2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Accelerating Dependent Cache Misses with an Enhanced Memory Controller

Milad Hashemi*, Khubaib†, Eiman Ebrahimi‡, Onur Mutlu§, Yale N. Patt*

*The University of Texas at Austin †Apple ‡NVIDIA §ETH Zürich & Carnegie Mellon University

ABSTRACT

On-chip contention increases memory access latency for multi-core processors. We identify that this additional latency has a substantial effect on performance for an important class of latency-critical memory operations: those that result in a cache miss and are dependent on data from a prior cache miss. We observe that the number of instructions between the first cache miss and its dependent cache miss is usually small. To minimize dependent cache miss latency, we propose adding just enough functionality to dynamically identify these instructions at the core and migrate them to the memory controller for execution as soon as source data arrives from DRAM. This migration allows memory requests issued by our new Enhanced Memory Controller (EMC) to experience a 20% lower latency than if issued by the core. On a set of memory intensive quad-core workloads, the EMC results in a 13% improvement in system performance and a 5% reduction in energy consumption over a system with a Global History Buffer prefetcher, the highest performing prefetcher in our evaluation.

1. Introduction

On-Chip Latency. The large latency disparity between performing computation at the core and accessing data from off-chip DRAM has been a major performance bottleneck for decades. However, in the current multi-core era, the *effective* latency of accessing memory has increased due to on-chip interference. Figure 1 separates the delay incurred by a DRAM request into (a) the average time that the request takes to access DRAM and return data to the chip and (b) all other on-chip delays that the request incurs after missing in the LLC, for the *SPEC CPU2006* benchmark suite. We simulate a quad-core processor where each core has a 256-entry reorder buffer (ROB) and 1 MB of last level cache (LLC).

In Figure 1, benchmarks are sorted in ascending memory intensity. For the memory intensive applications to the right of *leslie*, defined as having an MPKI (misses per thousand instructions) of over 10, the actual DRAM access is less than half of the total latency of the memory request. Most of the effective memory latency is due to on-chip delay.

The on-chip latency overhead shown in Figure 1 is due to shared resource contention among the multiple cores. This contention happens in the shared on-chip interconnect, cache, and DRAM buses, row-buffers, and banks. Others [12, 15–17, 30, 31, 41, 42] have pointed out the effect of such interference on performance, and noted that this effect will increase as the number of cores increases [5, 27, 35].

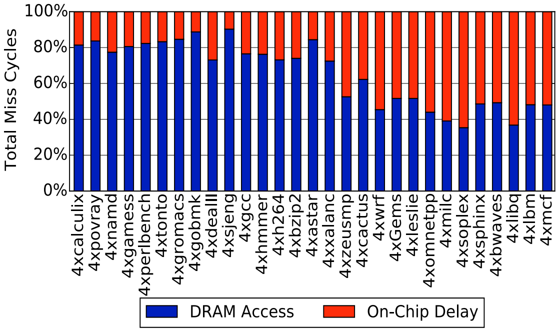


Figure 1: Breakdown of total memory access latency into DRAM latency and on-chip delay.

Criticality of Dependent Cache Misses. The impact of on-chip latency on processor performance is magnified when a cache miss has a dependent load that also results in a cache miss. These dependent cache misses are common in pointer chasing applications and prevent the core from making forward progress since the effective memory access latencies of both misses are serialized. Figure 2 shows the percentage of total LLC misses that are dependent on a prior LLC miss for *SPEC CPU2006*. The application with the highest fraction of dependent cache misses (*mcf*) has an IPC of just 0.3, the lowest performance of all benchmarks in the suite.

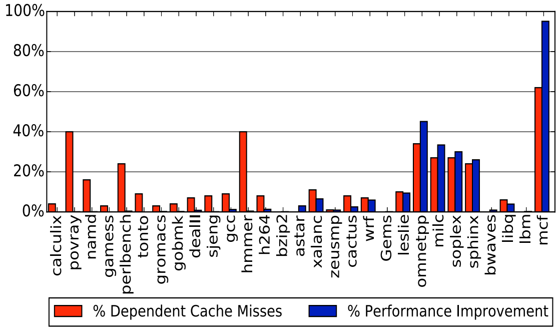


Figure 2: Percentage of LLC misses dependent on a prior LLC miss and the performance increase if the miss is a LLC hit.

Figure 2 also shows the performance increase of these benchmarks if all of the dependent cache misses had been LLC hits. The memory intensive benchmarks with a significant number of dependent cache misses experience large performance gains. For example, *mcf's* performance increases by 95%. Hence, decreasing the latency of these dependent cache misses is critically important to performance.

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.46

Prefetching. Several techniques have attempted to reduce the effect of dependent cache misses on performance. The most common is prefetching. Figure 3 shows the percent of all dependent cache misses that are prefetched by three different prefetchers: a global history buffer (GHB) prefetcher [43], a stream prefetcher [57], and a Markov prefetcher [25] for the memory intensive *SPEC CPU2006* benchmarks. The average percentage of all dependent cache misses that are prefetched is small, under 20% on average.

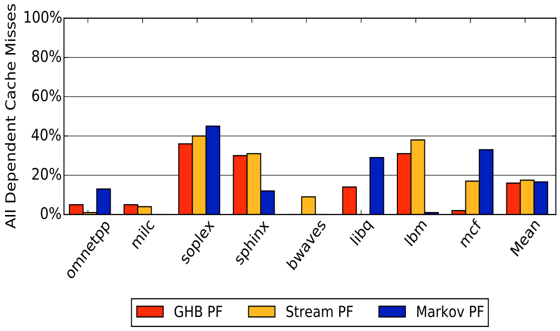


Figure 3: Percentage of dependent cache misses that are prefetched with a GHB, stream, and Markov prefetcher.

There are good reasons for this. Prefetchers have difficulty with dependent cache misses because their addresses are data dependent, leading to patterns that are hard to capture. Moreover, inaccurate and untimely prefetch requests lead to a large increase in bandwidth consumption, a significant drawback in a bandwidth constrained multi-core system. The GHB, stream, and Markov prefetchers increase bandwidth consumption by 20%, 22%, and 42% respectively.

Note that pre-execution techniques such as Runahead Execution [14, 38] and Continual Flow Pipelines [58] target prefetching *independent* cache misses. Unlike dependent cache misses, independent misses only require source data that is available on chip. These operations can be issued and executed by an out-of-order processor as long as the ROB is not full. Runahead/CFP discard or defer slices of operations that are dependent on a miss (including any dependent cache misses) to generate memory level parallelism with new independent cache misses.

1.1. The Enhanced Memory Controller (EMC)

The observations above suggest that dependent cache misses require a different acceleration mechanism. We have found that the number of operations between a cache miss and its dependent cache miss is usually small. If a dependent cache miss is likely, we propose using a dynamic dataflow walk to identify these operations. This dependence chain is then migrated to an enhanced memory controller (EMC) where it is executed immediately after the source data arrives from DRAM. This allows the EMC to generate cache misses faster than the core, thereby reducing the on-chip delay observed by the memory requests.

With this mechanism, some of the operations in the ROB are executed at the core, while others are executed remotely at the EMC. Figure 4 provides a high level view of partitioning a sequence of instructions between the EMC and the core.

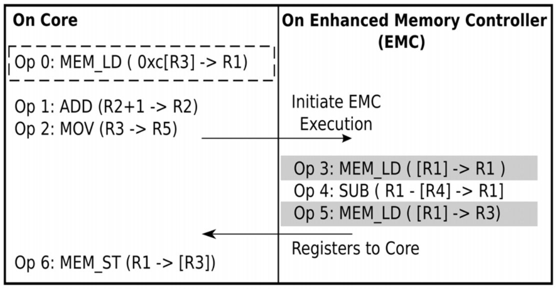

On Core
On Enhanced Memory Controller (EMC)
Op 0: MEM_LD (0xc[R3] -> R1)
Op 1: ADD (R2+1 -> R2)
Op 2: MOV (R3 -> R5)
Initiate EMC Execution
Op 3: MEM_LD ([R1] -> R1)
Op 4: SUB (R1 - [R4] -> R1]
Op 5: MEM_LD ([R1] -> R3)
Registers to Core
Op 6: MEM_ST (R1 -> [R3])

Figure 4: A sequence of 7 instructions. Instruction 0 is a cache miss and is surrounded by a dashed box. Dependent cache misses to be executed at the EMC are shaded gray.

