A Low Power Front-End for Embedded Processors Using a Block-Aware Instruction Set

Ahmad Zmily

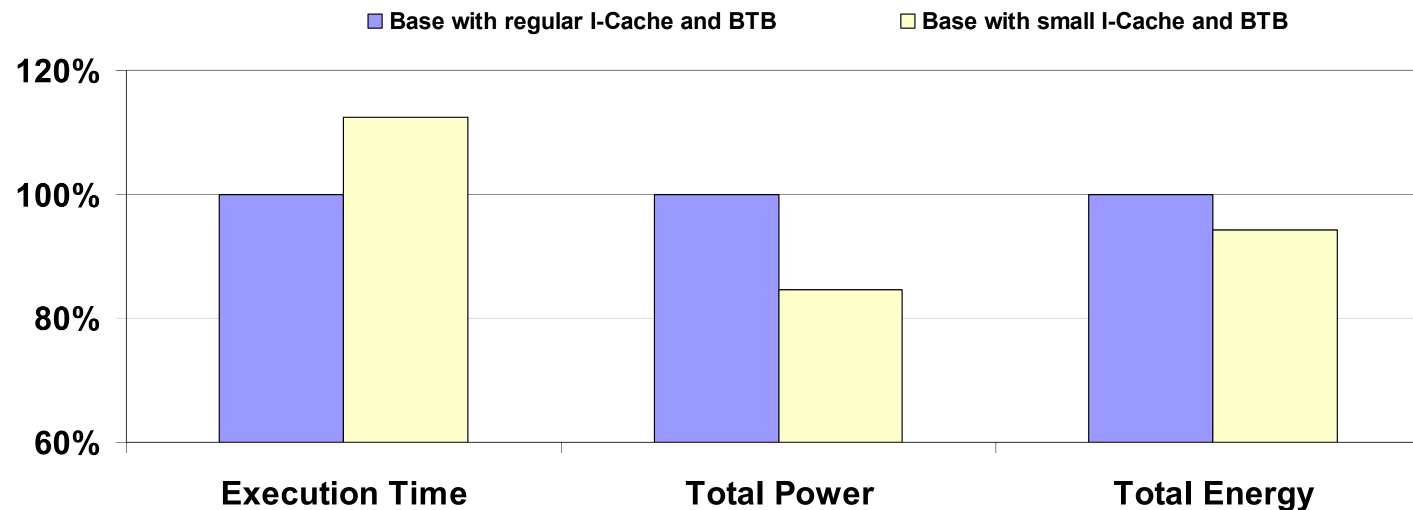
Computer System Lab
Stanford University

Motivation

- Processor front-end engine
  - Performs control flow prediction & instruction fetch
  - Sets upper limit for performance
    - Cannot execute faster than you can fetch
- Energy and Power efficiency
  - Determines battery life-time
  - Cost of cooling and packaging
- Front-end consumes significant budget of total power
  - Large memory arrays accessed nearly every cycle
    - Instruction cache, predictors, BTB
  - Arrays are sized to achieve good overall performance
- Reduce the size of the front-end structures?

The Problem

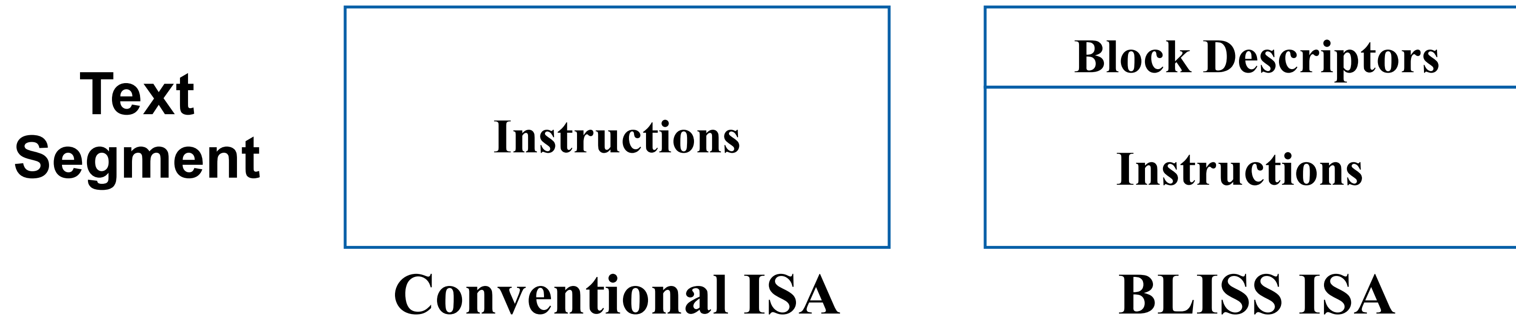
- Xscale with small front-end structures
  - 16% decrease in total processor power
- The cost for MediaBench programs
  - 12% performance loss



