Improving Instruction Delivery with a Block-Aware ISA

Ahmad Zmily, Earl Killian, **Christos Kozyrakis**

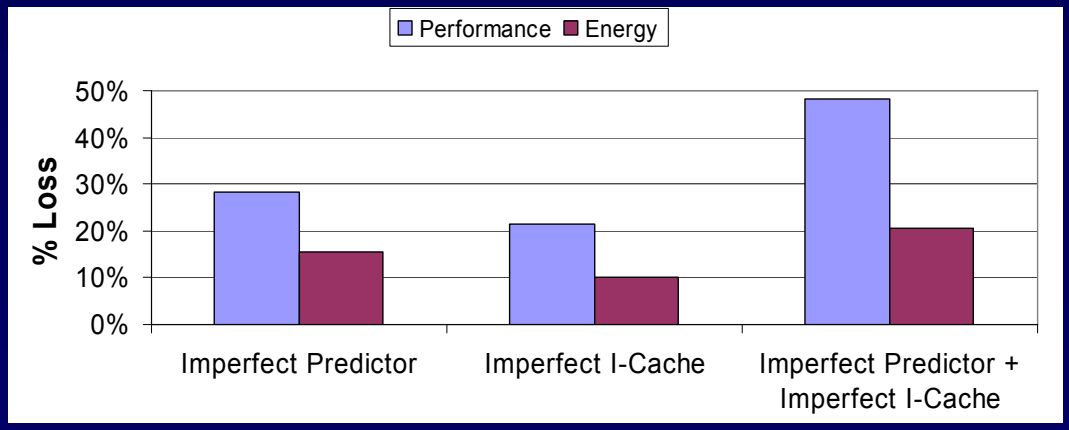
Computer Systems Laboratory

Stanford University

http://csl.stanford.edu

Motivation

- Processor front-end engine
  - Performs control flow prediction & instruction fetch
  - Sets upper limit for performance
    - Cannot execute instructions faster than you can fetch them!
- Front-end detractors
  - Control-flow mispredictions that lead to pipeline flushing
  - Instruction cache misses
  - Multi-cycle instruction cache accesses



Outline

BLISS

- A block-aware instruction set architecture
  - Allows software to help with hardware challenges
  - Decouples control-flow prediction from instruction fetching
  - Allows fast & accurate instruction delivery with simple hardware

- Talk outline
  - BLISS overview
    - Instruction set and front-end microarchitecture
    - Performance optimizations
    - Energy optimizations
  - Experimental results
    - 20% performance and 14% energy improvements
    - Outperforms hardware-only block-based front-ends
  - Conclusions

BLISS Instruction Set

- Explicit basic block descriptors (BBDs)
  - Stored separately from instructions in the text segment
  - Describe control flow and identify associated instructions
- Execution model
  - PC always points to a BBD, not to instructions
  - Atomic execution of basic blocks

32-bit Descriptor Format

- **Type**: type of terminating control-flow instruction
  - Fall-through, jump, jump register, forward/backward branch, call, return
- **Offset**: displacement for PC-relative branches and jumps
  - Offset to target basic block descriptor
- **Length**: number of instruction in the basic block
  - 0 to 15 instructions
  - Longer basic blocks use multiple descriptors
- **Instruction pointer**: address of the first instruction in the block
  - Remaining bits from TLB
- **Hints**: optional compiler-generated hints
  - This study: branch hints (biased taken/non-taken branches)
  - Other uses: code density, power savings, VLIW techniques, …

BLISS Code Example

```
numeqz=0;
for (i=0; i<N; i++)
  if (A[i]==0) numeqz++;
  else foo();
```

