A Framework to Accelerate Sequential Programs on Homogeneous Multicores

Christopher W. Fletcher
MIT CSAIL
cwfletch@mit.edu

Rachael Harding
MIT CSAIL
rhardin@mit.edu

Omer Khan
University of Connecticut
khan@uconn.edu

Srinivas Devadas
MIT CSAIL
devadas@mit.edu

ABSTRACT

This paper presents a light-weight dynamic optimization framework for homogeneous multicores. A dynamic optimizer speculates that certain program paths are *hot* at runtime, optimizes those paths as if they were a single basic block, and provides the application a way to fetch the optimized code. Our approach offloads dynamic optimization to a Partner core and strives to minimize dedicated hardware overhead. Cheaply implementing dynamic optimization with a Partner core is possible because (1) dynamic optimizers are naturally *loosely-coupled* and (2) hot paths are repetitive by nature. These two properties make the optimization process time-insensitive and require minimal dedicated hardware to implement.

We show how our system enables different SPEC06int applications to execute on average 52 % of their dynamic instructions from within traces. We then show that our system is resilient to the latency of the optimization process, to the placement of the Partner core on the chip, and to the Partner core clock frequency. Given the insight that quality of results is resilient to Partner core frequency scaling, we present a 2-core design point that—without additional compiler optimizations—improves power dissipation by 7 % compared to a 1-core baseline and only degrades performance by 2 %. With the mechanisms we present, our results are attainable with < 50 Bytes of dedicated hardware.

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. While this approach has been shown to work well for data-level parallelism, performance gain tapers off for applications with more serial code or complicated inter-thread communication requirements. In light of this *parallelism wall*, researchers have begun to use otherwise idle cores to augment cores that run user workloads.

Dynamic optimization is a technique in which frequently executed *hot paths* are recompiled at runtime into contiguous *traces* [4] to exploit optimizations which static compilers are not aggressive enough to realize. Optimizations applied in previous work such as Dynamo and Trident have reported significant speedups [1, 18]. However, these dynamic optimizers require a large amount of memory (over 50 KBytes), comparable in size to today's L1 caches, to track, store and expand traces.

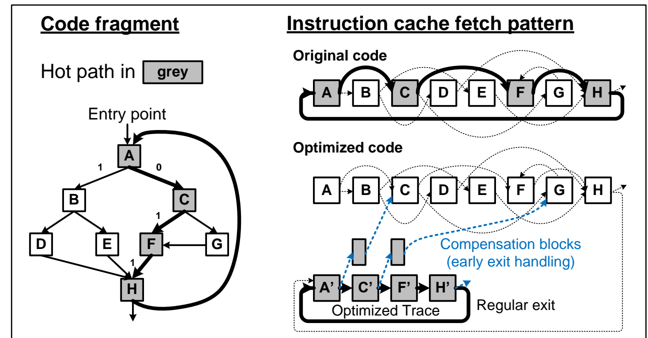

Figure 1: Code layout (in instruction cache memory) before and after trace optimizations. Thick lines correspond to the hot path through the application. In addition to instruction re-layout, each basic block in the trace may be different from the original basic block. Compensation blocks and early exit handling are discussed in Section 1.1.

We observe that while dynamic optimization is a memory-intensive process, it is naturally *loosely-coupled*. Like prefetching, dynamic optimization can take place in the background. This non-blocking property makes multicores well suited to support dynamic optimization because the application and helper thread(s) can be run on separate cores, putting less pressure on the application core's private memory hierarchy and pipeline.

In this paper we present a framework to support dynamic optimizations on homogeneous multicores. In our system, a single-threaded application utilizes two cores: an application (App) core and a Partner core. The App core (1) runs the unmodified application binary and (2) sends messages describing *hot paths* to the Partner core. We define a hot path as a sequence of consecutive basic blocks that execute atomically.[1] The Partner core, which runs a fixed Helper thread, expands hot paths in its data cache–creating contiguous *traces* [13]–and sends select traces back to the App core where they can be fetched in place of normal instructions. To illustrate the concept, Figure 1 shows an example loop before and after trace optimizations.

We focus on homogeneous multicores in this work because of their reduced verification costs, ease of design, and greater

[1]That is, if control flow enters the hot path, there is high probability that control flow will enter each basic block in the path.

code compatibility. Homogeneous multicores also increase scheduling flexibility in that the Helper thread can be scheduled to any core relative to the App core. Minimizing additional hardware is an important factor for any proposal in a homogeneous multicore setting because cost is replicated per-core.

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., [10]).

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

We evaluate our system as a trace cache in which hot paths are laid out contiguously without applying any additional compiler optimizations and find that we have comparable performance to a baseline system. While we do not apply optimizations in this work, we forecast that by applying similar optimizations as previous work inside the Helper thread, significant performance improvements could be achieved.

