A Low-Overhead Dynamic Optimization Framework for Multicores

Christopher W. Fletcher[1], Rachael Harding[1], Omer Khan[2], Srinivas Devadas[1]

[1]Massachusetts Institute of Technology; Cambridge, MA, USA
[2]University of Connecticut; Storrs, CT, USA

CSAIL
Presented during the proceedings of PACT'2012, September 20th

Dynamic Optimization

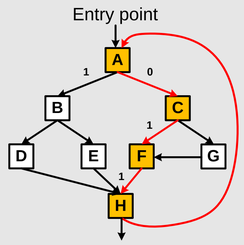
"Applications spend 90% of their time in 10% of a static binary" - Pareto principle

Dynamic optimization:

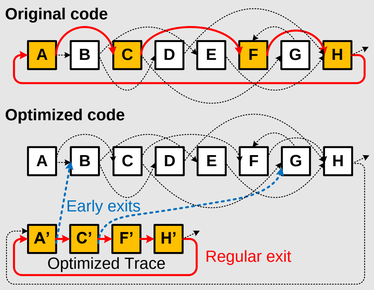
1.) **Profiles applications for *hot paths*** *(consecutive basic blocks that are evaluated together with high probability)*
2.) **Optimizes those hot paths** into contiguous ***traces*** at runtime
3.) **Enables applications to execute instructions from traces**

Code fragment

Hot path in **orange**



Instruction cache fetch pattern

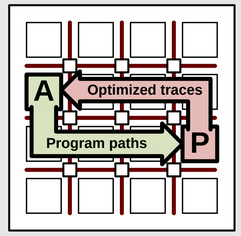


Partner Core Motivation

Dynamic optimization is **loosely coupled** and **memory intensive**
- Profiling/optimization steps do not block application (similar to prefetching)
- Optimizer working sets are similar in size to modern L1 caches (32-64 KB)

These properties motivate a Partner core-based design:
1.) The application runs on the "App" core
2.) The dynamic optimizer runs on a Partner core

High level:



A = App core
P = Partner core

Research question:

Is it possible for the Partner core to keep up (rate match) with the App core, given minimal dedicated hardware support?

Why try to minimize dedicated hardware?

Homogeneous multicores

Benefits:

Reduced verification costs, ease of design, greater code compatibility
***** Additionally, Dynamic optimization can be scheduled to *any* core

Any dedicated hardware cost is replicated per-core.

Results

Method

Chip architecture: Homogeneous multicore
Core model: In order, single issue
Memory: 32 KB L1 I/D Caches (4 ways), 1 MB shared/unified L2 (8 ways)
Metric: Trace coverage, the % of dynamic instructions executed within traces

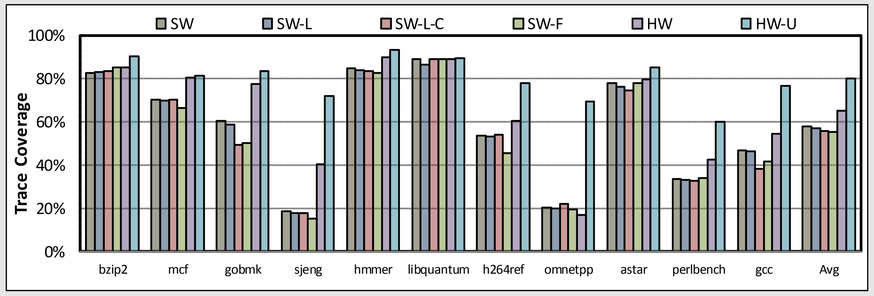
Rate Matching

	Max Trace Length (BB/instr)	Trace Occurrence Threshold	L1 TCache Capacity (in traces)	Network Latency (cycles)	Network Buffer Depth (in flits)
SW	16/128	8	16	16	16
SW-L	16/128	8	16	256	16
SW-L-C	16/128	8	16	256	256
SW-F	16/128	8	16	16	16
HW	16/128	1	16	1	∞
HW-U	22/1024	1	512	1	∞

System Configurations

Other:

- Partner core clock frequency reduced by 10x
- L1 TCache is fully-associative; `helper_thread.o` is a single-cycle (magic) operation



Network

Messages Started Due to:

Backwards Branches / Function Calls	55%
Trace Early Exits	30%
Trace Regular Exits	15%

Other Network Statistics:

Msgs. Dropped due to Network Contention	36%
Messages Shorter than Branch Limit	50%
Average Network Injection Rate	> 100 cycles
Average Network Utilization	12% - 29%

