A Low-Overhead Dynamic Optimization Framework for Multicores*

Christopher W. Fletcher†, Rachael Harding†, Omer Khan‡, and Srinivas Devadas†
†Massachusetts Institute of Technology; Cambridge, MA, USA
‡University of Connecticut; Storrs, CT, USA
{cwfletch, rhardin, devadas}@mit.edu, khan@uconn.edu

ABSTRACT

This paper argues for a "less is more" design philosophy when integrating dynamic optimization into a multicore system. The primary insight is that dynamic optimization is inherently *loosely-coupled* and can therefore be supported on multicores with very *low-overhead* by using a Partner core. We exploit this property by designing a dynamic optimizer composed of a two-core partnership that requires a minimal amount of dedicated hardware and is resilient to (a) reducing the Partner core's clock frequency, (b) changing the Partner core's placement on the multicore die and (c) varying the latency of dynamic optimization operations.

Categories and Subject Descriptors

C.5 [**Computer Systems Organization**]: Computer System Implementation—*General*

Keywords

Dynamic optimization, Multicores, Helper threads

1. INTRODUCTION

This paper presents a low-overhead dynamic optimization framework for multicores. A dynamic optimizer (a) monitors an application (App) for *hot* program paths at runtime, (b) re-compiles hot paths into optimized/contiguous traces [1] and (c) provides the application a mechanism to execute from within traces.

Dynamic optimization is a naturally *loosely-coupled* yet memory-intensive process. The canonical dynamic optimization flow consists of two entities: the App which runs in one context and a Helper thread which monitors the App and provides recompilation support once hot paths are detected. Like prefetching, dynamic optimization can take place in the background; the App does not depend on the Helper thread to make forward progress. This non-blocking property makes multicores well suited to support dynamic optimization because the App and Helper thread can be run on separate App and Partner cores, putting less pressure on the App core's private memory hierarchy and pipeline. To put this in perspective, dynamic optimizers traditionally need a large amount of memory (over 50 KBytes) to track, store and expand traces [1, 3, 5], which is comparable in size to today's L1 caches. The open question is whether a Helper thread running on a separate Partner core can *rate match* the App core; that is, can it run in the background and still deliver traces that the App will be able to use.

Duesterwald et al. made an important observation that when profiling application hot paths, simple heuristics achieve on par the same quality of results as complex heuristics, yet incur significantly less storage [2]. Our work extends this result by exploiting how the dynamic optimization process is *latency and noise tolerant* due to hot paths being repetitive by nature.

To exploit latency and noise tolerance, we show how to implement dynamic optimization on a multicore with very small area, power and performance overheads (Section 2). In particular we show how with little dedicated hardware, large inter-core network latencies and a reduced power envelope, the Partner core can still detect, expand and return a majority of program hot paths before the main application changes phase (Section 3).

2. SYSTEM ARCHITECTURE

Our system changes program execution at *run-time only* and works with unmodified application binaries. Once an application is loaded onto the App core, the operating system spawns a fixed Helper thread on some other core (the Partner core). The high-level system is shown in Figure 1 and is broken into the following main components:

(1) Network. A mechanism for the App core to send data to the Partner core, and vice versa. In our system, the network behaves like a first-in-first-out queue which can be replaced by a conventional Network on Chip (NoC).

(2) Hot path FSM (App core). A hardware finite state machine (FSM) that detects candidate hot paths and writes messages describing those hot paths to the network. The FSM starts a new message when (a) it is not in the middle of building another message and (b) the App fetches a backwards branch (as done in [2]), sees a function call, or a trace currently executing on the App core exits. Each hot path message consists of the PC address for the start of the hot path, a bit vector representing up to 16 taken/not-taken consecutive branch directions on the path (BR), and a path length field (each message is $\sim$ 64 bits in size). If the network between cores is full when a message is ready to be sent, the message is dropped (instead of being stored in a dedicated table).

(3) Helper thread (Partner core). A software routine that reads and records hot path messages from the network. When a message has been seen *enough* times (8 in this paper), the Helper thread uses the message's PC and BR to reconstruct the sequence of dynamic instructions that make up the hot path, optimizes those instructions as a trace, caches the trace in the Partner core data cache, and sends the trace back to the App core through the network. Each trace is $\leq$ 128 instructions in length (longer traces are cut short) and the Helper thread stalls while it waits for traces to be sent through the network to the App core.

*This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1122374.

Copyright is held by the author/owner(s).
PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.
ACM 978-1-4503-1182-3/12/09.