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 any compiler optimization 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 absolute jump instruction back to the application. Otherwise, the trace *regular exited*, taking a jump back to the application at the bottom of the trace.

Dynamic optimization systems are typically evaluated via system speedup, trace coverage (*coverage* for short), trace early exit rate and average trace size. Coverage is the percent of dynamic instructions that are executed from within traces, not counting instructions added to each trace as overhead (e.g., the early/regular exit jump instructions).[2] We measure both *coverage* and end-to-end performance and do not focus on early exit rate because traces that early exit still make forward progress in the application (unlike rollback schemes like Replay [10]).

[2]Dynamic optimization potential correlates to coverage—i.e. if coverage is 0%, no trace optimization can speedup the App core.

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, 10, 18]. Replay [10] and Trident [18] 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 speedup applications in a multicore environment [17, 2, 16, 6]. 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, 16, 6] or trace early exits are taken [17], the system must rollback somehow. 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 [7, 9]. Changhee et al. [7] study loosely-coupled helper threads in multicores but limit their scope to prefetching. Lau et al. [9] 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. [9] mentions several possible application domains—not including dynamic optimization—and also performs a case study on prefetching.

2. SYSTEM ARCHITECTURE

Our system changes program execution at *run-time only* and works with unmodified program binaries. 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. We assume that the application is compiled to the MIPS ISA for the rest of the paper. The system can be summarized by the following components:

1. Network, Section 2.1: A mechanism for the App core to send data to the Partner core, and vice versa.

2. *Hot path FSM* (HP-FSM, located on the App core), Section 2.2: A hardware finite state machine that detects hot paths, encodes them as *hot path messages* (HPMs) and writes them to the network. Each HPM consists of the PC address for the start of 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.

3. *Helper thread* (Partner core), Section 2.3: A software routine that monitors HPMs received from the network. When a message has been seen *enough* times, the Helper thread uses the message's PC and BR to reconstruct the sequence of dynamic instructions that

