Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads

Milad Hashemi*, Onur Mutlu§, Yale N. Patt*

*The University of Texas at Austin §ETH Zürich

ABSTRACT

Runahead execution pre-executes the application's own code to generate new cache misses. This pre-execution results in prefetch requests that are overwhelmingly accurate (95% in a realistic system configuration for the memory intensive SPEC CPU2006 benchmarks), much more so than a global history buffer (GHB) or stream prefetcher (by 13%/19%). However, we also find that current runahead techniques are very limited in coverage: they prefetch only a small fraction (13%) of all runahead-reachable cache misses. This is because runahead intervals are short and limited by the duration of each full-window stall. In this work, we explore removing the constraints that lead to these short intervals. We dynamically filter the instruction stream to identify the chains of operations that cause the pipeline to stall. These operations are renamed to execute speculatively in a loop and are then migrated to a Continuous Runahead Engine (CRE), a shared multi-core accelerator located at the memory controller. The CRE runs ahead with the chain continuously, increasing prefetch coverage to 70% of runahead-reachable cache misses. The result is a 43.3% weighted speedup gain on a set of memory intensive quad-core workloads and a significant reduction in system energy consumption. This is a 21.9% performance gain over the Runahead Buffer, a state-of-the-art runahead proposal and a 13.2%/13.5% gain over GHB/stream prefetching. When the CRE is combined with GHB prefetching, we observe a 23.5% gain over a baseline with GHB prefetching alone.

1. Introduction

Runahead execution for out-of-order processors [29,30,33,34] is a proposal that reduces the impact of main memory access latency on single-thread performance. In runahead, once the processor stalls, it uses the instruction window to continue to fetch and execute operations. The goal of runahead is to generate new cache misses, thereby turning subsequent demand requests into cache hits instead of cache misses. Runahead pre-executes the application's own code and therefore generates extremely accurate memory requests. Figure 1 shows the prefetch accuracy of runahead when compared to a stream [45] and a global-history buffer (GHB) prefetcher [36] on the memory intensive *SPEC CPU2006* benchmarks (sorted from lowest to highest memory intensity).[1] The stream and GHB prefetchers both use prefetcher throttling [45] to dynamically reduce the number of inaccurate requests. Prefetch accuracy is measured as the percentage of all prefetched cache-lines that have been accessed by the core prior to eviction from the last level cache (LLC).

Overall, runahead requests have 95% average accuracy. This is significantly higher than the dynamically throttled GHB and stream prefetchers. Yet, despite high accuracy, runahead prefetches only a small portion of all *runahead-reachable* memory requests and consequently results in only a fraction of the *Oracle* performance gain, as shown in Figure 2. A runahead-reachable memory request is defined as a request that is *not* dependent on off-chip source data at issue. The Oracle turns all of these runahead-reachable misses into LLC hits. 85% of all LLC misses on average are runahead-reachable.

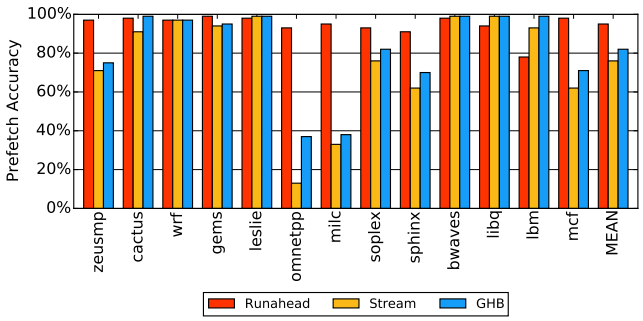

Figure 1: Average prefetch accuracy.

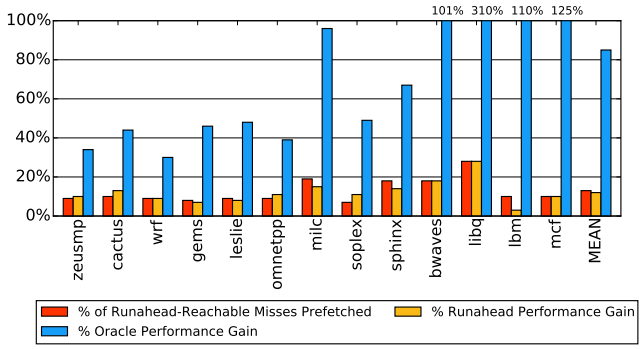

Figure 2: Runahead coverage/performance vs. Oracle.

Runahead results in a 12% performance gain (vs. an 85% Oracle gain) while prefetching 13% of all runahead-reachable misses (vs. 100% for the Oracle). This *runahead performance gap* between runahead and the Oracle is because the total number of cycles that the core spends in each runahead interval is small, as shown in Figure 3. Runahead intervals are less than 60 cycles long on average in a machine with a 256-entry reorder buffer (ROB). A short runahead interval significantly

[1]Simulated on an aggressive 4-wide out-of-order processor with a 256-entry reorder buffer (ROB) and 1MB LLC.

978-1-5090-3508-3/16/$31.00

limits both the number of pre-executed instructions and the number of new cache misses generated per runahead interval [28,33]. However, even though each full-window stall is short, the core is still memory-bound. A 256-entry ROB machine with runahead still spends over 50% of total execution time waiting for data from main memory.

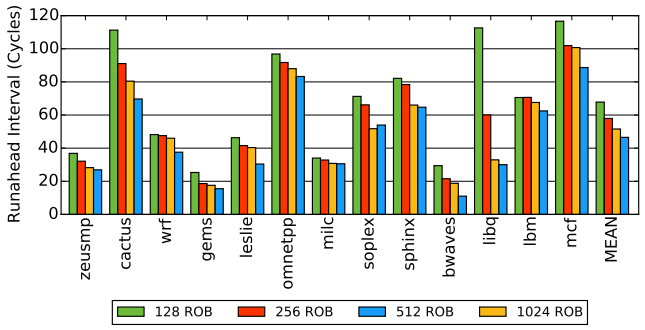

Figure 3: Average number of cycles per runahead interval.

Runahead is a reactive technique that imposes constraints on how often and how long the core is allowed to pre-execute operations. First, the core is required to completely fill its reorder buffer before runahead begins. This limits how often the core can enter runahead mode, particularly as ROB size increases (Figure 3). Second, the runahead interval terminates when the runahead-causing cache miss is serviced. This limits the duration of each runahead interval. Since runahead uses the pipeline only when the main thread is stalled, these constraints are necessary to maintain maximum performance when the main thread is active. However, despite high accuracy, these constraints force prior runahead policies to run for very short intervals. Prior runahead proposals improve runahead efficiency in single-core systems [16, 31, 33], but do not solve the major problem: short runahead intervals dramatically limit runahead performance gain.

In this paper, we explore removing the constraints that lead to these short intervals. The goal is a proactive policy that uses runahead to accurately prefetch data so that the core stalls less often. To this end, we propose the notion of *Continuous Runahead*, where the instructions that cause LLC misses are executed speculatively for *extended intervals* to prefetch data. The implementation of Continuous Runahead involves exploring three major challenges: Continuous Runahead instruction supply, Continuous Runahead hardware, and Continuous Runahead control.

Continuous Runahead Instruction Supply. Which instructions should be executed during runahead? We show that the most critical cache misses to prefetch with runahead are those that cause the pipeline to stall most frequently. We dynamically filter the instruction stream down to the chains of operations that generate the addresses of these critical misses. These dependence chains are then renamed to execute *continuously* in a loop.

Continuous Runahead Hardware. What hardware should Continuous Runahead dependence chains execute on? It is possible to use full simultaneous multi-threading (SMT) thread contexts or idle cores as the substrate for Continuous Runahead. However, we argue that this is inefficient and that full cores are over-provisioned for Continuous Runahead dependence chains. Instead, we use specialized hardware for the implementation of Continuous Runahead [15]. The Continuous Runahead Engine (CRE) speculatively executes renamed dependence chains as if they were in a loop, accurately generating new cache misses. The CRE is a lightweight runahead accelerator (with a 2% quad-core area overhead) located at the memory controller so a single CRE can be efficiently multiplexed among the cores of a multi-core chip.