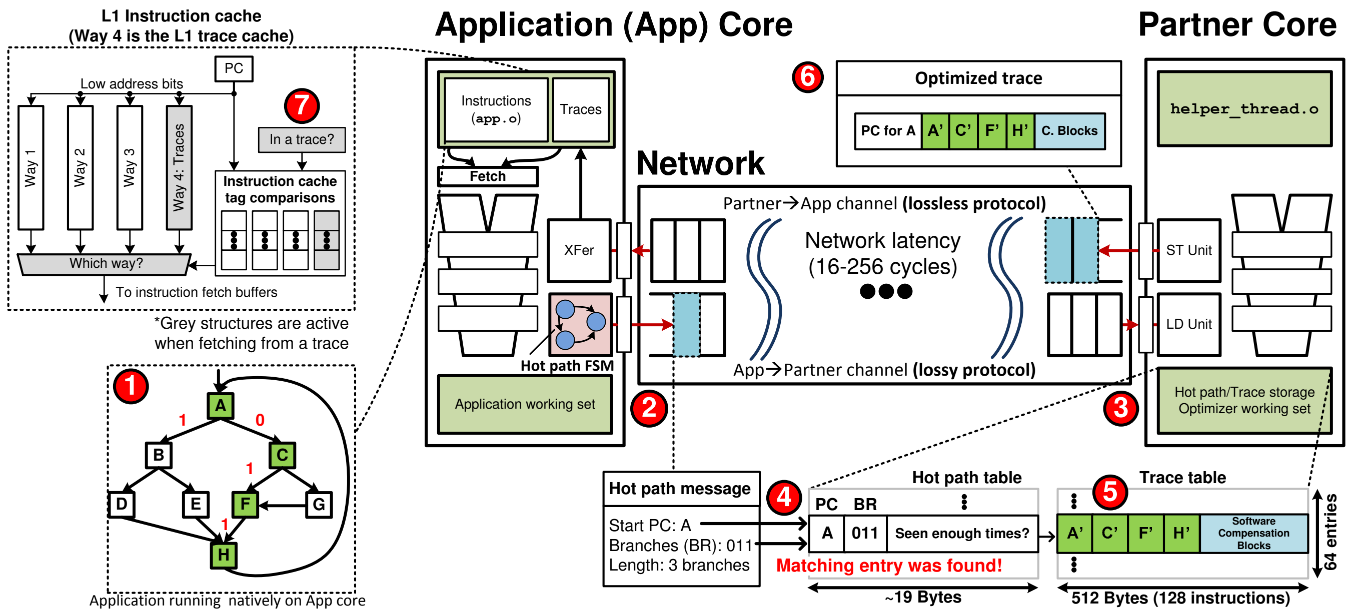

Figure 1: An end-to-end example showing our system optimize the small loop shown in (1). ((1)-(2)) The hot path FSM detects that $\{A, C, F, H\}$ is hot (H is a backwards branch) and sends a hot path message to the Partner core. The Helper thread reads the hot path message ((3)), sees that the message has been seen enough times to warrant optimization ((4)), and ((5)) creates an optimized trace for the message. Finally, the trace and its start PC are sent back to the App core ((6)) and fetched in place of regular instructions ((7)).

(4) L1 trace cache (App core). A hardware buffer from which traces, sent by the Partner core, can be fetched and executed in place of regular instructions. We implement the L1 trace cache as a way in the L1 instruction (I)Cache, managed as a direct-mapped structure with a small number of entries (16 for this paper). The direct-mapped lookup is cheap in terms of dedicated hardware cost ($\sim$ 14 Bytes) and combinational latency (several additional gate delays). When the Helper thread is disabled, the trace way can be used to store normal instructions.

3. RESULTS

We evaluate our system over the SPEC06-int benchmark suite using the SESC simulator [4], and simulate for 3 billion instructions with a 1-20 billion instruction warmup depending on the benchmark. Both the App and Partner core have mid-range core models (2-issue out-of-order with a 64-entry reorder buffer) with a private/shared cache hierarchy (32 KB, 4-way, single-cycle L1 I/D caches and a 1 MB, 8-way, 10 cycle unified L2 cache).

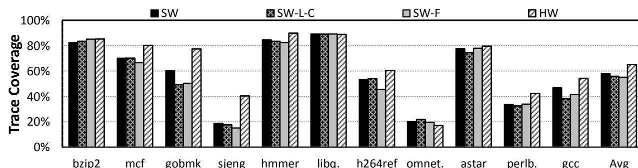

Figure 2: Trace coverage for the representative system design points discussed in Section 3.

Figure 2 shows the *trace coverage* (the percent of dynamic instructions executed from within traces) for each benchmark. Higher trace coverage is better: a dynamic optimizer's effectiveness scales with how many instructions the App executes from within traces. SW implements a software-based Helper thread on a Partner core that is adjacent to the App core on the chip (inter-core latency $= 16$ cycles). SW-L-C is the same as SW, but the Partner core is scheduled to the "other side" of the chip (the latency and capacity in the network is 256 cycles and 256 buffer slots, respectively). SW-F is the same as SW, but the Partner core clock frequency is $10\times$ less than the App core and network. HW is an *idealized* Helper thread that performs the same dynamic optimization routine as SW but in a single cycle and with no network latency/bandwidth overheads. Furthermore, HW assumes a fully associative L1 trace cache (which avoids conflict misses but requires a dedicated fully-associative lookup on the App core).

Overall, SW, SW-L-C and SW-F maintain competitive trace coverage with HW (and have a 7.2%, 9.2% and 9.9% difference, respectively). As another point of comparison, SW attains on average 5.8% more coverage on bzip2, mcf and gcc relative to [3], which also evaluates these benchmarks and assumes large hardware structures to support the trace collection process. Most benchmarks (all except bzip2, omnet and h264ref) show slight dips in coverage for SW-L-C/SW-F relative to SW, which is caused by the increase in round-trip trace collection latency. The three outlier benchmarks suffer from L1 trace cache thrashing: when round-trip latency increases, the Helper thread is able to process fewer hot paths per program phase, which leads to fewer L1 trace cache evictions. Especially for omnet, which has repetitive phases and many hot paths per phase, slowing down the Helper thread reduces the number of traces that are still hot yet evicted by (possibly cold) traces.

4. REFERENCES

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, 2000.

[2] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *SIGPLAN Not.*, 2000.

[3] Sanjay J. Patel, Tony Tung, Satarupa Bose, and Matthew M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, 2000.

[4] Jose Renau. Sesc: Superescalar simulator. Technical report, 2002.

[5] Weifeng Zhang, Brad Calder, and Dean M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, 2005.