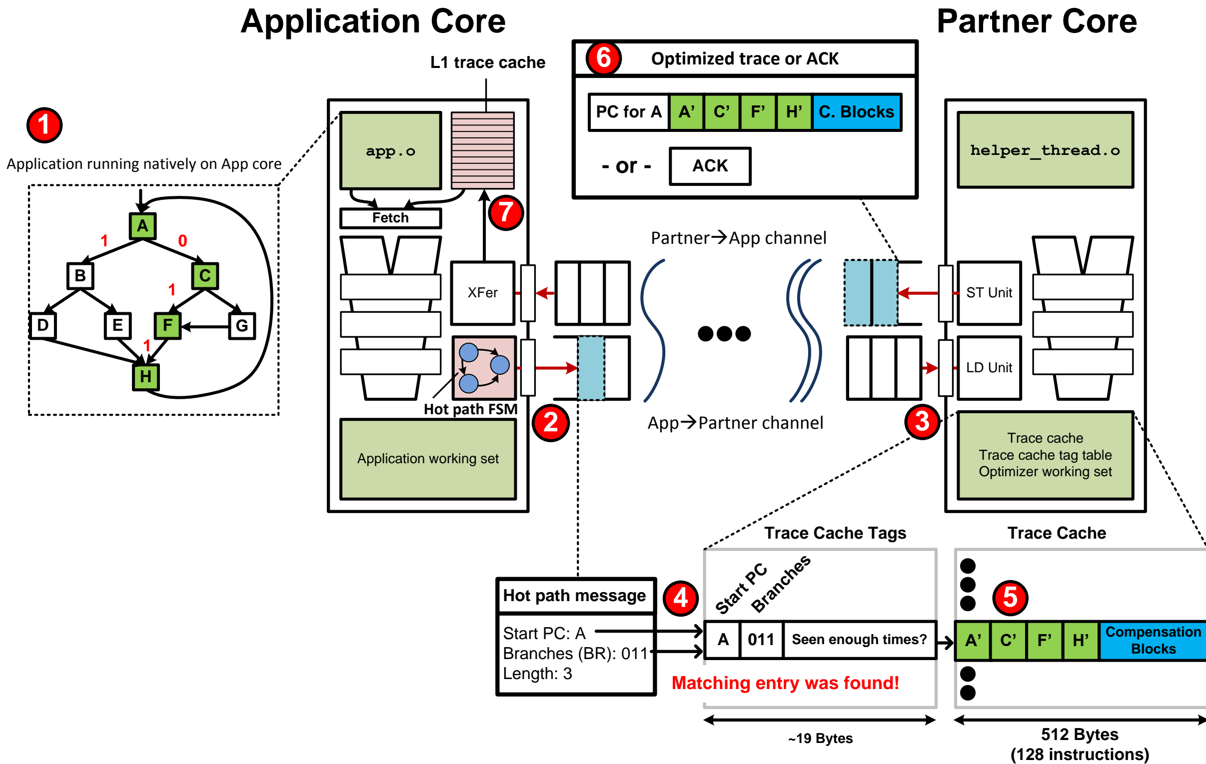

Figure 2: An end-to-end example showing our system optimizing the small loop shown in 1. (1-2) The hot path FSM detects that $\{A, C, F, H\}$ is hot and sends a hot path message to the Partner core. The helper thread reads the hot path message (3), sees that the message has been seen enough times to warrant optimization (4), and (5) creates a trace for the message. Finally, the message and its start PC are sent back to the App core (6) and fetched in place of regular instructions (7). In between steps 2 and 7, the hot path FSM does not send additional messages into the network.

comprise the hot path, optimizes those instructions as a trace, caches the trace in the Partner core data (D)Cache, and sends the trace back to the App core. After processing *every* HPM, the Helper thread sends an acknowledgement (ACK) back to the App core (if a trace is sent back, that trace is also treated as an ACK).

4. *L1 trace (T)Cache* (App core), Section 2.4: A buffer from which traces, sent by the Partner core, can be fetched and executed in place of regular instructions.

To guide the reader we show an end-to-end example, that we will refer back to throughout the rest of the section, in Figure 2.

2.1 Network Communication Between Cores

The App and Partner cores are connected through a pair of channels implemented on top of an unmodified 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[3] written to a channel will eventually be received at the other end. We assume that flow-control units (flits, the atomic unit of physical transport) are 32 bits wide.

2.1.1 App→Partner Channel

At any time, the App→Partner channel transports at most one HPM created by the HP-FSM. With our HPM structure (PC, BR, and the length field), each message is broken up into two flits when being transported over the network.

If the HP-FSM tries to inject an HPM into the channel when the channel is full, the HP-FSM drops the HPM. Once the HP-FSM successfully writes an HPM, it will not write additional HPMs until the App core has received an ACK from the Partner core.

Waiting for an ACK from the Partner core serves two purposes. First, it minimizes load on the App→Partner channel. Between any two HPMs, the Partner core runs a software routine that (potentially) optimizes a trace and sends an ACK back before the next HPM is written. This process takes approximately 2500 cycles on average. Second, when an HPM arrives at the Partner core network ingress, the Helper thread is guaranteed to be able to consume the HPM immediately, which avoids backlogging the network while the Helper is processing older HPMs.

An open question that we address is whether sending so few messages through the network is sufficient to capture application behavior. Based on the SPEC06 benchmarks we use to study our proposal, the average basic block has between 5-10 instructions. Thus, the HP-FSM can potentially capture more information about the application and generate new network traffic once every $5 * |BR| = 80$ to

[3] If the channel is full, the writer cannot write more data to it.

$10 * |BR| = 160$ instructions (we assume that traces are limited to 15 branches in our evaluation). In our evaluation, we show how the ACK-based implementation performs competitively with schemes that process HPMs at a faster rate.

2.1.2 Partner→App Channel

At any time, the Partner→App channel may transport either a trace or an ACK 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. Furthermore if the Partner→App channel fills, the Helper thread stalls until space is available.

We force the Helper thread to stall for two reasons. First, if any portion of a trace is lost in transit, the trace is not functionally correct. Second, stalling the Partner core while a trace is sent over the channel 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.

2.2 Hot Path FSM (HP-FSM)

The HP-FSM (summarized in Figure 3) generates and sends HPMs to the Partner core. The HP-FSM 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*). This is Dynamo's NET/MRET heuristic and is a hint that the program has started a loop [3]. For example, a new backwards branch message can start in Figure 2 whenever the HP-FSM (❷) detects that basic block H transitions to A.

2. is not executing in a trace and executes a jump-and-link (function call) instruction. Function calls can potentially be inlined by Partner core optimizations and can make good optimization targets, but do not necessarily correspond to backwards branches.

3. is executing in a trace and exits from that trace (after an early or regular exit). When a trace exits, its exit point is likely a hot path as well [1]. By creating new messages when we exit from traces early, our system can adapt to branch bias changes by creating adjacent traces.

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. In Figure 2 ❶-❷, if path $\{A, C, F, H\}$ is repeatedly taken, BR will be {011} (read left to right) and {0111011} if the branch limit is set to 3 and 7, respectively. Messages sent before the branch limit has been reached (called *short messages*) represent varying numbers of basic blocks. Short messages improve coverage and prevent code from being unreachable because of mis–aligned traces. They can also create L1 TCache fragmentation, which is discussed in Section 2.4.