BLISS

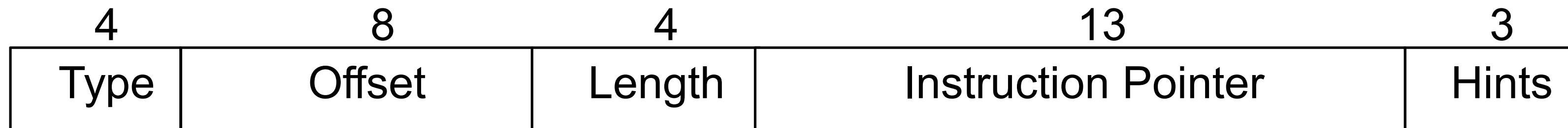
- Focus of this paper
  - Low-power using small front-end structures
  - Eliminate performance degradation through optimizations
- A block-aware instruction set architecture (BLISS)
  - Decouples control-flow prediction from instruction fetching
  - Allows software to help with hardware challenges
- Talk outline
  - BLISS Overview
  - Front-End Optimizations
  - Tools and Methodology
  - Evaluation for Embedded Processors
  - Conclusions

Block-Aware Instruction Set



- BLISS = Block-aware 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

32-bit Basic Block Descriptor Format



- **Type**: type of terminating control-flow instruction
  - Fall-through, jump, jump register, 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
- **Instruction pointer**: address of the first instruction in the block
  - Remaining bits from TLB
- **Hints**: optional compiler-generated hints

BLISS Code Example

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

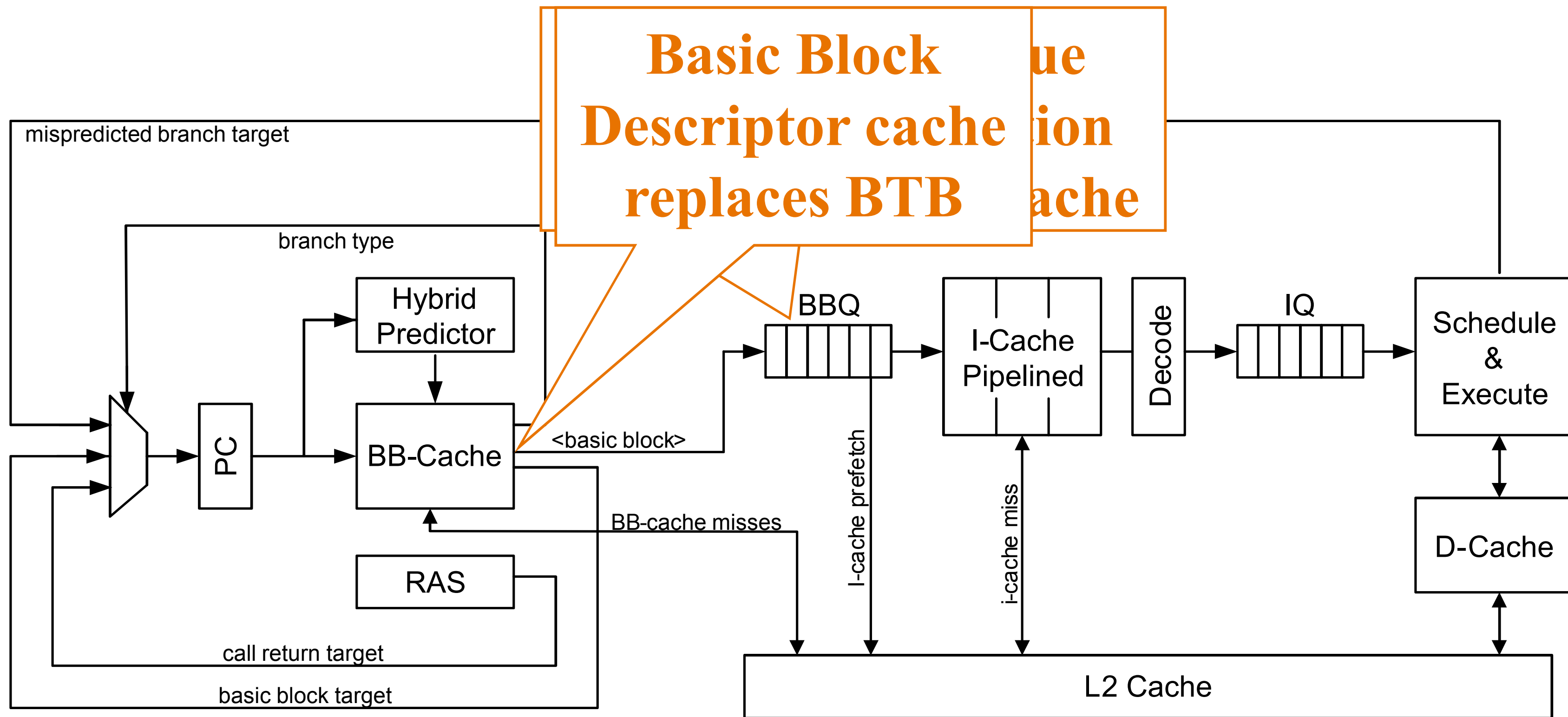
- Example program in C-source code:
  - Counts the number of zeros in array a
  - Calls foo() for each non-zero element