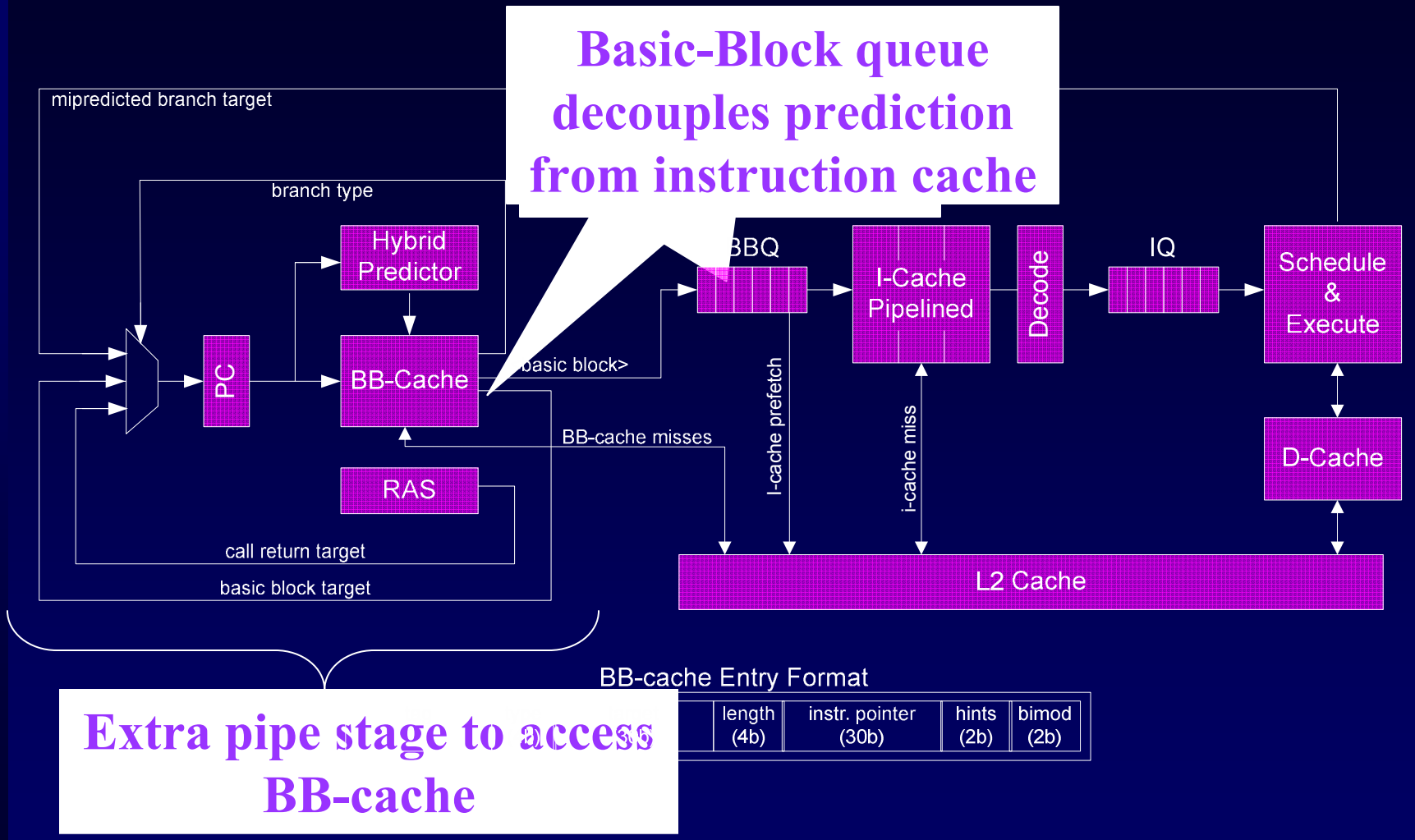
- Example program in C-source code
  - Counts the number of zeros in array A
  - Calls foo() for each non-zero element

