Filtered Runahead Execution with a Runahead Buffer

Milad Hashemi
The University of Texas at Austin
miladhashemi@utexas.edu

Yale N. Patt
The University of Texas at Austin
patt@ece.utexas.edu

ABSTRACT

Runahead execution dynamically expands the instruction window of an out of order processor to generate memory level parallelism (MLP) while the core would otherwise be stalled. Unfortunately, runahead has the disadvantage of requiring the front-end to remain active to supply instructions. We propose a new structure (the Runahead Buffer) for supplying these instructions. We note that cache misses are often caused by repetitive, short dependence chains. We store these dependence chains in the runahead buffer. During runahead, the runahead buffer is used to supply instructions. This generates 2x more MLP than traditional runahead on average because the core can run further ahead. It also saves energy since the front-end can be clock-gated, reducing dynamic energy consumption. Over a no-prefetching/prefetching baseline, the result is a performance benefit of 17.2%/7.8% and an energy reduction of 6.7%/4.5% respectively. Traditional runahead with additional energy optimizations results in a performance benefit of 12.1%/5.9% but an energy increase of 9.5%/5.4%. Finally, we propose a hybrid policy that switches between the runahead buffer and traditional runahead, maximizing performance.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles - *Microprocessors and Microcomputers*

Keywords

Runahead Execution, Memory Wall, Energy Efficiency

1. INTRODUCTION

The latency of accessing off-chip memory is a key impediment to improving single-thread performance. Figure 1 demonstrates the magnitude of the problem across the SPEC CPU2006 benchmark suite. We simulate a 4-wide superscalar, out-of-order processor with a 192-entry reorder buffer (ROB) and 1MB of last level cache (LLC).The percent of total core cycles that the core is stalled waiting for memory is displayed on the y-axis. The benchmarks are sorted from lowest to highest memory intensity, defined by LLC misses per thousand instructions (MPKI).

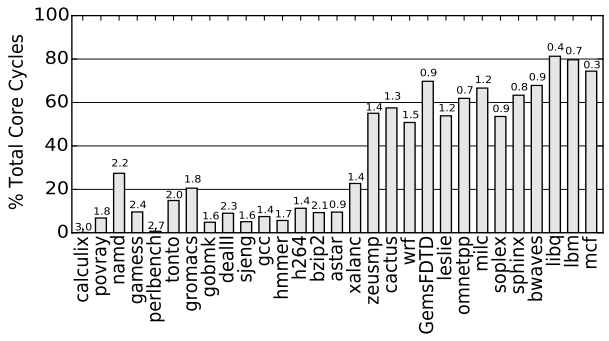

Figure 1: Percent of total cycles where the core is stalled and waiting for memory for the SPEC CPU2006 benchmark suite. The benchmarks are sorted by memory intensity. The average instructions per cycle (IPC) of each benchmark is listed on top of each bar.

As Figure 1 exhibits, the applications with the lowest IPC also generally spend the most time waiting for memory. We define any application to have high memory intensity if it has a MPKI of over 10. This includes roughly one third of the SPEC06 applications (any application to the right of *GemsFDTD* in Figure 1). All of these memory intensive applications spend over half of their total cycles stalled waiting for memory and largely have an IPC of under one.

This memory latency problem is commonly referred to as the memory wall [38, 37]. One technique that has been proposed to reduce the effect of the memory wall on single-thread performance is runahead execution [10, 25]. In runahead, once a core is stalled and waiting for memory, the processor's architectural state is checkpointed and the front-end continues to fetch and execute instructions. This creates a prefetching effect. The processor is able to use the application's own code to uncover additional cache misses when it would otherwise be stalled, thereby reducing the effective memory access latency of the subsequent demand request.

Runahead targets cache misses that have source data available on-chip but cannot be issued by the core due to limitations on the size of the reorder buffer. Therefore, instructions executed in runahead cannot be dependent

MICRO-48, December 05 - 09, 2015, Waikiki, HI, USA

ACM 978-1-4503-4034-2/15/12 $15.00
DOI: http://dx.doi.org/10.1145/2830772.2830812.

on a prior instruction that is a cache miss. We define "source data" as all of the data that is necessary for a memory instruction to generate an address that results in a cache miss. Figure 2 displays the fraction of all cache misses that have source data available on-chip for the SPEC06 benchmark suite.

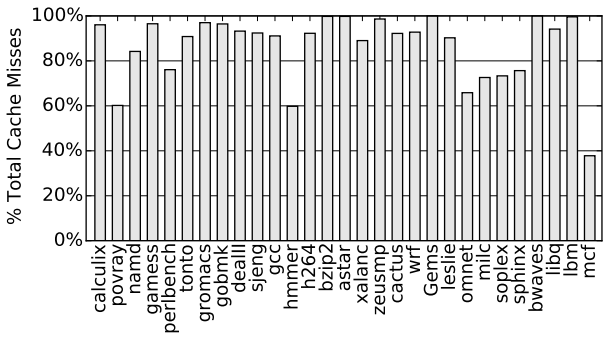

Figure 2: Percent of all cache misses that have source data available on chip.

We find that runahead targets the large majority of all cache misses for the SPEC06 benchmark suite, as Figure 2 shows that most source data is available on chip. However, runahead execution requires the front-end to remain on throughout the period of time when the core would be otherwise stalled. As front-end power consumption can reach 40% of total core power [1], this can result in a significant energy overhead. We seek to reduce this overhead with a new mode for runahead. In this work, we observe that most of the dependence chains that lead to cache misses in runahead execution are repetitive. We propose dynamically identifying these chains and using them to runahead with a new structure that we call a runahead buffer. This results in two benefits. First, by targeting only the filtered dependence chain, we frequently generate more MLP than the baseline runahead scheme by running further ahead. Second, by clock-gating the front-end during runahead we incur a much lower energy cost than the baseline runahead scheme.
We make the following contributions in this paper:

- We propose a mechanism to automatically identify a dependence chain to use during runahead execution, based on the PC of an outstanding cache miss.
- Using this mechanism, we dynamically fill dependence chains into a structure we call a Runahead Buffer. The operations in this buffer are then used to runahead when the core is otherwise stalled. We show that a system with a runahead buffer results in up to a 17.2% average performance increase along with a 6.7% *decrease* in energy consumption over the memory intensive SPEC06 benchmarks. This is compared to a 14.3% average performance increase and 9.0% *increase* in energy consumption for a system that uses traditional runahead execution. When a stream prefetcher is added to the system, we observe a 37.5% performance increase over the baseline. Traditional runahead and the runahead buffer result in speedups of 48.3% and 48.2% respectively with a stream prefetcher.
- We propose a policy to dynamically switch between traditional runahead and the runahead buffer. This policy maximizes the benefits of each mode, selecting the runahead buffer when it is best to do so and traditional runahead otherwise. With this hybrid policy we observe a 21.0% performance increase and 2.3% decrease in energy consumption without prefetching and a 51.5% performance increase with a stream prefetcher over the no-prefetching baseline.