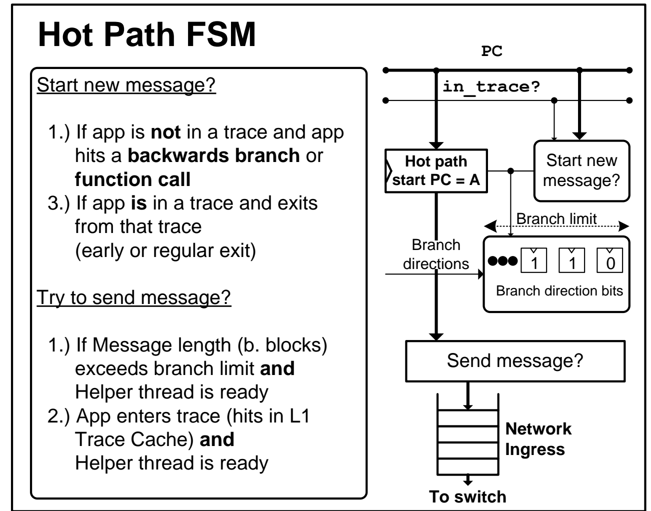

Figure 3: The hot path FSM (HP-FSM); shown as ❷ in Figure 2.

If the Helper thread is ready (i.e., the HP-FSM received an ACK), the HP-FSM sends the HPM. Otherwise, the message is discarded and the HP-FSM begins forming a new message. One can think of the HP-FSM as profiling the application *on the side* as the application runs: under no circumstance will the HP-FSM stall the application.

Putting the ideas from this section together, Figure 4 shows how the HP-FSM helps build traces for a nested loop.

2.3 Helper Thread: Trace expansion and optimization

The Helper thread tracks HPMs, decides when a hot path is worth expanding into a trace, and sends traces back to the App core. When the Helper thread is spawned, it sets the *occurrence threshold* (explained below) and creates two 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 capacity of both the tag table and TCache are set statically.

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[4] its read port (Figure 2, ❸) until an HPM arrives, at which point the Helper thread

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

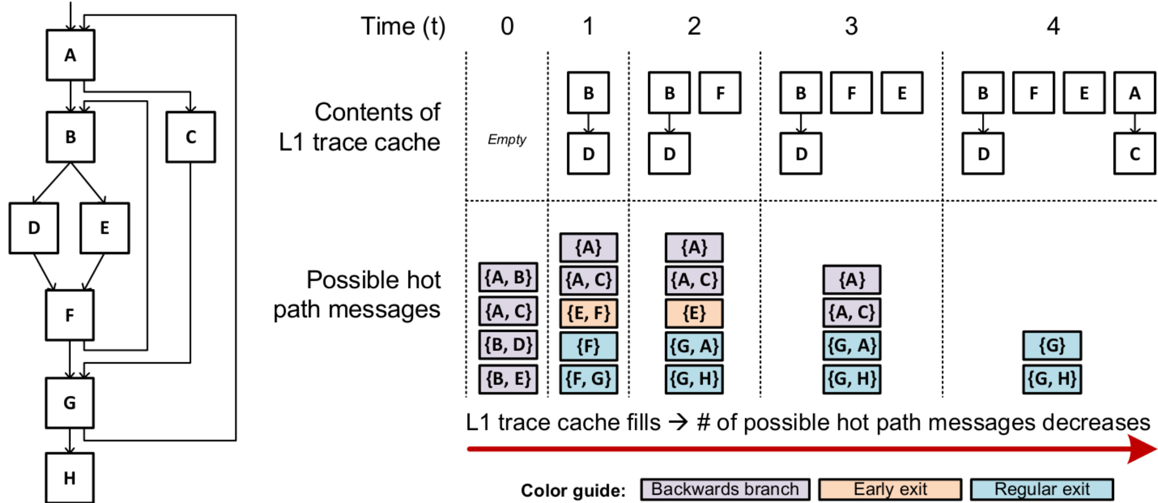

Figure 4: Hot path message formation for the nested loop shown to the left (the branch limit is set to 1 for simplicity). Initially ($t = 0$), the L1 TCache is empty and only backwards branch messages can be formed. When the application is in the inner loop ($t = 1$–3), early and regular exits from the trace corresponding to $\{B, D\}$ cause the HP-FSM to form new messages to further cover the inner loop. Once the inner loop terminates and the Helper thread has detected that the path from A to B is hot, the L1 TCache will start filling with traces (such as the trace corresponding to $\{A, B\}$) that are not a part of the inner loop ($t = 4$). Given enough training time, the L1 TCache will cover all basic blocks in the nested loop and the HP-FSM will cease to create new messages. (*Note on notation*: the right arrow for each branch corresponds to the taken direction and hot path message {A, C} means "the hot path message with start PC = A and BR = {1}.")

performs the following actions:

1. **Lookup {PC, BR} in the TCache tag table.** If the entry is not present, evict 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.[5] 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 (6) and return to message polling behavior.