Continuous Runahead Control. How should Continuous Runahead be controlled to maintain high accuracy/coverage? We demonstrate that the accuracy of CRE requests can be controlled by a "chain update interval" parameter that regulates when the CRE can receive and switch to a new dependence chain to execute.

This paper makes the following contributions:

- We show that while runahead execution is extremely accurate, the number of generated memory requests is limited by the duration of each runahead interval. We demonstrate that runahead interval length significantly limits the performance of state-of-the-art runahead proposals.
- To solve this problem, we introduce the notion of Continuous Runahead. We demonstrate that by running ahead continuously, we increase prefetch coverage from 13% to 70% of all runahead-reachable cache misses.
- In order to implement Continuous Runahead, we develop policies to dynamically identify the most critical dependence chains to pre-execute. We show that dependence chain selection has a significant impact on runahead performance. These dependence chains are then renamed to execute in a loop and migrated to a specialized compute engine where they can run ahead continuously. This Continuous Runahead Engine (CRE) leads to a 21.9% single-core performance gain over prior state-of-the-art techniques on the memory intensive *SPEC CPU2006* benchmarks.
- In a quad-core system, we comprehensively evaluate the CRE with and without traditional prefetching. The CRE leads to a 43.3% weighted speedup gain over a no-prefetching baseline and a 30.1% reduction in system energy consumption. This is a 13.2% gain over the highest performing prefetcher (GHB) in our evaluation. When the CRE is combined with a GHB prefetcher, the result is a 23.5% gain over a baseline with GHB prefetching alone.

2. Related Work

Runahead. This paper builds on the concepts that were proposed in runahead execution. In runahead [9, 29, 30, 34], once the back-end of a processor is stalled due to a full reorder buffer, the state of the processor is checkpointed and the front-end continues to fetch instructions. These instructions are executed if source data is available. Some proposals do not maintain pre-executed results [29, 32], while other related

proposals like Continual Flow Pipelines do [2, 32, 46]. Prior proposals have also combined runahead with SMT [39, 40, 50]. The main goal of runahead is to generate additional LLC misses to prefetch future demand requests. Hashemi et al. [16] identify that the only operations that are required to generate these cache misses are in the address generation chain of the load. Executing only these *filtered* dependence chains during runahead results in more cache misses per runahead interval. All of these prior proposals are only able to "run ahead" for the short intervals that the pipeline is stalled (Figure 3). This is the first work that we are aware of that evaluates runahead execution for extended intervals.

Pre-Execution. This paper is also related to prior proposals that dynamically identify code segments or "helper threads" to pre-execute to generate new cache misses. For a helper-thread to be effective, it needs to execute ahead of the main-thread. In prior work [1, 6, 47, 52], this is done by using a dynamically filtered version of the main-thread, where unimportant instructions have been removed as the helper thread (such that the helper thread can run much faster than the main thread). These helper-threads are then executed using either: a full core, SMT contexts, or a separate back-end. All of these prior proposals consider only single-core systems.

Prior work has proposed using two full processors to execute an application [12, 47, 52]. Slipstream uses a filtered A-stream to run ahead of the R-stream, but does not filter code down to address generation chains. Dual-core execution does not *explicitly* filter the instruction stream, but instead uses runahead in one core to prefetch for the other. The power and performance overhead of using two cores to execute one application is significant. We show that one optimized CRE located near-memory can be effectively shared by all cores.

Annavaram et al. [1] add hardware to extract a dependent chain of operations that are likely to result in a cache miss from the front-end during decode. These operations are prioritized and execute on a separate back-end. This reduces the effects of pipeline contention on these operations, but limits runahead distance to operations that the processor has already fetched.

Collins et al. [6] propose a dynamic mechanism to extract helper-threads based on address-generation chains from the back-end of a core. To do so, retired operations are filtered through new hardware structures. The generated helper threads are then stored in a large cache and run on one of 8 full SMT thread contexts, potentially contending with the main thread for resources.

Prior work also proposes compiler/programmer driven helper-thread generation [4, 7, 21, 25, 53] or dynamic compilation techniques [24, 51]. Statically-generated helper threads can run on idle-cores of a multi-core processor [3, 20].

In contrast to prior work, we develop algorithms to identify the most critical load to accelerate with pre-execution and dynamically generate a filtered dependence chain for only that load. The dependence chain is then speculatively executed with minimal control overhead on specialized hardware that is shared among all cores. This allows the dependence chain to continuously run ahead of the core without contending for pipeline resources. We demonstrate that the additional hardware required to execute a dependence chain is small when added to prior compute-near-memory proposals [15].

Prefetching. Pre-execution is a form of prefetching. Many prior works have used pattern-matching prefetchers to predict future misses based on past memory access patterns. Stream/stride prefetchers are simple structures that can have large memory bandwidth utilization but prefetch simple access patterns accurately [13, 19, 37]. Correlation prefetchers maintain large tables that link past miss addresses to future miss addresses to capture complicated access patterns [5,18,22,41,44]. The global-history buffer [36] uses a two-level mechanism that reduces correlation table overhead. Content-directed prefetching [8, 10] greedily prefetches by dereferencing values that could be memory addresses. We evaluate the CRE in comparison to on-chip prefetchers, as well as in conjunction with them.

3. Continuous Runahead

The only operations that need to be executed during runahead are in the address dependence chain of a load that is likely to result in a cache miss. Such dependence chains can be dynamically identified and have been shown to have three properties [16]. First, if a dependence chain generates a cache miss, the same sequence of operations is likely to generate additional cache misses in the near future. Second, dependence chains are generally short, under 32 micro-operations (uops) on average. Third, dependence chains generally contain only simple integer operations for pointer arithmetic. One example of a dependence chain is shown in Figure 4. Four uops produce the value in R1, which is then used to access memory. The load that results in the cache miss is dashed.

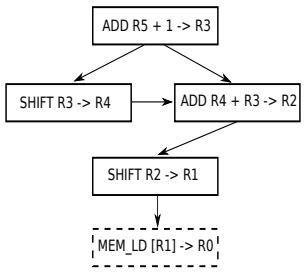

Figure 4: Example dependence chain adapted from *mcf*.

Using a dependence chain, we argue that it is possible to execute it continuously in a loop to prefetch new data. However, for high accuracy, it is important to make the correct choice of *which* dependence chain to use. We examine this problem in Section 3.1.

After identifying a dependence chain, it can be executed continuously on an SMT thread context or an idle core. However, Section 3.2 argues that these prior solutions are over-provisioned. Instead, a single, tailored Continuous Runahead

Engine (CRE) can be shared by multiple cores. Section 4.2 then shows that the CRE can be controlled in an interval-based fashion to maintain high prefetch accuracy. As dependence chain selection and accuracy are single-thread decisions, we first focus on a single-core system and then explore multi-core policies in Section 6.2.

3.1. Dependence Chain Selection

To generate a dependence chain, the original Runahead Buffer policy [16] speculates that at a full window stall, a different dynamic instance of the load that is blocking retirement is currently available in the reorder buffer. This second load is then used to complete a backwards data-flow walk to determine a dependence chain to use during runahead. While this is a straightforward and greedy mechanism, it is not clear that it always produces the best dependence chain to use during runahead. In this section, we explore relaxing the constraints of the original policy with three new policies for dependence chain generation that use the hardware implementation from the Runahead Buffer proposal.

PC-Based Policy. The original policy restricts itself to using a dependence chain that is available in the reorder buffer. For the PC-Based policy, this restriction is relaxed. The hardware maintains a table of all PCs that cause LLC misses. For each PC, the hardware also maintains a list of all of the unique dependence chains that have led to an LLC miss in the past. During a full-window stall, the Runahead Buffer is loaded with the dependence chain that has generated the most LLC misses for the PC of the operation that is blocking retirement.

Maximum-Misses Policy. The original policy constrains itself to using a dependence chain based on the PC of the operation that is blocking retirement. In the Maximum-Misses policy, instead of using this PC, the hardware searches the entire PC-miss table for the PC that has caused the most LLC misses over the history of the application so far. This assigns priority to the loads that miss most often. At a full window stall, the dependence chain that has led to the most cache misses for the PC that has caused the most cache misses is loaded into the Runahead Buffer and runahead begins.