This paper is organized as follows. We discuss prior related work in Section 2. Section 3 details the background of runahead execution and several observations that lead to the runahead buffer. We discuss the microarchitecture of the runahead buffer in Section 4 and our experimental methodology in Section 5. Our evaluation is in Section 6 and we conclude with Section 7.

2. RELATED WORK

There is a large body of prior work that uses hardware prefetching to reduce data-access latency for demand requests. This work can be generally divided into two categories: prefetchers that predict future addresses based on current and prior memory access patterns, and prefetching effects that are based on pre-execution of code. We first discuss prior work related to memory address prediction.

Prefetchers that uncover stream or stride patterns [11, 14, 27] require a small amount of hardware overhead and are commonly implemented in modern processors today [2]. These prefetchers can significantly reduce data access latency for predictable data access patterns, but can have heavy memory bandwidth requirements as well as difficulties with complex access patterns.

More advanced hardware prefetching techniques such as correlation prefetching [5, 13, 17, 31, 28] aim to reduce average memory access latency for more unpredictable cache misses. These prefetchers work by maintaining large tables that correlate past cache miss addresses to future cache misses. The global-history buffer [26] is a form of prefetching that uses a two-level indexing scheme to reduce the need for large correlation tables. Content-directed prefetching [8] does not require additional state to store pointers, but greedily prefetches using values that it believes to be addresses. This can result in a large number of inaccurate prefetch requests and poor prefetch timeliness.

Prior work has also pre-executed code segments from the running application with the goal of prefetching future addresses. These "helper threads" can be either obtained dynamically, or are generated by the compiler or programmer, and can run on a core or specialized hardware. Many prior works have researched statically generated helper threads: Collins et al. [7] generate

helper-threads with compiler analysis and require free hardware thread-contexts to execute them. Other work also constructs helper threads manually or with a compiler [40, 4, 20]. Kim and Yeung [16] discuss techniques for the static compiler to generate helper threads. Statically-generated helper threads have also been proposed to run on idle-cores of a multi-core processor [15, 3].

Dynamic generation of helper threads has also been explored. Collins et al. [6] propose a mechanism to extract helper-threads dynamically from the back-end of a processor by identifying dependence-chains. To remove helper-thread generation from the critical path of the processor, they propose adding large, post-retirement, hardware structures that all operations are filtered through. Once the helper threads are generated, they run on free SMT thread contexts, requiring the front-end to fetch and decode operations.

Slipstream [34] dynamically uses two threads to execute an application. The A-stream runs a filtered version of the application ahead of the R-stream. The A-stream can then communicate performance hints such as branch-directions or memory addresses for prefetching back to the R-stream. However, Slipstream does not target dependence chains during instruction filtering, instead removing unimportant stores and highly biased branches. Dynamic compilation techniques have also been pursued for generating helper threads [39, 19].

We propose an extension to runahead execution for out-of-order processors [24, 25]. Runahead allows the core to continue fetching and executing operations while the core is stalled and has the advantage over other dynamic pre-execution techniques of not requiring free hardware thread contexts to execute helper threads. Runahead discards all completed work when data for the blocking cache miss returns from memory. Techniques similar to runahead have also been proposed that do not discard completed work [33, 12]. We propose a new runahead policy that identifies the exact dependence chains to runahead with. We show that this increases performance and energy-efficiency.

3. BACKGROUND

As Figure 2 shows, the majority of all cache misses have source data available on-chip. Two main factors prevent an out-of-order processor from issuing these cache misses early enough to hide the effective memory access latency of the operation. The first factor is the limited resources of an out-of-order processor. An out-of-order core can only issue operations up to the size of its reorder buffer. Once this buffer is full, generally due to a long-latency memory access, the core can not issue additional operations that may result in a cache miss. The second factor is branch prediction. Assuming that limited resources are not an issue, the out-of-order processor would have to speculate on the instruction stream that generates the cache misses. However, prior work has shown that even wrong-path memory requests are generally beneficial for performance [23]. We consider solutions to the first factor in this work.

Runahead execution for out-of-order processors [25] is one solution to the first factor, the limited resources of an out-of-order processor. Runahead is a dynamic hardware mechanism that effectively expands the reorder buffer. Once the retirement of instructions is stalled by a long-latency memory access, the processor takes several steps.

First, architectural state, along with the branch history register and return address stack, are checkpointed. Second, the result of the memory operation that caused the stall is marked as poisoned in the physical register file. Once this has occurred, the processor continues fetching and executing instructions with the goal of generating additional cache misses.

Any operation that uses poisoned source data propagates the poison flag to its destination register. Store operations cannot allow data to become globally observable, as runahead execution is speculative. Therefore, a special runahead cache is maintained to hold the results of stores and forward this data to runahead loads. While runahead execution allows the core to generate additional MLP, it has the downside of requiring the front-end to be on and remain active when the core would be otherwise stalled, using energy. We examine this tradeoff in Section 3.1.

3.1 Runahead Observations

To uncover new cache misses, traditional runahead issues all of the operations that are fetched by the front-end to the back-end of the processor. Many of these operations are not relevant to calculating the address necessary for a subsequent cache miss. The operations required to execute a cache miss are encapsulated in the dependence chain of the miss. These are the only operations that are necessary to generate the memory address that causes the cache miss. In Figure 3 we compare the total number of operations executed in runahead to the number of operations that are actually in a dependence chain that is required to generate a cache miss.

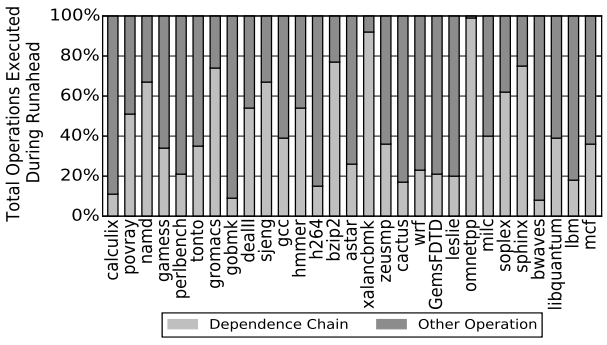

Figure 3: The average percentage of operations executed in runahead that are necessary to cause cache misses.

As Figure 3 shows, there are cases (*omnetpp*) where all of the executed instructions in runahead are necessary to uncover a cache miss. However, for most applications, this is not the case. For example, in *mcf* only

36 % of the instructions executed in runahead are necessary to cause a new cache miss. Ideally, runahead would only fetch and execute these required instructions, executing other operations is a waste of energy.

To observe how often these dynamic dependence chains vary, during each runahead interval, we trace the dependence chain for each generated cache miss. This chain is compared to all of the other dependence chains for cache misses generated during that particular runahead interval. Figure 4 shows how often each dependence chain is unique, i.e. how often a dependence chain has not been seen before in the current runahead interval.

