

# A Framework to Accelerate Sequential Programs on Homogeneous Multicores

Christopher W. Fletcher<sup>1</sup>, Rachael Harding<sup>1</sup>, Omer Khan<sup>2</sup>, and Srinivas Devadas<sup>1</sup>

<sup>1</sup>Massachusetts Institute of Technology, Cambridge, MA, USA, {cwfletch, rhardin, devadas}@mit.edu

<sup>2</sup>University of Connecticut, Storrs, CT, USA, khan@uconn.edu

**Abstract**—This paper presents a light-weight dynamic optimization framework for homogeneous multicores. Our system profiles applications at runtime to detect *hot* program paths, and offloads the optimization of these paths to a Partner core. Our work contributes two insights: (1) that the dynamic optimization process is highly insensitive to runtime factors in homogeneous multicores and (2) that the Partner core's view of application hot paths can be noisy, allowing the entire optimization process to be implemented with very little dedicated hardware in a multicore.



## 1 INTRODUCTION

With forecasts predicting hundreds of cores in the near future, designers have an ever increasing number of parallel processing units at their disposal. One way to use these additional cores is to parallelize applications further. Due to the *parallelism wall*, however, researchers have begun to use spare cores to augment other cores that run (serial) user workloads.

Dynamic optimization is a traditional approach to speedup serial workloads in single core and multi-threaded environments. A dynamic optimizer's job is to detect frequently executed application code (called *hot paths*) and recompile that code into contiguous *traces* [4]. This process occurs at runtime, allowing dynamic optimizers to exploit *input data-dependent* optimizations that static compilers cannot realize. Indeed, previous dynamic optimizers have reported significant speedups [1], [19].

The problem with previous frameworks is their required *memory footprint* and *dedicated hardware overhead*—typically over 50 KBytes—to track, store and expand traces [19], [11]. In emerging processors such as homogeneous multicores, adding dynamic optimizer-specific hardware storage to each core is clearly undesirable. Furthermore, storing both optimizer and application working set in shared L1 cache causes significant contention, given multicore systems' typical L1 capacities.

In this paper we present a framework to support dynamic optimizations on homogeneous multicores. The key insight enabling our work is that while dynamic optimization is a memory-intensive process, it is naturally *loosely-coupled*. Like prefetching, dynamic optimization can take place in the background and does not block the application. On a homogeneous multicore, this means the application and optimizer (implemented as a helper thread(s)) can be run on separate cores (*the App and Partner cores*), thereby preventing pipeline and memory contention between the two.

This paper contributes the following insights:

(1) That the dynamic optimization process is highly insensitive to runtime factors in a homogeneous multicore.

(2) That a dynamic optimizer's view of application hot paths can be noisy, yet still capture the "big picture," since the most beneficial hot paths occur thousands or millions of times.

Using these insights, we develop a 2-core dynamic optimization system that:

(1) Consumes less power than a baseline system with a single core running an application without our framework.

(2) Maintains comparable trace coverage to previous dynamic optimization systems that require significant dedicated hardware (e.g., [11]).

(3) Is implemented using < 50 Bytes of dedicated hardware per core.

### 1.1 Trace structure

Throughout this work, traces are defined as single-entry, multi-exit blocks of instructions as in Dynamo [1] (thereby allowing the system to adopt the types of optimizations used in Dynamo). Branches remain in the trace, with their direction possibly reversed so that the "not taken" direction stays on the trace. If a branch is taken, we say the trace has *early exited*, transferring control flow to a software compensation block attached to the end of the trace, which contains an direct jump back to the application. Otherwise, the trace *regularly exited*, taking a jump back to the application at the bottom of the trace.

### 1.2 Related work

Previous work has studied dynamic optimization in single core and simultaneous multithreading (SMT) environments, using customized hardware or software memory to support the optimization process [1], [11], [19]. Replay [11] and Trident [19] store and consolidate hot trace description messages in dedicated hardware predictor tables. Like our work, Trident is also an *event-driven* dynamic optimizer but monitors events in hardware tables, while we perform these operations in software. Additionally, Trident is based in a complex core/SMT setting where the application and helper thread run on the same core. Dynamo is a pure software system that runs as a software interpreter until it detects a hot path, and stores optimized traces in software memory along with the application [1].

Dynamic parallelization is another approach to speed up applications in a multicore environment [18], [2], [17],

[7]. These approaches identify parallel code within an application and create micro threads on independent cores to run that parallel code. Micro threads are speculative—if data dependencies are violated [2], [17], [7] or trace early exits are taken [18], the system must rollback. In contrast, our system focuses on optimizing sequential code and executes software compensation routines instead of performing complete rollbacks.