BLISS Code Example

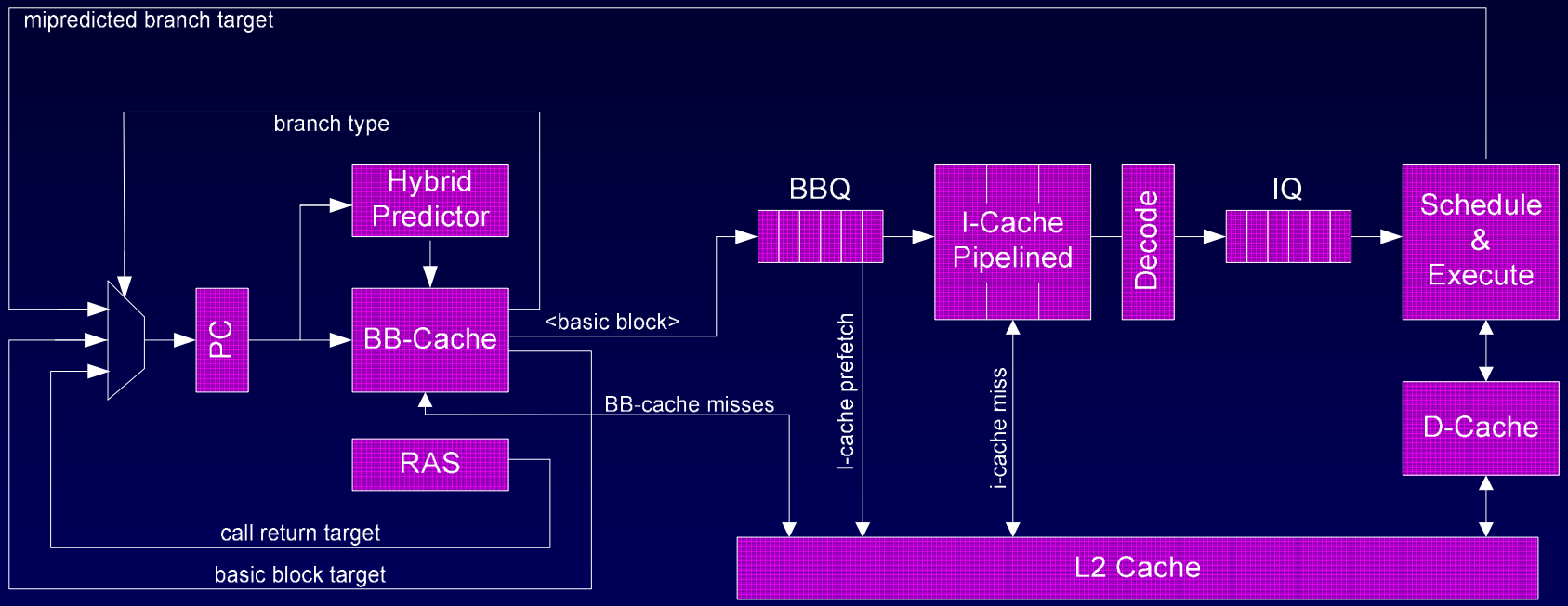
BBD1: FT , --- , 1	addu r4,r0,r0
BBD2: B_F , BBD4, 2	L1: lw r6,0(r1) bneqz r6,L2
BBD3: J, BBD5, 1	addui r4,r4,1 j L3
BBD4: JAL, FOO, 0	L2: jal FOO
BBD5: B_B, BBD2, 2	L3: addui r1,r1,4 bneq r1,r2,L1

- All jump instructions are redundant
- Several branches can be folded in arithmetic instructions
  - Branch offset is encoded in descriptors

