

**Reducing Memory Access Latency via an
Enhanced (Compute Capable) Memory Controller**

Milad Hashemi Khubaib Eiman Ebrahimi Onur Mutlu Yale N. Patt



**High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240**

**TR-HPS-2015-001
September 2015**

This page is intentionally left blank.

Reducing Memory Access Latency via an Enhanced (Compute Capable) Memory Controller

Milad Hashemi¹, Khubaib¹, Eiman Ebrahimi², Onur Mutlu³ and Yale N. Patt¹

¹Department of Electrical and Computer Engineering, The University of Texas at Austin
²NVIDIA

³Department of Electrical and Computer Engineering, Carnegie Mellon University
¹{miladh, khubaib, patt}@hps.utexas.edu, ²eebrahimi@nvidia.com, ³onur@cmu.edu

September 30, 2015

Abstract

Processor cores are seeing an increase in effective cache miss latency as the number of cores in a multi-core chip increases, and on-chip contention correspondingly increases. This paper identifies an important subset of latency-critical cache misses: those that will result in a cache miss but are dependent on a prior cache miss. We propose accelerating the execution of these cache misses in a manner that is transparent to the programmer, by adding compute-capability to the memory controller. Our new enhanced memory controller executes the dependent cache misses as soon as the source data arrives from DRAM, bypassing on-chip contention and decreasing DRAM row buffer conflict rate. The result, on a set of memory intensive quad/eight core workloads, is an improvement in system performance of 15%/17% respectively and a 20% reduction in memory request latency.

1 Introduction

The large latency disparity between performing computation at the core and accessing data from off-chip memory is a key impediment to system performance. While raw DRAM access latency remains roughly constant [18], decreasing by only 26% over the last 12 years, the *effective* DRAM latency as seen by the core is rising. This problem is known as the “memory wall” [39, 38] and is due to two factors: high core frequency, and increasing levels of on-chip shared-resource contention in the multi-core era. Examples of this contention include: on-chip interconnect, shared cache, DRAM row-buffer, and DRAM bank contention. Due to these factors, DRAM accesses are a performance bottleneck, particularly for single threaded applications that have difficulty hiding long-latency operations with instruction-level parallelism due to limited reorder buffer size.

The impact of the memory wall on processor performance is magnified when a cache miss has dependent memory operations that will also result in a cache miss. These operations, dependent cache misses, form chains of long-latency operations that fill the reorder buffer and prevent the core from making forward progress. Exacerbating the problem, these dependent cache misses often have data-dependent addresses that are difficult to prefetch. Figure 1 shows the percentage of total last level cache (LLC) misses that are dependent on a prior LLC miss for the SPEC CPU2006 benchmark suite. We simulate an aggressive out-of-order processor with a 256 instruction reorder buffer and 1MB of last level cache. The benchmark suite is sorted in ascending memory intensity. The application with the highest fraction of dependent cache misses, mcf, also has the lowest performance across the entire benchmark suite, with an IPC of just 0.3.

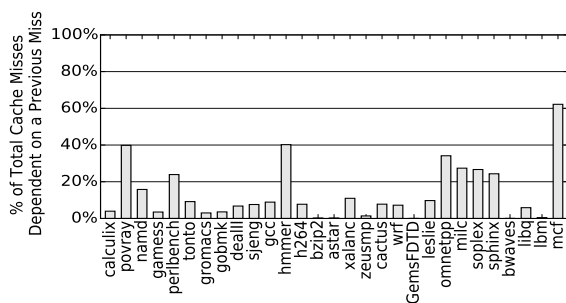


Figure 1: Percent of total Last Level Cache (LLC) misses dependent on a prior LLC miss, sorted by memory intensity.

We aim to reduce the latency of issuing these cache misses in a manner that is transparent to the programmer by adding limited compute capability to the memory controller. We modify the core to automatically identify the operations that are in the dependence chain of a cache miss. This dependence chain is then transparently migrated to the enhanced memory controller (EMC) so that data does not have to flow back to the core before the critical dependent cache miss is issued. Once the data from the cache miss arrives, the EMC executes the dependent operations. This results in two benefits. First, the EMC can generate cache misses faster than the core by bypassing on-chip contention, thus reducing the effective latency seen by memory operations. Second, our proposal increases the likelihood of a memory access hitting an open row buffer before the row can be closed by a competing request to a different row in the same DRAM bank from a different core. Thus, the enhanced memory controller can decrease the chance of a row-buffer conflict and increase the row buffer hit rate and system performance.

We make the following contributions in this paper:

- We explore partitioning code between the core and the memory controller. We propose a mechanism to automatically identify and migrate the dependence chain of a cache miss to a compute capable, enhanced memory controller (EMC).

We propose that these operations that are dependent on a cache miss can 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 2a shows one dynamic instance where there are a small number of simple integer operations between the source and dependent miss. We find that this trend holds over the memory intensive applications of SPEC06. Figure 2b 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 enhanced memory controller (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.

We therefore explore mechanisms to tailor the memory controller to execute dependent chains of operations such as those listed in Figure 2a. The added compute capability is described in detail in Section 3.1. Since the instructions have already been fetched and decoded at the core and are sitting in the reorder buffer, 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 hardware. This process is described in Section 3.2. The chain of decoded uops is then sent to the EMC.

Once the cache line arrives from DRAM for the original source miss the chain of dependent uops are executed and the live-outs are sent back to the core. We discuss the details of execution at the EMC in Section 3.3.

3 Mechanism

A quad-core chip multiprocessor (CMP) that uses our proposed enhanced memory controller is shown in Figure 3a. 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 [13]. Our proposal adds two pieces of hardware to the processor: a dependence chain-generation unit at each of the cores and limited compute capability at the memory controller. We first focus on the compute hardware that we add to the memory controller.

3.1 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 3.1.1). For the back-end, we use 2 ALUs and provide a minimal set of caching and virtual address translation capabilities (Section 3.1.2). Figure 3b provides a high level view of the added compute microarchitecture.