Stall Policy. Due to overlapping memory access latencies in an out-of-order processor, the load with the highest number of total cache misses is not necessarily the most critical load operation. Instead, the most important memory operations to accelerate are those that cause the pipeline to stall most often. For this new policy, the hardware tracks each PC that has caused a full-window stall and every dependence chain that has led to a full-window stall for every PC. Each PC has a counter that is incremented when a load operation blocks retirement. At a full-window stall, the hardware searches the table for the PC that has caused the most full-window stalls. The dependence chain that has resulted in the most stalls for the chosen PC is then loaded into the runahead buffer.

Figure 5 shows performance results for using these new policies during runahead on a single core system (Table 1). Overall, all new policies result in performance gains over the original Runahead Buffer policy. However, the Stall policy generally outperforms the other policies that track LLC misses, increasing performance by 7.5% on average over the original policy. One of the reasons for this is shown in Figure 6, which plots the number of distinct instructions that cause full-window stalls/LLC-misses and the number of operations that cause 90% of all full-window stalls/LLC-misses. On average, only 94 different operations cause full-window stalls per benchmark and 19 operations cause 90% of all full-window stalls. This is much smaller than the total number of operations that cause all/90% of cache misses, particularly for *omnetpp* and *sphinx*. We find that tracking the number of full-window stalls provides an effective filter for identifying the most critical loads, a similar observation to MLP-based policies from prior work [38].

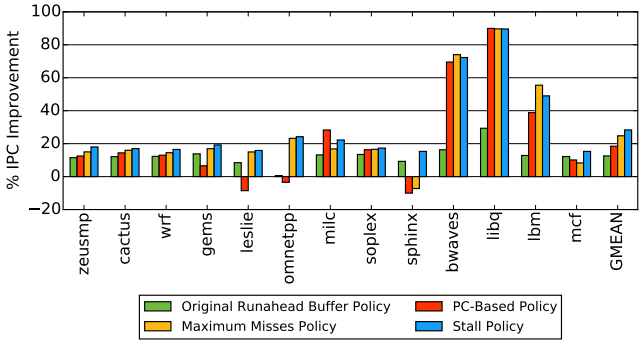

Figure 5: Effect of runahead buffer dependence chain selection policy on performance over a no-prefetching baseline.

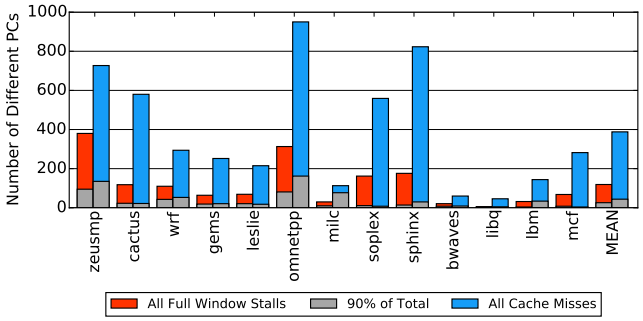

Figure 6: The number of different PCs that account for all LLC misses/full window stalls and 90% of all LLC misses/full window stalls.

To further reduce the storage requirements of the Stall policy, we make two observations. First, Figure 6 shows that 90% of all full window stalls are a result of 19 static instructions on average. Therefore, instead of storing every PC that causes a full-window stall, we maintain new hardware, a 32-entry cache of PCs. Each cache entry contains a counter that is incremented every time that PC causes a full-window stall.

Second, Figure 7 shows that it is unnecessary to store *all* dependence chains that have caused a full-window stall in the past. We vary the number of dependence chains that are stored for each miss PC from 1 to 32. Performance is normalized to a system that stores *all* chains for each miss PC. On average, storing 32 or 64 chains provides only marginally

higher performance than storing only one chain. Both *leslie* and *sphinx* achieve maximum performance with only one stored dependence chain, suggesting that recent path history is more important than storing prior dependence chains for these applications. Using this data, we maintain only the last dependence chain that the pipeline has observed for the PC that has caused the pipeline to stall the most often. Section 4.1 describes the details of identifying this chain.

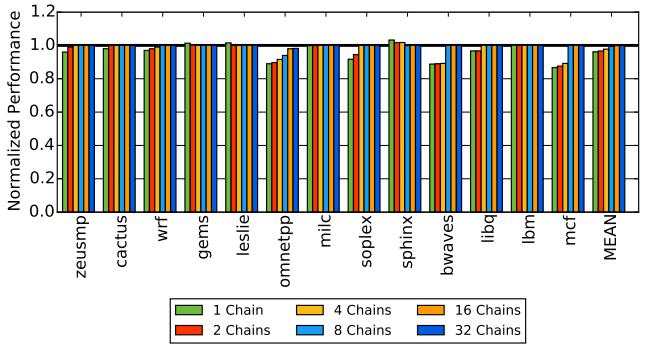

Figure 7: Sensitivity to the number of stored dependence chains per PC normalized to storing all chains.

3.2. The Continuous Runahead Engine (CRE)

While Figure 5 shows that intelligent dependence chain selection policies improve runahead performance over the state-of-the-art, our goal is to run ahead using these dependence chains for extended time intervals. To accomplish this goal, it is possible to use SMT thread contexts or idle cores to execute dependence chains. However, this has two drawbacks.

First, a full hardware thread context could be used by the operating system to schedule non-speculative work. Second, full hardware thread contexts are over-provisioned for executing filtered dependence chains [15]. For example, dependence chains are generally short, which means that they do not require large register files. Dependence chains frequently contain only integer operations, which means that they do not need a processor with a floating point or vector pipeline. Dependence chains can also be dynamically identified in the back-end of an out-of-order core, so a separate front-end that fetches and decodes instructions is unnecessary.

Therefore, instead of using full thread contexts, we argue that it is preferable to use specialized hardware to implement the notion of Continuous Runahead. Prior work exploits the properties of dependence chains (listed above) to design specialized compute capability for the memory controller [15]. The memory controller is an advantageous location for executing dependence chains because it provides low-latency access to main-memory and is centrally located and easily multiplexed among different cores. Prior work uses this compute capability to accelerate operations that are dependent on off-chip data. Such acceleration of dependent memory operations in the memory controller is orthogonal to runahead, as runahead relies on having all source data available to generate new cache misses (i.e., runahead prefetches *independent* cache misses, not *dependent* cache misses [15, 31]).

We propose using such an implementation in the memory controller to continuously execute runahead dependence chains. This requires two small modifications over [15]: 1) Slightly larger buffers to hold full runahead dependence chains (The two 16-uop buffers in [15] become one 32-uop buffer). 2) Correspondingly, a slightly larger physical register file to hold the register state for the runahead dependence chain (The two 16-entry physical register files in [15] become one 32-entry buffer).

The CRE contains a small data cache (4kB) along with a 32-entry data TLB per core. All memory operations executed at the CRE are translated using the CRE-TLB. The CRE-TLB does not handle page faults, instead chain execution is halted. This allows the CRE-TLB to provide control over inaccurate dependence chains. The core is responsible for updating CRE-TLB entries. We add a bit per TLB-entry at the core to track remote CRE-TLB entries (assisting in shootdowns). The CRE-TLB is updated in two cases: 1) Every time a dependence chain is generated, the TLB-entry of the load used to generate the chain is sent to the CRE. 2) CRE-TLB misses are sent to the core to be serviced and if the PTE is present in the core TLB, it is sent to the CRE. Data prefetched by the CRE is directly installed in the LLC.

The datapath of the Continuous Runahead Engine (CRE) is shown in Figure 8. Our CRE implementation has the same area overhead as [15] (2% of total quad-core area, 10.4% of a full core) because our modifications are area-neutral. The main evaluation of this paper does not use the dependent miss acceleration proposed by this prior work [15]. We only use a similar implementation as [15] to perform runahead processing. However, we can combine the CRE with dependent miss acceleration (using the same hardware) to achieve larger benefits. We evaluate this combination in Section 6.5.

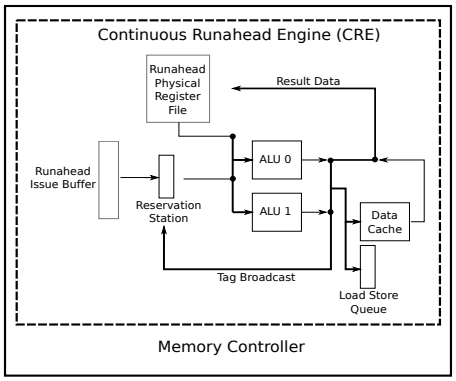

Figure 8: Continuous Runahead Engine datapath.