BLISS Decoupled Front-End

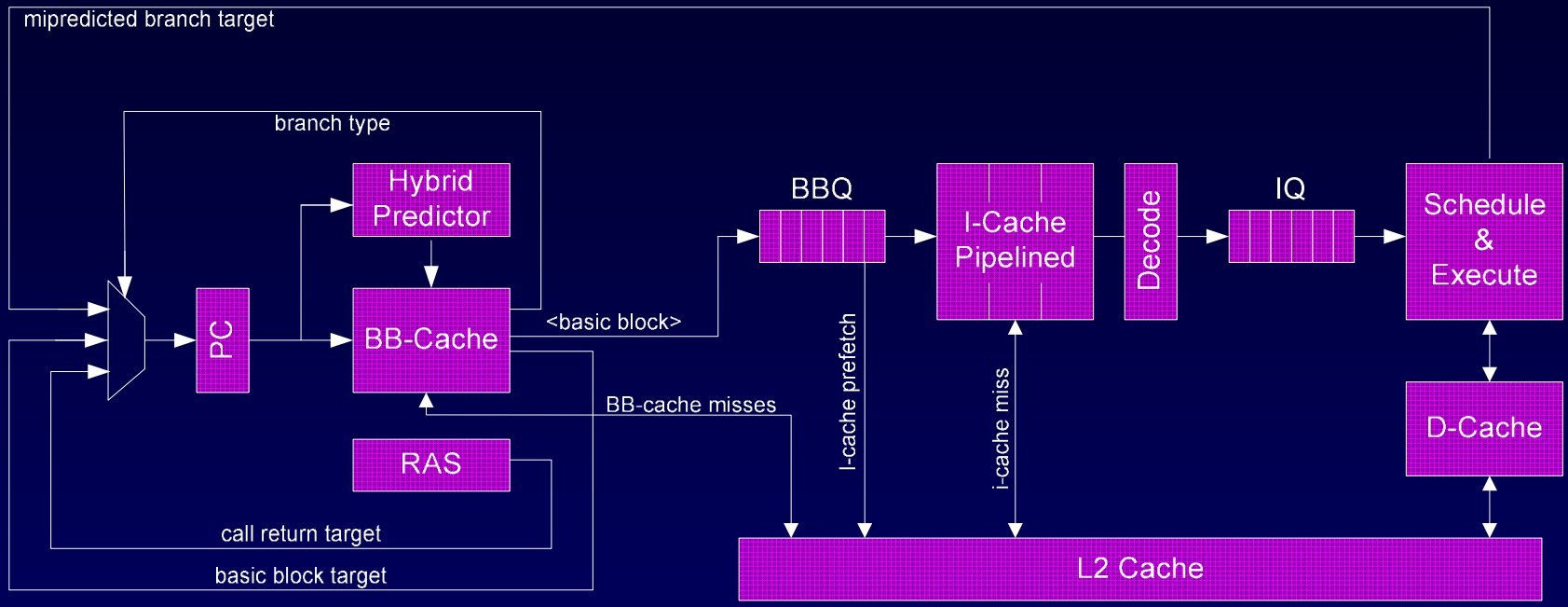


Front-End Operation: BB-cache Hit



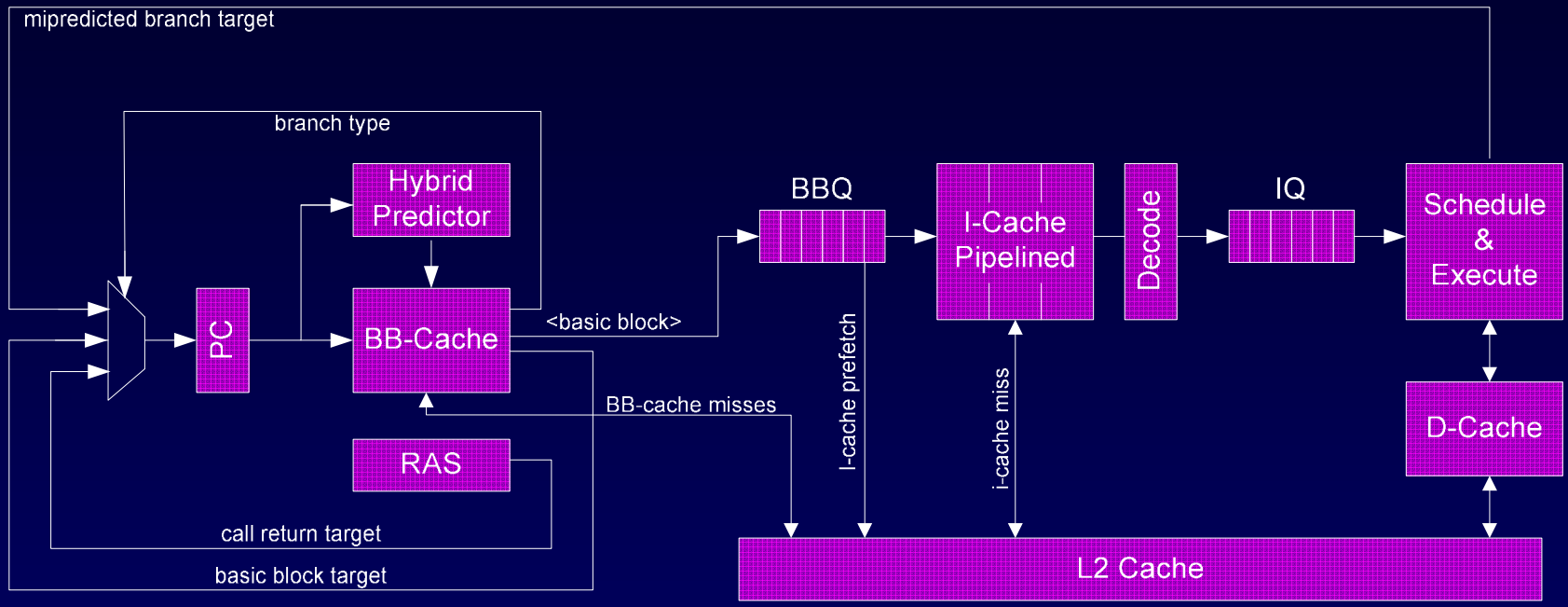
- Push descriptor & predicted target in BBQ
  - Instructions fetched and executed later (decoupling)
- Continue fetching from predicted BBD address
  - Hybrid predictor accessed in following cycle to verify speculation

Front-End Operation: BB-cache Miss



- Wait for refill from L2 cache
  - Calculate 32-bit instruction pointer & target on refill
- Back-end only stalls when BBQ and IQ are drained
  - Can hide significant portion of L2 cache latency