3.1.1 Front-End

The front-end of the EMC consists of multiple small uop buffers. Each of these buffers supports executing a single dependence chain, and can hold up to 16 uops (based on Figure 2b). With

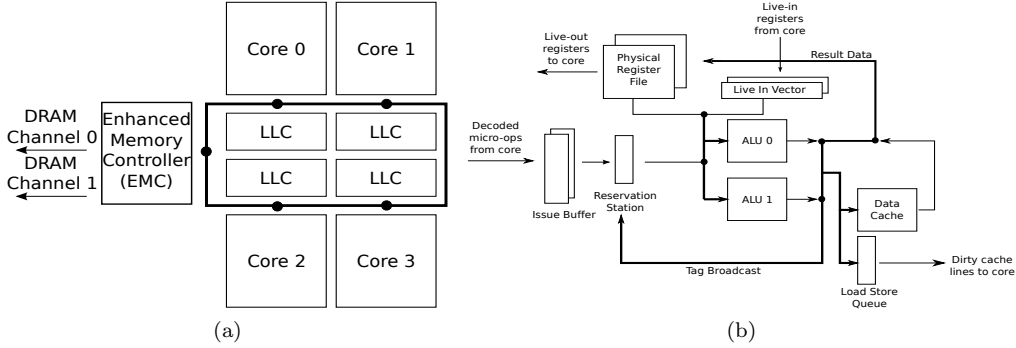


Figure 3: (a) A high level view of a quad-core CMP with an Enhanced Memory Controller. Each core has a ring stop, denoted by a dot, that is also connected to a slice of the shared last level cache. (b) The microarchitecture of the EMC. Boldfaced lines indicate a shared bus.

multiple buffers, the EMC can be shared between the cores of a multi-core processor. Ready uops are issued in a round-robin fashion out of each of these buffers and into the reservation stations when reservation station entries are available. The front-end of the EMC consists only of this buffer, it does not contain any fetch, decode, or register rename hardware. The chains of operations are renamed for the EMC using the out of order capabilities of the core (Section 3.2).

3.1.2 Back-End

As the EMC is targeting pointer-arithmetic, 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 another cache miss. The core is creating a *filtered* chain of operations for the EMC to execute (Section 3.2), only the operations that are required to generating the address for the dependent cache miss are included in the uop chain.

These filtered dependence chains are executed on a 2-wide back-end. To achieve comparable levels of memory level parallelism as the base out-of-order core, which our exploration shows is important for many memory-intensive workloads, the EMC has the capability to issue and execute uops out-of-order. This requires the back-end to support out-of-order issue and wakeup with a small reservation station (8-entries) and common data bus (CDB). In Figure 3b the CDB is denoted by the result and tag broadcast buses. A small load/store queue is maintained at the EMC to be able to execute memory operations out of order. We support executing stores at the EMC due to how common register spills/fills are in x86.

Each of the issue buffers in the front-end is also 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), operations arrive at the enhanced memory controller correctly renamed to use the physical registers of the EMC.

Operations are not retired at the EMC, only executed. Retirement state is maintained at the ROB of the home core and physical register data is transmitted back to the core for in-order retirement. These operations are not re-executed at the core. Thus, a portion of the operations in the reorder buffer are executed at the core, while others are executed at the EMC. Figure 4 provides a high-level view of partitioning the instruction stream between the EMC and the core with a simple sequence of 7 uops from *milc*, a memory intensive SPEC06 application.

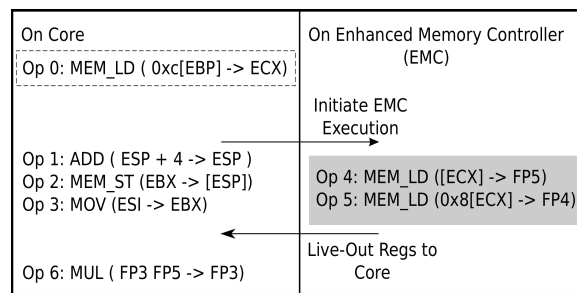


Figure 4: A sequence of 7 micro-ops from *milc*. Operation 0 results in a cache miss and is surrounded by a dashed box. The dependent cache misses to be executed at the memory controller are shaded gray.

In Figure 4, Op 0 results in a cache miss. Ops 1, 2, and 3 are independent of the result of Op 0 and execute during the period of time that Op 0 is waiting for data from memory. Ops 4 and 5 are dependent on Op 0 and will result in a cache miss when issued.

The core transmits these two uops to execute at the EMC instead of the core. When EMC execution completes, FP4 and FP5 are returned to the core so that execution can continue. Section 3.2 describes in detail the process of identifying and generating the chain of dependent operations to be executed at the enhanced memory controller.

3.1.3 Caches

The EMC contains no instruction cache, but it does contain a small data cache that holds the most 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 line to track the cache lines that the EMC holds.

3.1.4 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

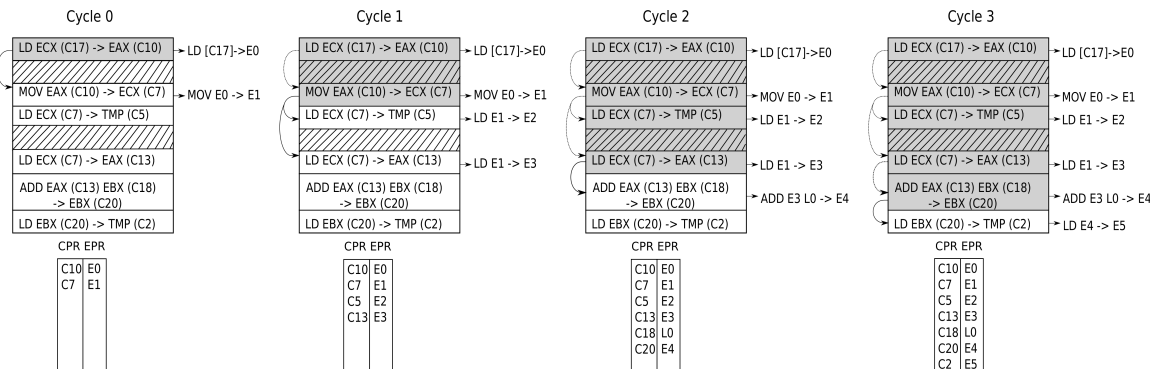


Figure 5: Chain generation using the chain of micro-ops from Figure 2a over four cycles. Two structures, the reorder buffer (ROB) and register remapping table (RRT) are shown. Physical registers are noted using parenthesis (CPR denotes Core Physical Register, EPR denotes EMC Physical Register). Processed operations are shaded after every cycle.

by the EMC for each core. The PTEs of the home 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 TLB entries resident at the EMC during the TLB shutdown process. Before a chain is executed, the core sends the EMC the PTE for the source miss if it is determined not to be resident at the EMC TLB. 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.