In Figure 4, instruction 0 is the first cache miss. Instructions 1 and 2 are independent of instruction 0 and therefore execute at the core while instruction 0 is waiting for data from memory. Instructions 3, 4, and 5 are dependent on instruction 0. The core recognizes that instructions 3 and 5 will likely miss in the LLC, i.e., they are dependent cache misses, and so transmits instructions 3, 4, and 5 to execute at the EMC. When EMC execution completes, R1 and R3 are returned to the core so that execution can continue. To maintain the sequential execution model, operations sent to the EMC are not retired at the EMC, only executed. Retirement state is maintained at the ROB of the core and physical register data is transmitted back to the core for in-order retirement.

We make the following contributions in this paper:

- We propose the first mechanism to identify the chains of instructions that generate dependent cache misses at runtime and migrate their execution to a compute capable, enhanced memory controller (EMC). The EMC is motivated by the combination of three observations: 1) dependent cache misses are latency critical operations that are hard to prefetch, 2) the number of instructions between a source cache miss and a dependent cache miss is often small, 3) on-chip contention is a substantial portion of memory access latency in multi-core systems.
- We show that since the EMC is located near memory, it minimizes on-chip latency by executing dependence chains immediately when source data arrives from DRAM and issuing requests directly to DRAM. Memory requests issued by the EMC observe 20% lower latency than if issued by the core. A quad-core system with an EMC results in a 13% performance gain over a GHB prefetcher.
- We develop the design of the EMC, which implements the minimum functionality required to efficiently execute the dependence chains that generate dependent cache misses. The EMC requires 10.4% of the area of a full out-of-order core (a 2% total quad-core area overhead). The EMC maintains the traditional sequential execution model along with conventional cache coherence and virtual memory support.

2. Related Work

To discuss related work, we separate all cache misses into two categories. Dependent cache misses, which the EMC accelerates, and independent cache misses, which the EMC does not target. Prior work has researched two main approaches to reduce the effective memory latency observed by these two types of misses: prefetching (to access the address before a demand request), and moving computation closer to memory.

Stream or stride prefetchers [6, 20, 26, 57] lock on to simple access patterns and require a small amount of hardware overhead. These prefetchers effectively prefetch only independent cache misses (Figure 3). More advanced hardware prefetching techniques such as correlation prefetching [8, 25, 29, 56] do target dependent cache misses. These prefetchers maintain large tables that correlate past miss addresses to future miss addresses. The global-history buffer [43] is a form of correlation prefetching that uses a two-level indexing scheme to reduce the need for large correlation tables. Roth et al. [48] identify stable dependence patterns between pointers, and store this information in a correlation table. These pattern based prefetchers are oblivious to the control flow of the main thread of execution and are consequently bandwidth inefficient. They rely on past behavior re-occurring in the future and therefore cannot target all dependent cache misses.

Other hardware prefetchers specifically target the pointers that lead to cache misses [39, 49, 62]. Content-directed prefetching [11, 18] greedily prefetches by dereferencing values that could be memory addresses.

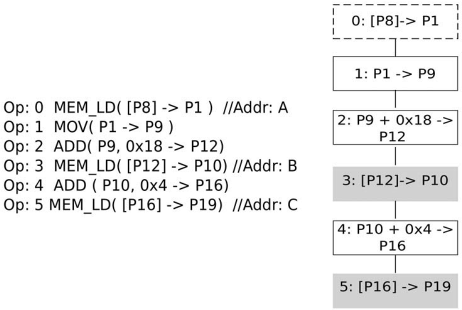
Another form of prefetching uses the application's own code to spawn speculative threads [9, 10, 64] or other forms of precomputation [4, 34, 60] to execute ahead of the demand access stream, generating independent cache misses. Similarly, Runahead Execution [14, 38, 40] and Continual Flow Pipelines [58] discard dependent cache misses to generate new independent cache misses. Solihin et al. [55] combine correlation prefetching and an extra execution context by proposing that a user level thread executing either in a DRAM chip or at the memory controller can leverage DRAM capacity to store the large correlation tables required for correlation prefetching.

Prior work has also considered enhancing the memory controller. Carter et al. [7] and Seshadri et al. [52] propose enhancements to the memory controller that include address remapping, prefetching and gather/scatter capabilities. Memory-side prefetching moves the hardware that prefetches data from the chip closer to DRAM [3, 22]. More generally, fabricating logic and memory on the same process has been proposed [19, 21, 28, 44, 53, 59] and recently revisited with 3D-stacked memory that incorporates a logic layer underneath DRAM layers [32], e.g., Hybrid Memory Cube (HMC) [45]. Industry and academia are also pursuing different methods of integrating compute and memory controllers [13,50,51]. Prior work has proposed performing computation inside the logic layer of 3D-stacked DRAM [1, 2, 63], but none has targeted automatically accelerating dependent cache misses.

To our knowledge, this is the first work that proposes adding compute capability to the memory controller to transparently accelerate chains of dependent cache misses. Our proposal differs from prior work in that we do not prefetch data; all of the requests sent by the EMC are demand requests. A dependent chain of computation is automatically extracted from the core and dynamically moved closer to memory. This allows the EMC to reduce access latency for all dependent cache misses, not just requests that can be easily prefetched.

3. Motivation

Figure 5 presents one example of the problem that we target. We adapt a dynamic sequence of micro-operations (uops) from *mcf*. The uops are shown on the left and the data dependencies, omitting control uops, are illustrated on the right. Core physical registers are denoted by a 'P'. Assume a scenario where Operation 0 is an outstanding cache miss. We call this uop a source miss and denote it with a dashed box. Operations 3 and 5 would result in cache misses when issued, shaded gray. However, their issue is blocked as their parent Operation 1 has a data dependence on the result of the source miss, Operation 0. Operations 3 and 5 are delayed from execution until the data from Operation 0 returns to the chip and flows back to the core through the interconnect and cache hierarchy. Yet, there are a small number of relatively simple uops between Operation 0 and Operations 3/5.



```
Op: 0  MEM_LD( [P8] -> P1 )  //Addr: A
Op: 1  MOV( P1 -> P9 )
Op: 2  ADD( P9, 0x18 -> P12)
Op: 3  MEM_LD( [P12] -> P10) //Addr: B
Op: 4  ADD ( P10, 0x4 -> P16)
Op: 5 MEM_LD( [P16] -> P19)  //Addr: C
```

Figure 5: Dependent cache misses: dynamic sequence of micro-ops based on the left, the dataflow graph is shown on the right. A, B, C represent cache line addresses.

We propose that the operations that are dependent on a cache miss be executed as soon as the source data enters the chip, at the memory controller. This avoids on-chip interference and reduces the overall latency to issue the dependent memory requests.

Figure 5 shows one instance where there are a small number of simple operations between the source and dependent miss. We find that this trend holds over the memory intensive *SPEC CPU2006* applications. Figure 6 shows the average number of operations in the dependence chain between a source and dependent miss, if a dependent miss exists. A small number of operations between a source and dependent miss means that the EMC does not have to do very much work to uncover a cache miss and that it requires a small amount of input data to do so.

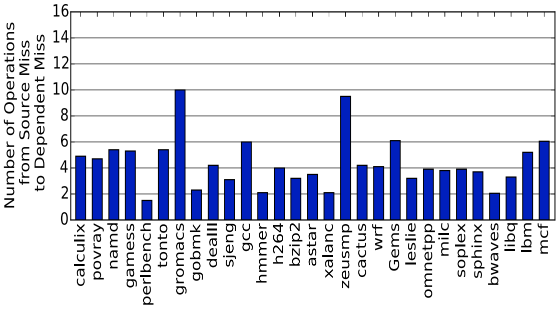


Figure 6: Average number of dependent operations between a source miss and dependent miss.