Front-End Operation: Misprediction



- Flush pipeline including BBQ and IQ
- Restart from correct BBD address
  - Start fetching BBDs while recovering back-end state

Performance Optimizations (1)

- I-cache misses can be tolerated
  - BBQ provides early view into instruction stream
  - Guided instruction prefetch
- I-cache is not in the critical path for speculation
  - BBDs provide branch type and offsets for speculation
  - Multi-cycle I-cache does not affect prediction accuracy
    - Latency only visible on mispredictions
- Similar to previous decoupled front-end work
  - [Calder et.al. 94], [Stark et.al. 97], [Reinman et.al. 01] , …

Performance Optimizations (2)

- Better target prediction
  - L2 backs up target buffer on capacity misses
  - No cold misses for PC-relative branch targets
  - Compiler hints for branches (optional)
- Better direction prediction
  - All PCs refer to basic block boundaries
    - Denser representation leads to less interference
  - Judicious use and training of predictor
    - No predictor access for fall-through or jump blocks
    - Selective use of hybrid predictor if branch hints are available
- Overall up to 41% less pipeline flushes with BLISS
  - Without complicated hardware control

Energy Optimizations

- Energy saved on mispredicted instructions
  - Due to better target and direction prediction
  - The saving is across the whole processor pipeline
    - 15% of energy wasted on mispredicted instructions
- Instruction cache optimizations
  - Access only the necessary words in I-cache
  - Serial access of tags and data in I-cache
  - Merged I-cache accesses waiting in the BBQ
- Judicious use and training of predictor
  - No predictor access for fall-through or jump blocks
  - Selective use of hybrid predictor if branch hints are available

Evaluation Methodology

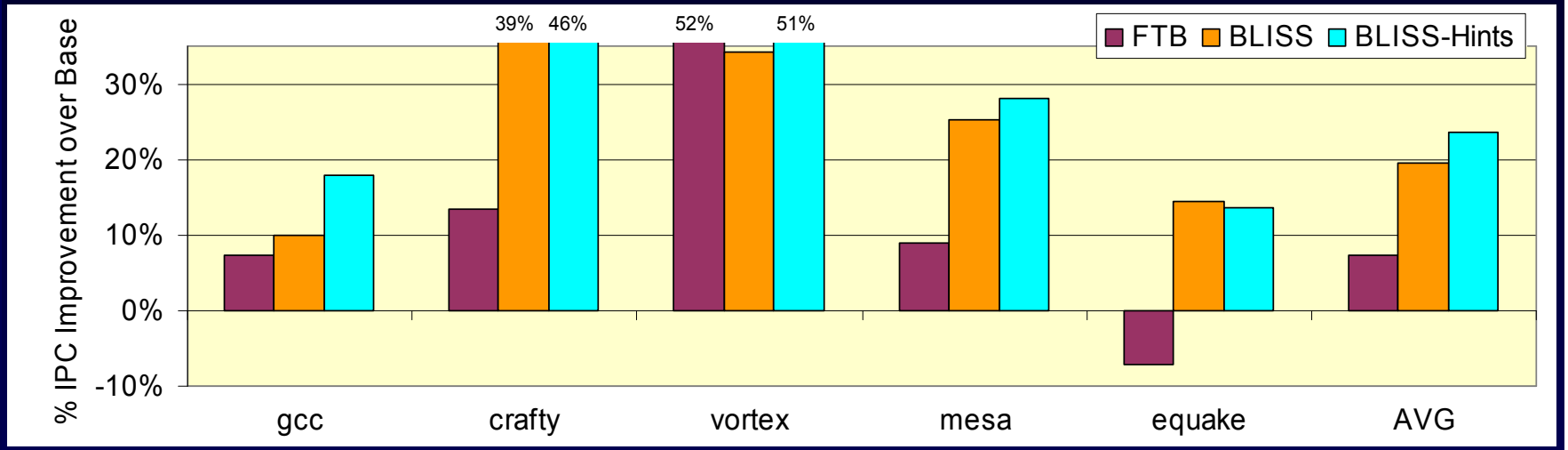
- 8-way superscalar processor
  - Out-of-order execution, two-level cache hierarchy
  - Simulated with Simplescalar & Wattch toolsets
  - SpecCPU2K benchmarks with reference datasets
- BLISS code generation
  - Binary translation from MIPS executables
  - 16% reduction in static code size by eliminating redundancy
- Comparison: fetch-target-block (FTB) [Reinman et. al. 2001]
  - Similar to BLISS but pure hardware implementation
  - Hardware creates and caches block and extended blocks
    - Optimistic approach on FTB misses to help block detection
  - Similar performance and energy optimizations applied