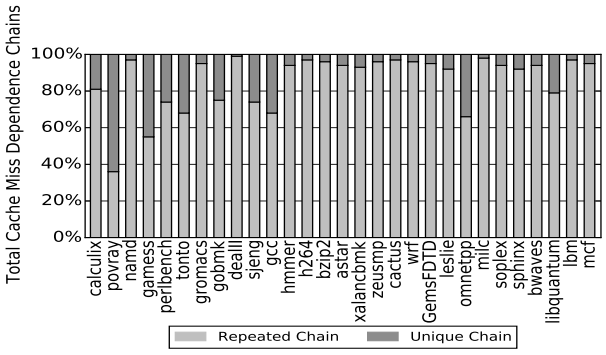

Figure 4: The average percentage of unique and repeated dependence chains leading to a cache miss in a given runahead interval.

As Figure 4 demonstrates, we find that most dependence chains are repeated, not unique, in a given runahead interval. This means that if an operation with a given dependence chain generates a cache miss, it is highly likely that a different dynamic instance of that instruction with the same dependence chain will generate another cache miss in the same interval. This is particularly true for the memory intensive applications on the right side of Figure 4.

We also find that each of these dependence chains are on average, reasonably short. Figure 5 lists the average length of the dependence chains for the cache misses generated during runahead in micro-operations (uops).

With the exception of *omnetpp*, all of the memory intensive applications in Figure 5 have an average dependence chain length of under 32 uops. Several benchmarks, including *mcf, libquantum, bwaves,* and *soplex*, have dependence chains of under 20 operations. Considering that the dependence chains that lead to cache misses during runahead are short and repetitive, we propose dynamically identifying these chains from the reorder buffer when the core is stalled. Once the chain is determined, we runahead by executing operations from this dependence chain. To accomplish this, we place the chains in a runahead buffer, similar to a loop buffer [9]. As the dependence chain is made up of decoded uops, the runahead buffer is able to feed these decoded ops directly into the back-end. Section 4 discusses how the chains are identified and the hardware structures required to support the runahead buffer.

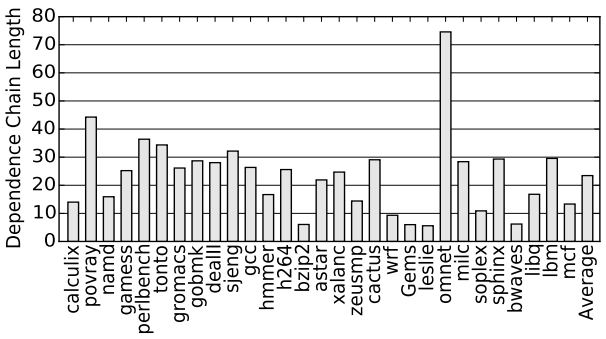

Figure 5: The average length of a dependence chain leading to a cache miss during runahead in uops.

4. RUNAHEAD BUFFER MICROARCHITECTURE

4.1 Hardware Modifications

To support the runahead buffer, small modifications are required to the traditional runahead scheme. A high-level view of a standard out-of-order processor is shown in Figure 6. We consider the front-end to include the fetch and decode stages of the pipeline. The back-end consists of the rename, select/wakeup, register read, execute and commit stages. To support traditional runahead execution, the shaded modifications are required. The physical register file must include poison bits so that poisoned source and destination operands can be marked. This is denoted in the register read stage. Additionally, the pipeline must support checkpointing architectural state, so that normal execution can recommence when the blocking operation returns from memory, and a runahead cache for forwarding store data as in [25]. These two changes are listed in the execute stage.

The runahead buffer requires two further modifications to the pipeline: the ability to dynamically generate dependence chains in the back-end and the runahead buffer, which holds the dependence chain itself. We also describe a small dependence chain cache in Section 4.4 to reduce how often chains are generated.

To generate and read filtered dependence chains out of the ROB, we use a pseudo-wakeup process. This requires every decoded uop to be available in the ROB and for the PC and destination register field to be searchable. Both the PC and destination register are already part of the ROB entry of an out-of-order processor. Destination register IDs are necessary to reclaim physical registers at retirement. Program counters are stored to support rolling back mispredicted branches and exceptions [30]. However, decoded uop information can be discarded upon instruction issue. We add 4-bytes per ROB entry to maintain this information until retirement.

We use both architectural register ids and physical register ids during the psuedo-wakeup process and runahead buffer execution. Physical register ids are used

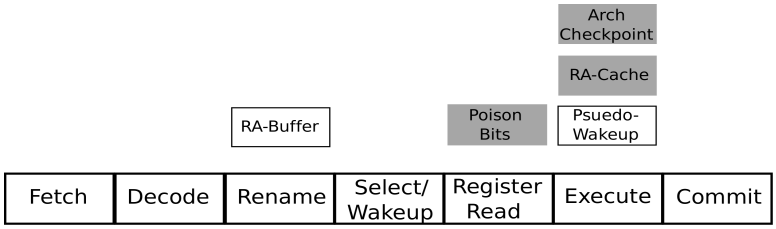

Figure 6: The runahead buffer pipeline.

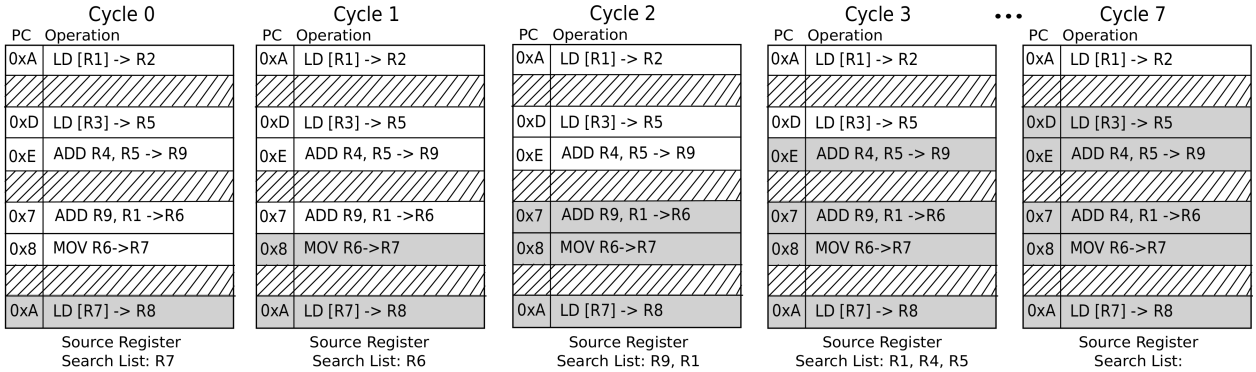

Figure 7: The dependence chain generation process. Listed register IDs correspond to physical registers. Only relevant uops are shown, other uops are hashed out.

during the dependence chain generation process. The runahead buffer itself is placed in the rename stage, as operations issued from the buffer are decoded but need to be renamed for out-of-order execution. Architectural register ids are used by the renamer once the operations are issued from the runahead buffer into the back-end of the processor. We describe the pseudo-wakeup process for generating dependence chains in Section 4.2.