We therefore tailor the memory controller to execute dependent chains of operations such as those listed in Figure 5. Section 4.1 describes the required additional compute capability in detail. Since the instructions have already been fetched and decoded at the core and are waiting in the instruction-window, the core can automatically determine the uops to include in the dependence chain of a cache miss by leveraging the existing out-of-order execution machinery. Section 4.2 describes this process. The chain of decoded uops is then sent to the EMC. Once the data arrives from DRAM for the original miss, the EMC executes the dependent operations. Section 4.3 describes the details of EMC execution.

4. Mechanism

Figure 7 shows a quad-core chip that uses our proposed enhanced memory controller. The four cores are connected with a bi-directional ring. The memory controller is located at a single ring-stop, along with both memory channels, similar to Intel's Haswell microarchitecture [24]. Our proposal adds two pieces of hardware to the processor: limited compute capability at the memory controller (Section 4.1) and a dependence chain-generation unit at each of the cores (Section 4.2).

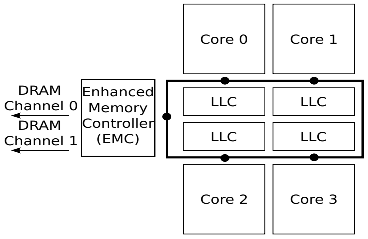


Figure 7: A high level view of a quad-core processor with an Enhanced Memory Controller. Each core has a ring stop, denoted by a dot, which is also connected to a slice of the shared last level cache.

4.1. EMC Compute Microarchitecture

We design the EMC to have the minimum functionality required to execute the pointer-arithmetic that generates dependent cache misses. Instead of a front-end, we utilize small uop buffers (Section 4.1.1). For the back-end, we use 2 ALUs and provide a minimal set of caching and virtual address translation capabilities (Section 4.1.2). Figure 8 provides a high level view of the EMC microarchitecture.

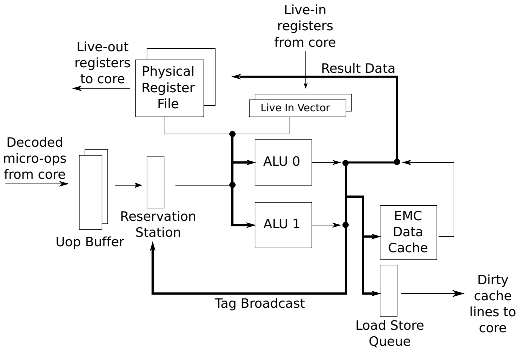


Figure 8: The microarchitecture of the EMC.

4.1.1. Front-End. The front-end of the EMC consists of two small uop buffers that can each hold a single dependence chain of up to 16 uops. With multiple buffers, the EMC can be shared between the cores of a multi-core processor. The front-end of the EMC consists only of this buffer, it does not contain any fetch, decode, or register rename hardware. Chains of dependent operations are renamed for the EMC using the out-of-order capabilities of the core (Section 4.2).

4.1.2. Back-End. As the EMC targets the pointer-arithmetic that generates dependent cache misses, it is limited to executing a subset of the total uops that the core is able to execute. Only integer operations are allowed (Table 1). Floating point and vector operations are not allowed. This simplifies the microarchitecture of the EMC, and enables the EMC to potentially execute fewer operations to get to the dependent cache miss. When the core creates a *filtered* chain of operations for the EMC to execute, only the operations that are required to generate the address for the dependent cache miss are included in the uop chain.

These filtered dependence chains are issued from the uop buffers to the 2-wide back-end. Our exploration shows that, for maximum performance, it is important to exploit the memory level parallelism present in the dependence chains. Therefore, the EMC has the capability to issue non-blocking memory accesses. This requires a small load/store queue (LSQ) along with out-of-order issue and wakeup using a small 8-entry reservation station and a common data bus (CDB). In Figure 8, the CDB is denoted by the result and tag broadcast buses. We support executing stores at the EMC due to how common register spills/fills are in the x86 ISA.

Each of the uop buffers in the front-end is allocated a private physical register file (PRF) that is 16 registers large and a private live-in source vector. As the out-of-order core has a much larger physical register file than the EMC (256 vs. 16 registers), we rename operations at the core to use the smaller physical register set of the EMC.

4.1.3. Caches. The EMC contains no instruction cache, but it does contain a small 4kB data cache that holds the most

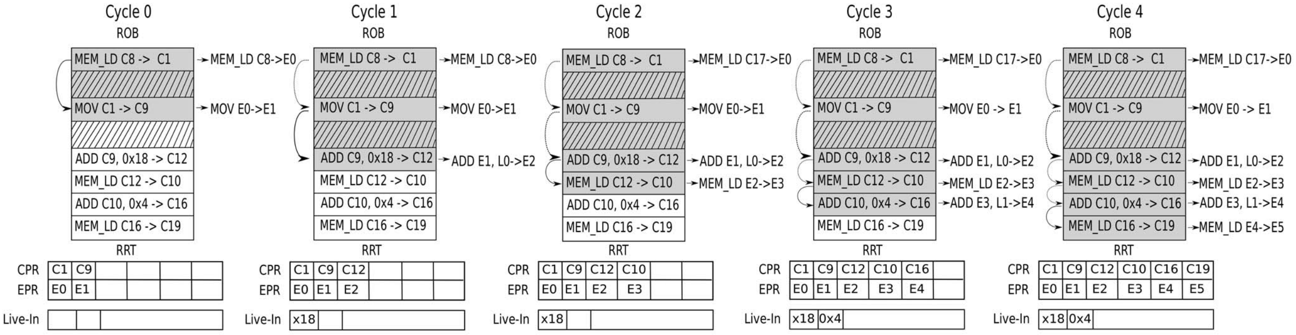


Figure 9: Chain generation using the chain of micro-ops from Figure 5. Three structures are shown: the reorder buffer (ROB), register remapping table (RRT), and live-in vector. CPR: Core Physical Register, EPR: EMC Physical Register. Processed operations are shaded after every cycle.

recent lines that have been transmitted from DRAM to the chip to exploit temporal locality. Cache coherence for this cache is maintained at the inclusive last-level cache by adding an extra bit to each directory entry for every cache line to track the cache lines that the EMC holds.

4.1.4. Virtual Address Translation. Virtual memory translation at the EMC occurs through a small 32 entry TLB for each core. The TLBs act as a circular buffer and cache the page table entries (PTE) of the last pages accessed by the EMC for each core. The PTEs of the core add a bit to each TLB entry to track if a page translation is resident in the TLB at the EMC. This bit is used to invalidate EMC TLB entries during the TLB shootdown process. Before a chain is sent to the EMC, the bit is also used to check if the PTE for the source miss is resident at the EMC TLB. If it is not, the core sends the source miss PTE to the EMC along with the dependence chain. The EMC does not handle page-faults: if the PTE is not available at the EMC, the EMC halts execution and signals the core to re-execute the entire chain.

4.2. Generating Chains of Dependent Micro-Operations

We leverage the out-of-order execution capability of the core to generate the short chains of operations that the EMC executes. This allows the EMC to have no fetch, decode, or rename hardware, as shown in Figure 8, significantly reducing its area and energy consumption.

The core can generate dependence chains to execute at the EMC once there is a full-window stall due to a LLC miss blocking retirement. If this is the case, we use a 3-bit saturating counter to determine if a dependent cache miss is likely. This counter is incremented if any LLC miss has a dependent cache miss and decremented if any LLC miss has no dependent cache misses. If either of the top 2-bits of the saturating counter are set, we begin the following process of generating a dependence chain for the EMC to accelerate.