Front-end Parameters

	Base	FTB	BLISS
Fetch Width	8 instructions	1 (extended) block	1 basic block
Target Predictor	BTB: 2K entries 4-way 1 cycle access	FTB: 2K entries 4-way 1 cycle access	BB-cache: 2K entries 4-way 1 cycle access 8 entries per line
Decoupling Queue	–	4 Entries	
I-cache Latency	32 KBytes, 4-way 2-cycle access pipelined		

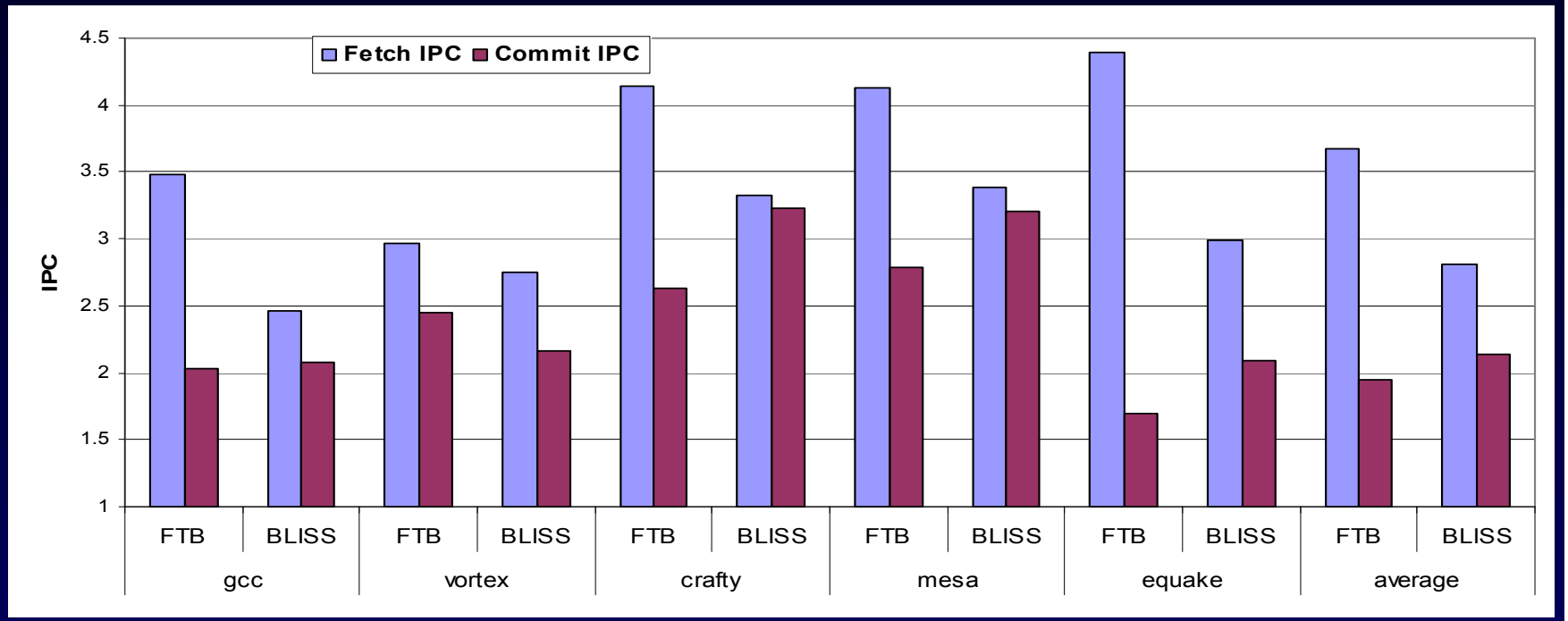
- BTB, FTB, and BB-cache have exactly the same capacity
  - Same number of SRAM bits needed for implementation
  - Nearly identical access latency