Helper threads running in spare hardware contexts have been studied extensively, primarily in a prefetching context [8], [10]. Changhee et al. [8] study loosely-coupled helper threads in multicores but limit their scope to prefetching. Lau et al. [10] present a Partner core framework where the Partner core is a different (typically weaker) type of core. We assume the same microarchitecture for both the App and Partner core. [10] mentions several possible application domains—not including dynamic optimization—and also performs a case study on prefetching.

## 2 SYSTEM ARCHITECTURE

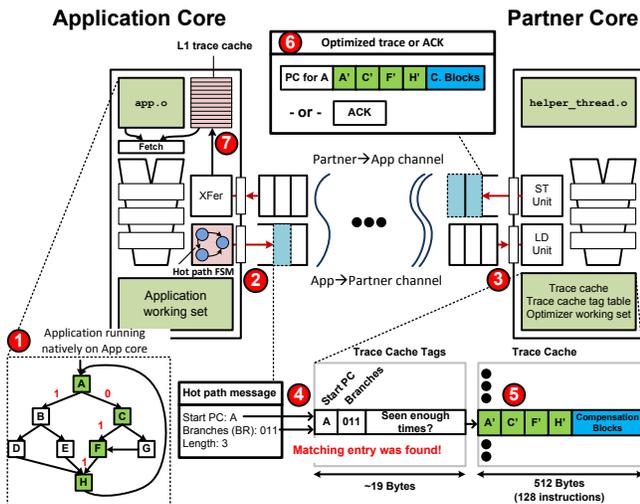


Fig. 1: An end-to-end example showing our system optimizing the small loop shown in 1.

Our system changes program execution at *run-time only* and works with unmodified program binaries. We will assume a MIPS-like ISA for the rest of the paper, although the ideas presented do not impose this. Once an application is loaded onto the App core, the operating system spawns a fixed Helper thread on the Partner core. Alternatively, the Partner core code can be stored in non-volatile read-only memory on the chip where it can be deployed to different cores as needed.

To guide the reader we show an end-to-end example in Figure 1, that we will refer back to throughout the rest of the section. The major components of the system are the App and Partner cores, communication channels between the cores (3, 6 in the figure), the Hot Path Finite State Machine (HP-FSM) which profiles the applications running on the App core (2), and the L1 trace cache which allows for traces to be fetched on the App core (7).

The channels connecting App and Partner cores can be implemented using an unmodified, general-purpose NoC. Each channel is modeled as an in-order, first-in-first-out buffer with latency, buffer depth, and flow

control. The communication channels are *lossless*: any message successfully written to a channel will eventually be received at the other end.

### 2.1 2 Hot Path FSM (HP-FSM)

The HP-FSM, on the App core, starts the trace creation process by creating digests of which paths are “hot” in the application. Each digest (called a hot path message, or HPM) consists of the start PC for the hot path, a bit vector (BR) representing taken/not-taken branch directions on the path, and a length field indicating the number of valid branch direction bits. The HP-FSM runs alongside the application and starts a new message when the App core:

(1) is not executing in a trace and takes a branch whose target address is less than the current PC (a *backwards branch*).

(2) is not executing in a trace and executes a jump-link (function call) instruction.

(3) is executing in a trace and exits from that trace (after an early or regular exit).

These start conditions are all hints that the application is executing a hot path, as discussed in [5].

Once a message starts, other start-of-trace events are ignored until the current message is complete. This “greedy” behavior allows the HP-FSM to be implemented with just enough memory to buffer the current message. If the HP-FSM encounters a loop, the path encoded in the message is effectively an unrolled version of the loop. When a new message begins, the current App core PC is written to a dedicated register and the BR register is reset. For subsequent branch instructions, taken/not-taken bits are shifted into BR in the order that those branches appear in the instruction stream.

The HP-FSM completes its current message when the number of branches in the hot path reaches a statically determined *branch limit*, or when the App core starts executing from a trace. Messages completed before the branch limit has been reached (called *short messages*) prevent code from being unreachable because of mis-aligned traces.

### 2.2 3 App → Partner Channel

The App→Partner channel transports completed HPMs from the App to Partner core. With our HPM structure (PC, BR, and the length field), each HPM is < 64 bits and will likely be broken into several flits, depending on network width.