We use the dynamic micro-op sequence from Figure 5 to demonstrate the chain generation process, illustrated by Figure 9. This process takes a variable number of cycles based on dynamic chain length (5 cycles for Figure 9). As the uops are included in the chain they are stored in a buffer maintained at the core until the entire chain has been assembled. At this point the chain is transmitted to the EMC.

For each cycle, we show three structures in Figure 9: the reorder buffer of the home core (ROB), the register remapping table (RRT), and a live-in source vector. The RRT is functionally similar to a register alias table and maps core physical registers to EMC physical registers. The operations in the chain have to be remapped to a smaller set of physical registers so that the EMC can execute them. The live-in source vector is a shift register that holds the input data necessary to execute the chain of operations. We only show a relevant portion of the ROB and omit irrelevant operations by denoting them with stripes.

In Figure 9, the cycle 0 frame shows the source miss at the top of the ROB. It has been allocated core physical register number 1 (C1) to use as a destination register. This register is remapped to an EMC register using the RRT. EMC physical registers are assigned using a counter that starts at 0 and saturates at the maximum number of physical registers that the EMC contains (16). In the example, C1 is renamed to use the first physical register of the EMC (E0) in the RRT.

Once the source miss has been remapped to EMC physical registers, chains of decoded uops are created using a forward dataflow walk that tracks dependencies through renamed physical registers. The goal is to mark uops that would be ready to execute when the source miss has completed. Therefore, the load that has caused the cache miss is *pseudo* "woken up" by broadcasting the tag of the destination physical register onto the common data bus (CDB) of the home core. A uop wakes up when the physical register tag of one of its source operands matches the tag that is broadcast on the CDB and all other source operands are ready. Pseudo waking up the uop does not execute or commit the uop; it simply broadcasts the uop's destination tag on the CDB to pseudo wake up dependent instructions.

In the example, there is only a single ready uop to broadcast in Cycle 0. The destination register of the source load (C1) is broadcast on the CDB. This wakes up the second operation in the chain, which is a MOV instruction that uses C1 as a

Algorithm 1: Dependence Chain Generation

```
//Process the source uop at ROB full stall;
Allocate EPR for destination CPR of uop in RRT;
Add uop to chain and broadcast destination CPR tag;
for each dependent uop do
    if uop Allowed and (all source CPRs ready or in RRT) then
        //Prepare the dependent uop to send to EMC;
        for each source operand do
            if CPR ready then
                Read data from PRF into live-in vector;
            else
                EPR = RRT[CPR];
            end
        end
        Allocate EPR for destination CPR in RRT;
        Add uop to chain and broadcast destination CPR tag;
        if Total uops in Chain == 16 then
            break;
        end
    end
end
Send filtered chain of uops and live-in data to EMC;
```

Figure 10: Dependence chain generation. CPR: Core Physical Register. EPR: EMC Physical Register. RRT: Register Remapping Table.

source register. It reads the remapped register id from the RRT for C1, and uses E0 as its source register at the EMC. The destination register (C9) is renamed to E1.

Operations continue to “wake up” dependent operations until either the maximum number of operations in a chain is reached, or there are no more operations to awaken. Thus, in the next cycle, the core broadcasts C9 on the CDB. The result of this operation is shown in cycle 1, when an ADD operation is woken up. This operation has two sources, C9 and an immediate value, 0x18. The immediate is shifted into a live-in source vector, which will be sent to the EMC along with the chain. The destination register C12 is renamed to E2 and written into the RRT.

In the example, the entire process takes five cycles to complete. In cycle 4, once the final load is added to the chain, a filtered portion of the execution window has been assembled for the EMC to execute. These uops are read out of the instruction window and sent to the EMC for execution. Algorithm 1 describes our mechanism for dynamically generating a chain of dependent uops. Note that dependence chain generation terminates when either all dependent operations have been identified or the maximum chain length (16 uops) is reached. Dependence chains frequently contain multiple levels of indirection (dependent loads). Out-of-order issue allows the EMC to react to dynamic hit/miss information, minimizing request latency.

4.3. EMC Execution

To start execution, the EMC takes two inputs: the source vector of live-in registers and the executable chain of operations. The EMC does not commit architectural state, it executes the chain of uops speculatively and sends the destination physical registers back to the core. Two special cases arise with respect to control operations and memory operations. First, we discuss control operations.

The EMC does not fetch instructions and is sent the branch-predicted stream that has been fetched in the ROB. We send branch directions along with computation to the EMC so that the EMC does not generate wrong path memory requests. If the EMC determines that the dependence chain it is executing contains a mispredicted branch, it stops execution and notifies the core of the mispredicted branch. The EMC has the capability to detect branch mispredictions made by the core, but it cannot restart from the correct path.

For memory operations, a load first queries the EMC data cache: if it misses in the data cache it generates an LLC request. However, the EMC can predict if a load is going to result in a cache miss. This enables the EMC to directly issue the request to memory if it is predicted to miss in the LLC, thus saving the latency to access the on-chip cache hierarchy. To enable this capability we keep an array of 3-bit counters for each core, similar to [47]. The PC of the miss-causing instruction is used to hash into the array. On a miss, the corresponding counter is incremented; a hit decrements the counter. If the counter is above a threshold, the load is sent directly to memory without accessing the LLC.

A store is included in the dependence chain only if it is a register spill. This is determined by searching the home core LSQ for a corresponding load with the same address (fill) during dependence chain generation. A store executed at the EMC writes its data value into the EMC LSQ.

Load and store operations are retired in program order back at the home core. When a memory operation is executed at the EMC, it sends a message on the address ring back to the core. The core snoops this request and populates the relevant entry in the LSQ. This serves two purposes. First, if a memory disambiguation problem arises (i.e., there is a store to the same address as a load executed at the EMC), execution of the chain can be canceled. Second, for consistency reasons, stores executed at the EMC are not made globally observable until the store has been drained from the home core store-queue in program order.

While executing chains of instructions remotely requires these modifications to the core, transactional memory implementations that are built into current hardware [23] provide many similar guarantees for memory ordering.

Once a dependence chain has completed execution, the live-outs, including the store data from the LSQ, are sent back to the core. Physical register tags are broadcast on the home core CDB, and execution on the home core continues. As the home core maintains all instruction state for in-order retirement, any *exceptional* event (e.g., branch misprediction, EMC TLB-miss, EMC exception) causes the home core to re-issue and execute the entire chain normally without shipping it to the EMC.

Core	4-wide issue, 256-entry ROB, 92-entry reservation station, hybrid branch predictor, 3.2 GHz clock rate
L1 Caches	32 KB I-Cache, 32 KB D-Cache, 64 byte lines, 2 ports, 3 cycle latency, 8-way, write-through.
L2 Cache	Distributed, shared, 1MB 8-way slice per core, 18-cycle latency, write-back. 4-Core: 4 MB. 8-Core: 8MB.
Interconnect	2 Bi-directional rings: control (8 bytes)/data (64 bytes). 1 cycle core to LLC slice bypass. 1 cycle ring links.
EMC Compute	2-wide issue. 8-entry reservation station. 32-entry TLB per core. 4kB data cache, 4-way, 2-cycle access, 1-port. 4-Core: 2 contexts. 8-Core: 4 contexts total. Each context contains: 16-entry uop buffer, 16-entry physical register file, 16-entry live-in vector, 8 LSQ-entries. Micro-op size: 6 bytes in addition to any live-in source data.
EMC Instructions	Integer: add/subtract/move/load/store. Logical: and/or/xor/not/shift/sign-extend.
Memory Controller	Batch Scheduling [42]. 4-Core: 128-entry memory queue. 8-Core: 256-entry memory queue.
Prefetchers	Stream: 32 streams, distance 32. Markov: 1MB correlation table, 4 addresses per entry. GHB G/DC: 1k-entry buffer, 12KB total size. All configurations use FDP [57]: dynamic degree 1-32, prefetch into LLC.
DRAM	DDR3 [36], 1 Rank of 8 Banks/Channel, 8KB Row-Size, CAS 13.75ns, bank-conflicts & queuing delays modeled, 800 MHz bus. 4-Core: 2 Channels. 8-Core: 4 Channels.

