Lawrence Berkeley National Laboratory
LBL Publications

Title
Last Level Collective Hardware Prefetching for Data-Parallel Applications

Permalink
https://escholarship.org/uc/item/1tz16957

ISBN
9781538622933

Authors
Michelogiannakis, George
Shaf, John

Publication Date
2017-12-01

DOI
10.1109/hipc.2017.00018

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Last Level Collective Hardware Prefetching For Data-Parallel Applications

George Michelogiannakis and John Shalf
Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720
Email: {mihelog, jshalf}@lbl.gov

***Abstract*—With rapidly increasing parallelism, DRAM performance and power have surfaced as primary constraints from consumer electronics to high performance computing (HPC) for a variety of applications, including bulk-synchronous data-parallel applications which are key drivers for multi-core, with examples including image processing, climate modeling, physics simulation, gaming, face recognition, and many others. We present the last-level collective prefetcher (LLCP), a purely hardware last-level cache (LLC) prefetcher that exploits the highly correlated prefetch patterns of data-parallel algorithms that would otherwise not be recognized by a prefetcher that is oblivious to data parallelism. LLCP generates prefetches on behalf of multiple cores in memory address order to maximize DRAM efficiency and bandwidth, and can prefetch from multiple memory pages without expensive translations. Compared to well-established other prefetchers, LLCP improves execution time by 5.5% on average (10% maximum), increases DRAM bandwidth by 9% to 18%, decreases DRAM rank energy by 6%, produces 27% more timely prefetches, and increases coverage by 25% at minimum.**

I. Introduction

Technology improvements combined with power and clock frequency constraints create a drive towards greater parallelism in order to continue historical growth in compute performance [1], [2], [3], [4], [5], [6]. Continued performance growth intensifies stress on memory, but DRAM technology improvements are not projected to scale fast enough to meet future demands [7], [8], [9], [10], [11] due to daunting manufacturing and data movement constraints. Even today, numerous applications reach the memory bandwidth ceiling using just a fraction of cores available in a modern chip multiprocessor (CMP), such as 48 cores generating 300GB/s of memory bandwidth demand in the case of graphics [12]. To make things worse, projections state that chip pins that directly correlate to available bandwidth to off-chip memory increase by 10% every year, whereas processing capacity doubles every 18 months [13]. In addition, DRAM power consumption is already critical. For instance, a DDR4 DIMMs consume about 35pJ per bit, meaning that a system with only 0.2 bytes per FLOP memory bandwidth requires over 160mW of DRAM power [2]. Likewise, memories already require tens to hundreds of core cycles per memory access [7]. 3D stacked memory promises improvements in bandwidth and energy, but the underlying technology and associated power, latency, and bandwidth will continue to be challenged by performance scaling [14], and capacity and cost concerns necessitate that most of the memory in future large-scale systems will remain “conventional” DRAM [14].

Memory performance constraints are particularly limiting for bulk-synchronous data-parallel single program multiple data (SPMD) execution where all compute elements are employed in tandem to speed up a single kernel. In this work we focus on memory system optimizations for bulk-synchronous data-parallel SPMD execution for CMPs because they are at the core of a wide variety of critical and diverse applications from consumer-grade electronics to high performance computing (HPC). This family of workloads includes image processing, machine learning, physics simulation, climate modeling, and others [8]. In fact, data-parallel applications have been cited as the biggest drivers for multi-core because of their promise to take the most advantage of parallelism in the future [15]; applications in future multi-cores, even consumer and mobile, are expected to follow data-parallelism [15]. Because of the amount of data processed, data-parallel applications are particularly stressful to main memory. In fact, many important applications today are limited by memory bandwidth or latency [16], [17], [11]. Even worse, emerging applications will be more sensitive to main memory bandwidth and latency than today [18]. Because the performance of a memory-bound application is roughly proportional to the rate at which its memory requests are served [19], techniques to increase memory bandwidth directly impact application execution time [11].

Data prefetching is currently the defacto solution for latency hiding in modern CMPs [20], [21], [22]. Hardware data prefetchers observe the memory access stream and predict what data should be moved closer to the cores before the data is actually requested by the cores. However, modern last-level cache (LLC) prefetchers on many-core architectures are oblivious to the highly structured data access patterns that are inherent in SPMD execution [23], [24], and so are unable to effectively preserve memory address order across groups of cores. That is partly because LLC prefetchers typically operate in the physical address space and performing reverse translations for prefetching is prohibitively expensive [25]. Therefore, prefetching from a different memory page than that of the request that initiated the prefetch is a major challenge. Low-level cache prefetchers do not have this opportunity because they do not get exposed to the access streams of other cores.