To prevent filling the network with HPMs, we use Partner core-controlled flow control to limit the number of HPMs written to the network at a time. After the HP-FSM writes a single HPM to the network, it will not write another HPM until the Partner core has sent back an ACK, indicating that it has received the last HPM.<sup>1</sup> In addition to reducing network congestion and guaranteeing that the Partner core will always immediately consume HPMs at its network ingress, the ACK scheme prevents the Partner core from reading “stale” HPMs [5].

1. Also, if the HP-FSM tries to inject an HPM into the channel when the channel is full for other reasons, the HP-FSM drops the HPM and resets.

### 2.3 ④-⑤ Trace expansion & optimization

The Helper thread reads incoming HPMs from the network, decides when a hot path is worth expanding into a trace, and constructs/sends traces back to the App core. The Helper thread maintains two statically-sized software structures: a *trace (T)Cache tag table* and a *trace (T)Cache*. One TCache entry of size *trace size* instructions is allocated for each tag table entry to cache its corresponding trace.

The TCache tag table is fully-associative with least-recently-used (LRU) replacement. Table entries are indexed by {PC, BR}, allowing the Helper thread to track multiple program paths originating from the same start PC. Each tag table entry contains an *expanded* flag and an *occurrence count* (explained below).

To start, the Helper thread polls<sup>2</sup> the App→Partner channel until a HPM arrives, at which point the Helper thread performs the following actions:

(1) **Lookup {PC, BR} in the TCache tag table.** If the entry is not present, evict (if the tag table is full) an entry and allocate a new entry with *occurrence count* = 1 and *expanded* = *false*. If the entry is present (④), increment its *occurrence count*. If *occurrence count* = *occurrence threshold* and *expanded* = *false*, reset *occurrence count* to 1 and proceed with steps 2-4 below. If *occurrence count* = *occurrence threshold* and *expanded* = *true*, reset *occurrence count* and skip to step 4. If none of the above conditions hold, the Helper thread writes an ACK message to the network (or stalls until the network has enough space to allow this action to proceed) and then returns to the initial message polling state.

(2) **Trace expansion.** Expand the HPM into a contiguous sequence of instructions, forming a trace. The Helper thread copies instructions from the App core’s instruction address space into the TCache, starting at the start PC (⑤) and using BR to decide whether to follow each branch. If the Helper thread reads an indirect function return (e.g., *jr* in MIPS) it stops expanding the trace *unless* a matching call-and-link (e.g., *jal*) has appeared at some point earlier in the trace from which the PC target can be derived. Traces are always prematurely terminated at other indirect branches (e.g., *jalr*); we found these instructions to be rare in our benchmarks.

(3) **Trace pre-processing.** Remove direct jumps and matching call/return pairs from the trace and change branch directions so that “not-taken” keeps control flow on the trace. For each branch instruction, create a software compensation block at the end of the trace which jumps back to the application if the branch is taken.

(4) Write the full contents of the trace, along with its starting PC, to the network (see next section) and return to message polling behavior. The full software routine takes approximately 2500 cycles.

### 2.4 ⑥ Partner → App Channel

The Partner → App channel transports traces and ACKs from the Partner core to the App core. Each ACK consists of a single flit. Each trace consists of the trace’s start PC followed by the trace itself (each instruction takes up one flit and each trace may be hundreds of instructions). While the Helper thread is writing the trace on the Partner→App channel, it cannot do other work.

2. Alternatively, the Partner core can go to sleep when it starts to poll, and wakeup via interrupt when a message arrives.

Furthermore if the Partner→App channel fills, the Helper thread stalls until space is available, since the trace is not functionally correct if any portion of a trace is lost in transit. We note that stalling the Partner core does not degrade application performance as it will not block the App core.

Like the Partner core in the case of the App→Partner channel, the App core will always consume any network flits (either an ACK or trace) as soon as they are available at the App core network ingress.

### 2.5 ⑦ Trace Execution on the App Core

Upon arriving at the App core, ACK messages are consumed and traces are moved to a structure called the *L1 trace cache*, or *L1 TCache*. Conceptually, the L1 TCache is the first level cache for the Helper thread’s software TCache.