4.2 Entering Runahead

Once a miss has propagated to the top of the reorder buffer, as in the traditional runahead scheme, runahead execution begins and the state of the architectural register file is checkpointed. This also triggers creation of the dependence chain for the runahead buffer. Figure 7 shows an example of this process with code from *mcf*. Control instructions are omitted in Figure 7 and not included in the chain, as the ROB contains a branch-predicted stream of operations. The dependence chain does not need to be contiguous in the ROB, only relevant operations are shown and other operations are hashed out.

In Figure 7, the load stalling the ROB is at PC:0xA. This load cannot be used for dependence chain generation as its source operations have likely retired. Instead, we speculate that a different dynamic instance of that same load is present in the ROB. This is based on the data from Figure 4 that showed that if a dependence chain generates a cache miss, it is very likely to generate additional cache misses.

Therefore, in cycle 0, we search the ROB for a different load with the same PC. To accomplish this, we modify the ROB to include a program-order based priority content addressable memory (CAM) for the PC and destination register ID field. This is similar to the CAM employed by the store-queue.

If the operation is found with the CAM, it is included in the dependence chain (denoted by shading in Figure 7). We track the uops that are included in the dependence chain with a bit-vector that includes one bit for every operation in the ROB. The source physical registers for the included operation (in this case R7) are maintained in a source register search list. These registers are used to generate the dependence chain.

During the next cycle, the destination registers in the ROB are searched using a CAM to find the uop that produces the source register for the miss. In this case, R7 is generated by a move from R6. In cycle 1, this is identified. R6 is added to the source register search list while the move operation is added to the dependence chain.

This process continues in cycle 2. The operation that produces R6 is located in the reorder buffer, in this case an ADD, and its source registers are added to the search list (R9 and R1). Assuming that only one source register can be searched for per cycle, in cycle 3 R4 and R5 are added to the search list and the second ADD is included in the dependence chain. This process is continued until the source register search list is empty, or the maximum dependence chain length (32 uops, based on Figure 5) is met. In Figure 7, this process takes 7 cycles to complete. In cycle 4 R1 finds no producers and in cycle 5 R4 finds no producing operations. In cycle 6, the load at address 0xD is included in the dependence chain, and in cycle 7 R3 finds no producers.

As register spills and fills are common in x86, loads additionally check the store queue to see if the load value is dependent on a prior store. If so, the store is included in the dependence chain and its source registers are added to the source register search list. We note that as runahead is speculative, the dependence chains are not required to be exact. We seek to generate a prefetching effect. While using the entire dependence chain is ideal, we find that capping the chain at 32 uops is sufficient for most applications. This dependence chain generation algorithm is summarized in Algorithm 1.

Once the chain is generated, the operations are read out of the ROB with the superscalar width of the backend (4 uops in our evaluation) and placed in the runahead buffer. Runahead execution then commences as in the traditional runahead policy.

Algorithm 1 Runahead Buffer dependence chain generation.
SRSL: Source Register Search List
ROB: Reorder Buffer
DC: Dependence Chain
MAXLENGTH: 32

```
if ROB Full then
  Get PC of op causing stall.
  Search ROB for another op with same PC.
  if Matching PC found then
    Add oldest matching op to DC.
    Enqueue all source registers to SRSL.
    while SRSL != EMPTY and
    DC < MAXLENGTH do
      Dequeue register from SRSL.
      Search ROB for op that produces register.
      if Matching op found then
        Add matching op to DC.
        Enqueue all source registers to SRSL.
        if Matching op is load then
          Search store buffer for load address.
          if Store buffer match then
            Add matching store to DC.
            Enqueue all source registers to SRSL.
          end if
        end if
      end if
    end while
    Fill runahead buffer with DC from ROB.
    Start runahead execution.
  end if
end if
```

4.3 Runahead Buffer Execution

Execution with the runahead buffer is similar to traditional runahead execution except operations are read from the runahead buffer as opposed to the front-end. The runahead buffer is placed in the rename stage. Since the dependence chain is read out of the ROB, operations issued from the runahead buffer are pre-decoded, but must be renamed to physical registers to support out-of-order execution. Operations are renamed from the runahead buffer at up to the superscalar width of the processor. Dependence chains in the buffer are treated as loops, once one iteration of the dependence chain is completed the buffer starts issuing from the beginning of the dependence chain once again. As in traditional runahead, stores write their data into a runahead cache so that data may be forwarded to runahead loads. The runahead buffer continues issuing operations until the data of the load that is blocking the ROB returns. The core then exits runahead, as in [25], and regular execution commences.

4.4 Dependence Chain Cache

We find that a cache to hold generated dependence chains can significantly reduce how often chains need to be generated prior to using the runahead buffer. We use a very small cache that is indexed by the PC of the operation that is blocking the ROB. Dependence chains are inserted into this cache after they are filtered out of the ROB. The chain cache is checked for a hit before beginning the construction of a new dependence chain. We disallow any path-associativity, so only one dependence chain may exist in the cache for every PC. As dependence chains can vary between dynamic instances of a given static load, we find that it is important for this cache to remain small. This allows old dependence chains to age out of the cache. Note that chain cache hits do not necessarily match the exact dependence chains that would be generated from the reorder buffer, we explore this further in Section 6.1.

4.5 Runahead Buffer Hybrid Policies

Algorithm 1 describes the steps necessary to generate a dependence chain for the runahead buffer. In addition to this algorithm, we propose a hybrid policy that chooses between traditional runahead and the runahead buffer with a chain cache. For this policy, if one of two events occur during the chain generation process, the core begins traditional runahead execution instead of using the runahead buffer. These two events are: an operation with the same PC as the operation that is blocking the ROB is not found in the ROB, or the generated dependence chain is too long (more than 32 operations).

If an operation with the same PC is not found in the ROB, we predict that the current PC will not generate additional cache misses in the near future. Therefore, traditional runahead will likely be more effective than the runahead buffer. Similarly, if the dependence chain is longer than 32 operations, we predict that the dynamic instruction stream leading to the next cache miss is likely to differ from the dependence chain that will be obtained from the ROB (due to a large number of branches). Once again, this means that traditional runahead is preferable to the runahead buffer, as traditional runahead can dynamically predict the instruction stream with the core's branch predictor, while the runahead buffer executes a simple loop. This hybrid policy is summarized in Figure 8 and evaluated in Section 6.1.