3.2 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 enhanced memory controller (EMC) executes. This allows the EMC to have no fetch, decode, or rename hardware, as shown in Figure 3b, thereby 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 2a to demonstrate the chain generation process, illustrated by Figure 5. This process takes a variable number of cycles based on dynamic chain length (4 cycles for Figure 5). 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 entire chain is transmitted to the EMC.

For each cycle we show two structures in Figure 5, the reorder buffer of the home core (ROB) and the register remapping table (RRT). The RRT is functionally similar to a register alias table and maintained for the proposed EMC at the home core. We only show a portion of the ROB and omit control operations by denoting them with stripes.

In Figure 5 the cycle 0 frame shows the source miss at the top of the ROB. It has been allocated core physical register number 10 (C10) to use as a destination register. However, since the EMC has many fewer physical registers than the large out-of-order core the operations in the chain have to be renamed to a smaller set of physical registers so that the memory controller can execute them. 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. This is equivalent to the maximum number of uops allowed in the chain. In the example, C10 is renamed to use the first physical register of the EMC (E0). This information is stored in the RRT, which is indexed by the physical register id and shown at the bottom of Figure 5.

Once the source miss has been renamed to EMC physical registers, chains of decoded uops are created by tracking dependencies through renamed physical registers. This process begins after a load is known to have missed in the last level cache and the ROB of the home core is full. The goal is to mark uops that would be ready to execute when the load 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. By pseudo waking up the uop it does not execute or commit the uop, it simply broadcasts its destination tag on the CDB. A variable number of uops are broadcast every cycle based on uop and functional unit availability, up to the back-end width of the home core.

In the example, there is only a single ready uop to broadcast in Cycle 0. The destination register of the source load (C10) is broadcast on the CDB. This wakes up the second operation in the chain, which is a MOV instruction that uses C10 as a source register. Before this operation can be included in the chain, the core must check if the EMC has the ability to execute the operation. As the MOV is an integer operation, it is included in the chain. It reads the remapped register id from the RRT for C10, and uses E0 as its source register at the EMC. The destination register (C7) is renamed to E1.

Operations continue to pseudo “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 C7 on the CDB. The result of this operation is shown in Cycle 1, two loads are woken up. Their destinations, C5 and C13 respectively, are renamed to E2 and E3 and written into the RRT.

In cycle 2, C5 and C13 are broadcast on the CDB. Broadcasting C13 causes the ADD operation to pseudo-wake up. However, the ADD has a second source register, C18. If the result of C18 has already been computed, or is ready, the ADD can be included in the chain. Otherwise, the ADD will not broadcast the tag of its destination register, C20, on the CDB. In this case, C18 is ready.

The value is read out of the core’s physical register file and packed into a live-in source vector, which will be sent to the EMC along with the chain. C18 is renamed to the first element in the source vector (L0). Thus, the ADD instruction uses two sources E3 and L0, and writes its result into E4.

In cycle 3, the ADD broadcasts C20 on the CDB and the final load can be added to the chain. Once the process has completed, a filtered portion of the execution window has been assembled for the EMC to execute. These uops are read out of the ROB and sent to the EMC for execution along with the live-in vector.

Algorithm 1 summarizes the algorithm for dynamically generating a filtered chain of dependent uops.

ALGORITHM 1: Dependence chain generation.

CPR denotes Core Physical Register.

EPR denotes EMC Physical Register.

RRT denotes Register Remapping Table.

```

Process the source uop at ROB full stall
Read source miss uop from ROB.
Allocate EPR for destination CPR of uop.
Update RRT.
Broadcast destination CPR tag on CDB.
for each dependent pseudo-woken up uop do
  if Total uops in Chain < MAXLENGTH
  and uop Type Allowed then
    Prepare the dependent uop to be sent to the EMC
    for each source operand do
      if source ready then
        Read data from physical register file.
        Pack data into live-in vector.
      else
        EPR = RRT[CPR]
      end if
    end for
    Allocate EPR for destination CPR of uop.
    Update RRT.
    Read uop from ROB to include in chain.
    Broadcast destination CPR tag of uop on CDB.
  end if
end for
Send filtered chain of uops and live-in vector to the EMC

```

3.3 EMC Execution

To start execution, the enhanced memory controller (EMC) takes two inputs: a source vector of live-in registers and an executable chain of operations, as described above in Section 3.2. The EMC also does not commit any 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 already sent a branch predicted stream, so it evaluates each condition and determines if the chain that it was sent to execute contains the correct path of execution. If the EMC realizes it is on the wrong-path, execution is stopped and the core is notified of the mis-predicted branch. We send control operations along with computation to the EMC so that the EMC does not generate wrong path memory requests if it is on the wrong path.

For memory operations, a load first queries the data cache, if it misses in the data cache it generates an LLC request. A store writes its value into the EMC LSQ. Loads and stores are retired in program order back at the home core. Every load or store executed at the EMC sends a request on the interconnect 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, for example there is a store to the same address as a load executed at the EMC in program order at the core, 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 [12] provide many similar guarantees for load/store ordering. Remote execution at the EMC is simpler than a transaction, as there is no chance for a conflict or rollback due to simultaneous execution. Leveraging these transactional memory capabilities provides a path towards fine-grained remote-code execution, as we propose with the EMC.

Once each 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 CDB, and execution on the main core continues.

As the home core maintains all instruction state for in-order retirement, any bad-event (branch misprediction, EMC TLB-miss, EMC exception) causes the home core to re-issue and execute the entire chain normally.

3.4 EMC Miss Predictor

The EMC has the ability to predict if any given 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 cache, 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 [29]. 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.