Table 1: System configuration.

4.4. Multiple Memory Controllers

We primarily consider a common quad-core processor design, where one memory controller has access to all memory channels from a single location on the ring (Figure 7). However, with large core counts, multiple memory controllers can be distributed across the interconnect. In this case, with our mechanism, each memory controller would be compute capable. On cross-channel dependencies (where one EMC generates a request to a channel located at a different enhanced memory controller), the EMC directly issues the request to the other memory controller without migrating execution of the chain. This cuts the core, a middle-man, out of the process. We evaluate this scenario with an eight-core CMP (Figure 11b) and compare the results to an eight-core CMP with a single memory controller (Figure 11a) in Section 6.2.

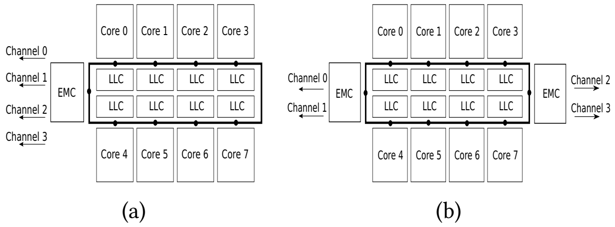


Figure 11: (a) Single memory controller. (b) Dual memory controller.

5. Methodology

We simulate three systems: a quad core system (Figure 7) and two eight-core systems (Figure 11). Table 1 lists the details of our system configurations. The cache hierarchy of each core contains a 32KB instruction cache and a 32KB data cache. The LLC is divided into 1MB cache slices per core. The interconnect is composed of two bi-directional rings, a control ring and a data ring. Each core has a ring-stop that is shared with the LLC slice.

We model three different prefetchers. A stream prefetcher [57] (based on the stream prefetcher in the IBM POWER4 [61]), a Markov prefetcher [25], and a global-history-buffer (GHB) based global delta correlation (G/DC) prefetcher [43]. Prior work has shown a GHB prefetcher to outperform a large number of other prefetchers [46]. We find that the stream prefetcher always increases performance when used with a Markov prefetcher, and therefore employ them together.

The baseline system uses a sophisticated memory scheduling algorithm, batch scheduling [42], and Feedback Directed Prefetching (FDP) [57] to throttle prefetchers. The parameters for the EMC listed in Table 1 (TLB size, cache size, number/size of contexts) have been chosen via sensitivity analysis. In the eight-core, dual memory controller case (Figure 11b), each EMC contains 2 issue contexts for 4 total contexts, and is otherwise identical to the EMC in the eight-core single memory controller configuration.

We separate the *SPEC CPU2006* benchmarks into two categories: high memory intensity and low memory intensity by MPKI. The classification of each benchmark is listed in Table 2. As the EMC is primarily intended to accelerate memory intensive applications, we focus on high memory intensity workloads in our evaluation. The EMC does not result in appreciable performance gains on most low memory intensity applications. Using the high intensity benchmarks, we randomly generate a set of ten quad-core workloads to evaluate (Table 3). Each benchmark appears only once in every workload combination. We additionally evaluate homogeneous quad-core workloads using four copies of each of the high memory intensity benchmarks. Eight-core workloads are two copies of the corresponding quad-core workload.

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

Table 2: *SPEC CPU2006* classification by memory intensity.

We use an in-house cycle accurate x86 simulator, which faithfully models core microarchitectural details, the cache hierarchy, and includes a detailed non-uniform access latency DDR3 memory system. We simulate each workload until every application in the workload completes at least 50 million instructions from a representative SimPoint [54].

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

Table 3: Quad-Core workloads.

We model chip energy using McPAT [33] and DRAM power using CACTI [37]. Shared structures dissipate static power until the completion of the entire workload. Event counters used for dynamic power computation are updated until each benchmark's completion. The EMC is modeled as a stripped down core and does not contain structures like an instruction cache, decode stage, register renaming hardware, or a floating point pipeline.

We model the chain generation unit by adding the following additional energy events corresponding to the chain generation process at each home core. Each of the uops included in the chain requires an extra CDB access (tag broadcast) due to the pseudo wake-up process. Each of the source operations in every uop requires a Register Remapping table (RRT) lookup, and each destination register requires an RRT write since the chain is renamed to the set of physical registers at the EMC. Each operation in the chain requires an additional ROB read when it is transmitted to the EMC. We model data and instruction transfer overhead to/from the EMC via additional messages sent on the ring.

6. Results

6.1. Quad-Core Evaluation

Figure 12 shows the performance of the quad-core system for workloads H1-H10. Performance gain due to the EMC over the no-prefetching baseline and each prefetching configuration is illustrated as a bold/hashed bar.

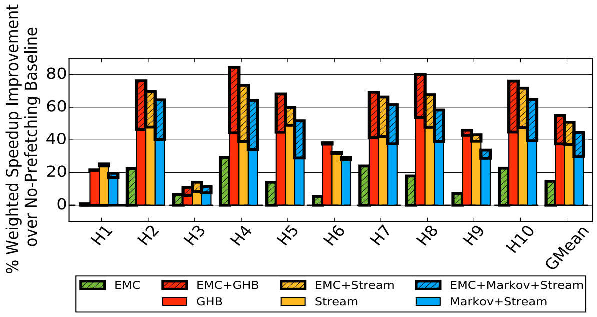


Figure 12: Quad-Core performance for workloads H1-H10.

On H1-H10, the EMC improves performance on average by 15% over a no-prefetching baseline, by 13% over a baseline with a GHB prefetcher, 10% over a baseline with stream prefetching, and by 11% over a baseline with both a stream and Markov prefetcher. Workloads that include a *SPEC CPU2006* benchmark with a high rate of dependent cache misses (Figure 2) such as *mcf* or *omnetpp* tend to perform well, especially when paired with other highly memory intensive workloads like *libquantum* or *bwaves*. Workloads with *lbm* tend not to perform well. *lbm* contains no dependent cache misses, leaving no room for improvement with EMC, and has a regular access pattern that utilizes most of the available bandwidth, particularly with prefetching enabled.

To provide insight into the performance implications of the EMC on homogeneous workloads, Figure 13 shows a system running four copies of each high memory intensity *SPEC06* benchmark. Overall, the EMC results in a 9.5% performance advantage over a no-prefetching baseline and roughly 8% over each prefetcher. *mcf* results in the highest performance gain, at 30% over a no-prefetching baseline. All of the benchmarks with a high rate of dependent cache misses show performance improvements with an EMC. These applications also generally observe performance degradations when prefetching is employed.

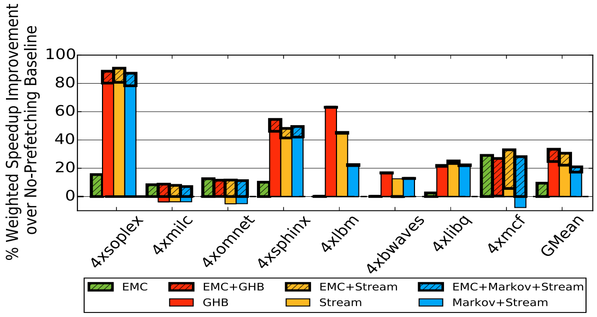


Figure 13: Quad-Core performance for homogeneous workloads.

6.2. Eight-Core Evaluation

We demonstrate the scalability of the EMC system. Figure 14 shows the performance benefit of using the EMC in an eight-core system. We evaluate both the single memory controller configuration (1MC, the first four bars in each workload) and the dual memory controller configuration (2MC, the second four bars in each workload).