Core	4-Wide Issue, 192 Entry ROB, 92 Entry Reservation Station, Hybrid Branch Predictor, 3.2 GHz Clock Rate.
Runahead Buffer	32-entry. Micro-op size: 8 Bytes. 256 Total Bytes.
Runahead Cache	512 Byte, 4-way Set Associative, 8 Byte Cache Lines.
Chain Cache	2-entries, Fully Associative, 512 Total Bytes.
L1 Caches	32 KB I-Cache, 32 KB D-Cache, 64 Byte Cache Lines, 2 Ports, 3 Cycle Latency, 8-way Set Associative, Write-back.
Last Level Cache	1MB, 8-way Set Associative, 64 Byte Cache Lines, 18-cycle Latency, Write-back, Inclusive.
Memory Controller	64 Entry Memory Queue.
Prefetcher	Stream [35]: 32 Streams, Distance 32, Degree 2. Prefetch into Last Level Cache. Prefetcher Throttling: FDP [32]
DRAM	DDR3[21], 1 Rank of 8 Banks/Channel, 8KB Row-Size, CAS 13.75ns, Bank-Conflicts & Queuing Delays Modeled, 800 MHz Bus. 2 Channels.

Table 1: System Configuration

High (MPKI >= 10)	mcf, libquantum, bwaves, lbm, sphinx3, omnetpp, milc, soplex, leslie3d, GemsFDTD
Medium (MPKI >2)	zeusmp, cactusADM, wrf
Low (MPKI <= 2)	perlbench, bzip2, gcc, gobmk, hmmer, sjeng, h264ref, astar, xalancbmk, gamess, gromac, namd, dealII, povray, calculix, tonto

Table 2: SPEC06 Workload Classification by Memory Intensity

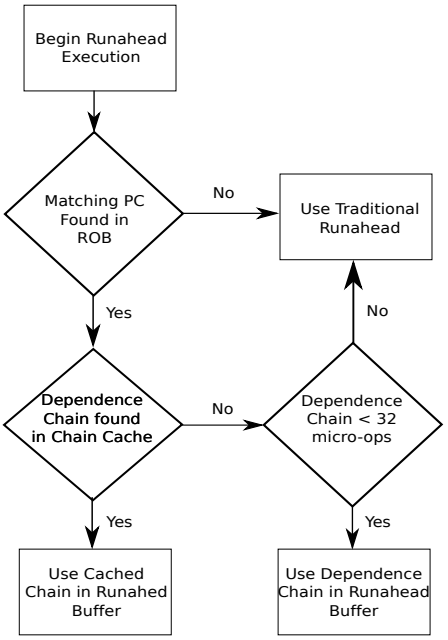

Figure 8: A flow chart of the hybrid policy.

4.6 Runahead Enhancements

We find that the traditional runahead execution policy significantly increases the total dynamic instruction count. This is due to repetitive and unnecessary runahead intervals as discussed in [24]. We implement the two hardware controlled policies from that paper. These policies limit how often the core can enter runahead mode. The first policy states that the core does not enter runahead mode unless the operation blocking the ROB was issued to memory less than a threshold number of instructions ago (we use 250). The goal of this optimization is to ensure that the runahead interval is not too short. It is important for there to be enough time to enter runahead mode and generate MLP. The second policy states that the core does not enter runahead unless it has executed further than the last runahead interval. The goal of this optimization is to eliminate overlapping runahead intervals. This policy helps ensure that runahead does not waste energy uncovering the same cache miss over and over again.

These policies are implemented in the runahead enhancements policy evaluated in Section 6.3 and the Hybrid policy described in Section 4.5. As the runahead buffer does not use the front-end during runahead, we find that these enhancements do not noticeably effect energy consumption for the runahead buffer policies.

5. METHODOLOGY

We use an execution driven, x86 cycle-level accurate simulator to model the runahead buffer. The front-end of the simulator is based on Multi2Sim [36]. The simulator faithfully models core microarchitectural details, the cache hierarchy, wrong-path execution, and includes a detailed non-uniform access latency DDR3 memory system. System details are listed in Table 1. The core uses a 192 entry reorder buffer. The cache hierarchy contains a 32KB instruction cache and a 32KB data cache with 1MB of last level cache. We model a stream prefetcher (based on the stream prefetcher in the IBM POWER4 [35]).

The runahead buffer we use in our evaluation can hold up to 32 micro-ops, this number was determined as best through sensitivity analysis. The dependence chain cache for the runahead buffer consists of two 32 micro-op entries. We additionally require a 24 byte bit vector to mark the operations in the ROB that are included in the dependence chain during chain generation, a sixteen element source register search list, and add 4-bytes per ROB entry to store micro-ops. The total storage overhead for the runahead buffer system is estimated at 1.7 kB.

We evaluate on the SPEC CPU2006 benchmark suite, but focus on the medium and high memory intensive applications (Table 2). Each application is simulated for 50 million instructions from a representative SimPoint [29]. Chip energy is modeled using McPAT 1.3 [18] and computed using total execution time, "runtime dynamic" power, and "total leakage power". McPAT models clock-gating the front-end during idle cycles for all simulated systems. DRAM power is modeled using CACTI 6.5 [22].

To enter runahead, both runahead and the runahead buffer require checkpointing the current architectural state. This is modeled by copying the physical registers pointed to by the register alias table (RAT) to a checkpoint register file. This occurs concurrently with dependence chain generation for the runahead buffer or before runahead can commence in the baseline runahead scheme. For dependence chain generation, we model a CAM for the destination register id field where up to two registers can be matched every cycle.

We model the runahead buffer dependence chain generation process with the following additional energy events. Before entering runahead, a single CAM on PCs of operations in the ROB is required to locate a matching load for dependence chain generation. Each source register included in the source register search list requires a CAM on the destination register ids of operations in the ROB to locate producing operations. Each load instruction included in the chain requires an additional CAM on the store queue to search for source data from prior stores. Each operation in the chain requires an additional ROB read when it is sent to the runahead buffer. The energy events corresponding to entering runahead are: a register alias table (RAT) and physical register reads for each architectural register and a write into a checkpoint register file.

6. RESULTS

We use instructions per cycle (IPC) as the performance metric for our single core evaluation. During our performance evaluation we compare to performance optimized runahead (without the enhancements discussed in Section 4.6) as these enhancements negatively impact performance. During our energy evaluation, we compare to energy optimized runahead, which uses these enhancements. We begin by evaluating the runahead buffer without prefetching in Section 6.1 and then with a stream prefetcher in Section 6.2.

6.1 Runahead Buffer Performance Results

Figure 9 shows the results of our experiments on the SPEC2006 benchmark suite. Considering only the low memory intensity applications in Table 2, we observe an average 0.8% speedup with traditional runahead. These benchmarks are not memory-limited and the techniques that we are evaluating have little to no-effect on performance. We therefore concentrate and the medium and high memory intensity benchmarks for this evaluation.

We evaluate four different systems against a no-prefetching baseline based on the parameters in Table 1. The "Runahead" system utilizes traditional out-of-order runahead execution. The "Runahead Buffer" system utilizes our proposed mechanism and does not have the ability to traditionally runahead. The "Runahead Buffer + Chain Cache" is the Runahead Buffer system but with an added cache that stores up to two, 32-operation dependence chains. The final system uses a "Hybrid" policy that combines the Runahead Buffer + Chain Cache system with traditional Runahead.