BLISS Code Example

Descriptor	Code
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



BB-cache Entry Format

tag	type (4b)	target (30b)	length (4b)	instr. pointer (13b)	hints (2b)	bimod (2b)

Agenda

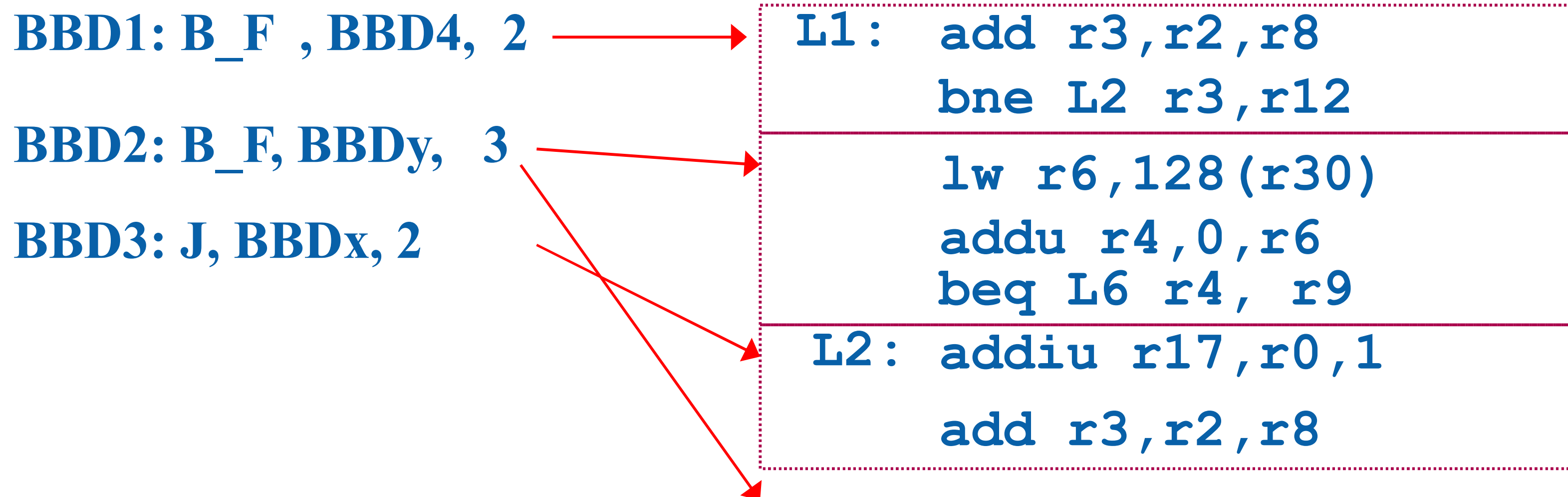
- Motivation
- BLISS Overview
- Front-End Optimizations
- Tools and Methodology
- Evaluation for Embedded Processors
- Conclusions

Instruction Reordering

- Idea: Reorder blocks to improve hit rate and utilization
  - Lay out closely executed blocks in chains using profiling
  - Adjust instruction pointer in block descriptor

Instruction Reordering

- Idea: Reorder blocks to improve hit rate and utilization
  - Lay out closely executed blocks in chains using profiling
  - Adjust instruction pointer in block descriptor

