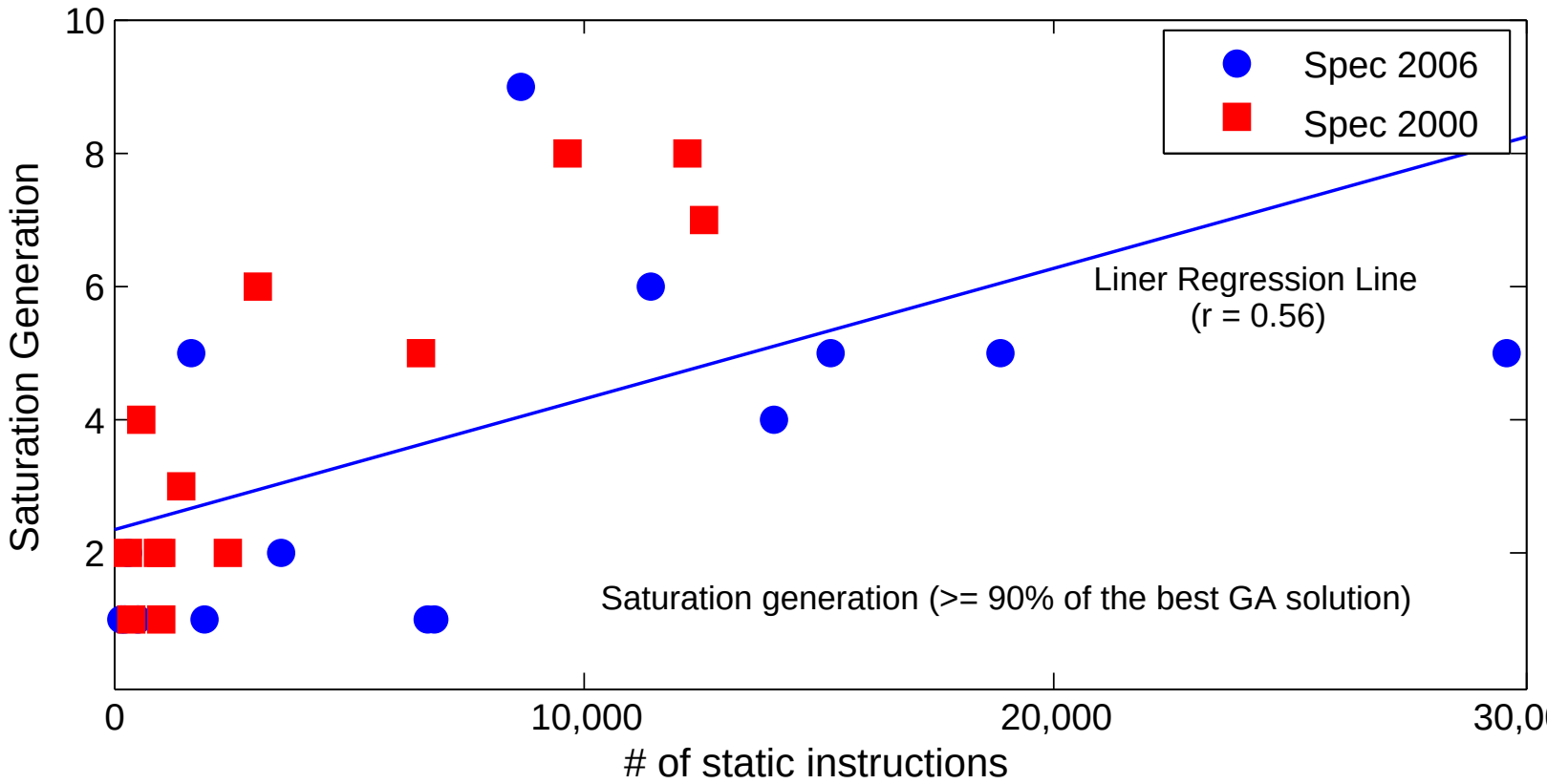
Self-tuned Look-ahead: Speedup Analysis

- A larger code (with more genes) takes slightly more time to evolve

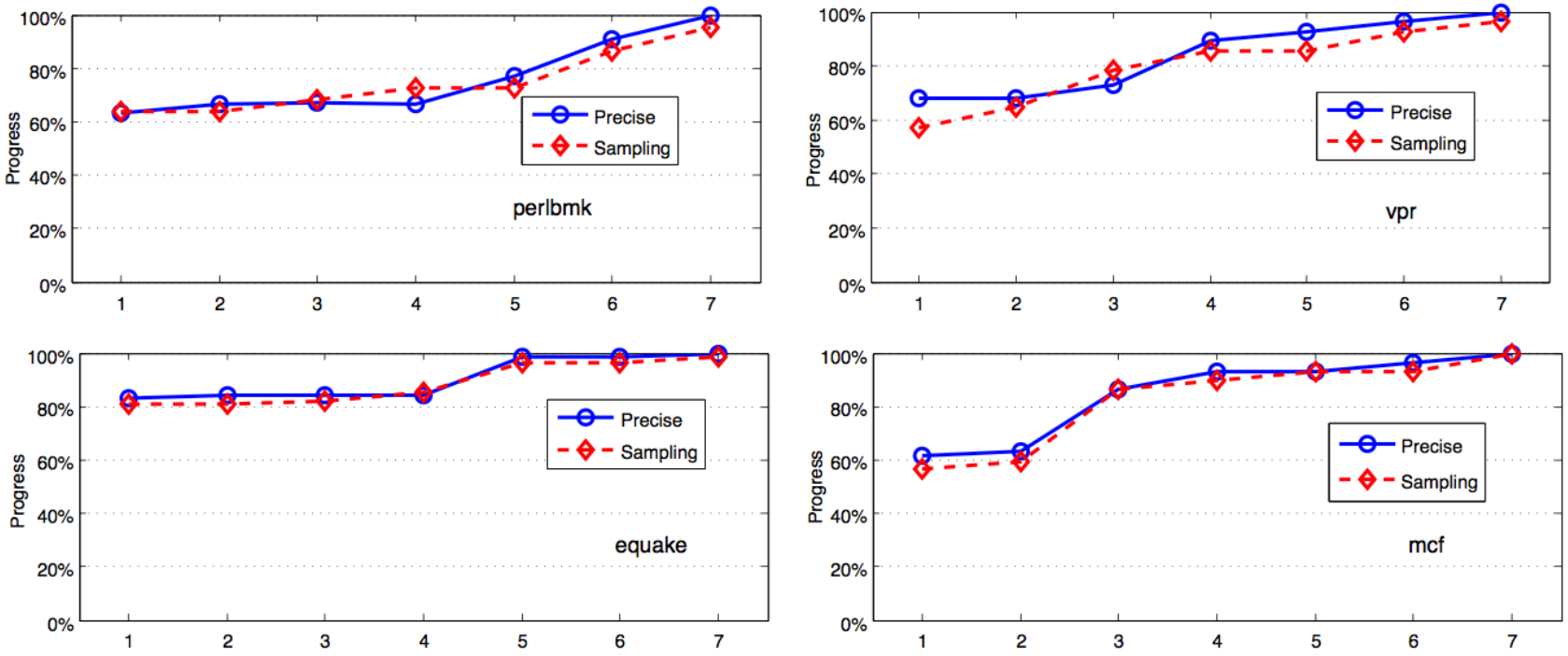


Self-tuned Look-ahead: Speedup Analysis

- Performance gain has strong correlation with # of generations