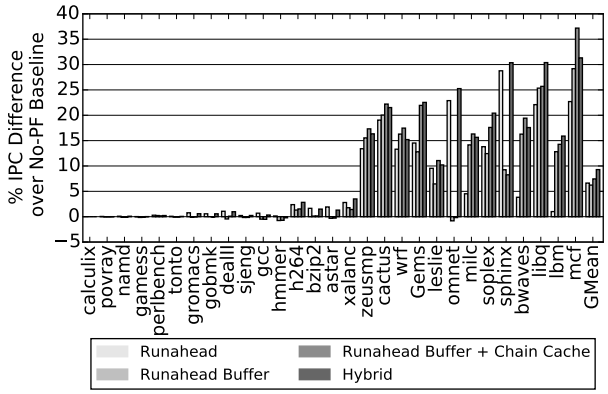

Figure 9: Performance of runahead configurations normalized to a No-Prefetching system. Benchmarks are sorted from lowest to highest memory intensity.

Considering only the medium and high memory intensity benchmarks, we observe performance improvements of 14.3%, 14.4%, 17.2% and 21.0% with traditional Runahead, the Runahead Buffer, Runahead Buffer + Chain Cache and Hybrid policy systems respectively. Traditional runahead performs well on *omnetpp* and *sphinx*, two benchmarks with longer average dependence chain lengths in Figure 5. The runahead buffer does particularly well on *mcf*, an application with short dependence chains, as well as *lbm* and *milc*, which have longer average dependence chains but a large number of unnecessary operations executed during traditional runahead (Figure 3).

By not executing these excess operations we find that the runahead buffer is able to generate more MLP than traditional runahead. Figure 10 shows the average number of cache-misses that are generated by runahead execution and the runahead buffer for the medium and high memory intensity SPEC06 benchmarks.

We observe that the runahead buffer generates over twice as many cache misses on average when compared to traditional runahead execution. Benchmarks where the runahead buffer shows performance gains over traditional runahead such as *zeusmp, cactus, milc, bwaves,* and *mcf* all show large increases in the number of cache misses produced by the runahead buffer. One application that does not perform well with the runahead buffer, *sphinx*, also shows a large increase in generated MLP. Figure 16, shows that this traffic is due to inaccurate memory requests. We mitigate this problem with our hybrid policy.

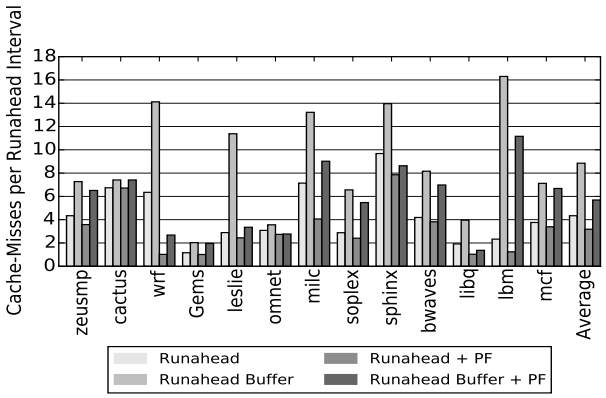

Figure 10: The average number of memory accesses per runahead interval generated by traditional runahead and the runahead buffer.

In addition to generating more MLP than traditional runahead on average, the runahead buffer also has the advantage of not using the front-end during runahead. The percent of total cycles that the front-end is idle and can be clock-gated with the runahead buffer are shown in Figure 11 for the medium and high memory intensity SPEC06 benchmarks.

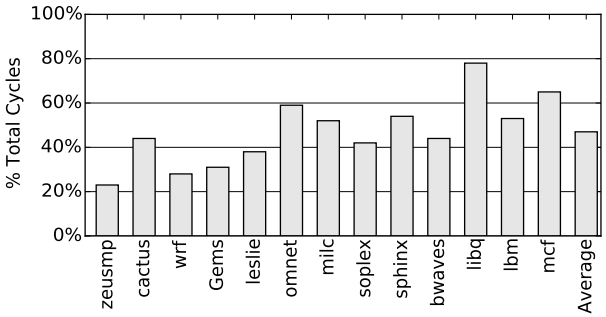

Figure 11: The percent of total cycles that the core is in runahead buffer mode.

On average, 47% of total execution cycles are spent in runahead buffer mode. By not using the front-end during this time, we reduce the dynamic energy consumption vs. traditional runahead execution on average, as discussed in the energy evaluation in Section 6.3.

Dependence Chain Cache: Looking beyond the simple runahead buffer policy, Figure 9 additionally shows the result of adding a small dependence chain cache to the runahead buffer system. This chain cache generally improves performance when added to the system, particularly for *mcf*, *soplex*, and *GemsFDTD*. Figure 12 shows the hit rate for the medium and high memory intensity applications in the chain cache.

The applications that show the highest performance improvements with a chain cache show very high hit rates in Figure 12, generally above 95%. The chain cache broadly improves performance over using a runahead buffer alone. We observe a slight performance drop in *sphinx*, an application where the runahead buffer does not perform better than traditional runahead.

The dependence chains in the chain cache do not necessarily match the exact dependence chains that would be generated from the reorder buffer. A chain cache hit is speculation that it is better to runahead with a previously generated chain than it is to take the time to generate a new chain. We find that this is an acceptable trade-off. In Figure 13, we analyze all chain cache hits to determine if the stored dependence chain matches the dependence chain that would be generated from the ROB.

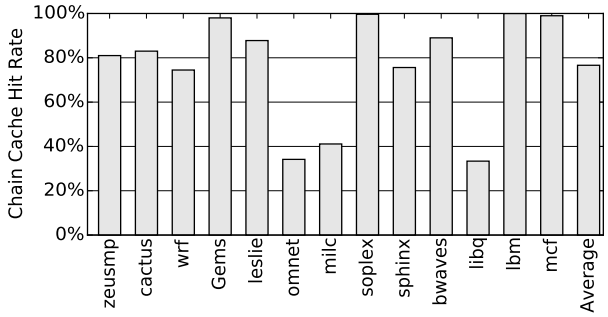

Figure 12: The chain cache hit-rate.

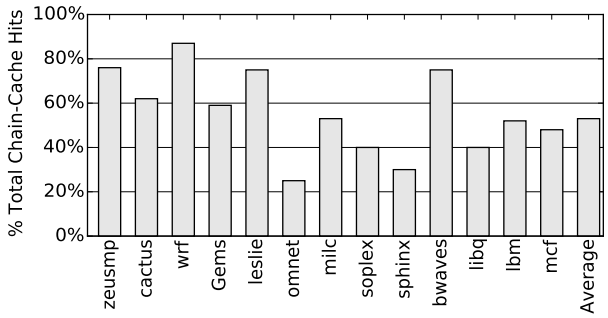

Figure 13: The percent of total chain cache hits that exactly match the dependence chain that would be generated from the ROB.