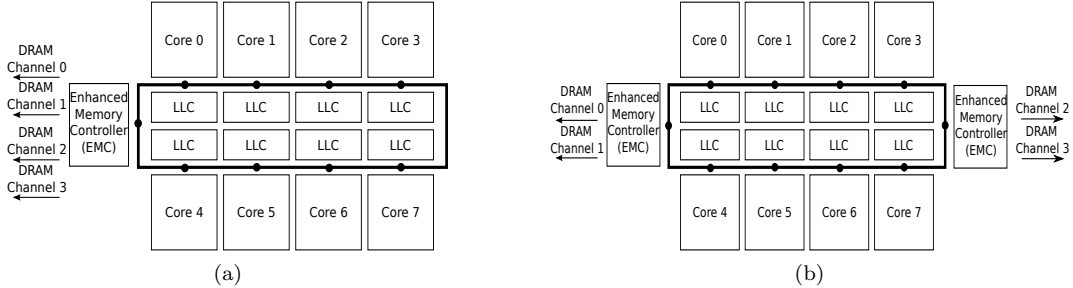


Figure 6: Eight-core configurations. (a) Single Memory Controller. (b) Dual Memory Controller.

If the counter is above a threshold the request is sent directly to memory. Note that a parallel request still has to be sent to the home core to populate a LSQ entry and to look-up the LLC in the case of a mis-predicted dirty line.

3.5 Micro-Op Cache

We observe that the chains of uops that are sent to the EMC are repetitive, and a small cache of chains can greatly reduce interconnect overhead. However, caching uops at the EMC produces additional complexity as the core must be certain that the cached chain is the same as the chain that it has scheduled for execution. This is a solvable problem, requiring additional hardware overhead, but we do not currently consider caching uop chains at the EMC for simplicity.

3.6 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, as shown in Figure 3a. 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 has generated a request to a channel located at a different enhanced memory controller) the EMC directly issues the request to the new memory controller without migrating execution of the chain. This cuts the core, a middle-man, out of the process (in the baseline the original request would have to travel back to the core and then on to the second memory controller). We evaluate this scenario with an eight-core CMP (Figure 6b) and compare the results to an eight-core CMP with a single memory controller (Figure 6a) in Section 5.2.

4 Methodology

We simulate three systems: a quad core system (Figure 3a) and two eight-core systems (Figure 6). The details of our system configurations are listed in Table 1. 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 a ring where a core can access the LLC slice at its own ring stop without getting onto the ring (using a bypass) to not overstate ring contention.

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 total. 8-Core: 8MB total.
Interconnect	2 Bi-Directional rings, control (8 bytes) and data (32 bytes). 1 cycle core to LLC slice bypass. 1 cycle latency between ring stops.
EMC Compute	2-wide issue. 8 Entry Reservation Stations. 32 Entry TLB per core. 64 Line 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. 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 [24]. 4-Core: 128 Entry Memory Queue. 8-Core: 256 Entry Memory Queue.
Prefetchers	Stream: 32 Streams, Distance 32, Degree 2. Markov: 1MB Correlation Table, 4 addresses per entry. GHB G/DC: 1k Entry Buffer, 12KB total size. All configurations: FDP [35], prefetch into Last Level Cache.
DRAM	DDR3[22], 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

We model three different prefetchers. A stream prefetcher (based on the stream prefetcher in the IBM POWER4 [37]), a Markov prefetcher [14], and a global-history-buffer (GHB) based global delta correlation (G/DC) prefetcher [25]. Prior work has shown a GHB prefetcher to outperform a large number of other prefetchers [28]. We find that the stream prefetcher always increases performance when used with a Markov prefetcher, and therefore employ them together.

The baseline memory controller uses a sophisticated scheduling algorithm, batch scheduling [24], and Feedback Directed Prefetching (FDP) [35] 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 6b), 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 SPEC06 benchmarks into three categories: high, medium, and low memory intensity by misses per thousand instructions (MPKI). The classification of each SPEC06 benchmark is listed in Table 2.

High (MPKI >10)	mcf, libquantum, bwaves, lbm, sphinx3, omnetpp, milc, soplex
Medium (MPKI >2)	zeusmp, cactusADM, leslie3d, GemsFDTD, 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

Using this table we randomly generate a set of 4-core workloads to evaluate. Each benchmark can only appear once in every workload combination. As the EMC is primarily intended to accelerate memory intensive applications, we focus on memory intensive workloads in our evaluation. We generate three sets of workloads, listed in Table 3.

The first set of 10 workloads contains 4 high memory intensity benchmarks. The second contains 5 workload combinations. In each, 2 benchmarks have a high memory intensity and 2 have a medium memory intensity. The third set of workloads also contains 5 workloads combinations. In each, 2 benchmarks have a high memory intensity and 2 have a low memory intensity. The eight-core workloads are two copies of the corresponding quad-core workload.

H1	bwaves+lbm+milc+omnetpp	M11	soplex+Gems+wrf+mcf
H2	soplex+omnetpp+bwaves+libq	M12	mcf+zeusmp+lbm+cactus
H3	sphinx3+mcf+omnetpp+milc	M13	Gems+wrf+mcf+omnetpp
H4	mcf+sphinx3+soplex+libq	M14	cactus+Gems+soplex+sphinx3
H5	lbm+mcf+libq+bwaves	M15	libq+leslie3d+wrf+lbm
H6	lbm+soplex+mcf+milc	L16	h264ref+lbm+omnetpp+povray
H7	bwaves+libq+sphinx3+omnetpp	L17	tonto+sphinx3+sjeng+mcf
H8	omnetpp+soplex+mcf+bwaves	L18	bzip2+namd+mcf+sphinx3
H9	lbm+mcf+libq+soplex	L19	omnetpp+soplex+namd+xalancbmk
H10	libq+bwaves+soplex+omnetpp	L20	soplex+mcf+bzip2+perlbench

Table 3: Workloads

We simulate these workloads using an in-house cycle-accurate x86 simulator. The simulator 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 has completed at least 50 million instructions from a representative SimPoint [31].

Chip energy is modeled using McPAT [19] and DRAM power is modeled using CACTI [23]. Static power of shared structures is dissipated until the completion of the entire workload. Dynamic counters stop updating upon 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 require a Register Remapping table (RRT) lookup, and each destination register requires a 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. Data and instruction transfer overhead to/from the EMC is taken into account via additional messages sent on the ring.