Performance



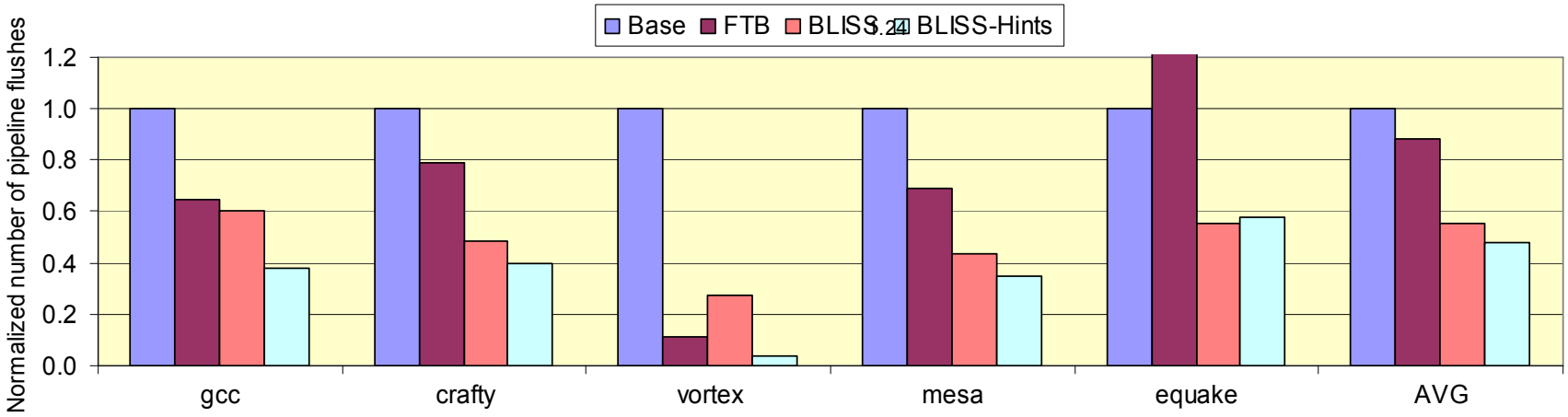
- Consistent performance advantage for BLISS
  - 20% average improvement over base
  - 13% average improvement over FTB
- Sources of performance improvement
  - 41% reduction pipeline flushes compared to base
  - 24% reduction in I-cache misses due to prefetching

FTB vs. BLISS



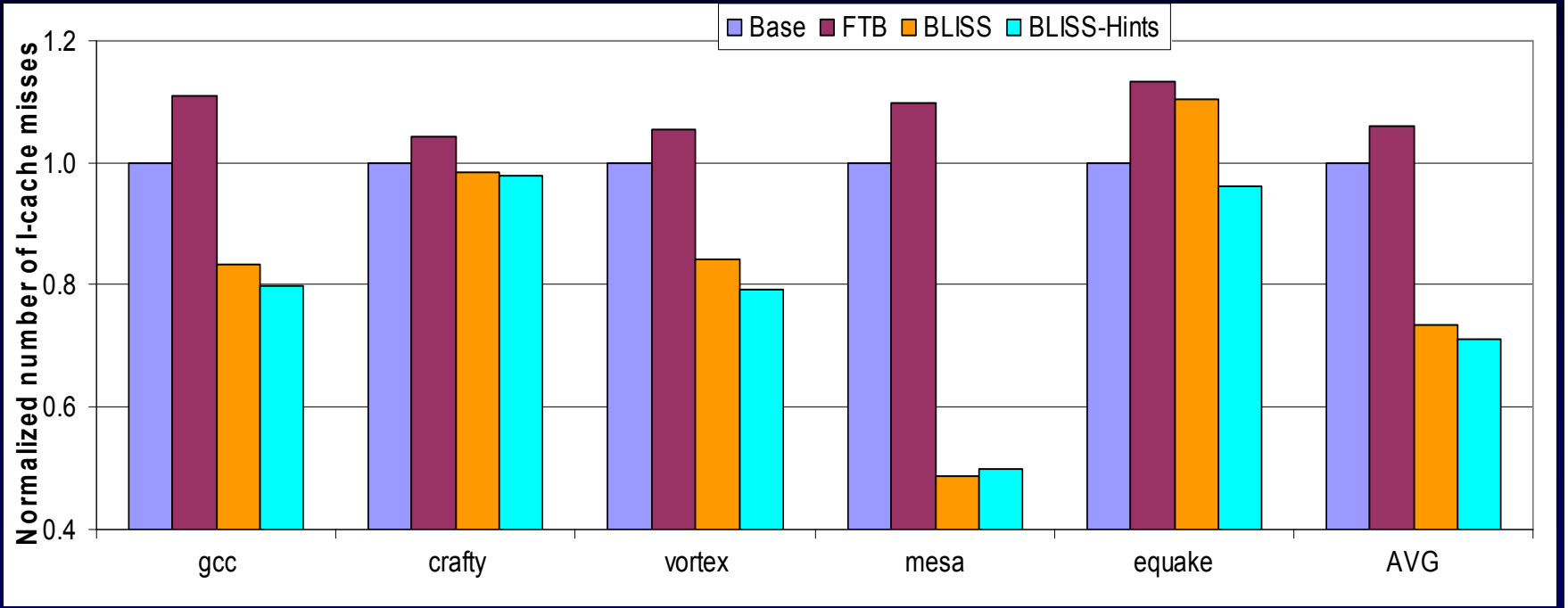
- FTB
  - Aggressive extended block formation ⇒ higher fetch IPC
  - Over-speculation on misses ⇒ lower commit IPC
- BLISS
  - Stall on misses, get accurate block descriptor from L2 cache
  - Balance between under-speculation and over-speculation