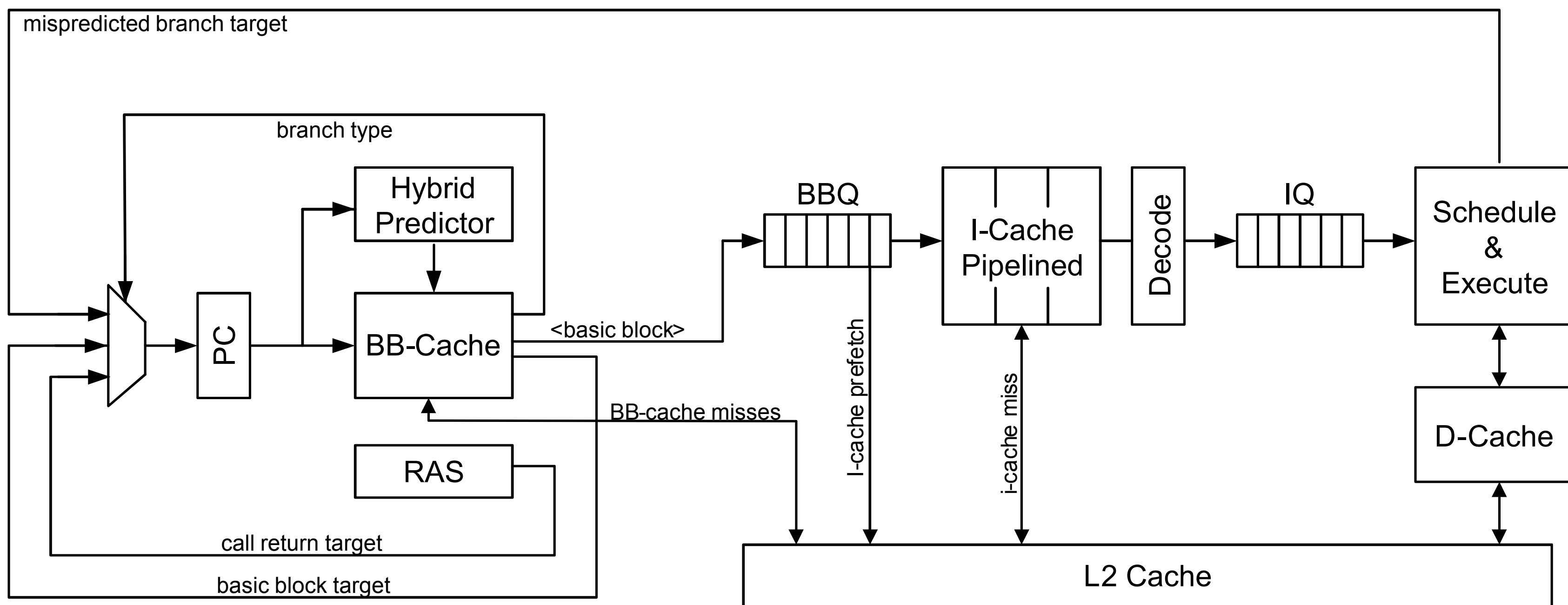


Instruction Cache

add	bne	lw	addu
jal	addiu	add	

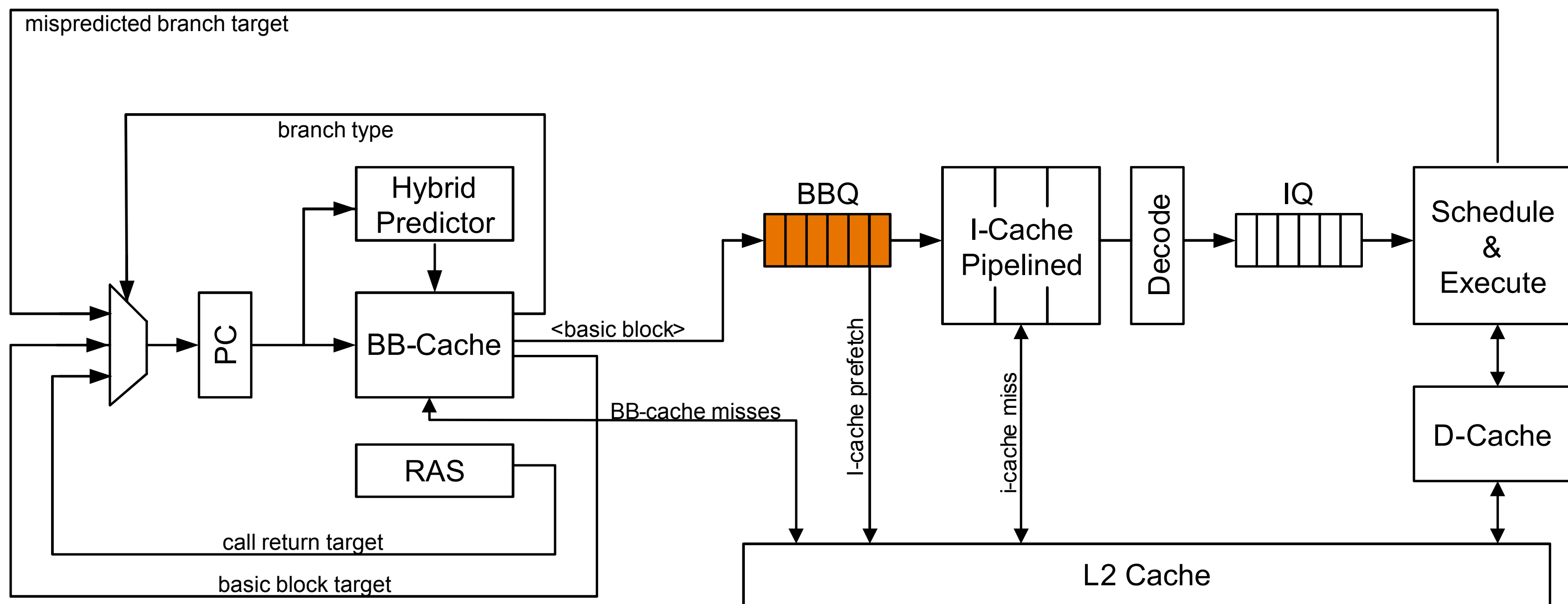
Instruction Prefetching

- BBQ decouples prediction from instruction fetching
  - Predictor runs ahead even when IQ full or I-cache miss
  - Stalls only on BB-cache miss or BBQ