[5]The PC target in indirect function returns can be derived from a matching call-and-link instruction.

2.4 Mechanism for Trace Execution on the App Core

Upon arriving at the App core, ACK messages are consumed and traces are moved to the *L1 TCache*. Conceptually, the L1 TCache is the first level cache for the Helper thread's software TCache, and has a fixed capacity.

For this work, we implement the L1 TCache using one of the ways in the L1 instruction (I)Cache (Figure 5). To minimize dedicated hardware cost, trace lookups within the dedicated ICache way are made in a direct-mapped fashion that uses the same logic already in place to support each way in the set-associative ICache as well as a small amount of custom hardware. 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

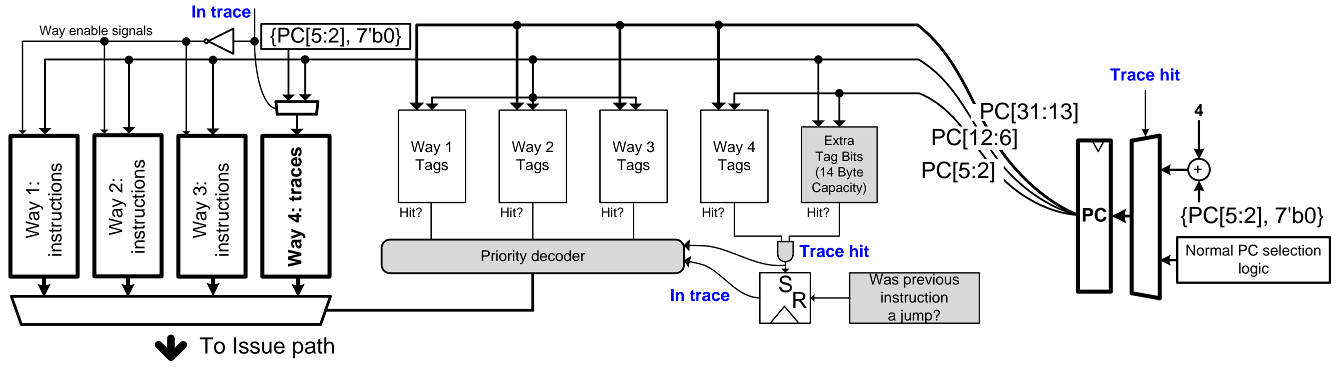

Figure 5: A dedicated-way, single-cycle L1 TCache design (grey structures are added to support our system). To determine a trace hit, the tag PC[31 : 13] is checked in the unmodified tag array for the trace way while PC[12 : 6] is compared in the "Extra tag Bits" table, whose capacity is $\frac{16\times7}{8} = 14$ Bytes if that the ICache way stores 16 traces at a time.

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 small direct-mapped structure with a 14 Byte capacity that is indexed in parallel with the tag arrays during the first fetch cycle (see Figure 5).

Aside from the dedicated table, the entire design adds several gate delays to the fetch cycle 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

To evaluate our system, we first investigate how the Partner core can be decoupled from the App core (Section 3.2). We then discuss the system's load on the network (Section 3.3), the Helper thread's impact (Section 3.4), dedicated hardware overhead (Section 3.5), and power usage (Section 3.6). To close, we perform a case study that evaluates application speedup when the Helper thread implements a traditional trace cache (Section 3.7).

3.1 Methodology

We simulate a subset of the SPEC06-int benchmark suite [14] using the SESC simulator [12]. SESC is a cycle-level simulator used in both uni- and multicore settings. SPEC is known primarily as a uniprocessor/server target, and conventional wisdom suggests that it is difficult to extract efficiency from a multicore for these benchmarks.

Unless otherwise stated, we compile each benchmark and helper thread to MIPS assembly with $-O3$ optimizations and simulate 3 billion instructions with a 1 to 20 billion instruction warm-up period, depending on the benchmark, using SPEC reference inputs. The only exceptions are *gcc* and *perl* which were compiled under $-O0$ and $-O2$ respectively because of simulator incompatibilities.

The baseline system configuration used in our results is shown in Table 1a and 1b. (We refer to the baseline configuration as SW for the remainder of the paper.) Both the App and Partner cores use the same core model. We chose the *branch limit* and *trace size* to be competitive with other systems [10]. Furthermore, the Helper thread's TCache capacity is set so that the entire TCache fits within the Partner core's L1 data cache.[6] Notice that after fixing the ICache and *trace size* accordingly, *L1 TCache capacity* becomes fixed to 16 traces.

3.2 Partner Core Decoupling

Figure 6 shows how the design points in Table 2, which reduce dedicated hardware/power overheads and increase scheduler flexibility, impact coverage.