Prediction Accuracy



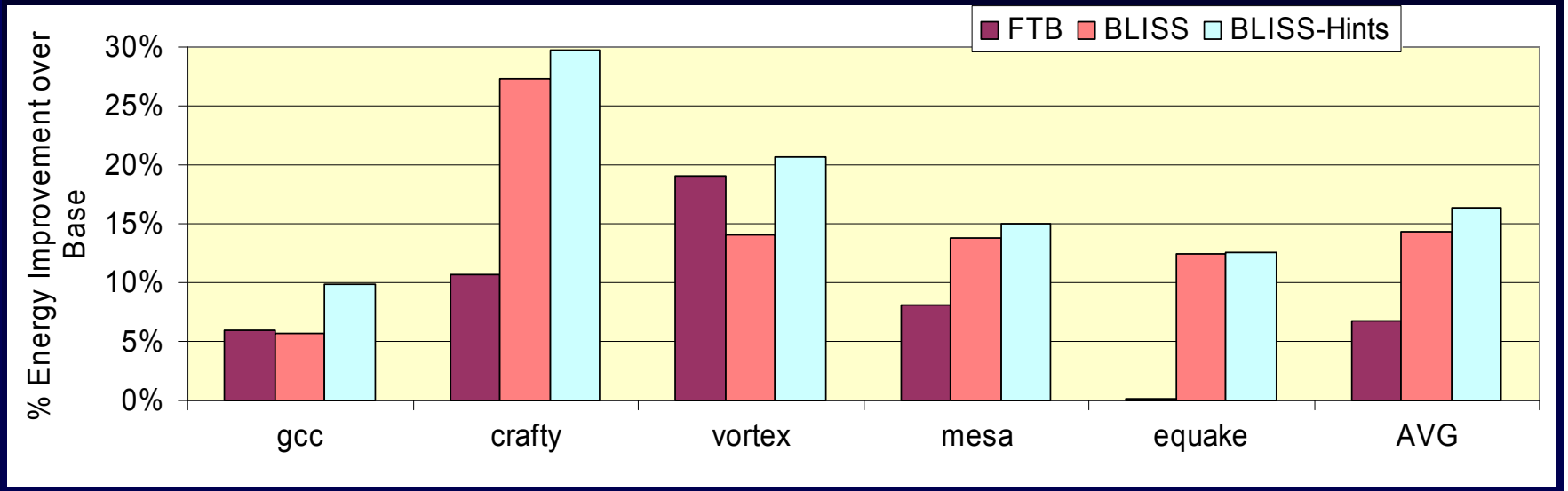
- BLISS lead to 41% reduction in mispredictions
  - Avoids over-speculation on BB-cache misses
  - Accurate indexing and training of the hybrid predictor
  - Dense PCs lead to 1% better prediction
  - 1.2% better prediction when hints are used

Instruction Cache



- BLISS reduces instruction cache misses
  - Dense static code
  - Prefetching using BBQ contents
  - Fewer mispeculated instructions requested

Total Chip Energy



- Total energy = front-end + back-end + all caches
- BLISS leads to 14% total energy savings over base
  - Front-end savings + savings from fewer mispredictions
- FTB is limited to 7% savings
  - Optimistic, large blocks needed to facilitate block creation
  - Over-speculation is bad for energy too

Conclusions

- BLISS: a block-aware instruction set
  - Defines basic block descriptors separate from instructions
  - Expressive ISA to communicate software info and hints
- Enabled optimizations
  - Better target and direction prediction accuracy
  - Tolerate I-cache misses
  - Less time and energy spent on overspeculation
- Results: improved performance **and** energy consumption
  - 20% performance and 14% energy over conventional
  - 13% performance and 7% energy over hardware-only scheme
  - Additional benefits from software hints