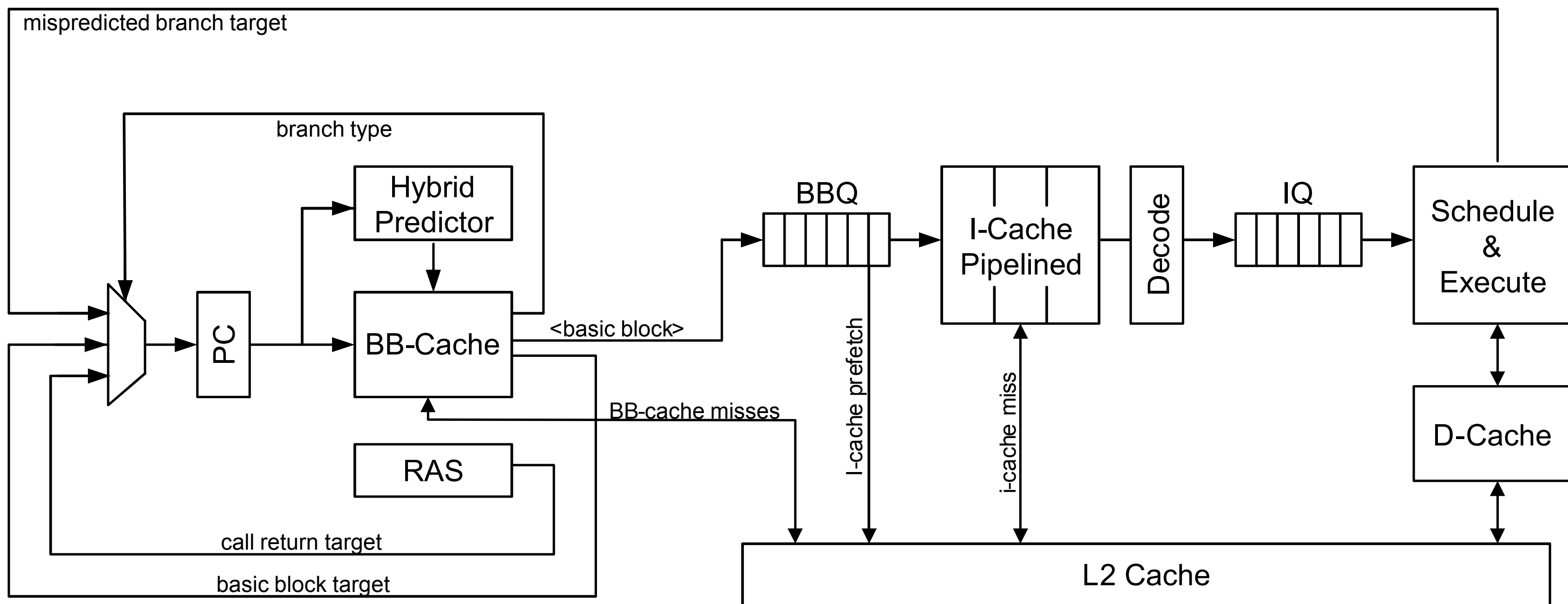
Instruction Prefetching

- BBQ provides early view into instruction stream
  - Guided instruction prefetch
  - I-cache misses can be tolerated



Instruction Prefetching

- Prefetches initiated for potential misses
  - Prop the cache when read port is idle
- Prefetched data in a buffer to avoid cache pollution
  - Pushed into the I-cache after first access



Unified Instruction Cache and BTB

- Programs exhibit different behavior
  - Susceptible to I-cache organization and size (e.g. rasta)
  - Susceptible to BTB organization and size (e.g. adpcm)

- A unified I-cache and BB-cache
  - Cache line has either BBDs or regular instructions
  - Single port accessed by BBD fetch or instruction fetch
    - Instruction fetch returns multiple instructions per cycle