4. CRE Implementation

Using the Stall policy, Section 4.1 describes how to dynamically identify the dependence chain to use for Continuous Runahead and rename it to a smaller physical register set so that it can be executed in a loop at the CRE. Section 4.2 then develops a mechanism for deciding how long each dependence chain should be executed at the CRE.

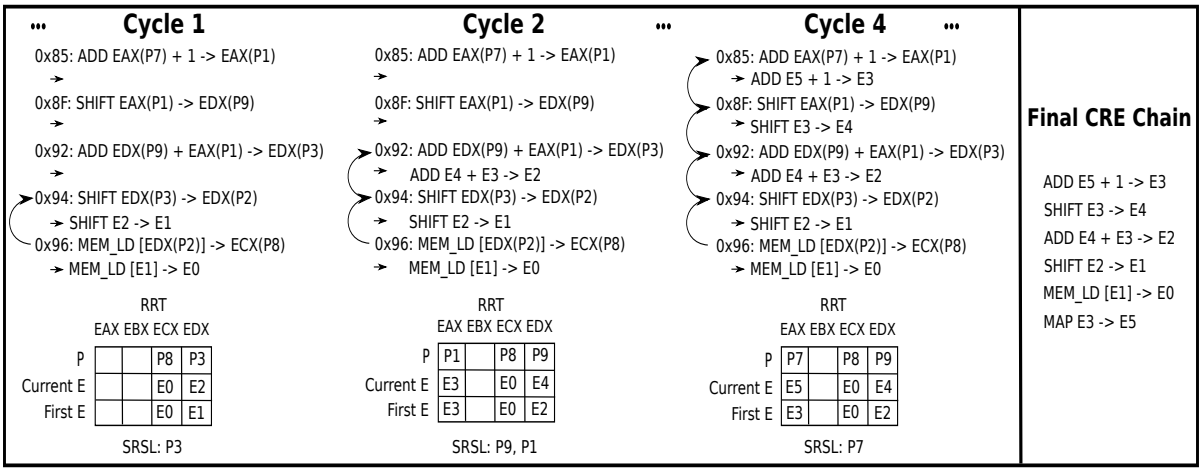

Figure 9: CRE chain generation process. P: Core Physical Register, E: CRE Physical Register.

4.1. Dependence Chain Generation

The Stall policy from Section 3.1 tracks the 32 PCs that have caused the most full-window stalls. We propose that if the processor is in a memory-intensive phase (i.e., the miss rate per thousand instructions (MPKI) is greater than 5 in our implementation), the core marks the PC that has caused the highest number of full-window stalls (using the 32-entry PC-stall cache). The next time a matching PC is issued into the back-end and the core is stalled, we begin a dependence chain generation process using a backwards dataflow walk.

This dataflow walk requires four modifications to the pipeline. First, it requires that all ROB destination register IDs and PCs be searchable. We model this using a CAM. However, there are other ways to associate destination physical registers to ROB entries, including an Intel P6 style-ROB design [17]. Second, we require a Register Remapping Table (RRT). The RRT is functionally similar to a Register Alias Table and is used to rename the dependence chain to a smaller set of physical registers since the CRE has far fewer physical registers than the core. Renaming the dependence chain at the core is advantageous as the chain has to be renamed only once instead of for every iteration at the CRE. Third, we require a 32-uop buffer to hold the last generated dependence chain along with its renamed source registers. Fourth, we maintain a source register search list (SRSL) to conduct the backwards dataflow walk. This additional harware adds an estimated 1kB of additional storage per core. The specifics of this dataflow walk are similar to the algorithm outlined by prior work [16], with the exception that operations are renamed to use the smaller physical register set of the CRE.

Figure 9 provides an example of the chain generation process using the code from Figure 4. In Figure 9, the load at PC 0x96 has been marked for dependence chain generation. Core physical registers are denoted with a 'P' while CRE physical registers use an 'E' and are allocated sequentially. In cycle 0 (not shown in Figure 9), the load at PC 0x96 is identified and the destination register P8 is mapped to E0 in the RRT. Source register P2 is mapped to E1 and enqueued to the SRSL. The RRT has three fields for each architectural register, the current renamed core physical register, the current renamed CRE physical register and the first CRE physical register that was mapped to the architectural register. This last entry is used to re-map the live-outs of each architectural register back to live-ins at the end of dependence chain generation, using a special MAP instruction (described below). The intuition, and the prediction we make, is that the register outputs of one iteration are potentially used as inputs for the next iteration (without any intervening code that changes the outputs). This is necessary to allow the dependence chain to execute as if it were in a loop. We limit chain length to a maximum of 32 operations.

In cycle 1, the core searches all older destination registers for the producer of P2. If an eligible operation is found, it is marked to be included in the dependence chain and its source registers are enqueued into the SRSL. An operation is eligible if it is an integer operation that the CRE can execute (Table 1, Row 6), its PC is not already in the chain, and it is not a call or return. Note that conditional branches do not propagate register dependencies and therefore do not appear in backwards dataflow walks.

The result of the search in cycle 1 is found to be a SHIFT and the source register of the shift (P3) is remapped to E2 and enqueued in the SRSL. This process continues until the SRSL is empty. In cycle 2, P9 and P1 are remapped to E4 and E3 respectively. In cycle 3 (not shown in Figure 9), the SHIFT at address 0x8F is remapped. In cycle 4, the ADD at address 0x85 is remapped and enqueues P7 into the SRSL.

In cycle 5 (not shown in Figure 9), P7 does not find any older producers. This means that architectural register EAX is a live-in into the chain. To be able to speculatively execute this dependence chain as if it were in a loop, a new operation is inserted at the end of the final dependence chain. This "MAP" operation moves the last live-out for EAX (E3) into the live-in for EAX (E5), thereby propagating data from one

dependence chain iteration to the next. Semantically, MAP also serves as a dataflow barrier and denotes the boundary between dependence chain iterations. MAP cannot be issued at the CRE until *all* prior operations are issued. Every live-in register to the chain generates a MAP operation.

In cycle 6, the SRSL is empty. The chain generation process is complete and the dependence chain has been identified. The entire chain can now be sent to the CRE along with a copy of the core physical registers that were renamed to CRE physical registers in the RRT.

4.2. CRE Dependence Chain Execution

When a dependence chain is sent to the Continuous Runahead Engine (CRE), it executes *continuously* as if in a loop. Executing the dependence chain in this fashion is speculating that since these instructions have caused a critical LLC miss in the past, they are likely to do so again in the future. However, we find that a dependence chain can not be executed indefinitely, as the application may move to a different phase where the selected dependence chain is not relevant. In this section, we identify how long to execute each dependence chain at the CRE and when to replace it.

When a dependence chain is constructed and sent to the CRE, a copy of the core physical registers that are required to execute the first iteration of the dependence chain are also sent. This serves to reset runahead at the CRE. For example, if the core sends a new dependence chain to the CRE at every full window stall, the runahead interval length at the CRE for each dependence chain is simply the average time between full-window stalls. At every new full window stall, the core will replace the dependence chain at the CRE (using the algorithm from Section 4.1) and reset the CRE to run ahead with the new dependence chain. Therefore, we define the *CRE update interval* as the time between dependence chain updates from the core.[2]

We find that CRE request accuracy (and correspondingly CRE performance) can be managed by the update interval that the core uses to send dependence chain updates to the CRE. Figure 10 explores how modifying the update interval impacts performance and runahead request accuracy. The x-axis varies the update interval based on the number of instructions retired at the core from one thousand instructions to 2 million instructions. There are two bars, the first bar is the average geometric mean performance gain of the memory intensive *SPEC CPU2006* benchmarks. The second bar is the request accuracy defined as the percentage of total lines fetched by the CRE that are touched by the core before being evicted from the LLC.

Figure 10 shows that both average performance and CRE request accuracy plateau between the 5k update interval and the 100k update interval. Between the 250k update interval and

[2]While the CRE could execute multiple dependence chains from one application, for *SPEC CPU2006*, we find that executing one dependence chain per interval increases runahead distance, maximizing performance [14].