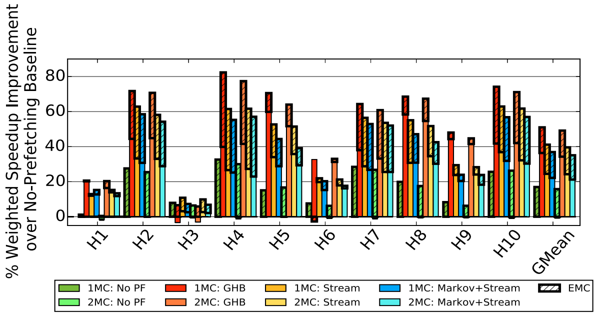


Figure 14: Eight-Core performance for workloads H1-H10.

Overall, the performance benefit of the EMC is slightly higher in the eight-core case than the quad-core case, due to a more heavily contended memory system. On the single memory controller configuration, EMC gains 17%, 13%, 14%, and 13% over the no-prefetching, GHB, stream, and Markov+stream prefetchers respectively in Figure 14. The dual memory controller baseline system shows a slight (-.8%) performance degradation over the single memory controller system, and EMC gains slightly less on average over each baseline (16%, 14%, 11%, 12% respectively) than the single memory controller, due to the overhead of communication between the EMCs. We do not observe a significant performance degradation by using two EMCs in the system.

6.3. Performance Analysis

To examine the reasons behind the performance benefit of the EMC we contrast workload H1 (1% performance gain) and H4 (33% performance gain). While we observe no single indicator for the performance improvement that the EMC provides, we identify three statistics that correlate to increased performance. First, we show the percentage of total cache misses that the EMC generates in Figure 15. As H1 and H4 are both memory intensive workloads, the EMC generating a larger percentage of the total cache misses indicates that its latency reduction features result in a larger impact on workload performance. The EMC generates about 10% of all of the cache misses in H1 and 22% of the misses in H4.[1]

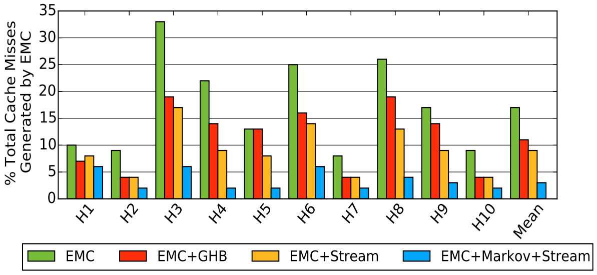


Figure 15: Fraction of all LLC misses generated by the EMC.

Second, we expect a reduction in DRAM contention for requests issued by the EMC. As requests are generated and issued to memory faster than in the baseline, a request can reach an open DRAM row before the row can be closed by a competing request from a different core. This results in a reduction in row-buffer conflicts. There are two different scenarios where this occurs. First, the EMC can issue a dependent request that hits in the same row-buffer as the original request. Second, multiple dependent requests to the same row-buffer are issued together and can coalesce into a batch. We observe that the first scenario occurs about 15% of the time while the second scenario is more common, occurring about 85% of the time on average.

Figure 16 shows the change in row-buffer conflict rate over the no-prefetching baseline for H1-H10. This statistic correlates to how much latency reduction the EMC achieves, as the latency for a row-buffer conflict is much higher than the latency of a row-buffer hit. The reduction in H1, less than 1%, is much smaller than the 19% reduction in H4.

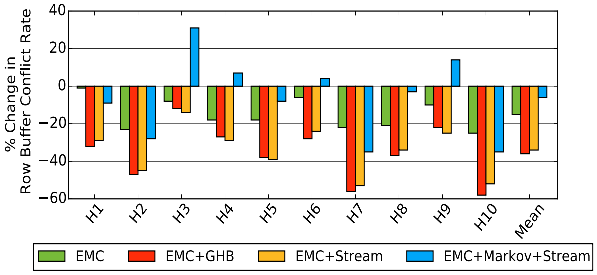


Figure 16: Change in row-buffer conflict rate with the EMC over a no-prefetching baseline.

Between these two factors, the fraction of total cache misses generated by the EMC and the reduction in row-buffer conflicts, it is clear that the EMC has a much smaller impact on H1 than on H4. One other factor is also important to note. The EMC exploits temporal locality in the memory access stream using a small data cache (Section 4.1.3). If the dependence chain executing at the EMC contains a load to data that has recently entered the chip, this will result in a very short-latency EMC data cache hit. Otherwise, the load may have to access the LLC (if a hit is predicted by the EMC miss predictor). Figure 17 shows that H1 has a much smaller hit rate in the EMC cache than H4.

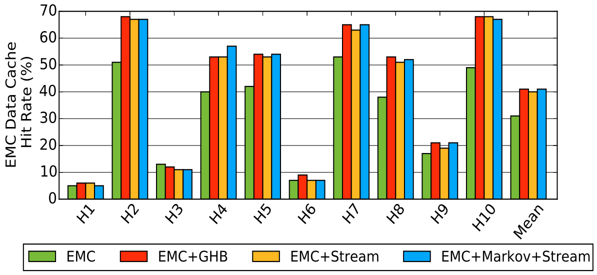


Figure 17: Hit rate at the EMC data cache.

These three statistics (the fraction of total cache misses generated by the EMC, the reduction in row-buffer conflict rate, and the EMC data cache hit rate) demonstrate why the performance gain in H4 is larger than the gain in H1.

The net result of the EMC is a raw latency difference for cache misses that are generated by the EMC and cache misses that are generated by the core. This is shown in Figure 18. Latency is given in cycles observed by the miss before dependent operations can be executed and is inclusive of accessing the LLC, interconnect, and DRAM. We find that a cache miss generated by the EMC observes a 20% lower average latency than a cache miss generated by the core.

[1] The Markov + Stream PF configuration generates 25% more memory requests than any other configuration on average, diminishing the impact of the EMC in Figure 15. This additional bandwidth consumption is also one reason that the Markov + Stream configuration results in lower relative performance when compared to the other prefetchers.

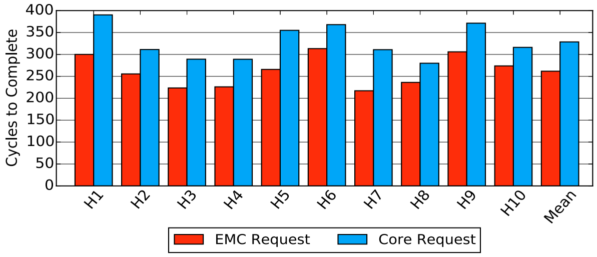


Figure 18: Latency observed by an LLC miss generated by the EMC vs. an LLC miss generated by the core for H1-H10.

The critical path of executing a dependent cache miss includes three areas where the EMC saves latency. First, in the baseline, the source cache miss is required to go through the fill path back to the core before dependent operations are executed. Second, the dependent cache miss must go through the on-chip cache hierarchy and interconnect before it can be sent to the memory controller. Third, the request must be selected by the memory controller to be issued to DRAM. We attribute the latency savings of EMC requests in Figure 18 to these three sources: bypassing the interconnect back to the core, bypassing cache accesses, and reduced contention at the memory controller. Figure 19 shows the average number of cycles saved by these three factors. We conclude that a large fraction of the savings come from reduced DRAM contention in many workloads, but the other two factors are also significant and sometimes dominant.

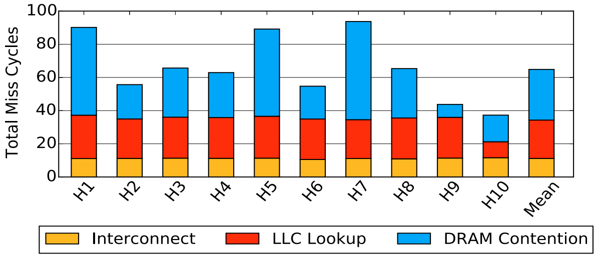


Figure 19: Average cycles saved by the EMC on a request.