We show HW and HW-U to illustrate the impact of SW's slower software Helper thread and realistic network model. In Table 2, the "magic network" has zero latency and infinite throughput. The "magic Helper thread" uses the same C code as the MIPS-based Helper thread, but performs the steps from Section 2.3 in a single cycle. Thus, HW and HW-U never drop HPMs due to network contention or waiting for ACKs. HW isolates the impact of the network model and Helper thread implementation, while keeping the App core L1 TCache size equal to SW. HW-U gives an upper bound on coverage, given the HP-FSM scheme from Section 2.2 and infinite/magic resources otherwise. To avoid L1 TCache conflict misses which partly negate the impact of a faster Partner core, both HW and HW-U assume a fully-associative L1 TCache (as opposed to the simpler direct-mapped design from Section 2.4).

Notice that SW attains only marginally lower coverage than HW ($\sim 7.8\%$ overall). Several benchmarks (e.g., gobmk,

[6] Correspondingly, we found that the Partner core had a $\sim 0 - .01\%$ miss rate in its L1 DCache for all of our benchmarks.

Table 2: System configurations used to test the extent to which the App and Partner cores can be decoupled.

Design point	How is the design point different than Table 1b?
SW-L	Network latency = 256
SW-F	Partner core frequency reduced by 10×
HW	Magic network and Helper thread, Occurrence count = 1, Fully-associative L1 TCache
HW-U	Magic network and Helper thread, Occurrence count = 1, Branch limit = 21, Trace size = 1024, L1 TCache capacity = 512, Fully-associative L1 TCache

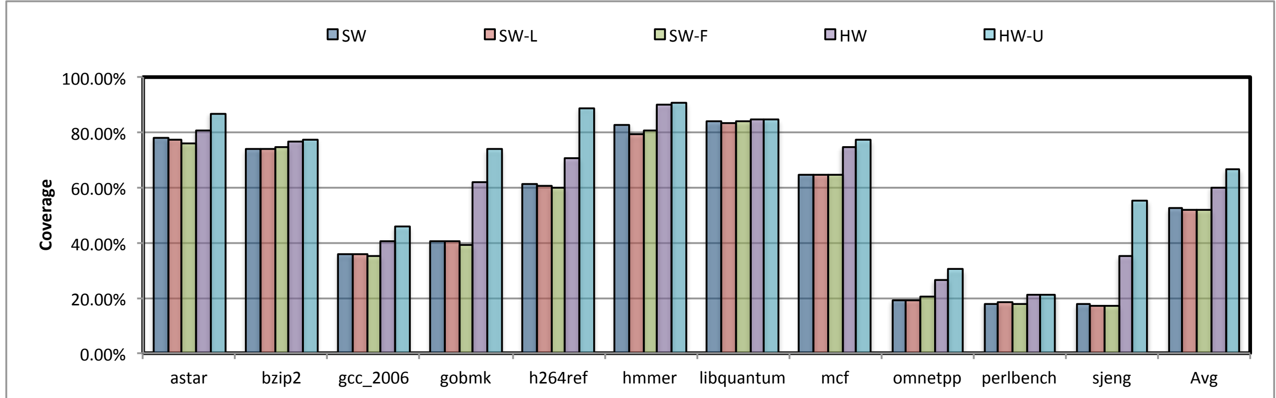

Figure 6: Coverage for the representative system design points discussed in Section 3.2.

Table 1: System configuration.

(a) Fixed architectural parameters which are independent of the dynamic optimization process.

Core model: 3 issue, 5 stage, in-order	
Memory Hierarchy (Private L1 I/D caches, unified/shared L2)	
L1 I/D, L2 capacity	32/32, 1024 KB
L1 I/D, L2 associativities	4/4, 8-way
Block size	64 B
L1 I, L1 D, L2 hit/miss delay	1/1, 2/3, 10/14 cycles
Delay to memory	100 cycles
Network	
Latency	16 cycles
Capacity	8 flits

(b) The dynamic optimization-specific parameters for the SW point.

Helper thread Implementation: C code compiled to MIPS	
Dynamic Optimization Parameters	
Branch limit	15 (16 basic blocks)
Trace size	512 B (128 instructions)
TCache capacity	64 traces
Occurrence threshold	8
L1 TCache arch.	1-way in the L1 ICache direct-mapped lookup
L1 TCache capacity	16 traces

sjeng, gcc) attain noticeable benefit from the ideal Helper thread. We observed that these benchmarks produce a large number of paths over sustained periods, making it more difficult for the Helper thread to rate match. For other benchmarks (e.g., bzip2, hmmer, libquantum), the software Helper thread can effectively match the idealized design. These benchmarks generally have more stable phases or smaller code bases and spend more time in loops, which causes the HP-FSM to send fewer messages as loop coverage increases (see Figure 4).