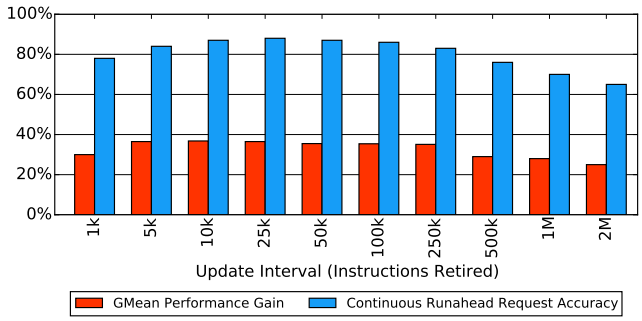

Figure 10: Continuous Runahead performance/accuracy sensitivity to CRE update interval.

the 2M instruction interval, both request accuracy and performance decrease. Average Continuous Runahead request accuracy in the plateau of the chart is at about 85%, roughly 10% less than original runahead request accuracy (Figure 1). As both accuracy and performance gain decrease above the 250k update interval, it is clear that allowing the CRE to run ahead for too long without an update has a negative effect on performance as the application can move to a different phase. Yet, the 1k update interval also reduces performance without a large effect on runahead request accuracy. This occurs because frequently resetting the CRE reduces runahead distance, causing a decrease in CRE effectiveness. To reduce communication overhead, it is advantageous to control the CRE at the coarsest interval possible while maintaining high performance. Therefore, based on Figure 10 we choose a 100k instruction update interval for runahead at the CRE.

5. Methodology

To simulate our proposal, we use an execution-driven, cycle-level x86 simulator. The front-end of the simulator is based on Multi2Sim [49]. The simulator faithfully models core microarchitectural details, the cache hierarchy, wrong-path execution, and includes a detailed non-uniform access latency DDR3 memory system. We evaluate on a single-core system in Section 6.1 and a quad-core system in Section 6.2. Each core has a 256 entry reorder buffer along with a cache hierarchy containing 32KB of instruction/data cache and a 1MB slice of shared last level cache (LLC) per core. The cores are connected with a bi-directional address/data ring. Each core has a ring-stop that is shared with its LLC slice. The memory controller has its own ring stop. Two prefetchers are modeled: a stream prefetcher [45, 48] and a global history buffer (GHB) prefetcher [36]. We find the GHB prefetcher to outperform other address correlation prefetchers with lower bandwidth overhead. We also compare against a system that uses the Runahead Buffer, the state-of-the-art runahead technique [16]. Table 1 describes our system.

The Continuous Runahead Engine (CRE) is located at the memory controller (and shares a ring-stop with the memory controller) in both the single and quad-core configurations. The CRE can execute only a subset of integer operations and has a 2-wide issue width with a small 4kB data cache. These

1: Core	4-Wide Issue, 256-Entry ROB, 92-Entry Reservation Station, Hybrid Branch Predictor, 3.2 GHz
2: L1 Caches	32 KB I-Cache, 32 KB D-Cache, 64 Byte Lines, 2 Ports, 3 Cycle Latency, 8-Way, Write-Through.
3: L2 Cache	Distributed, Shared, 1MB 8-Way Slice per Core, 18-Cycle Latency, Write-Back. Single-Core: 1MB Total. Quad-Core: 4 MB Total.
4: Interconnect	2 Bi-Directional Rings, Control (8 Bytes) and Data (64 Bytes). 1-Cycle Core to LLC Slice Bypass. 1 Cycle Latency between Ring-Stops. Single-Core: 2 Total Ring-Stops. Quad-Core: 5 Total Ring-Stops.
5: CRE	2-Wide Issue. 8-Entry Reservation Stations. 4KB Data Cache 4-Way, 2-Cycle Access, 1-Port. 32-Entry TLB per Core. 1 Continuous Runahead Dependence Chain Context with 32-Entry uop Buffer, 32-Entry Physical Register File. Micro-op Size: 8 Bytes.
6: CRE Instructions	Integer: add/subtract/move/load/store. Logical: and/or/xor/not/shift/sign-extend.
7: Memory Controller	Single-Core: 64-Entry Memory Queue. Quad-Core: 8: Batch Scheduling [35], 256-Entry Queue.
8: Prefetchers	Stream: 32 Streams, Distance 32. GHB G/DC [36]: 1k Entry Buffer, 12KB Total Size. All Configurations: FDP [45], Dynamic Degree: 1-32, Prefetch into Last Level Cache.
9: DRAM	DDR3 [26], 1 Rank of 8 Banks/Channel, 8KB Row-Size. Single-Core: 1-Channel, Quad-Core: 2-Channels. CAS 13.75ns, CAS = t_{RP} = t_{RCD} = CL. Other Modeled DDR3 Constraints: BL, CWL, $t_{RC,RAS,RTP,CCD,RRD,FAW,WTR,WR}$. 800 MHz Bus, Width: 8 B.

Table 1: System Configuration.

values have been determined via sensitivity analysis. CRE requests that miss in the CRE data cache query the LLC. We maintain coherence for the CRE data cache via the inclusive LLC for CRE loads. Instruction supply for the CRE consists of a 32-entry buffer which stores decoded dependence chains. The total quad-core storage overhead of the CRE is 10kB. To allow each core to use the Continuous Runahead Engine (CRE), dependence chain generation hardware is added to each core.

We divide the *SPEC CPU2006* benchmarks into three categories in Table 2: high, medium, and low memory intensity. The evaluation concentrates on the memory intensive workloads since runahead has little performance impact on low memory intensity workloads [16, 29]. From the high-memory intensity workloads, we randomly generate a set of quad-core workloads, shown in Table 3. We also evaluate the multi-core system on a set of workloads consisting of 4 copies of each of the high memory intensity workloads. We call these the "Homogeneous" workloads. We simulate each workload until every application in the workload completes at least 50 million instructions from a representative SimPoint [42].

High Intensity (MPKI >= 10)	omnetpp, milc, soplex, sphinx3, bwaves, libquantum, lbm, mcf
Medium Intensity (MPKI >=5)	zeusmp, cactusADM, wrf, leslie3d, GemsFDTD
Low Intensity (MPKI <5)	calculix, povray, namd, gamess, perlbench, tonto, gromacs, gobmk, dealII, sjeng, gcc, hmmer, h264ref, bzip2, astar, xalancbmk

Table 2: SPEC CPU2006 Classification by Memory Intensity.

We model chip energy with McPAT 1.3 [23], and DRAM power with CACTI 6.5 [27]. Dynamic counters stop updating once each benchmark completes. Shared structures dissipate static power until the completion of the entire workload. We model the CRE as a 2-wide back-end without certain structures such as a front-end, floating point pipeline, or register renaming hardware. Communication between the CRE and the core is modeled using extra messages on the address/data rings. We model the chain generation hardware at each core using additional energy events. Each operation included in the chain requires a CAM on destination register IDs to locate producer operations and extra ROB and physical register file reads. Each source register in every uop requires an RRT read. A destination register requires at least one RRT write. A MAP operation requires two RRT reads. We maintain a 32-entry buffer to store the last generated dependence chain. Operations are written into this buffer during chain generation and read from the buffer and packed into a data message when it is time to transmit the chain to the CRE.

H1	omnetpp+libq+sphinx3+milc
H2	soplex+milc+bwaves+libq
H3	bwaves+mcf+lbm+sphinx3
H4	mcf+milc+lbm+soplex
H5	libq+sphinx3+bwaves+lbm
H6	bwaves+sphinx3+libq+lbm
H7	soplex+bwaves+lbm+mcf
H8	mcf+soplex+omnetpp+lbm
H9	soplex+omnetpp+mcf+milc
H10	milc+libq+bwaves+mcf

Table 3: Quad-Core Workloads.

6. Results

We use instructions per cycle (IPC) as the performance metric for our single-core evaluation in Section 6.1. For the multi-core evaluation in Section 6.2 we use weighted speedup as the performance metric [11, 43].

6.1. Single-Core

Figure 11 shows the single-core performance results for the Continuous Runahead Engine (CRE). We compare six configurations against a no-prefetching baseline. We update the Runahead Buffer to use the Stall policy developed in Section 3.1 instead of the original policy [16]. As we have shown in Section 3.1, the Stall policy leads to a 7.5% performance gain over the original policy.