As we attribute a large fraction of the latency reduction to decreased DRAM contention, we demonstrate that the performance gain of the EMC cannot simply be obtained by increasing memory banks and bandwidth. Figure 20 shows the average sensitivity of H1-H10 to different DRAM systems, from 1 channel with 1 rank to 4 channels with 4 ranks per channel (scaling memory queue size commensurately).

For the 1 channel and 2 channel cases (up to 2 channels 4 ranks), the performance benefit of the EMC relative to the no EMC baseline increases as the number of banks increases. These configurations have highly contended DRAM systems which gives the EMC the opportunity to reduce memory access latency for dependent cache misses. At 2 channels 4 ranks and with the 4 channel configurations, the large amount of memory bandwidth causes some reduction in the benefit of the EMC. However, as H1-H10 are very memory intensive workloads, we observe steadily increasing performance and

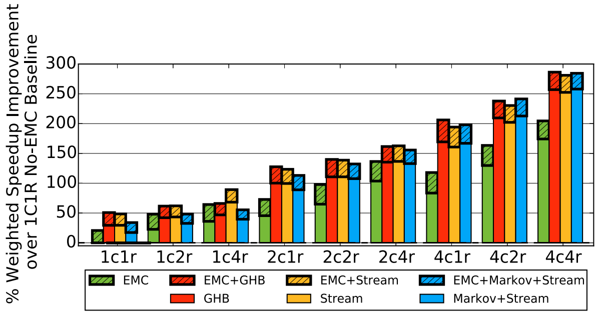


Figure 20: Performance sensitivity to varying memory channels and ranks over a 1 channel 1 rank (1C1R) baseline.

high bandwidth utilization through the 4 channel/4 rank configuration, particularly for the systems where prefetching is enabled. Even at 4 channels and 4 ranks, the EMC provides an 11% performance gain over the baseline.

6.4. Prefetching and the EMC

We analyze the interaction between the EMC and prefetching when they are employed together. The impact of prefetching on the EMC is shown by the fraction of EMC-generated cache misses that are also covered by prefetching. This is illustrated in Figure 21.

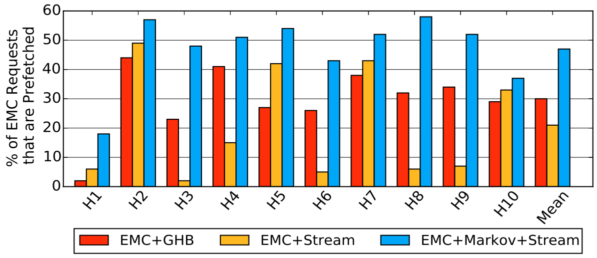


Figure 21: Percentage of cache misses generated by the EMC without prefetching that are converted into a cache hit with a prefetcher.

On average, the GHB/stream/Markov+stream prefetchers cover some fraction of the cache misses that the EMC generates. However, this fraction is relatively small: 30%, 21%, 48% of the memory requests that the EMC issued in the no-prefetching case are covered by the prefetchers, respectively. For the majority of EMC accesses, the EMC supplements the prefetcher by reducing the latency to access memory addresses that the prefetcher cannot predict ahead of time.

6.5. Enhanced Memory Controller Overhead

The interconnect overhead of the EMC consists of three main components: sending dependence chains and the source registers (live-ins) that these chains require to the EMC, and sending destination registers (live-outs) back to the core. Figure 22 shows the average length of the dependence chains that are executed at the EMC in uops.

The dependence chains executed at the EMC are short (under 10 uops, on average). These chains require an average of 6.4 live-ins. Transmitting the uops to the EMC results in a

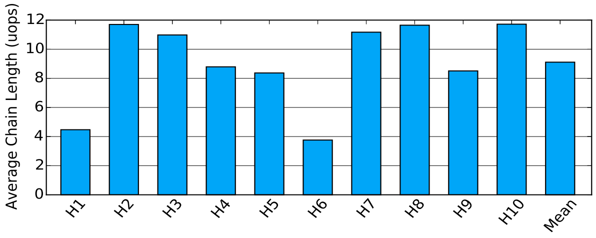


Figure 22: The average number of uops in each chain.

transfer of 1-2 cache lines of data on average. The average chain produces 8.8 live-outs, requiring a single cache line of data transfer back to the core. This relatively small amount of data transfer motivates why we do not see a performance loss due to the EMC. The interconnect overhead of the EMC for each executed chain is small and we migrate dependent operations only if dependent misses are likely. Overall, these messages result in a 33% average increase in data ring messages and a 7% increase in control ring requests for H1-H10. EMC requests are 25% of all data messages and 5% of all control ring messages. Due to the EMC, we observe a slight (4%) increase in LLC latency.

6.6. Energy and Area Evaluation

The energy results for the quad-core workloads are shown in Figure 23 and Figure 24 respectively. Both figures present the cumulative results for the energy consumption of the chip and DRAM as a percentage difference in energy consumption from the no-EMC, no-prefetching baseline.

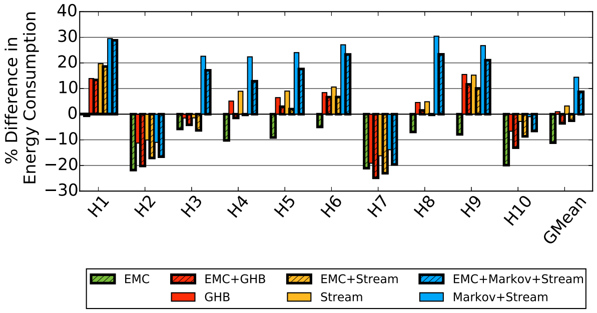


Figure 23: Energy consumption for workloads H1-H10.

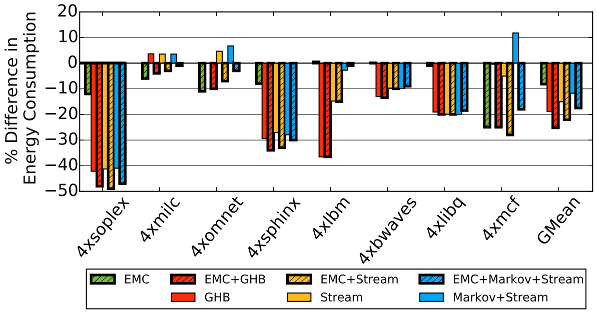


Figure 24: Energy consumption for homogeneous workloads.

Overall, we observe that the EMC reduces energy consumption on average by 11% for heterogeneous workloads H1-H10 and by 9% for the homogeneous workloads. This is due to two factors: a reduction in static energy consumption (as the performance improvement caused by the EMC decreases the total execution time of a workload), and dynamic energy savings due to the reduced row-buffer conflict rate.

In the prefetching configurations, all three of the prefetchers cause an increase in average energy consumption, particularly the Markov+stream configuration. This is due to inaccurate prefetches, which occur despite the fact that our baseline throttles inaccurate prefetchers [57]. In Figure 23, the GHB, stream, Markov+stream systems increase memory traffic by 18%, 20% and 52% respectively while the EMC increases traffic by only 8%. Similarly, in Figure 24 the prefetchers increase traffic by 12%, 8% and 45% respectively while the EMC increases traffic by only 3%. Analogous to the performance results, the systems with the EMC and prefetching combined result in lower energy consumption than systems with prefetching alone.

We estimate the entire area overhead of the EMC to be $2.2mm^2$ (including 5.9KB of additional storage), roughly 2% of total chip area. Over half of this additional area is due to the 4kB EMC cache. The small out-of-order engine constitutes 8% of the additional area, while the two integer ALUs make up 5%. McPAT estimates the area of a full out-of-order core as $21.2mm^2$, so the EMC is 10.4% of a full core. We implement the minimum functionality at the EMC to execute dependent cache miss chains. The out-of-order functionality is lightweight and the EMC does not contain large structures such as a floating-point pipeline or a front-end.