For this work, we implement the L1 TCache using one of the ways in the L1 instruction (I)Cache (Figure 2). To minimize dedicated hardware cost, trace lookups within the dedicated ICache way are made in a direct-mapped fashion that uses (mostly) the same logic already in place to support each way in a set-associative ICache. Each trace in the trace way occupies a fixed amount of space equal to *trace size* (Section 2.3). When the tag array for the trace way hits, (1) a small FSM (implemented as a set/reset latch) forces all subsequent ICache accesses to read from the trace way, (2) the PC register is loaded with offsets that point to the start of the trace in the ICache data array and (3) the entire ICache tag array plus the data arrays for the other cache ways shut off via clock gating. While inside of a trace, normal PC increment logic is used to index into the trace way and no tag array lookups are made (note that all branches along a trace are PC-relative). When a jump instruction, the sole end-of-trace condition, is executed, the PC is reset to a normal instruction address and the ICache tag array plus other data arrays are re-enabled.

When the App core indexes into the trace way tag array to see if it is time to enter a new trace, *the entire application PC must be compared against the candidate start of trace PC*. Because the start-of-trace location in the ICache data array does not necessarily correspond to where the first instruction in the trace would normally be mapped, it is not sufficient to perform a tag check on only the PC tag (upper) bits. To prevent *false positives*, we store the rest of each trace’s start PC in a direct-mapped structure that is indexed in parallel with the tag arrays during the fetch process (see Figure 2).

Aside from the dedicated table, the entire design adds several gate delays to the fetch stage critical path (for muxing and priority logic), saves power by shutting off tag and data array lookups while the App core executes inside of a trace, and reduces non-trace ICache capacity by one way.

## 3 RESULTS

We discuss the dedicated hardware overhead and power usage of our framework. A complete evaluation of our system can be found in [5]. There we show how the Partner core can be decoupled from the App core without seriously degrading quality of results. Notably, the number of dynamic instructions executed from within traces (called *trace coverage* in related work) is resilient to effects such as varying network and Helper Thread

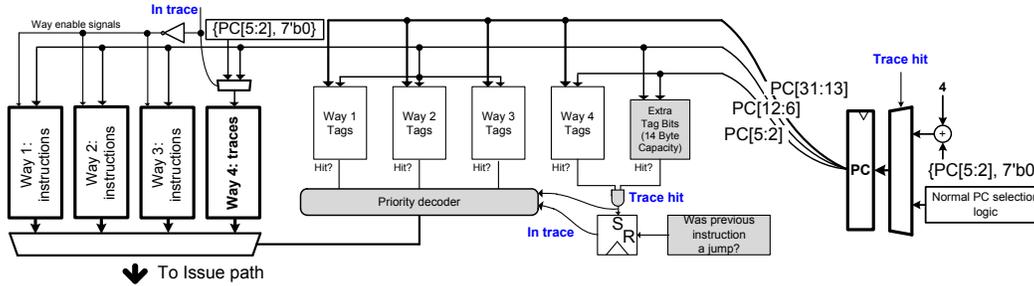


Fig. 2: A dedicated-way, single-cycle L1 TCache design (grey structures are added to support our system). To determine a trace hit, the tag  $PC[31 : 13]$  is checked in the unmodified tag array for the trace way while  $PC[12 : 6]$  is compared in the “Extra Tag Bits” table.

latency. Trace coverage’s latency tolerance allows power-saving techniques such as frequency scaling to be utilized with minimal coverage and performance impact. [5] also evaluates the performance of the system and shows that when the system is utilized as a traditional trace cache [4], speedup is on par with a single-core baseline, making our system a viable platform for further dynamic optimizations.

### 3.1 Dedicated Hardware Overhead

Our system’s two main structures are the HP-FSM and the L1 TCache. The HP-FSM requires  $\sim 9$  Bytes for PC, branch directions and FSM state. In the dedicated-way L1 TCache design (Section 2.5), 7 extra tag bits must be stored to make complete PC comparisons for the trace way, which requires 14 Bytes given a 16-entry L1 TCache (see Figure 2). While there are small additional overheads—such as the flag that indicates that we are in a trace—in total the system requires less than 50 Bytes of custom storage.

### 3.2 Power Usage

Table 1 summarizes the power usage of our 2-core framework, our framework with the Partner core operating at  $.1\times$  the clock frequency of the App core (2-CORE, SLOW), and a 1-core baseline without our framework. Details on the power derivation, and a discussion about the implications of the results, can be found in [5]. The 7% improvement in power dissipation is largely due to a more energy-efficient instruction fetch stage.

TABLE 1: System Power in Milliwatts.