5 Results

To measure the performance of a multi-core system, we use weighted speedup [32] as a metric, defined below.

$$Wspeedup = \sum_{i=0}^{n-1} \frac{IPC_i^{shared}}{IPC_i^{alone}} \tag{1}$$

We first show the performance results of the quad-core system (Section 5.1) and then the eight-core system (Section 5.2).

5.1 Quad-Core Evaluation

The performance of the quad-core system, represented in terms of weighted speedup deltas over a non-prefetching baseline, across the workload combinations listed in Table 3 is shown in Figures 7a and 7b. The performance gain due to the EMC over each no-prefetching/prefetching configuration is illustrated as a hashed bar.

On the memory intensive workloads, our enhanced memory controller (EMC) improves performance on average by 15% over a non-prefetching baseline, by 10% over a baseline with stream prefetching, 13% over a baseline with a GHB prefetcher and 11% over a baseline with both a stream and Markov prefetcher. Workloads M11-M15/L16-L20 show a smaller performance gain of 5% over the no-prefetching baseline, 4% over the stream prefetcher and 3% over the GHB and stream+Markov prefetcher.

To demonstrate the scalability of the EMC system, we show eight-core results next in Section 5.2.

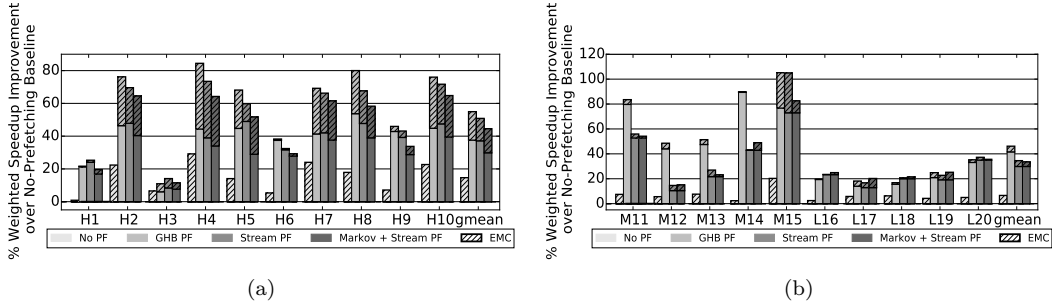


Figure 7: (a) Quad-Core performance increase relative to a no-prefetching baseline for workloads H1-H10. (b) Quad-Core performance increase relative to a no-prefetching baseline for workloads M11-L15 and L16-L20.

5.2 Eight-Core Evaluation

Figure 8a shows the performance benefit for 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).

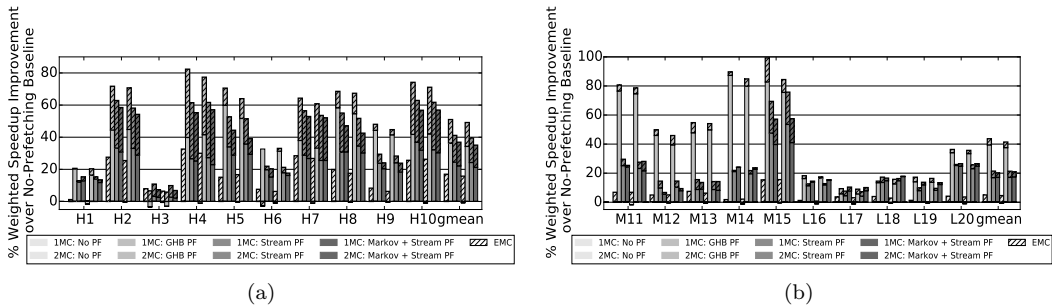


Figure 8: (a) Eight-Core performance relative to a no-prefetching baseline for workloads H1-H10. (b) Eight-Core performance relative to a no-prefetching baseline for workloads M11-M15 and L16-L20.

Overall, the performance benefit of the EMC is larger in the eight-core case than the quad-core case, due to a more heavily contested memory system. The single memory controller configuration gains 17%, 14%, 13%, and 13% over the no-prefetching, stream, GHB and stream+Markov prefetchers respectively in Figure 8a. The dual memory controller baseline system shows a slight (-.8%) performance degradation over the single memory controller system, and gains slightly less on average over each baseline (16%, 14%, 11%, 12% respectively) than the single memory con-

troller, due to the overhead of communication between the EMCs. We do not observe a significant performance degradation by using two enhanced memory controllers in the system.

For clarity, and ease of explanation (as the eight-core workloads are two of each quad-core workload) we explore the quad-core configuration in depth in Sections 5.3 through 5.6 to explain the benefits and drawbacks of the EMC.

5.3 Performance Analysis

To isolate the reasons behind the performance benefit of the EMC we compare several different statistics from workload 1, which results in a 1% performance gain, to workload 4, a 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 9a. As workloads 1 and 4 are both memory intensive workloads consisting of benchmarks with a high MPKI, 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 workload 1 and 22% of the misses in workload 4¹.

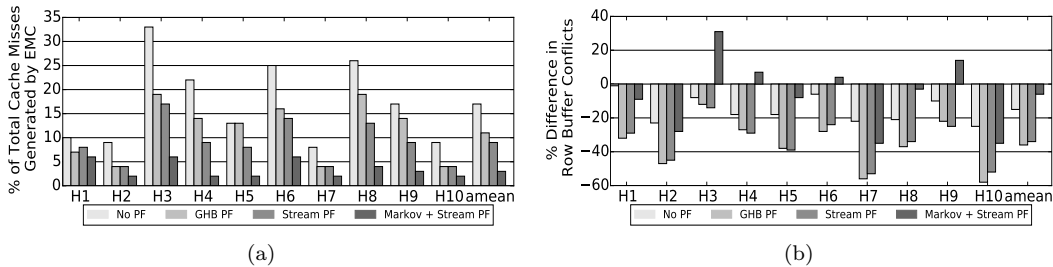


Figure 9: (a) The fraction of total cache misses generated by the EMC for Workloads H1 - H10 with and without prefetching. (b) The percent difference in row-buffer conflict rate over a no-prefetching baseline for workloads H1 - H10.