7. Conclusion

This paper makes a case for compute capable memory controllers. We introduce one mechanism for automatically offloading computation and mechanisms for communication between main processor cores and an EMC. We identify that dependent cache misses are latency critical operations. By transparently executing these dependent operations at the EMC instead of the core we observe a 20% reduction in effective memory access latency for these requests. This results in a 13% performance gain over a Global History Buffer prefetcher, the highest performing prefetcher in our evaluation. Future techniques can be built upon our framework that can use the EMC in different ways to exploit and enhance its capabilities. We believe that as memory continues to be an increasingly important bottleneck in future data-intensive workloads and systems, enhancing the memory controller and using it as an accelerator to improve memory access latency and efficiency will become increasingly important.

Acknowledgments

We wish to thank the anonymous reviewers, Carlos Villavieja, and Rustam Miftakhutdinov for valuable suggestions and feedback. We thank the other members of the HPS Research

Group for contributing to our working environment. We also wish to thank Intel, Oracle, and Microsoft for their generous financial support. Onur Mutlu acknowledges support from Google, Intel, and Seagate.

References

[1] J. Ahn *et al.*, "A Scalable Processing-in-memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.

[2] J. Ahn *et al.*, "PIM-Enabled Instructions: A Low-overhead, Locality-aware Processing-in-Memory Architecture," in *ISCA*, 2015.

[3] T. Alexander and G. Kedem, "Distributed Prefetch-Buffer/Cache Design for High Performance Memory Systems," in *HPCA*, 1996.

[4] M. Annavaram, J. M. Patel, and E. S. Davidson, "Data Prefetching by Dependence Graph Precomputation," in *ISCA*, 2001.

[5] M. Awasthi *et al.*, "Handling the Problems and Opportunities Posed by Multiple On-chip Memory Controllers," in *PACT*, 2010.

[6] J. Baer and T. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," in *Supercomputing*, 1991.

[7] J. Carter *et al.*, "Impulse: Building a Smarter Memory Controller," in *HPCA*, 1999.

[8] M. J. Charney and A. P. Reeves, "Generalized Correlation-Based Hardware Prefetching," Cornell Univ., Tech. Rep. EE-CEG-95-1, 1995.

[9] J. D. Collins *et al.*, "Dynamic Speculative Precomputation," in *MICRO*, 2001.

[10] J. D. Collins *et al.*, "Speculative Precomputation: Long-Range Prefetching of Delinquent Loads," in *ISCA*, 2001.

[11] R. Cooksey, S. Jourdan, and D. Grunwald, "A Stateless, Content-Directed Data Prefetching Mechanism," in *ASPLOS*, 2002.

[12] R. Das *et al.*, "Application-Aware Prioritization Mechanisms for On-Chip Networks," in *MICRO*, 2009.

[13] P. Dlugosch *et al.*, "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing," in *TPDS*, 2014.

[14] J. Dundas and T. Mudge, "Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss," in *ICS*, 1997.

[15] E. Ebrahimi *et al.*, "Coordinated control of multiple prefetchers in multi-core systems," in *MICRO*, 2009.

[16] E. Ebrahimi *et al.*, "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems," in *ASPLOS*, 2010.

[17] E. Ebrahimi *et al.*, "Parallel Application Memory Scheduling," in *MICRO*, 2011.

[18] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems," in *HPCA*, 2009.

[19] D. G. Elliott, W. M. Snelgrove, and M. Stumm, "Computational RAM: A Memory-SIMD hybrid and its application to DSP," in *CICC*, 1992.

[20] J. D. Gindele, "Buffer Block Prefetching Method," *IBM Technical Disclosure Bulletin*, 1977.

[21] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: The Terasys massively parallel PIM array," *IEEE Computer*, 1995.

[22] C. J. Hughes and S. Adve, "Memory-Side Prefetching for Linked Data Structures," in *Journal of Parallel and Distributed Computing*, 2001.

[23] "Intel Transactional Synchronization Extensions," http://software.intel.com/sites/default/files/blog/393551/sf12-arcs004-100.pdf, 2012.

[24] "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.

[25] D. Joseph and D. Grunwald, "Prefetching using Markov Predictors," in *ISCA*, 1997.

[26] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," in *ISCA*, 1990.

[27] Y. Kim *et al.*, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, 2010.

[28] P. M. Kogge, "EXECUBE-A New Architecture for Scaleable MPPs," in *ICPP*, 1994.

[29] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-Block Prediction and Dead-Block Correlating Prefetchers," in *ISCA*, 2001.

[30] C. J. Lee *et al.*, "Prefetch-Aware DRAM Controllers," in *MICRO*, 2008.

[31] C. J. Lee *et al.*, "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems," HPS Technical Report, Tech. Rep., 2010.

[32] D. Lee *et al.*, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *ACM TACO*, 2016.

[33] S. Li *et al.*, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO*, 2009.

[34] C.-K. Luk, "Tolerating Memory Latency through Software-Controlled Pre-execution in Simultaneous Multithreading Processors," in *ISCA*, 2001.

[35] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *USENIX Security*, 2007.

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

[37] N. Muralimanohar and R. Balasubramonian, "CACTI 6.0: A Tool to Model Large Caches," in *HP Laboratories, Tech. Rep. HPL-2009-85*, 2009.

[38] O. Mutlu *et al.*, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," in *HPCA*, 2003.

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

[40] O. Mutlu, H. Kim, and Y. N. Patt, "Techniques for Efficient Processing in Runahead Execution Engines," in *ISCA*, 2005.

[41] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.

[42] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.

[43] K. J. Nesbit and J. E. Smith, "Data Cache Prefetching Using a Global History Buffer," in *HPCA*, 2004.

[44] D. Patterson *et al.*, "A Case for Intelligent RAM," in *IEEE Micro*, 1997.

[45] J. T. Pawlowski, "Hybrid Memory Cube (HMC)," in *Hot Chips*, 2011.

[46] D. G. Perez, G. Mouchard, and O. Temam, "MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms," in *MICRO*, 2004.

[47] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," in *MICRO*, 2012.

[48] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based Prefetching for Linked Data Structures," in *ASPLOS*, 1998.

[49] A. Roth and G. S. Sohi, "Effective Jump-Pointer Prefetching for Linked Data Structures," in *ISCA*, 1999.

[50] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.

[51] V. Seshadri *et al.*, "Fast Bulk Bitwise AND and OR in DRAM," *IEEE CAL*, 2015.

[52] V. Seshadri *et al.*, "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses," in *MICRO*, 2015.

[53] D. E. Shaw *et al.*, "The NON-VON database machine: A brief overview," *IEEE Database Eng. Bull.*, 1981.

[54] T. Sherwood *et al.*, "Automatically Characterizing Large Scale Program Behavior," in *ASPLOS*, 2002.

[55] Y. Solihin, J. Lee, and J. Torrellas, "Using a User-Level Memory Thread for Correlation Prefetching," in *ISCA*, 2002.

[56] S. Somogyi *et al.*, "Spatial Memory Streaming," in *ISCA*, 2006.

[57] S. Srinath *et al.*, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," in *HPCA*, 2007.

[58] S. T. Srinivasan *et al.*, "Continual Flow Pipelines," in *ASPLOS*, 2004.

[59] H. S. Stone, "A Logic-in-Memory Computer," *IEEE TC*, 1970.

[60] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream Processors: Improving both Performance and Fault Tolerance," in *ASPLOS*, 2000.

[61] J. M. Tendler *et al.*, "POWER4 System Microarchitecture," *IBM Technical White Paper*, Oct. 2001.

[62] S. Yehia, J.-F. Collard, and O. Temam, "Load Squared: Adding Logic Close to Memory to Reduce the Latency of Indirect Loads with High Miss Ratios," in *MEDEA*, 2004.

[63] D. Zhang *et al.*, "TOP-PIM: Throughput-oriented Programmable Processing in Memory," in *HPDC*, 2014.

[64] C. Zilles and G. Sohi, "Execution-Based Prediction using Speculative Slices," in *ISCA*, 2001.