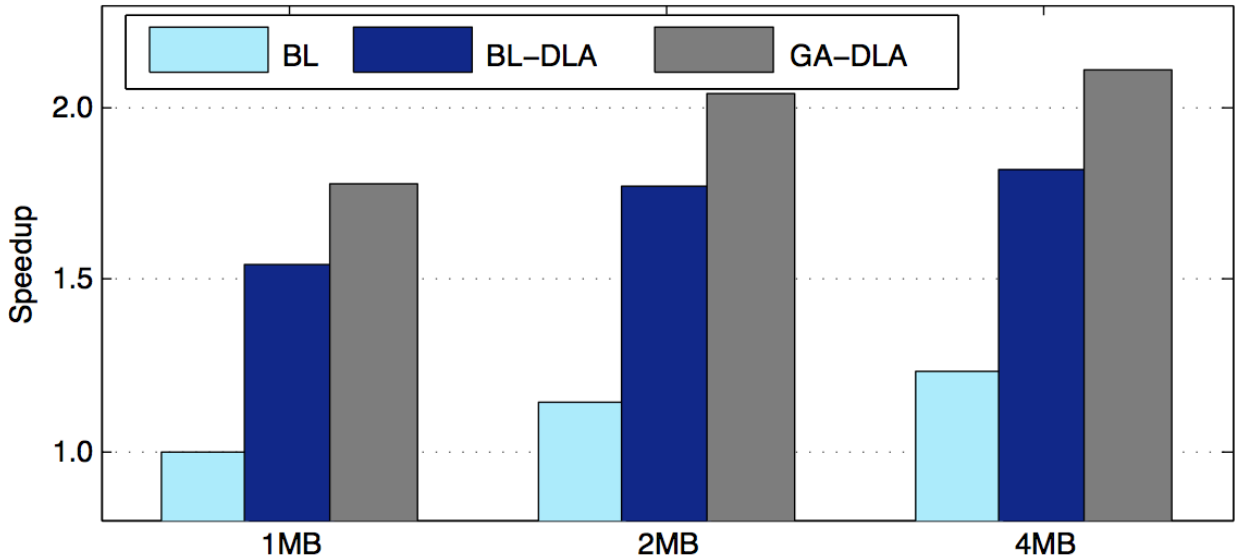


Sampling based Fitness Test



L2 Cache Sensitivity Study

- Speedup for various L2 caches is quite stable
  - 1.139x (1 MB), 1.133x (2 MB), and 1.131x (4 MB) L2 caches
- Avg. speedups, shown in the figure, are relative to single-threaded execution with a 1 MB L2 cache