In this paper, we describe the last-level collective

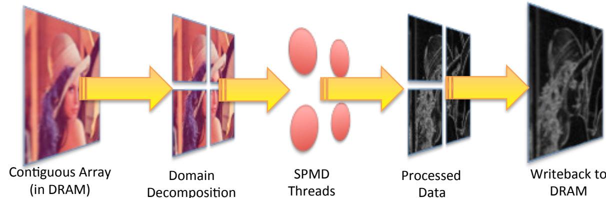

Figure 1: An example from a typical data-parallel image processing kernel.

prefetcher (LLCP), a LLC prefetcher that recognizes and exploits the highly correlated access patterns of data-parallel algorithms and coarse-grain parallelization. LLCP extends the strided prefetcher [26] to anticipate memory accesses by other cores that will request different parts of the same distributed array that the initiating core accesses first. The prefetches issued on behalf of different cores may reside in different physical memory pages, without the need for expensive address translations [25]. Furthermore, LLCP issues prefetch requests to memory on behalf of multiple cores *in memory address order*, which maximizes bandwidth and reduces power [27], [28], [29], [30], [31]. For applications without data-parallel access patterns, LLCP reverts to conventional strided prefetcher behavior [26]. Essentially, LLCP acts as a memory access accelerator for the critical class of data-parallel applications and requires no software intervention.

Compared against the best performing among the strided prefetcher [26], spatial memory streaming (SMS) [32], and global history buffer (GHB) [33], well-established prefetchers that target access patterns similar to LLCP, LLCP improves execution time by 5.5% on average (10% maximum), increases DRAM bandwidth by 9% to 18%, decreases DRAM rank energy by 6%, produces 27% more timely prefetches, and increases the number of prefetched data that are referenced over the number of cache misses (prefetching coverage) by 25% at minimum. LLCP achieves this array of improvements with no increase in complexity compared to competition, a marginal increase in logic area and power in the CMP, and with no software assistance.

II. Background and Motivation

A. SPMD and Data-Parallel Applications

Bulk-synchronous SPMD execution employs groups of cores executing essentially the same code in tandem with different inputs to speed up a single kernel. SPMD kernels typically rely on domain decomposition to divide up a large work array, which is a common way to expose parallelism as shown in Figure 1 [8], [34]. Each core computes on the data tiles it is assigned and then writes tiled results back to a contiguous array in main memory. Domain decomposition generalizes to problems of any dimension.

In this setting, a core requesting a tile provides a strong indication that the other tiles in the same distributed array should be prefetched to on-chip caches. Past work has already quantified the spatial and temporal localities inherent in data-parallel applications [23], [24], [34]. However, the predictability of this pattern is currently unexploited by modern prefetchers. It is spatial locality that we pursue with a prefetcher that is aware of data-parallel application access patterns. Even though there can be time skew between cores, because data-parallel applications use barriers between computation kernels [9], this skew typically is small enough such that prefetched data will not be evicted before they are used as long as the LLC can hold the working set for each core.

B. Memory Access Streams

In current CMPs and even in SPMD execution, cores access memory independently – causing requests to arrive unordered to the DRAM controller [35], [28]. Even though data-parallel applications typically use barriers between computation kernels [9], there are many loop iterations during computation phases. This effectively eliminates any coordination among cores when they access the DRAM to load or write tiles. The skew in the data access patterns is exacerbated by variability in core execution time caused by load imbalance, the system scheduler, and other factors [36]. This generates non-contiguous access patterns to the memory. Our claim is that the majority of bulk-synchronous SPMD codes would present a very highly ordered streaming data access pattern to the memory system were it not for this skew between load-store streams coming from these independent processor cores. Related work only partially addresses this problem or imposes substantial performance penalties.

In general, non-contiguous access patterns degrade DRAM bandwidth, latency, and power [28], [27], [29], [30], [31]. because they do not take advantage of pre-activated rows and therefore cause more row activations compared to sequential access patterns [28]. This is known as *overfetch* [28], [30]. Overfetch is detrimental to memory throughput, latency, and power because activating a new row requires charging bit lines, amplification by sense amplifiers, and then writing bits back to cells. As a result, in many workloads an open row is used only once or twice before being closed due to a row conflict [28]. Past work reports that, depending on the access pattern, as little as 14%–97% of peak memory bandwidth can actually be utilized [12]. Past work also reports that a random-order access stream compared to an in-order one lowers DRAM throughput by 25% for reads and 41% for writes, increases median latency by 23% for loads and 64% for stores, and increases power by 2.2× for loads and 50% for stores [27].

Modern memory controllers reorder requests in their transaction queues to reduce overfetching. However, they are typically passive elements which do not control how requests

arrive to them. Therefore, their degree of choice is limited to the entries in their finite-size transaction queues [19], [16], [37], [29], [38]. In a medium- to large-scale CMP where each core issues just a few tens of requests, this is enough to overwhelm nearly any modern DRAM controller's transaction queue.

C. Prefetchers

Prefetchers in low-level caches move data closer to the cores. LLC prefetchers move data from the DRAM to the LLC (off chip to on chip). Low-level cache prefetchers often suffer on bulk-synchronous applications with dense block arrays because the contiguous address stream is typically short [27] and confuses the prefetcher's filter heuristics. In addition, low-level cache prefetchers also issue requests independently of others and thus create out-of-order access patterns to the memory [39].

A simple and yet popular prefetcher is the strided prefetcher [26], [22]. Each read request arriving to the prefetcher creates or accesses a *stride prediction entry (SPE)*. When a load instruction requests address A, it is compared to the previous address the same load requested (B). The difference $A - B$ is the new *stride* S for that instruction. When the request for A arrives, the SPE is activated causing the prefetcher to issue $A + i \times S$ where i ranges from 1 to D where D is the *degree*, set by the prefetcher.