On average, the chain cache is reasonably accurate, with 53% of all dependence chains matching exactly. The two applications where the runahead buffer is not ideal, *omnetpp* and *sphinx*, show significantly less accurate chain cache hits than the other benchmarks.

Hybrid Policy: Lastly, the hybrid policy results in a average performance gain of 21.0% over the baseline. Figure 14 displays the fraction of time spent using the runahead buffer during the hybrid policy.

As Figure 14 shows, the hybrid policy favors the runahead buffer. 71% of the time the policy executes using the runahead buffer, the remainder is spent in traditional runahead. Applications that do not do well with the runahead buffer either the majority of the time (*omnetpp*), or a large fraction of the time (*sphinx*), executing traditional runahead execution. We conclude that the hybrid policy improves performance over the other schemes by using traditional runahead when it is best to do so (as in *omnetpp*) and leveraging the runahead buffer otherwise (as in *mcf*).

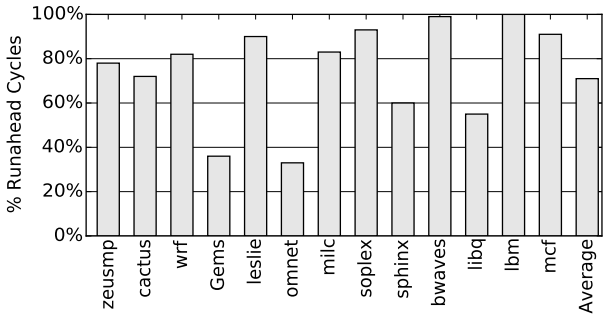

Figure 14: The percentage of cycles spent in runahead where the runahead buffer is used for the hybrid policy.

6.2 Comparison to Stream Prefetching

Figure 15 shows the effect of adding a stream prefetcher to the system for the memory intensive SPEC06 benchmarks.

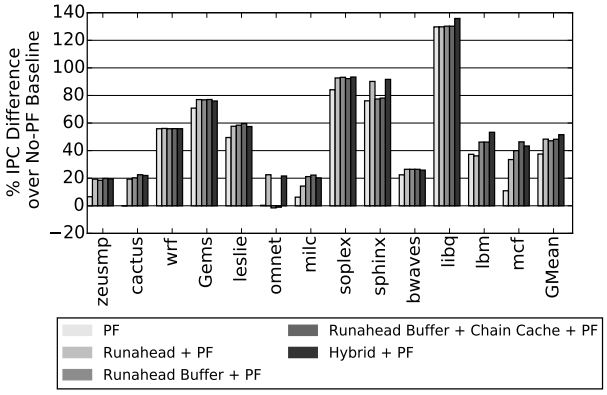

Figure 15: Performance with prefetching normalized to a No-Prefetching (No-PF) baseline system.

The stream prefetcher improves performance by 37.5% on average. Traditional runahead execution improves performance by 48.3% when combined with a stream prefetcher. The runahead buffer improves performance by 47.1% without a chain cache and 48.2% with a chain cache. The hybrid policy improves performance by 51.5%.

Runahead and the runahead buffer do well in cases where the stream prefetcher does not, such as *zeusmp, cactus,* and *mcf*. In the other cases, not including *wrf*, the hybrid policy is the highest performing policy on average. Runahead does not improve performance over prefetching in *wrf*. In general, we find that the runahead buffer in combination with prefetching improves core performance beyond using only a prefetcher. However, in addition to performance, the effect of prefetching on memory bandwidth is an important design consideration, as prefetching requests are not always accurate. Figure 16 quantifies the memory system overhead for prefetching and runahead.

On average, we find that the memory bandwidth requirements of runahead execution are small, especially when compared to a stream prefetcher. Traditional

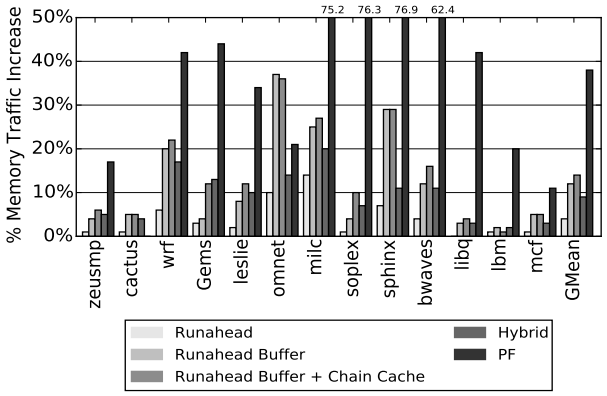

Figure 16: The number of additional DRAM requests generated by runahead and prefetching, normalized to the no-prefetching baseline.

runahead has a very small impact of memory traffic, increasing the total number of DRAM requests by 4%. This highlights the accuracy benefit of using fragments of the application's own code to prefetch. Using the runahead buffer alone increases memory traffic by 12%, including outliers like *omnetpp* and *sphinx* where the runahead buffer does not work well and increases memory traffic with inaccurate requests. The hybrid policy reduces overall DRAM traffic from 12% to 9%. Even with prefetcher throttling, the stream prefetcher has the highest DRAM overhead, issuing 38% more requests than in the baseline. While a stream prefetcher can significant increase performance, it also significantly increases memory traffic. We show in Section 6.3 that this excess memory traffic does not have a negative impact of energy-consumption in a single-core setting.

Figure 10 shows the effect that prefetching has on the amount of MLP that runahead can generate. As the prefetcher is able to prefetch some of the addresses that runahead generates, we find that a stream prefetcher reduces the MLP generated by traditional runahead by 27% on average and the runahead buffer by 36% on average. However, the runahead buffer is able to generate 80% more MLP than traditional runahead on average.

6.3 Energy Analysis

As discussed in Section 5, we evaluate system energy consumption using McPAT 1.3. The normalized results for the system without/with prefetching are shown in Figure 17/Figure 18 respectively.

Runahead alone drastically increases energy consumption due to very high dynamic instruction counts, as the front-end fetches and decodes instructions during periods where it would be otherwise idle. This observation has been made before [24], and several mechanisms have been proposed to reduce the dynamic instruction count, as discussed in Section 4.6. From this work, we implement the two hardware-based mechanisms that reduce the dynamic instruction count the most in "Runahead Enhancements". These mechanisms seek to eliminate short and overlapping runahead intervals.

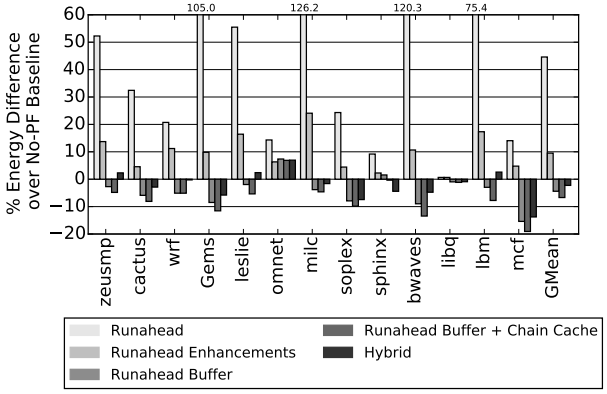

Figure 17: Normalized energy consumption for the system without prefetching.