By continuously running ahead with the CRE, we observe a 14.4% performance gain over the system that uses only

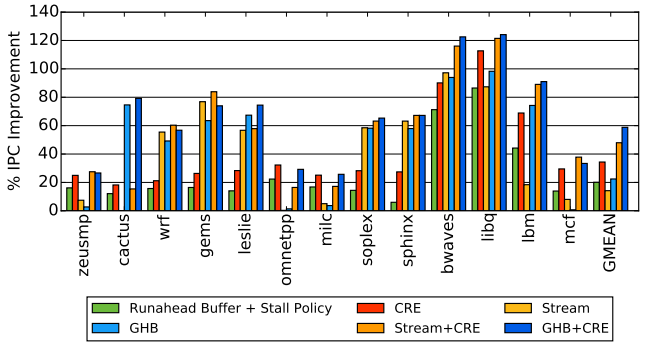

Figure 11: Single-core performance.

the Runahead Buffer, an 11.9% gain over the GHB prefetcher, and a 34.4% performance gain over the no-prefetching baseline. The CRE provides a performance win on all of the evaluated benchmarks and outperforms both the stream/GHB prefetchers. When the CRE is combined with prefetching, the CRE+GHB system (right-most bars) is the highest performing system, with a 36.4% average performance gain over the GHB prefetcher alone. The reason behind this performance gain is that the CRE is able to prefetch 70% of the runahead-reachable requests on average, as shown in Figure 12.

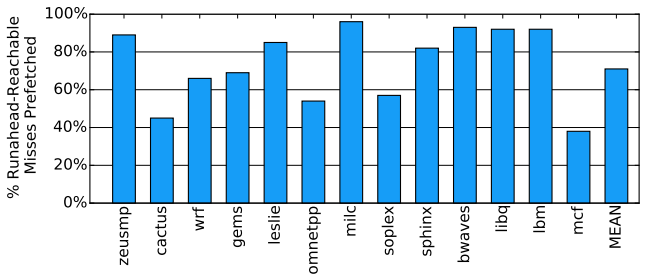

Figure 12: Percentage of runahead-reachable misses prefetched by the CRE.

Figure 13 shows that the CRE achieves this perfomance advantage with lower memory bandwidth overhead than the stream/GHB prefetchers. These prefetchers result in large bandwidth overheads on applications such as *omnetpp*, where they perform poorly. The CRE is more accurate than prefetching, especially on these applications, with a 7% average bandwidth overhead (vs. 11% and 22% overheads respectively for the stream and GHB prefetchers).

6.2. CRE Multi-Core Policies

While the CRE outperforms traditional pattern matching prefetchers in a single-core setting, it also requires additional hardware complexity (although the 10 kB storage overhead is less than the GHB prefetcher). The area overhead of the CRE is estimated at 10.4% of a full core and 7.8% of the total single-core chip area (based on McPAT area estimates). This overhead is more attractive when viewed in a multi-core setting where one CRE is shared among many different cores. We evaluate a quad-core system, where one CRE is located at the memory controller. We estimate the area overhead of the CRE in this configuration at 2% of total chip area.

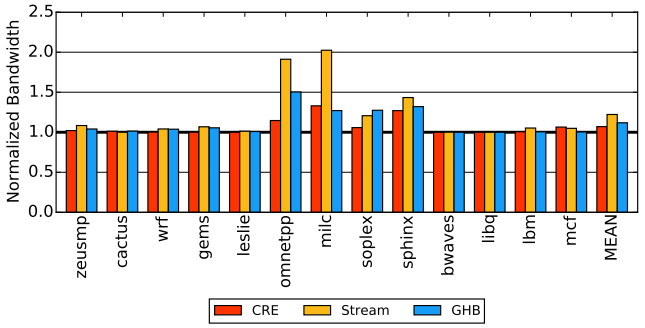

Figure 13: Single-core memory bandwidth consumption normalized to the no-prefetching system.

However, adding a single CRE to a quad-core processor means that the CRE is now a shared resource for which different cores contend. In this section, we evaluate three policies for determining which dependence chain to use at the CRE for a given interval. All three policies are interval based with an interval length of 100k instructions retired by the core that has provided a runahead dependence chain (Section 4.2). At the end of each interval, the CRE selects a new dependence chain to use for runahead. Dependence chains are generated by each core as described in Section 4.1.

The first policy is a round-robin policy. At the beginning of each interval, this policy chooses an *eligible core* in a round robin fashion. An eligible core has an MPKI above the threshold (MPKI > 5). The chosen core then provides the CRE with a dependence chain to use during Continuous Runahead. As explained above, this scheduling is repeated after the core that generated the dependence chain notifies the CRE that it has retired the threshold number of instructions.

The second policy is called the IPC policy. This policy uses the CRE to accelerate the application that is performing the worst. Therefore, at the beginning of each interval, the CRE schedules a dependence chain for Continuous Runahead from an eligible core with the lowest IPC in the workload.

The third policy is the Score policy. In this policy, we prioritize accelerating the dependence chain that is causing the workload to stall most. Recall from Section 3.1 that the Stall policy counts the number of times that each PC blocks retirement. At the beginning of each interval, the maximum stall count from each core is sent to the CRE and the CRE notifies the core with the highest stall count (or Score) to send a dependence chain for Continuous Runahead.

Figure 14 shows the performance results for these three policies for the heterogeneous workloads (H1-H10) and Figure 15 shows the results for the homogeneous workloads vs. a no-prefetching baseline. From this study, we find that the round-robin policy is the highest performing policy on average, across both the heterogeneous and the homogeneous workloads. Examining the homogeneous workloads in more detail, the round-robin policy is the highest performing policy on all workloads except for *4xlibquantum* where the Score policy performs best. The Score policy also comes close

to matching round-robin performance on *4xbwaves*. Both *libquantum* and *bwaves* have a very small number of PCs that cause full-window stalls (Figure 6). This indicates that the Score policy works best when there is a clear choice as to the dependence chain that is causing the workload to stall most.

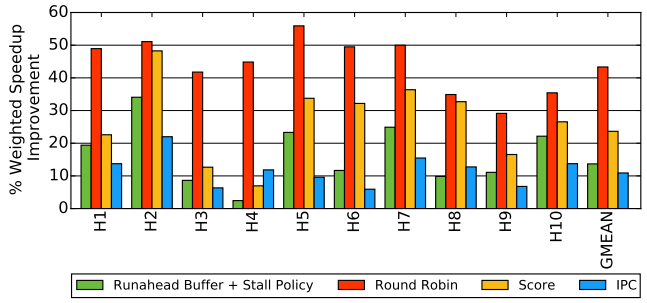

Figure 14: Heterogeneous workload policy evaluation.

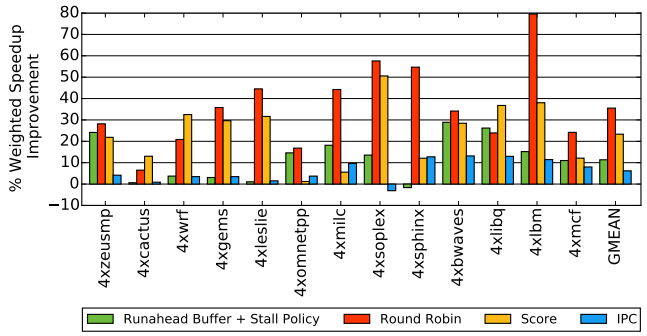

Figure 15: Homogeneous workload policy evaluation.

The Runahead Buffer + Stall policy results show that adding a Runahead Buffer to each core does not match the performance gains of the CRE policies. The Runahead Buffer is not able to run ahead for very long periods of time, reducing its performance impact. The IPC policy performs poorly. This is because it accelerates the benchmark with the smallest IPC. By doing this every interval, the IPC policy lengthens the number of cycles that the CRE executes a particular dependence chain. This interval is statically set to 100k retired instructions. A benchmark with a very low IPC takes longer to reach this threshold relative to the rest of the multi-core system. This means that the CRE runs ahead for more cycles than it would with a dependence chain from a different core, generating more runahead requests and resulting in a lower average runahead request accuracy.