SW-L shows how the Partner core's physical location on the chip relative to the App core makes a minimal .5% difference in coverage. The SW-L result coupled with the framework's near 0 network utilization implicates that our framework would operate well in a contended network.

SW-F shows how the Partner core's clock frequency reduces coverage by only .6% on average compared to SW. Partner core schemes require that two contexts run for a single application, which may lead to an overall decrease in energy efficiency if the performance gain is not large enough. Frequency scaling is one way to decrease the Partner core power overhead. We chose to reduce the frequency by 10× because it makes the Partner core's power consumption negligible (Section 3.6).

3.3 Network Load

Figures 7 and 8 show how the NoC is impacted by sending HPMs and traces between the App and Partner core.

Based on Figure 7, the primary HPM start condition is benchmark-dependent. However overall regular exit HPMs dominate followed closely by early exit message types. In Figure 8, we see that short messages make up a significant percentage (45% overall) of all messages created.

We found that our acknowledgement scheme restricts the number of HPMs in the system and reduces the network utilization to a fraction of a percent across all benchmarks and design points. As a result the framework also enjoys a low injection rate. This has good implications for the scalability of our system in a multi-workload setting in which foreign applications contribute network traffic.

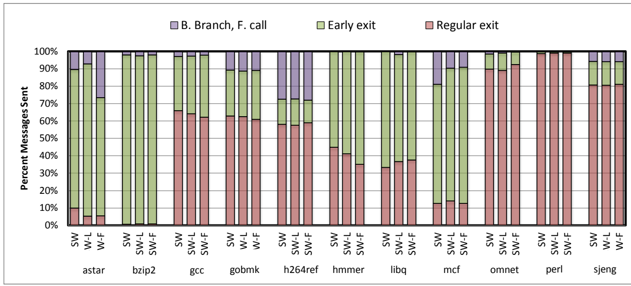

Figure 7: The hot path message breakdown by type.

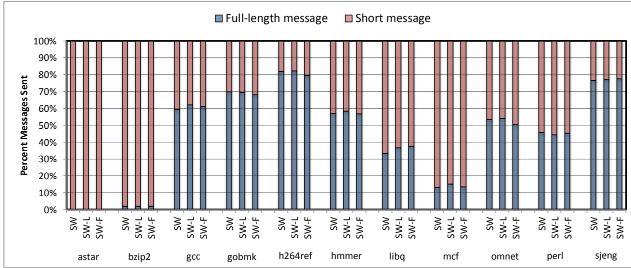

Figure 8: The hot path message breakdown by length. Full-length messages cover 16 basic blocks.

3.4 Helper Thread Latency

In order to demonstrate the effectiveness of our framework given different Helper thread implementations, we performed a study varying the number of cycles the Helper thread takes to return a trace, shown in Figure 9. To perform this study we modified the HW design point from Table 2 so that the Helper thread performs the steps from Section 2.3 in a set number of cycles. In comparison, the SW Helper thread takes on the order of 1000 cycles to return a trace.

We observe that in most of the benchmarks even a 10,000 cycle Helper thread has comparable coverage to an ideal, single-cycle Helper thread. On average coverage only degrades by 2.7%. This shows how the Helper thread's view of hot paths is *noise* tolerant—the Helper thread takes a long time to process HPMs yet keeps pace with the ideal single-cycle HW. However at 100,000 cycles, there is a sudden drop in coverage across most benchmarks. This is because for benchmarks such as *astar*, *bzip2* and *gcc* the usefulness of a trace declines over time as the benchmarks proceeds in its execution. That is, the Helper thread does not rate match well with the application with such a high latency. An exception to this is *libquantum*, which executes the same set of loops millions of times so even with high latency the Helper thread can return useful traces in time for them to still be executed millions of times.

3.5 Dedicated Hardware Overhead

Our system's two main structures are the HP-FSM and the L1 TCache. The HP-FSM requires ~ 9 Bytes for PC, branch directions and FSM state. In the dedicated-way L1 TCache design (Section 2.4), 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 5). While there are small additional overheads–such as the flag

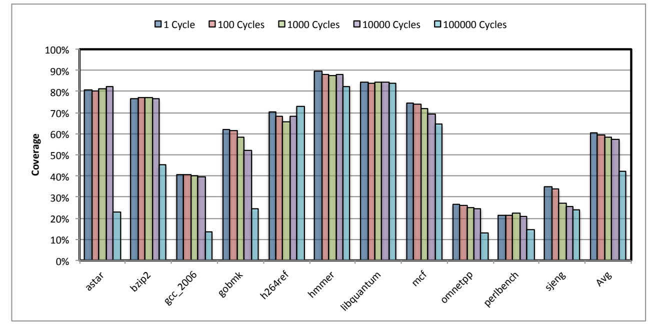

Figure 9: Coverage for a Helper thread that requires varying cycles to return traces.

that indicates that we are in a trace–in total the system requires less than 50 Bytes of custom storage.