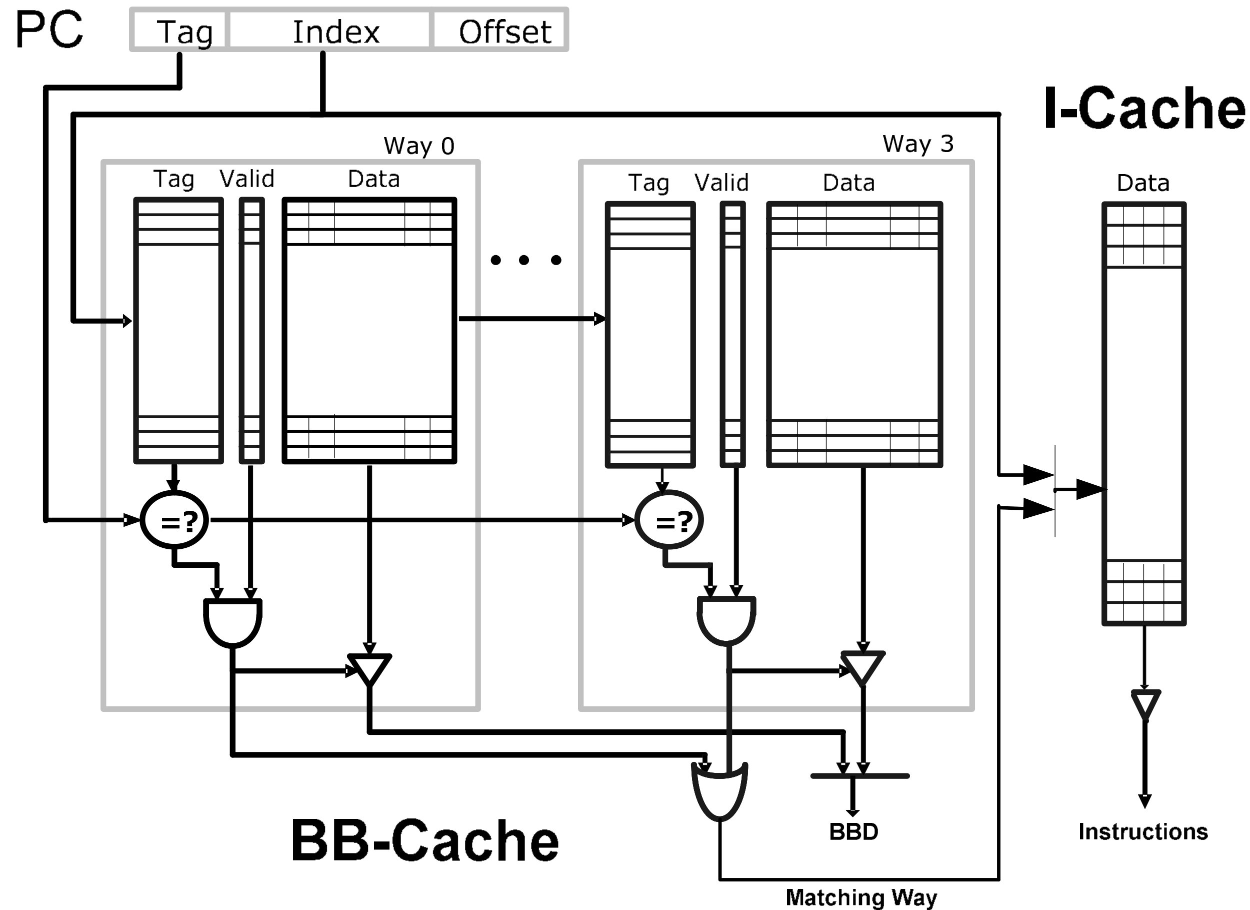
- Difficult with a conventional front-end
  - Same PC used to access I-cache & BTB
    - More conflict misses
    - Need to store extra information to differentiate the two types
  - Sharing single port is difficult
    - Basic-Block Boundaries are not known before decoding

Tagless Instruction Cache

- Idea: exploit tag checks on descriptor accesses
  - Improves I-cache access time, energy, and area

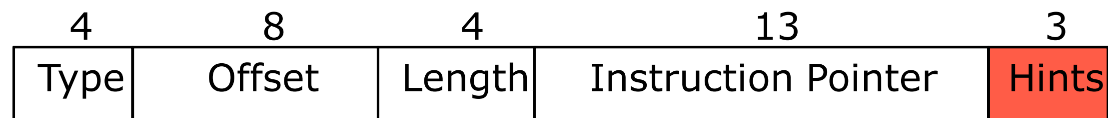
Tagless Instruction Cache

- Idea: exploit tag checks on descriptor accesses
  - Improves I-cache access time, energy, and area



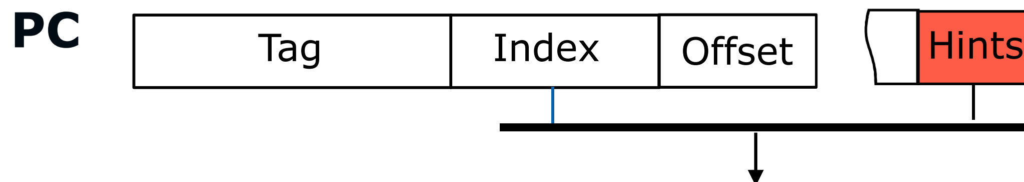
Cache Hints

- General mechanism to attach compiler generated hints
  - Basic-Block granularity
  - No effect on instruction footprint



- Cache placement hints
  - At what cache level it is profitable to place data
  - Heuristic: exclude infrequent and/or high mis-rates blocks

- Cache redistribute hints
  - Hints used as part of the cache index



Agenda

- Motivation
- BLISS Overview
- Front-End Optimizations
- Tools and Methodology
- Evaluation for Embedded Processors
- Conclusions

Evaluation Methodology

- Intel XScale PXA270 processor
  - Single issue in-order execution
  - Simulated with Simplescalar & Wattch toolsets
  - MediaBench benchmark suite