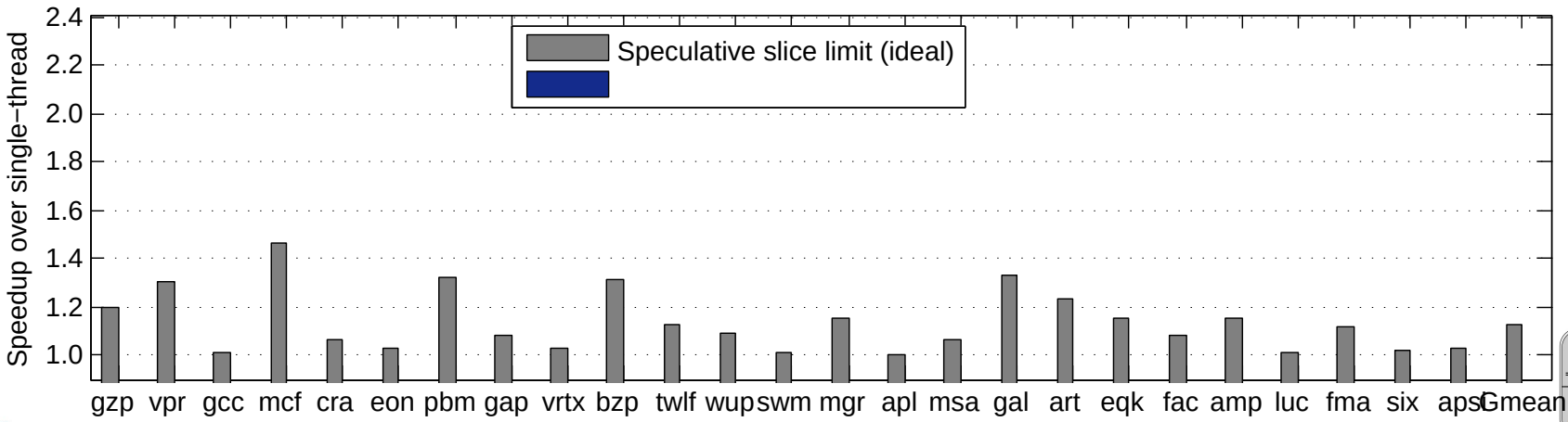


Look-ahead Skeleton: Size and Other Stats

	crafty	eon	gap	gzip	mcf	perlbmk	twolf	vortex	vpr	ammp	art	equake	fma3d	lucas	Average
Baseline look-ahead skeleton (%dyn)	87.14	72.29	76.22	64.72	59.95	78.98	81.05	58.10	67.06	66.60	54.33	32.86	79.50	32.21	65.07
GA tuned look-ahead skeleton (%dyn)	82.66	65.83	67.79	57.35	52.61	70.22	78.25	56.31	59.01	63.22	41.11	30.06	73.50	28.53	59.03
Total program instructions (static)	57568	79730	74650	23205	18286	120529	53936	95121	42089	43154	25588	25639	249464	103235	72299
Instructions in 100m window (static)	12543	6562	4130	1424	381	9692	2456	12230	1061	958	582	1041	3098	319	4034
Individual weak instructions (static)	172	57	211	207	117	417	110	398	261	223	173	628	104	55	224
Instructions removed using GA (static)	51	15	37	56	20	30	24	37	33	24	36	40	35	12	32

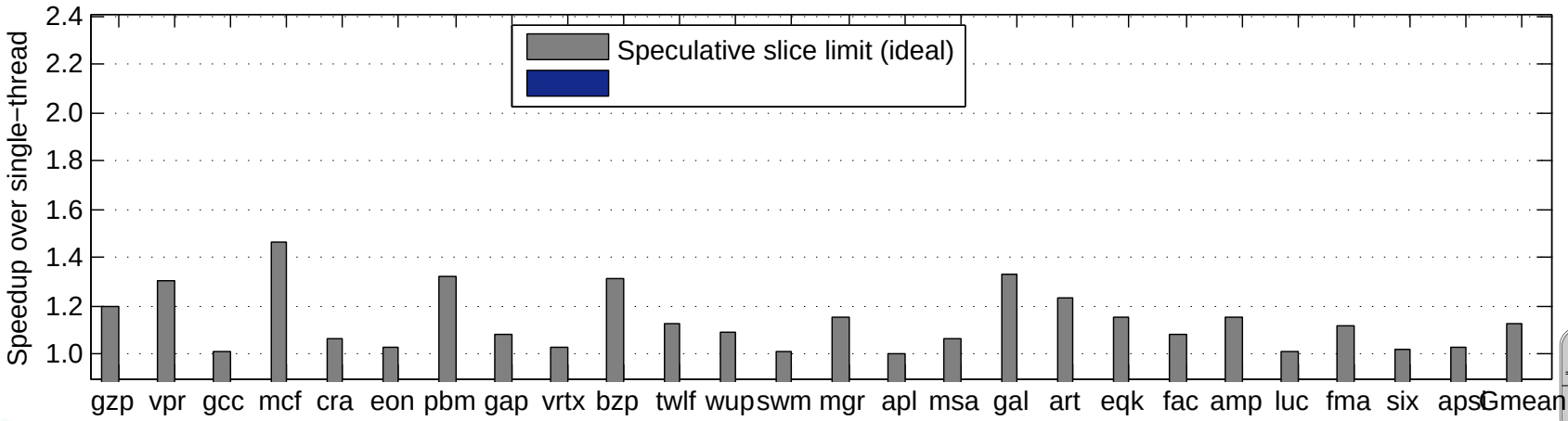
Practical Advantages of Decoupled Look-ahead

- Micro helper thread based approach:
  - Targets top cache misses and branch mispredictions (low coverage)
  - Support for quick spawning and register communication (not trivial)



Practical Advantages of Decoupled Look-ahead

- Micro helper thread based approach:
  - Targets top cache misses and branch mispredictions (low coverage)
  - Support for quick spawning and register communication (not trivial)
- Decoupled look-ahead approach:
  - Easy to disable, low management overhead on main thread
  - Natural throttling to prevent run-away prefetching, cache pollution



Practical Advantages of Decoupled Look-ahead

- Micro helper thread based approach:
  - Targets top cache misses and branch mispredictions (low coverage)
  - Support for quick spawning and register communication (not trivial)
- Decoupled look-ahead approach:
  - Easy to disable, low management overhead on main thread
  - Natural throttling to prevent run-away prefetching, cache pollution