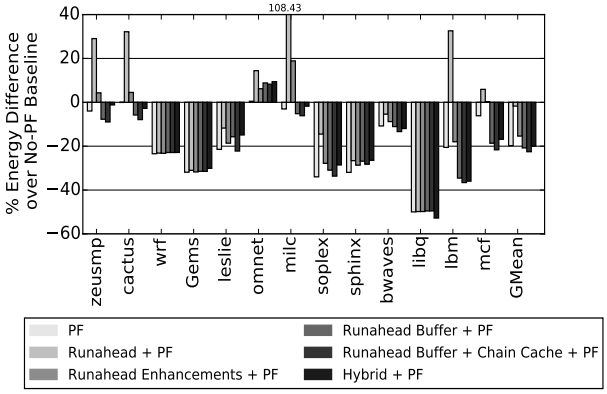

Figure 18: Normalized energy consumption for the system with prefetching.

With these enhancements, we observe drastically lower energy consumption with a 2.1% average degradation of runahead performance vs. the baseline (2.6% with prefetching). Traditional runahead increases system energy consumption by 44% and the system with the runahead enhancements increases energy consumption by 9% on average.

The runahead buffer reduces dynamic energy consumption by leaving the front-end idle during runahead periods. This allows the runahead buffer system to decrease average energy consumption by 4.4% without a chain cache and 6.7% with a chain cache. The hybrid policy decreases energy consumption by 2.3% on average. The runahead buffer decreases energy consumption more than the hybrid policy because the hybrid policy spends time in the more inefficient traditional runahead mode to maximize performance.

When prefetching is added to the system, we find that runahead + prefetching decreases energy consumption by 1.7% on average. Adding the runahead enhancements to traditional runahead decreases energy consumption by 15.4% over the no-prefetching baseline. As prefetching decreases energy consumption by 19.5% on average, both traditional runahead schemes increase energy consumption over a prefetching baseline. The runahead buffer designs further decrease energy consumption when combined with a stream prefetcher. The runahead buffer, runahead buffer + chain cache, and hybrid policy result in 20.8%, 22.5%, and 19.9% energy reductions from the baseline respectively. As in the no-prefetching scenario, we conclude that the runahead buffer + chain cache is the most energy efficient form of runahead execution.

7. CONCLUSION

We present an approach to increase the effectiveness of runahead execution for out-of-order processors. We identify that many of the operations that are executed in traditional runahead execution are unnecessary to generate cache-misses. Using this insight, we instead dynamically generate filtered dependence chains that only contain the operations that are required for a cache-miss. We find these chains to generally be short. The operations in a dependence chain are read into a buffer and speculatively executed as if they were in a loop when the core would be otherwise idle. This allows the front-end to be idle for 47% of the total execution cycles of the medium and high memory intensity SPEC06 benchmarks on average.

With this runahead mechanism, we generate over twice as much MLP on average as traditional runahead execution. This leads to a 17.2% performance increase and 6.7% decrease in energy consumption over a system with no-prefetching. Traditional runahead execution results in a 14.3% performance increase and 9.5% energy increase, assuming additional optimizations. Additionally, we propose a hybrid policy that uses traditional runahead alongside the runahead buffer when it is advantageous to do so. This policy increases performance, resulting in a 21.0% performance gain, but consumes additional energy, leading to a 2.3% energy reduction against a no-prefetching baseline. With prefetching, we observe an additional 10.7% and 14% performance gain for the runahead buffer and hybrid policy and a 3.0% and .4% energy decrease respectively. We observe that our policies require a fraction of the bandwidth consumption of a stream prefetcher.

Overall, we find that dynamically identifying specific instructions to runahead with results in a more capable and energy efficient version of runahead execution. The Runahead Buffer is a small structure, requiring 1.7 kB of total storage, that increases performance for memory latency-bound, single-threaded applications.

8. ACKNOWLEDGMENTS

We thank the members of the HPS research group and the anonymous reviewers for their comments and suggestions. Many thanks to Onur Mutlu, Khubaib, and Doug Carmean for helpful technical discussions. We gratefully acknowledge the support of the Cockrell Foundation and Intel Corporation.

9. REFERENCES

[1] "NVIDIA Tegra 4 Family CPU Architecture," http://www.nvidia.com/docs/IO/116757/NVIDIA_Quad_a15_whitepaper_FINALv2.pdf, 2013, [Online; Page 13; Accessed 8-May-2015].

[2] "Intel 64 and IA-32 Architectures Optimization Reference Manual," http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2014, [Online; Page 54; Accessed 4-May-2015].

[3] J. A. Brown, H. Wang, G. Chrysos, P. H. Wang, and J. P. Shen, "Speculative precomputation on chip multiprocessors," in *In Proceedings of the 6th Workshop on Multithreaded Execution, Architecture, and Compilation*, 2001.

[4] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "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, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic speculative precomputation," in *MICRO-34*, 2001.

[7] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: long-range prefetching of delinquent loads," in *ISCA-28*, 2001.

[8] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," in *ASPLOS-10*, 2002.

[9] Cray Research, Inc., "Cray-1 computer systems, hardware reference manual 2240004," 1977.

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

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

[12] A. Hilton and A. Roth, "Bolt: Energy-efficient out-of-order latency-tolerant execution," in *HPCA-16*, 2010.

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

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

[15] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Inter-core prefetching for multicore processors using migrating helper threads," in *ASPLOS-16*, 2011.

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

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

[18] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO-42*, 2009.

[19] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham, "Dynamic helper threaded prefetching on the Sun UltraSPARC CMP Processor," in *MICRO-38*, 2005.

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

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

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

[23] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt, "Understanding the effects of wrong-path memory references on processor performance," in *Third Workshop on Memory Performance Issues*, 2004.

[24] O. Mutlu, H. Kim, and Y. N. Patt, "Techniques for efficient processing in runahead execution engines," in *ISCA-32*, 2005.

[25] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *HPCA-9*, 2003.

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

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

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

[29] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS-10*, 2002.

[30] J. Smith and G. Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1609–1624, Dec 1995.

[31] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *ISCA-33*, 2006.

[32] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *HPCA-13*, 2007.

[33] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," in *ASPLOS-11*, 2004.

[34] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: improving both performance and fault tolerance," in *ASPLOS-9*, 2000.

[35] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy, "POWER4 system microarchitecture," *IBM Technical White Paper*, Oct. 2001.

[36] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: a simulation framework for cpu-gpu computing," in *PACT-21*, 2012.

[37] M. V. Wilkes, "The memory gap and the future of high performance memories," *SIGARCH Comput. Archit. News*, vol. 29, no. 1, pp. 2–7, 2001.

[38] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," in *SIGARCH Comput. Archit. News*, March 1995.

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

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