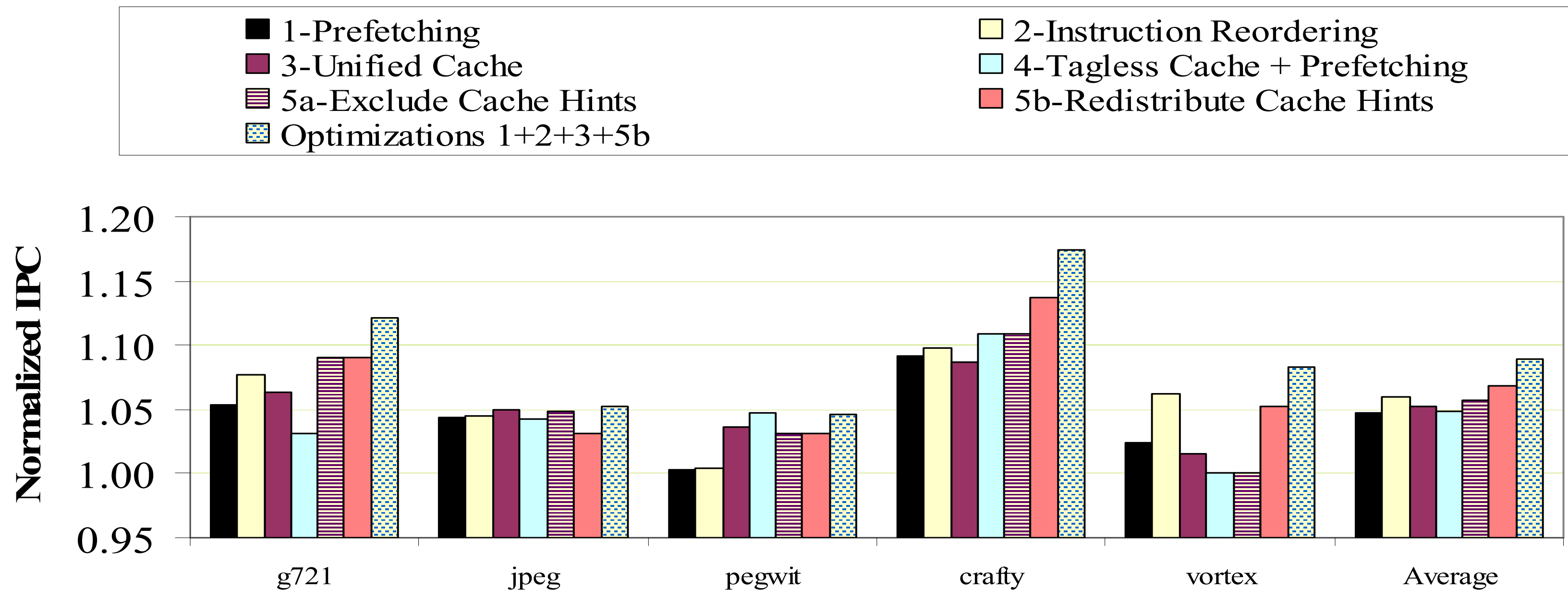
- BLISS code generation
  - Static binary translation from MIPS executables
  - Front-end optimizations performed during translation

- Instruction Reordering
  - Pettis and Hansen block-level positioning

Agenda

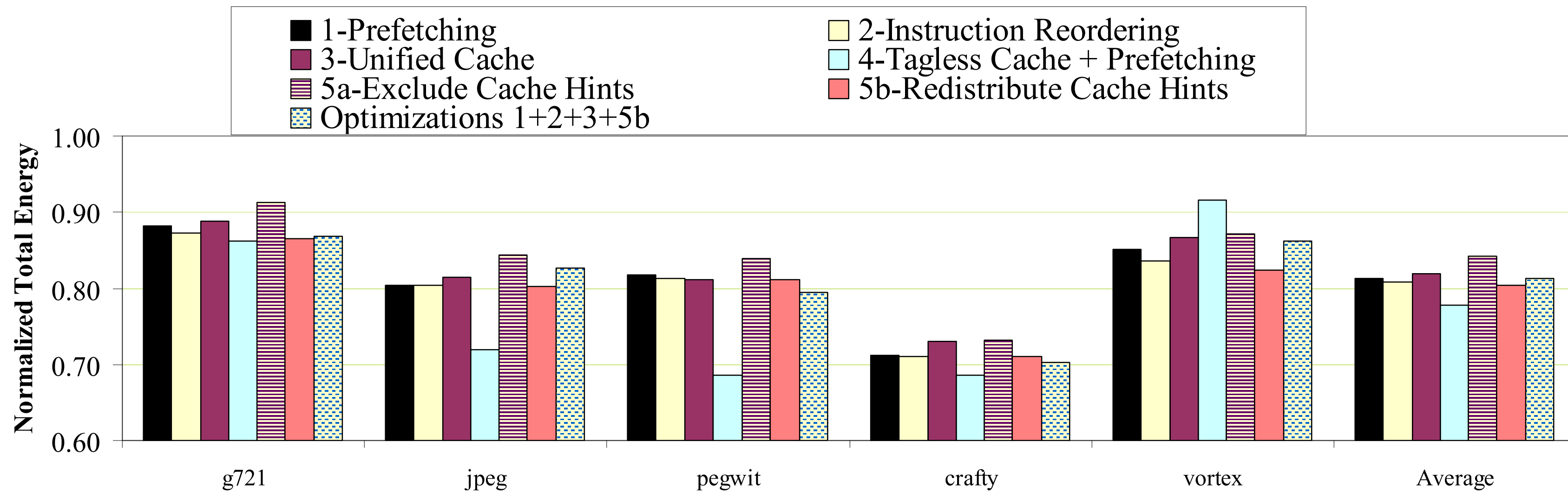
- Motivation
- BLISS Overview
- Front-End Optimizations
- Tools and Methodology
- Evaluation for Embedded Processors
- Conclusions