Additionally, we expect a reduction in row-buffer conflicts with the EMC, as requests are generated and issued to memory faster than in the baseline. Thus, a request can reach an open row before the row can be closed by a competing request. 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.

¹The Markov + Stream PF configuration generates 25% more memory requests than any other configuration on average, diminishing the impact of the EMC in Figure 9a and one reason for lower relative performance

Figure 9b shows the difference in row-buffer conflict reduction. This statistic correlates to how much total latency reduction the EMC is able to achieve, as the latency for a row-buffer conflict is much higher than the latency of a row-buffer hit. The reduction in workload 1, less than 1% is much smaller than the 19% reduction seen from workload 4.

Between these two factors, the percent 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 performance in workload 1 than workload 4. One other factor is also important to note, the data cache located at the EMC effectively converts a long-latency LLC lookup into a very short-latency cache hit. Figure 10a shows that Workload 1 has a much smaller hit rate in the data cache than Workload 4.

These three factors are major reasons behind why the performance gain in Workload 4 is much more significant than the performance gain in Workload 1.

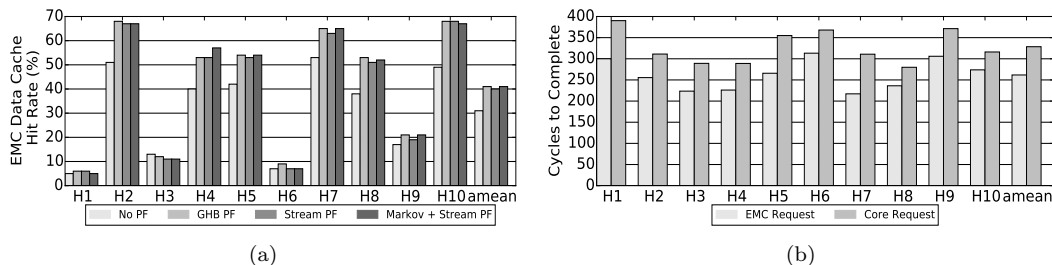


Figure 10: (a) The data cache hit rate at the EMC. (b) The latency observed by a cache miss generated by the EMC vs a cache miss generated by the core for Workloads 1-10 without prefetching.

We also show the raw latency difference for cache misses that are generated by the enhanced memory controller (EMC) and cache misses that are generated by the core in Figure 10b. 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. As such, a cache miss generated by the EMC observes a 20% lower latency than a cache miss generated by the core on average. Figure 11a illustrates why this is the case. In the baseline, a request issued by the core must access the global interconnect before querying the LLC (Steps #1/2). After a miss in the LLC, the request accesses the global interconnect again and is sent to the memory controller (Steps #3/4). After the DRAM access is complete, the data returns to the chip and accesses the interconnect again before it is installed in the cache hierarchy and used by the core (Steps #5,6,7,8).

With a compute capable memory controller, data does not have to return to the core before execution can begin (Steps #7/8). Additionally, the miss predictor at the EMC allows EMC requests to bypass the LLC, removing all global interconnect access from the critical path of the miss (Steps #1,2,3).

We attribute the latency difference between the EMC and the baseline in Figure 10b to these three sources: bypassing the interconnect back to the core, bypassing the LLC access, and a higher

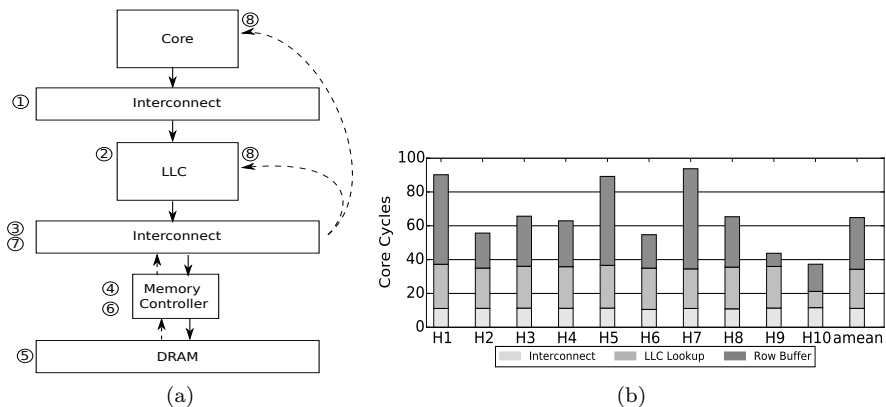


Figure 11: (a) Steps to uncover and service a cache miss. Solid lines denote the outgoing request and dashed lines denote data returning. (b) The average number of cycles saved by the EMC on each memory request.

percentage of row-buffer hits. The average number of cycles saved by these factors are shown in Figure 11b.

As we attribute much of the increase in performance to a reduction in row-buffer conflicts, we demonstrate that the performance gain of the EMC cannot simply be obtained by increasing memory banks and bandwidth. Figure 12 shows the sensitivity of the memory intensive workloads to different DRAM configurations, from 1 channel with 1 rank to 4 channels with 4 ranks per channel. 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. This is because more banks present the EMC with a greater opportunity to create a row-buffer hit. At 2 channels 4 ranks and 4 channels, the large amount of memory bandwidth causes some reduction in the benefit of the EMC. However, even at 4 channel/4 ranks, our proposal provides a 11% performance gain over the baseline.

5.4 Prefetching and the Enhanced Memory Controller

In this section we discuss the interaction between the EMC and prefetching when they are employed together. Figure 9a shows that the fraction of total cache misses that are generated by the EMC with prefetching is, on average, about 2/3 of the fraction of total cache misses generated without prefetching. However, the total number of memory requests is different between the prefetching and the non-prefetching case. This is because the prefetcher generates many memory requests, some requests are useful while others are useless. Thus, the impact of prefetching on the EMC is more accurately illustrated by considering how many fewer cache misses the EMC generates when prefetching is on versus when prefetching is off. This fraction is shown below in Figure 13.