Figure 16 shows the average performance improvement if the static 100k instruction interval length is changed to a dynamic interval length (called "throttled" in the Figure). To accomplish this, we track runahead request accuracy, similar to FDP [45]. Runahead fills set an extra-bit in the tag-store of each LLC cache line and MSHR entry. Upon eviction from the LLC, the CRE is notified (and increments a counter) if a runahead-fetched line was touched by the core. The counter is reset at the beginning of each runahead interval. Based on the counter, the CRE dynamically determines the length of the next runahead interval for that core. If request accuracy is above 95%, a 100k retired instruction interval is used. If it is greater than 90% or 85%, a 50k or 20k interval length is used respectively. Accuracy below 85% leads to a 10k interval length.

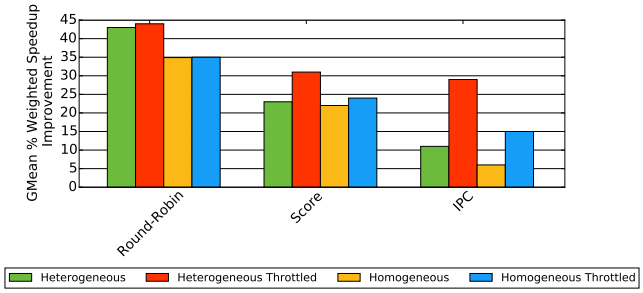

Figure 16: CRE performance using dynamic throttling.

Using a dynamic runahead interval length, the low-performing IPC policy shows the largest improvement, with a performance increase from 6% on the homogeneous workloads to 15%. On the heterogeneous workloads, IPC policy performance increases from 11% to 29%. However, the round-robin policy is still the highest performing policy with a 44% gain on the heterogeneous workloads and a 35% gain on the homogeneous workloads. Since a dynamic interval length provides negligible performance gain for the round-robin policy, we use the static 100k instruction threshold with the round-robin policy for the remainder of this evaluation.

6.3. Multi-Core Performance

Figures 17 and 18 show the performance results of a quad-core system (Table 1) using the CRE with a round-robin policy.

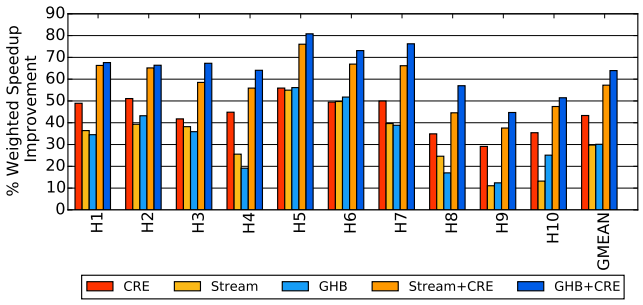

Figure 17: Heterogeneous workload performance.

The CRE results in a 35.5% performance gain on the homogeneous workloads and a 43.3% gain on the heterogeneous workloads over the no-prefetching baseline. The stream prefetcher results in a 23.9%/29.8% gain respectively. The GHB prefetcher gains 45% on the homogeneous workloads, due to large performance gains on *cactusADM* and 30.1% on the heterogeneous workloads. The highest performing system in both Figure 17 and 18 is the GHB + CRE configuration with a 55.8%/63.9% gain over the no-prefetching baseline.

Many of the trends from the single-core evaluation in Figure 11 hold on the quad-core homogeneous workloads. The CRE outperforms prefetching on *4xzeusmp*, *4xomnetpp*,

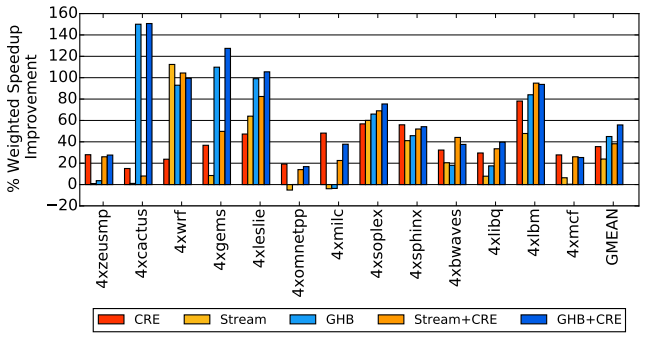

Figure 18: Homogeneous workload performance.

4xsphinx, and *4xmcf*. These are applications with low average prefetch accuracy and larger code footprints (Figure 6). The CRE outperforms or roughly matches prefetching performance on all heterogeneous workloads (Figure 17).

6.4. Multi-Core Energy and Overhead

Figure 19/20 show the energy results for the quad-core system running the heterogeneous/homogeneous workloads normalized to a no-prefetching baseline. Each bar is split into two components, showing static energy and dynamic energy.

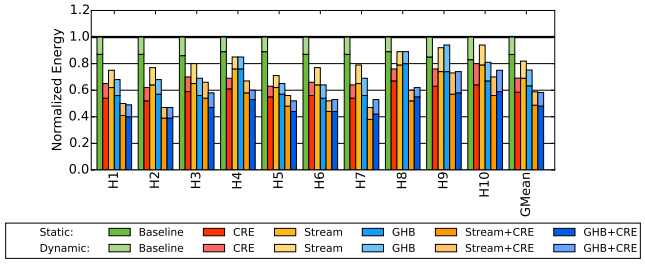

Figure 19: Heterogeneous workload energy consumption.

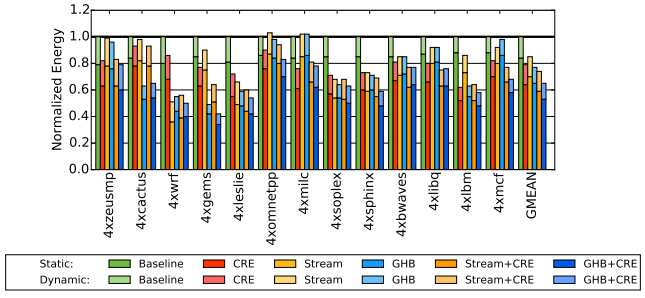

Figure 20: Homogeneous workload energy consumption.

As the evaluated workloads are all memory intensive, they all have low dynamic activity factors and run for longer than the single-core workloads due to contention in the memory system [35]. For example, the multi-core run of *4xmcf* runs for 42% more cycles than the single core run of *mcf* (Section 6.1). This causes energy consumption to be dominated by static energy. For *4xmcf*, static energy is 76.9% of total system energy consumption. These large static energy contributions relative to the small static energy cost of the CRE (2% of chip area in the multi-core case) mean that the large performance improvements from Section 6.3 translate to large energy reductions. We find that the system with the CRE reduces energy consumption by 22% on the homogeneous workloads and 30% on the heterogeneous workloads. These reductions correlate well to the relative performance gains of each workload. Overall, the GHB + CRE system is the highest performing and the most energy efficient.

The dynamic overhead of the CRE can be broken up into two components: shipping dependence chains to the CRE for execution and the increased pressure that CRE memory requests place on the LLC. Since CRE dependence chain updates occur at a coarse interval and we find that CRE dependence chains are short (14.5 uops long on average), the interconnect overhead of sending instructions to the CRE is low, under .01% of total data ring activity on average. However, the CRE causes an appreciable increase in the average number of LLC requests since CRE loads that miss in the CRE data cache query the LLC. We observe an average CRE data cache hit rate of 56%, leading to a 35% average increase in LLC requests.

6.5. Dependent Miss Acceleration

Prior work has proposed using compute hardware at the memory controller to reduce latency for cache misses that are dependent on off-chip data [15]. This is orthogonal to runahead execution, as runahead relies on having all source data available on-chip to generate new cache misses (i.e., runahead prefetches *independent* cache misses). Therefore, adding the prior *dependent* cache miss acceleration proposal to the CRE results in additional performance benefit. Dependent miss acceleration requires only the addition of two 16-uop dependent miss contexts to hold chains of operations that contain dependent cache misses. The total additional storage cost is 384 bytes over the CRE.

Figure 21 shows the performance benefits of the CRE and dependent miss acceleration on the homogeneous workloads. Overall, combining the CRE with dependent miss acceleration outperforms either mechanism alone, showing that the two techniques provide complimentary benefits.