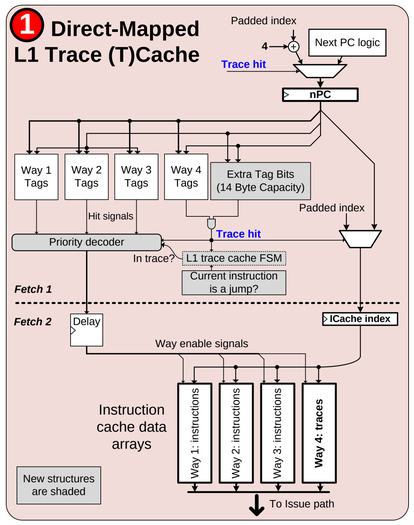
Branch Prediction Case Study

- App/Partner core system + static branch predictor has same branch prediction accuracy as single core baseline + hybrid branch predictor
- Our system's performance breaks even with a baseline using the same branch predictor

Systems adds < 50 Bytes of dedicated hardware per core

System

1 Direct-Mapped L1 Trace (T)Cache



2 Hot Path FSM

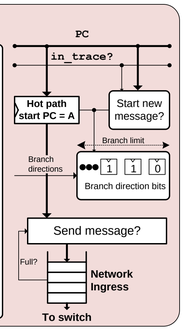
Start new message?

1.) If app is **not** in a trace and app hits a **backwards branch** or **function call**
3.) If app **is** in a trace and exits from that trace (early or regular exit)

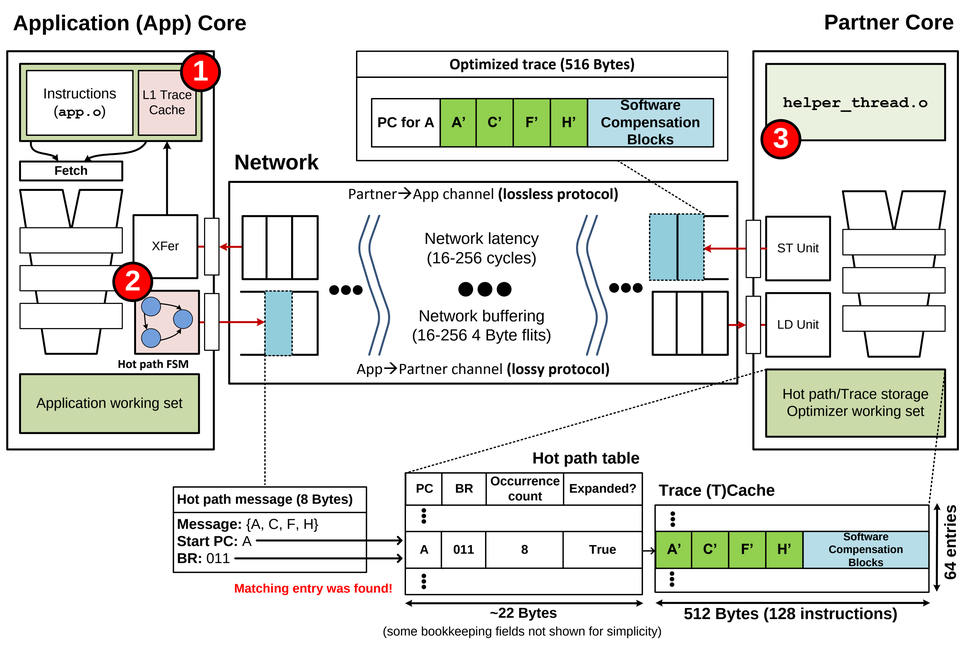
Try to send message?

1.) If Message length (b. blocks) exceeds branch limit
2.) App enters trace (hits in L1 Trace Cache)

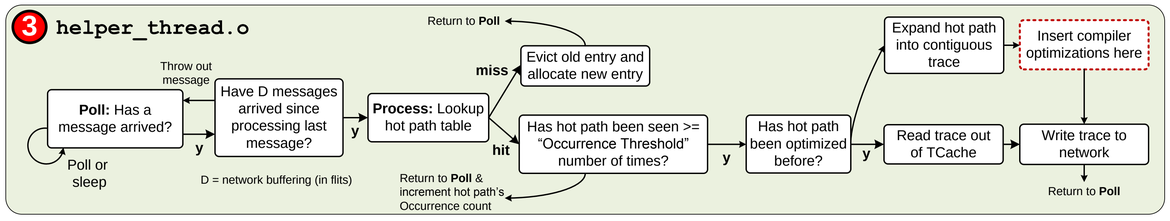
If network is full, drop message & reset FSM.



Application (App) Core / Partner Core



3 helper_thread.o



Example: Nested Loops (trace length = up to 2 basic blocks)