On average, the Stream/GHB/Markov+Stream prefetchers can prefetch about 21%, 30%, 48%

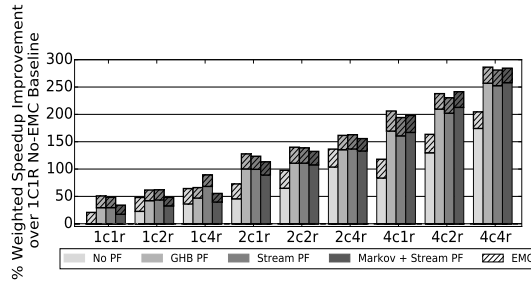


Figure 12: Performance sensitivity to varying memory channels and ranks over a 1 channel 1 rank (1C1R) baseline.

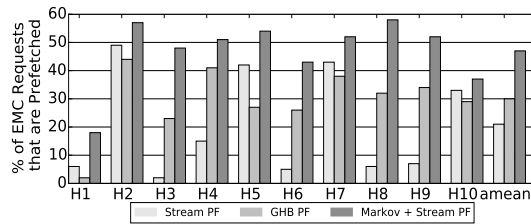


Figure 13: The percentage of cache misses generated by the EMC without prefetching that have been converted into a cache hit when employing a prefetcher.

of the requests that the EMC issued in the non-prefetching case respectively, thus converting what would have been a cache miss to a cache hit. This shows that prefetching does diminish the benefit of the EMC to some extent, but the EMC also supplements the prefetcher by reducing the latency to access memory addresses that the prefetcher can not predict ahead of time.

5.5 Enhanced Memory Controller Overhead

We now quantify the overhead of the enhanced memory controller (EMC). Overall, we observe an 33% average increase in data ring activity across Workloads H1-H10 while using the EMC. This overhead consists of two main components: shipping source registers (live-ins) and uops to the EMC and destination registers (live-outs) back from the EMC.

Figure 14a shows the average number of live-ins for each of the workloads in Figure 7a. On average, this results in less than a cache line of input data shipped to the EMC per executed chain, a relatively small amount of input data.

Figure 14b shows the average chain length in terms of uops. The chain length defines both the number of uops which must be sent to the EMC, and the number of registers that must be shipped back to the core. This is because we ship all physical registers back to the core as described in Section 3.3 and each uop produces a live-out/physical register.

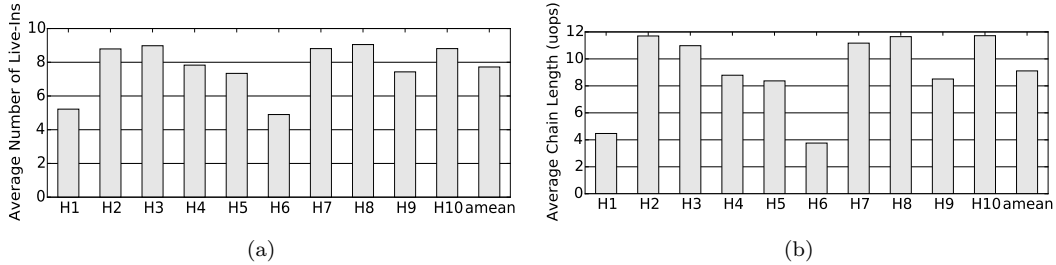


Figure 14: (a) The number of live-ins on average for each workload (H1-H10). (b) The average number of uops in each chain executed at the EMC for each workload (H1-H10).

Again, the destination registers that are shipped back to the home core result in roughly a cache line of data. Transmitting the micro-operations results in a transfer of 1-2 cache lines on average. 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 accelerate the issue and execution of integer dependent operations only if they exist.

5.6 Energy and Area Evaluation

The energy results for Workloads H1-H10 and M11-L20 are shown in Figure 15a and Figure 15b 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.

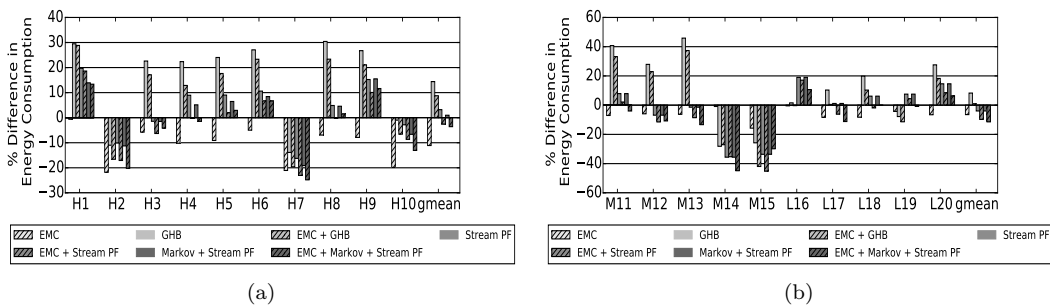


Figure 15: (a) Energy consumption difference relative to a no-prefetching baseline for workloads H1-H10. (b) Energy consumption difference relative to a no-prefetching baseline for workloads M11-L20.

Overall, we observe that the EMC is able to reduce energy consumption (Chip+DRAM) on average by about 11% for the memory intensive workload set and by about 5% for Workloads 11-20. We find 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 DRAM system.

Based on McPAT, the entire area overhead of the EMC is $5.0mm^2$, (including 7.5KB of additional storage) roughly 5% of total chip area. As McPAT estimates the area of an out-of-order core including I/D-Caches as $21.22mm^2$, and a typical design includes one memory controller for at least four cores, the area overhead/core is 6%. This additional area leads to a 6.5% increase in static power. The peak dynamic power of the chip is estimated to increase by 5.4%.

In the prefetching cases, Figure 15a illustrates the cost of prefetching. On average, all three of the prefetchers we evaluated cause an increase in energy consumption. This is due to inaccurate prefetches, which occur despite the fact that our baseline throttles inaccurate prefetchers (FDP). As in the performance results, we also observe that the prefetcher and EMC combined result in better energy efficiency than just using the prefetcher.

6 Related Work

To our knowledge, this is the first work that proposes adding compute capability to the memory controller to automatically reduce memory access latency for chains of dependent demand requests. Related prior work also attempts to reduce the latency seen by a cache miss using prefetcher improvements and/or moving computation closer to memory. Here we will briefly discuss these prior papers.

Advanced hardware prefetching techniques such as correlation prefetching [4, 14, 17, 34] aim to reduce average memory access latency by issuing requests for data that the processor is predicted to use ahead of the demand access stream. Other hardware prediction based mechanisms [5, 30] attempt to also predict future memory accesses by storing additional data regarding pointers and access patterns.