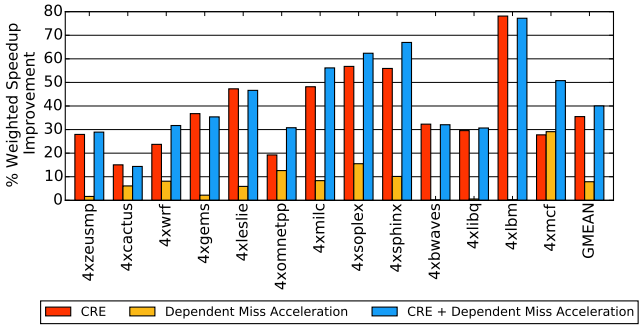

Figure 21: CRE + dependent miss acceleration on the homogeneous workloads.

7. Conclusion

We introduce the notion of Continuous Runahead to overcome a fundamental limitation of runahead execution: short runahead distance. We develop a low-cost implementation of this notion with two key ideas. First, we show that the

most critical dependence chain to accelerate with Continuous Runahead can be dynamically identified with low cost at runtime. Second, we show that this critical dependence chain can be *repeatedly* executed to generate new cache misses with a specialized Continuous Runahead Engine (CRE) located at the memory controller. Our results demonstrate that the CRE increases runahead prefetch coverage from 13% of all runahead-reachable cache misses (with traditional runahead) to 70%. This leads to a 21.9% performance gain over the state-of-the-art runahead mechanism on a single-core system. In a quad-core system, the CRE increases performance by 43.3% over a no-prefetching baseline and by 13.2% over the highest performing prefetcher (a GHB prefetcher) on a set of heterogeneous, memory intensive workloads. We conclude that Continuous Runahead is an effective hardware mechanism that transparently accelerates memory intensive applications.

Acknowledgments

We wish to thank the anonymous reviewers and Carlos Villavieja for valuable suggestions and feedback. We thank the members of the HPS Research Group for contributing to our working environment. We wish to thank Intel, Oracle, and Microsoft for their generous financial support. Onur Mutlu acknowledges support from Google, Intel, and Seagate.

References

[1] M. Annavaram *et al.*, "Data prefetching by dependence graph precomputation," in *ISCA-29*, 2001.

[2] R. D. Barnes *et al.*, "Beating in-order stalls with flea-flicker two-pass pipelining," in *MICRO-36*, 2003.

[3] J. A. Brown *et al.*, "Speculative precomputation on chip multiprocessors," in *MTEAC-6*, 2001.

[4] R. S. Chappell *et al.*, "Simultaneous subordinate microthreading (SSMT)," in *ISCA-26*, 1999.

[5] M. J. Charney and A. P. Reeves, "Generalized correlation-based hardware prefetching," Cornell Univ., Tech. Rep. EE-CEG-95-1, 1995.

[6] J. D. Collins *et al.*, "Dynamic speculative precomputation," in *MICRO-34*, 2001.

[7] J. D. Collins *et al.*, "Speculative precomputation: long-range prefetching of delinquent loads," in *ISCA-28*, 2001.

[8] R. Cooksey *et al.*, "A stateless, content-directed data prefetching mechanism," in *ASPLOS-10*, 2002.

[9] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *ICS-11*, 1997.

[10] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *HPCA-15*, 2009.

[11] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE MICRO*, no. 3, pp. 42–53, 2008.

[12] A. Garg and M. C. Huang, "A performance-correctness explicitly-decoupled architecture," in *MICRO-41*, 2008.

[13] J. D. Gindele, "Buffer block prefetching method," *IBM Technical Disclosure Bulletin*, vol. 20, no. 2, pp. 696–697, Jul. 1977.

[14] M. Hashemi, "On-chip mechanisms to reduce effective memory access latency," Ph.D. dissertation, The University of Texas at Austin, 2016.

[15] M. Hashemi *et al.*, "Accelerating dependent cache misses with an enhanced memory controller," in *ISCA-43*, 2016.

[16] M. Hashemi and Y. N. Patt, "Filtered runahead execution using a runahead buffer," in *MICRO-48*, 2015.

[17] G. Hinton *et al.*, "The microarchitecture of the Pentium® 4 processor," in *Intel Technology Journal, Q1*, 2001.

[18] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *ISCA-24*, 1997.

[19] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *ISCA-17*, 1990.

[20] M. Kamruzzaman *et al.*, "Inter-core prefetching for multicore processors using migrating helper threads," in *ASPLOS-16*, 2011.

[21] D. Kim and D. Yeung, "Design and evaluation of compiler algorithms for pre-execution," in *ASPLOS-10*, 2002.

[22] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction and dead-block correlating prefetchers," in *ISCA-28*, 2001.

[23] S. Li *et al.*, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO-42*, 2009.

[24] J. Lu *et al.*, "Dynamic helper threaded prefetching on the Sun UltraSPARC CMP Processor," in *MICRO-38*, 2005.

[25] C.-K. Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *ISCA-28*, 2001.

[26] "Micron Technology MT41J512M4 DDR3 SDRAM Datasheet Rev. K, April 2010," http://download.micron.com/pdf/datasheets/dram/ddr3/2Gb_DDR3_SDRAM.pdf.

[27] N. Muralimanohar and R. Balasubramonian, "CACTI 6.0: A tool to model large caches," in *HP Laboratories, Tech. Rep. HPL-2009-85*, 2009.

[28] O. Mutlu, "Efficient runahead execution processors," Ph.D. dissertation, The University of Texas at Austin, 2006.

[29] O. Mutlu *et al.*, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *HPCA-9*, 2003.

[30] O. Mutlu *et al.*, "Runahead execution: An effective alternative to large instruction windows," *IEEE MICRO*, vol. 23, no. 6, pp. 20–25, 2003.

[31] O. Mutlu *et al.*, "Address-value delta (AVD) prediction: Increasing the effectiveness of runahead execution by exploting regular memory allocation patterns," in *MICRO-38*, 2005.

[32] O. Mutlu *et al.*, "On reusing the results of pre-executed instructions in a runahead execution processor," *Computer Architecture Letters*, 2005.

[33] O. Mutlu *et al.*, "Techniques for efficient processing in runahead execution engines," in *ISCA-32*, 2005.

[34] O. Mutlu *et al.*, "Efficient runahead execution: Power-efficient memory latency tolerance," *IEEE MICRO*, vol. 26, no. 1, pp. 10–20, 2006.

[35] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *ISCA-35*, 2008.

[36] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *HPCA-10*, 2004.

[37] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *ISCA-21*, 1994.

[38] M. K. Qureshi *et al.*, "A case for MLP-aware cache replacement," in *ISCA-33*, 2006.

[39] T. Ramírez *et al.*, "Runahead threads to improve SMT performance," in *HPCA-14*, 2008.

[40] T. Ramírez *et al.*, "Efficient runahead threads," in *PACT-19*, 2010.

[41] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *ASPLOS-8*, 1998.

[42] T. Sherwood *et al.*, "Automatically characterizing large scale program behavior," in *ASPLOS-10*, 2002.

[43] A. Snavely and D. M. Tullsen, "Symbiotic job scheduling for a simultaneous multithreading processor," in *ASPLOS-9*, 2000.

[44] S. Somogyi *et al.*, "Spatial memory streaming," in *ISCA-33*, 2006.

[45] S. Srinath *et al.*, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *HPCA-13*, 2007.

[46] S. T. Srinivasan *et al.*, "Continual flow pipelines," in *ASPLOS-11*, 2004.

[47] K. Sundaramoorthy *et al.*, "Slipstream processors: improving both performance and fault tolerance," in *ASPLOS-9*, 2000.

[48] J. M. Tendler *et al.*, "POWER4 system microarchitecture," *IBM Technical White Paper*, Oct. 2001.

[49] R. Ubal *et al.*, "Multi2Sim: a simulation framework for cpu-gpu computing," in *PACT-21*, 2012.

[50] K. Van Craeynest *et al.*, "MLP-aware runahead threads in a simultaneous multithreading processor," in *HiPEAC-17*, 2009.

[51] W. Zhang, D. M. Tullsen, and B. Calder, "Accelerating and adapting precomputation threads for effcient prefetching," in *HPCA-13*, 2007.

[52] H. Zhou, "Dual-core execution: Building a highly scalable single-thread instruction window," in *PACT-14*, 2005.

[53] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *ISCA-28*, 2001.