Performance Analysis



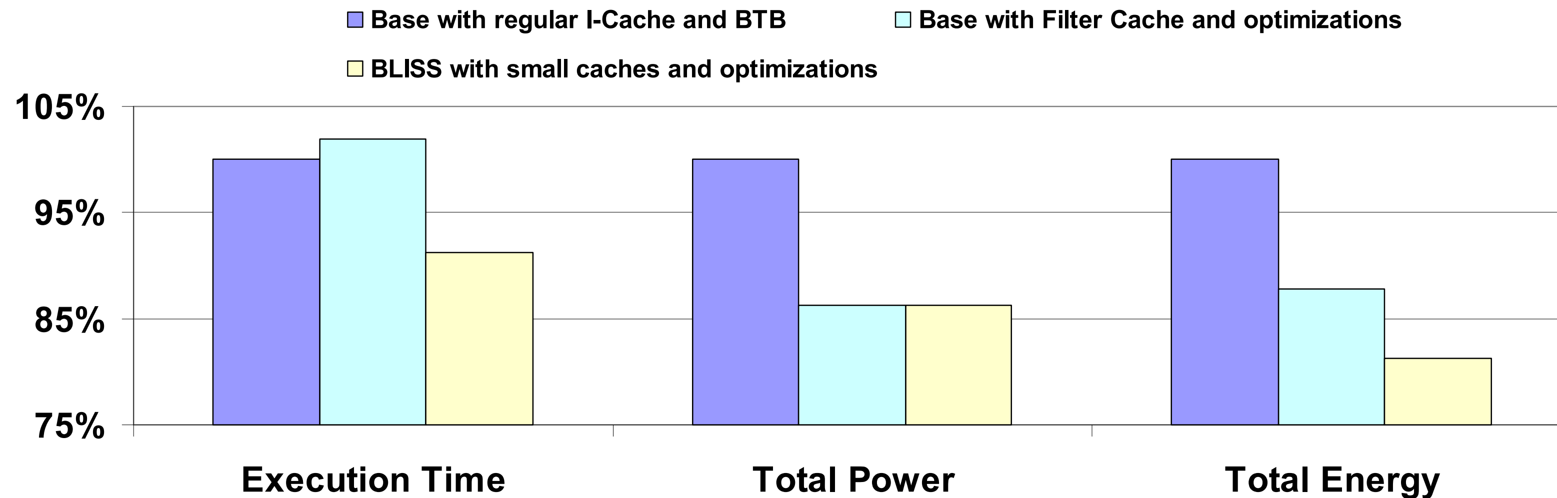
- Instruction Prefetching & Reordering ⇨ consistent performance
- Unified I-cache and BTB ⇨ for programs stressing BTB
- Tagless I-cache ⇨ for programs with BB size of 4 instructions
- Cache hints ⇨ consistent performance

Total Energy Analysis



- Tagless I-cache achieves lowest energy
  - Except for vortex due to its large BBs
- Combination leads to 19% total energy savings over base

BLISS Vs. Filter Cache



- Filter cache (tiny cache) proposed by kin et al.
  - Using similar optimizations
- BLISS achieves similar power reduction with
  - 9% performance improvement
  - 19% total energy improvement

Conclusions

- BLISS: a block-aware instruction set
  - Block descriptors separate from instructions
  - Expressive ISA to communicate software info and hints
- Enabled front-end optimizations
  - Efficient instruction reordering
  - Accurate instruction prefetching
  - General mechanism to implement cache hints
  - Unified instruction cache and BTB
  - Tagless instruction cache
- Result: Low-Power + Performance **+** Energy
  - 9% performance improvement
  - 16% total power improvement
  - 19% total energy improvement

Questions?

Microarchitecture parameters

	XScale PXA270	
	Base	**BLISS**
Fetch Width	1 inst/cycle	1 BB/cycle
Regular BTB	64-entry, 4-way	64-set, 4-way
Small BTB	16-entry, 2-way	16-set, 4-way
Regular I-cache	32 KBytes, 32-way, 32B Blocks, 2-cycle access	
Small I-cache	2 KBytes, 2-way, 32B Blocks, 2-cycle access	
BBQ	–	4 entries
Execution	single-issue, in-order with 1 INT & 1 FP unit	
Predictor	256-entry bimod with 8 entry RAS	
IQ/RUU/LSQ	16/32/32 entries	
D-cache	32 KBytes, 4-way, 32B blocks, 1 port, 2-cycle access	
L2-cache	128 KBytes, 4-way, 64B blocks, 1 port, 5-cycle access	
Main memory	30-cycle access	