	Total	App Core	App Fetch	Partner Core	L2	Net-work
2-CORE	151	67	25	75	9	$\sim 0$
2-CORE, SLOW	80	67	26	4	9	$\sim 0$
1-CORE	86	77	33	0	9	0

## 4 CONCLUSION

In this work we presented light-weight and low-overhead mechanisms to enable dynamic optimization on homogeneous multicores. To deliver competitive quality of results, our system relies on the fact that dynamic optimization is *loosely-coupled* by nature. We showed how this property makes the system resilient to the Partner core’s operating frequency. We predict that these properties also allow for a flexible Helper thread implementation which can allow a variety of dynamic optimizations without any hardware modifications to our framework. We leave implementing compiler-style

optimization passes inside the Helper Thread to future work. As the world adopts multicore, we believe that this flexibility that comes for free in a dynamic optimization setting makes dynamic optimization an attractive use for spare silicon, especially in situations when parallelism delivers diminishing returns.

## REFERENCES

- [1] V. Bala, E. Duesterwald, S. Banerjia. *Dynamo: A transparent dynamic optimization system*. Proceedings of the conference on Programming language design and implementation (PLDI), 2000.
- [2] M. DeVuyst, D. M. Tullsen, S. W. Kim. *Runtime parallelization of legacy code for a transactional memory system*. Proceedings of the International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC), 2011.
- [3] E. Duesterwald, V. Bala. *Software Profiling for Hot Path Prediction: Less is More* Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2000.
- [4] J. A. Fisher. *Trace Scheduling: A Technique for Global Microcode Compaction* IEEE Transactions on Computers, July 1981.
- [5] C. W. Fletcher, R. Harding, O. Khan, S. Devadas. *A Framework to Accelerate Sequential Programs on Homogeneous Multicores* Computation Structures Group Memo CSG-Memo-510, June 2013. <http://csg.csail.mit.edu/pubs/memos/Memo-510/memo510.pdf>
- [6] S. Galal, M. Horowitz. *Energy-Efficient Floating-Point Unit Design* IEEE Transactions on Computers, 2011.
- [7] B. Hertzberg, K. Olukotun. *Runtime Automatic Speculative Parallelization*. Proceedings of the International Symposium on Code Generation and Optimization (CGO), 2011.
- [8] C. Jung, D. Lim, J. Lee, Y. Solihin. *Helper thread prefetching for loosely-coupled multiprocessor systems*. Proceedings of the Parallel and Distributed Processing Symposium (IPDPS), 2006.
- [9] A. B. Kahng, Bin Li, Li-Shiuan Peh, K. Samadi. *ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration*. Design, Automation and Test in Europe Conference and Exhibition, 2009.
- [10] E. Lau, J. E. Miller, I. Choi, D. Yeung, S. Amarasinghe, A. Agarwal. *Multicore Performance Optimization Using Partner Cores*. Proceedings of the USENIX workshop on hot topics in parallelism (HOTPAR), 2011.
- [11] S. J. Patel, S. S. Lumetta. *Replay: A Hardware Framework for Dynamic Optimization*. IEEE transactions on computers, Vol. 50, No. 6, June 2001.
- [12] S. J. Patel, T. Tung, S. Bose, M. M. Crum. *Increasing the Size of Atomic Instruction Blocks using Control Flow Assertions*. Proceedings of the International Symposium on Microarchitecture (MICRO), 2000.
- [13] J. Renau. *SESC simulator*. <http://sesc.sourceforge.net/index.html>, 2002.
- [14] E. Rotenberg, S. Bennett, J. Smith. *Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching*. Proceedings of the Annual International Symposium on Microarchitecture (MICRO), 1996.
- [15] Standard Performance Evaluation Corporation. *SPEC CPU benchmark suite*. <http://www.spec.org/osg/cpu2006>.
- [16] S. Thoziyoor, N. Muralimanohar, N. P. Jouppi. *CACTI 5.0* <http://www.hpl.hp.com/research/cacti/>
- [17] C. Wang, Y. Wu, E. Borin, S. Hu, W. Liu, D. Sager, T. Ngai, J. Fang. *Dynamic parallelization of single-threaded binary programs using speculative slicing*. Proceedings of the International Conference on Supercomputing (ICS), 2009.
- [18] J. Yang, K. Skadron, M. L. Soffa, K. Whitehouse. *Feasibility of Dynamic Binary Parallelization*. Proceedings of the USENIX Workshop on Hot Topics in Parallelism (HOTPAR), 2011.
- [19] W. Zhang, B. Calder, D. Tullsen. *An event-driven multithreaded dynamic optimization framework*. Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), 2005.