3.6 Power Usage

We modeled our system's power with CACTI and Orion [15, 8] projected on 45nm technology using the parameters provided in Table 1a. The design components and their corresponding dynamic energy are listed in 3.

Table 3: Dynamic Energy

Design Component	Dynamic Energy
FPU/ALU	1.48 pJ/operation
L1 TCache	0.142 nJ/access
L1 ICache	0.236 nJ/access
L1 DCache	0.404 nJ/access
L2 Cache	2.06 nJ/access
Router+Link	0.11 pJ/flit

We derived the FPU energy per operation from [5] and use it as an upper bound for all arithmetic operations. The total dynamic power used by our 2-core system as averaged across all our benchmarks is 151 mW. We break down the power in Figure 4. By scaling down the frequency of the Partner core to $\frac{1}{10}$ of the application core, the total dynamic power is 80 mW. In comparison, the application-only baseline without our framework uses 86 mW. This 7% improvement in power dissipation is largely due to a more energy-efficient fetch stage. The drop in power in the fetch stage by the App core from the baseline is 24% for SW and 21% for SW-F. This is because when executing in a trace, only the dedicated ICache way is activated and all the other ways are turned off, leading to tremendous power savings. Due to the low amount of network traffic generated by our framework, the power consumed by the network in negligible.

Table 4: System Power in Milliwatts.

	Total	App Core	App Fetch	Partner Core	L2	Network
SW	151	67	25	75	9	~ 0
SW-F	80	67	26	4	9	~ 0
BASELINE	86	77	33	0	9	0

3.7 Performance

To evaluate the performance our system, we compare the performance of our framework to generate traces [4] against the same system without our framework. The Helper thread generates traces from hot paths without applying any additional optimizations. The purpose of this study is to show that our system does not significantly degrade performance, allowing any additional optimizations to only improve performance.

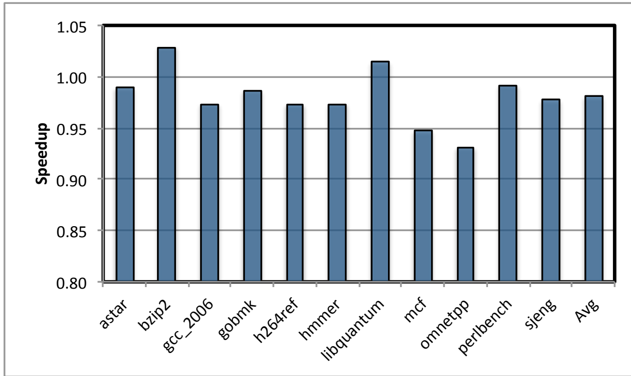

Figure 10: Speedup across SPEC benchmarks.

The effects of laying out frequently executed code fragments contiguously in the L1 TCache are shown in Figure 10. To calculate performance, we compare the number of cycles our framework requires to complete 500 million instructions to the number of cycles required to reach the same logically equivalent point in a baseline system that only runs the application without sending messages to a Partner Core.[7] Performance degradation in benchmarks including mcf and omnetpp is due to trace format: at each trace exit point, an extra jump + delay slot instruction (usually a no-op) are executed. The addition of these two instructions can cause slowdown, which is especially pronounced in benchmarks such as astar, which frequently early exits (Figure 7). When these extra instructions were eliminated from our simulator's timing model, our system performed at least as well as the baseline on all benchmarks. On average we see a performance degradation of 2.5%, which could be easily recouped by applying optimizations on top of the trace cache. We leave this demonstration to future work.

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 and location on the chip. 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.

[7] Because the Helper thread may add or remove instructions from a trace, the number of instructions executed by our framework and the baseline are not always equivalent.

5. 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] S. Galal, M. Horowitz. *Energy-Efficient Floating-Point Unit Design* IEEE Transactions on Computers, 2011.

[6] B. Hertzberg, K. Olukotun. *Runtime Automatic Speculative Parallelization.* Proceedings of the International Symposium on Code Generation and Optimization (CGO), 2011.

[7] 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.

[8] 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.

[9] 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.

[10] S. J. Patel, S. S. Lumetta. *Replay: A Hardware Framework for Dynamic Optimization.* IEEE transactions on computers, Vol. 50, No. 6, June 2001.

[11] 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.

[12] J. Renau. *SESC simulator.* http://sesc.sourceforge.net/index.html, 2002.

[13] 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.

[14] Standard Performance Evaluation Corporation. *SPEC CPU benchmark suite.* http://www.spec.org/osg/cpu2006.

[15] S. Thoziyoor, N. Muralimanohar, N. P. Jouppi *CACTI 5.0* http://www.hpl.hp.com/research/cacti/

[16] 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.

[17] 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.

[18] 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.