Another form of prefetching involves spawning speculative thread contexts [7], [42, 6] or other forms of precomputation [2, 20, 40, 36] to execute ahead of the demand access stream. These contexts normally execute filtered portions of the instruction stream. [33] 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 the available DRAM capacity to store the large correlation tables required for correlation prefetching.

Content-directed prefetching [8] does not require additional state to store pointers or additional execution contexts, but greedily prefetches using values that it believes to be addresses. This results in a large number of useless prefetches. [10] developed mechanisms to throttle inaccurate content-directed prefetchers.

Prior work has also considered enhancing the memory controller. [3] proposed enhancements to the memory controller that include address remapping and prefetching capabilities.

Memory-side prefetching moves the hardware that prefetches data from the chip closer to the main memory system [1, 11]. [41] track dependencies between two static instructions and then dynamically insert new instructions to execute them closer to memory. More generally, fabricating logic and memory on the same process has been proposed [16, 26] and recently revisited with Micron’s Hybrid Memory Cube [27]. To this end, industry is also pursuing different methods of integrating compute and memory controllers [9].

Our proposal differs from prior work in that we do not prefetch data, all of the requests sent by the EMC are demand requests, instead 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 requests that may be difficult to prefetch accurately without the negative effects of prefetching (inaccurate/untimely prefetch requests wasting bandwidth). As we are transferring execution, our proposal shares similarities with execution migration schemes [15, 21], which also move execution closer to data. However, execution migration has been primarily concerned with reducing the overhead of cache coherence in a shared memory multiprocessor. We focus on reducing memory access latency with very fine-grained dynamic migration.

7 Conclusion

We explore partitioning code between the core and the memory controller. This results in an approach to reduce the effective latency observed by memory operations by executing small chains of micro-ops at the memory controller instead of the core. These chains of operations are dynamically identified by the existing out-of-order unit once a cache miss is known to have occurred, and transparently migrated to an Enhanced Memory Controller for execution.

By executing the dependent operations at the EMC instead of the home core we observe a 15% average gain in weighted speedup and a 11% reduction in average energy consumption over a system without prefetching. Additionally, we observe a 13% performance gain over a GHB prefetcher, the highest performing prefetcher in our evaluation.

This paper makes a case for compute capable memory controllers. We introduce one mechanism for offloading computation and mechanisms for communication between main processor cores and an EMC. Other techniques can be built upon this framework that can exploit the proposed substrate in different ways to amortize its complexity. For example, while the current mechanism does not require programmer intervention, exposing the EMC to the expert programmer can further accelerate latency-critical, memory-bound code. As main memory latencies remain roughly constant architects must search for other avenues to reduce the effective latency seen by memory operations for applications that cannot hide memory access latency with parallelism.

References

- [1] T. Alexander and G. Kedem. Distributed prefetch-buffer/cache design for high performance memory systems. In *HPCA-2*, 1996.
- [2] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *ISCA-29*, 2001.
- [3] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *HPCA-5*, 1999.
- [4] M. J. Charney and A. P. Reeves. Generalized correlation-based hardware prefetching. Technical Report EE-CEG-95-1, Cornell Univ., 1995.
- [5] J. D. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *MICRO-35*, 2002.
- [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-X*, 2002.
- [9] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. 2014.
- [10] E. Ebrahimi, O. Mutlu, and Y. N. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *HPCA-15*, 2009.
- [11] C. J. Hughes and S. Adve. Memory-side prefetching for linked data structures. In *Journal of Parallel and Distributed Computing*, 2001.
- [12] Intel Transactional Synchronization Extensions. <http://software.intel.com/sites/default/files/blog/393551/sf12-arcs004-100.pdf>, 2012.
- [13] 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 2-3; Accessed 14-April-2014].
- [14] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA-24*, 1997.

- [15] O. Khan, M. Lis, S. Devadas, O. Khan, M. Lis, and S. Devadas. Em2: A scalable shared-memory multicore architecture. In *MIT CSAIL TR 2010-030*, 2010.
- [16] P. M. Kogge. Execube-a new architecture for scaleable mpps. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 01*.
- [17] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *ISCA-28*, 2001.
- [18] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu. Tiered-latency dram: A low latency and low cost dram architecture. In *HPCA-19*, 2013.
- [19] 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.
- [20] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *ISCA-28*, 2001.
- [21] P. Michaud. Exploiting the cache capacity of a single-chip multi-core processor with execution migration. In *Software, IEE Proceedings-*, pages 186–195, 2004.
- [22] MT41J512M4 DDR3 SDRAM Datasheet Rev. K Micron Technology, Apr. 2010. http://download.micron.com/pdf/datasheets/dram/ddr3/2Gb_DDR3_SDRAM.pdf, 2010.
- [23] N. Muralimanohar and R. Balasubramonian. Cacti 6.0: A tool to model large caches. In *HP Laboratories, Tech. Rep. HPL-2009-85*, 2009.
- [24] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA-35*, 2008.
- [25] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *HPCA-10*, 2004.
- [26] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent ram. In *IEEE Micro*, March 1997.
- [27] J. T. Pawlowski. Hybrid Memory Cube (HMC). In *Proceedings of Hot Chips*, volume 23, 2011.
- [28] D. G. Perez, G. Mouchard, and O. Temam. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *MICRO-37*, 2004.
- [29] 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-45*, 2012.

- [30] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *ISCA-26*, 1999.
- [31] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X*, 2002.
- [32] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *ASPLOS-IX*, 2000.
- [33] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *ISCA-29*, 2002.
- [34] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *ISCA-33*, 2006.
- [35] 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.
- [36] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: improving both performance and fault tolerance. In *ASPLOS-9*, 2000.
- [37] J. M. Tandler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Technical White Paper*, Oct. 2001.
- [38] M. V. Wilkes. The memory gap and the future of high performance memories, March 2001.
- [39] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. In *SIGARCH Comput. Archit. News*, March 1995.
- [40] C. Yang and A. R. Lebeck. Push vs. pull: Data movement for linked data structures. In *ICS-2000*, 2000.
- [41] 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. *MEDEA*, 2004.
- [42] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *ISCA-28*, 2001.