Strided prefetchers maintain a SPE for every load instruction and each core, and use the program counter (PC) or cache block addresses to differentiate between instructions. Each entry contains a base address, the identifier of the core, the stride S, and degree D. In addition, SPEs carry a confidence value that has to be above a threshold (*CONFTHRESH*) for the SPE to produce prefetches. Confidence increases by *CONFINC* if, at the time the request for A arrives, the newly-calculated stride S matches the old stride (old value of S). Otherwise, confidence decreases by *CONFDEC*. New SPEs are assigned an initial confidence value *CONFINIT*. Confidence values have a minimum *MINCONF* and a maximum *MAXCONF*.

The strided prefetcher maintains its SPEs in the reference prediction table (RPT). The RPT is indexed by a hash function that takes as input the load instruction's PC or cache block index [26]. The RPT is typically set-associative [40] such as to allow multiple SPEs with the same hash function value. SPEs are evicted when a new SPE is created using a replacement policy such as least recently used (LRU).

Current LLC prefetchers typically do not prioritize memory bandwidth and also do not accurately capture access patterns created by data-parallel applications. In part, this is because LLC prefetchers typically operate in the physical address space. Thus, spanning memory pages in a single prefetch activation requires a reverse translation to the virtual address space [25] which makes such approaches impractical. There is a substantial opportunity to exploit the correlated behavior of cores that are executing SPMD code. Currently, even state-of the art prefetchers are unable to do so because of this assumption of independence.

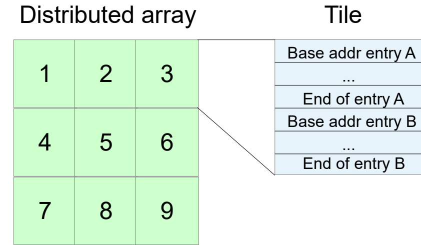

Figure 2: A single tile is much larger than a typical value of D. Only twice as large is shown, therefore two SPEs (A and B) would prefetch the entire tile.

III. LLCP: Last-Level Collective Prefetcher

A. Predictability of Data-Parallel Algorithms

Figure 2 shows that tiles are typically much larger than SPE degrees (D). It is impractical to set D large enough to fetch the entire tile at once because this may create LLC contention. Therefore, SPEs have to be activated from multiple memory requests at different times to prefetch the entire tile. Also, the majority of modern mappings of tiles to memory addresses do not preserve contiguous memory address order of data [41]; mappings that do so tend to have negative side effects to performance or programmability. Because tiles do not consist of contiguous address spaces, the request stream generated by prefetchers that only prefetch within a single core's tile cannot be in contiguous address order.

In addition, tiles tend to be much larger than memory pages (4KB is a common memory page size). Given that LLC prefetchers predominantly operate in the physical address space, a LLC prefetcher trying to prefetch an entire tile would require multiple translations between the virtual and physical address spaces. This would be unfavorably costly, even if a last-level translation lookaside buffer (TLB) is available [25]. Therefore, prefetchers tend to stop prefetching at page boundaries which limits prefetching effectiveness.

B. Overview

Figure 3 shows the desired functionality. When a core requests its tile, part of or the entire tile is prefetched, similarly to the strided prefetcher. However, LLCP also prefetches the equivalent parts of other tiles into the LLC if the application generates data-parallel memory access patterns. LLCP fetches each tile from the memory page it resides in, which may differ from the memory page the triggering request is in. Finally, all data across cores are fetched in memory address order to maximize memory bandwidth.

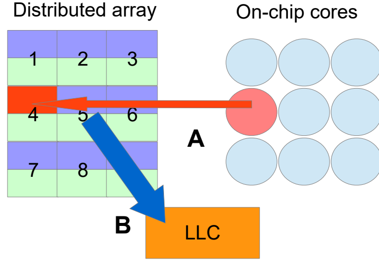

Figure 3: A core requests the first part of its next tile (A). LLCP fetches the equivalent parts of other tiles (in purple) in memory address order (B), anticipating other cores' requests.

To produce a prefetch stream in address order, LLCP maintains all SPEs associated with the same distributed array sorted by base address in the physical space. Therefore, in the example of Figure 2, there is one SPE per tile and SPEs are associated to each other and sorted by base address. When a prefetch for a distributed array initiates, SPEs are used in an interleaved manner. If $Base_i$ is the base address of the ith SPE in memory address order, S_i its stride, D_i its degree, and N the number of SPEs associated with the same distributed array, the generated access stream is

$$Base_1, ..., Base_N, ..., (Base_1 + S_1), ..., (Base_N + S_N), ..., \\ (Base_1 + S_1 \times D_1), ..., (Base_N + S_N \times D_N) \quad (1)$$

This assumes all S_i are equal; this is true for distributed arrays that map all tiles to addresses the same way.

In bulk-synchronous SPMD execution, it is the same instructions (with the same PC) but from different cores that access different parts of the same distributed array. Therefore, SPEs with the same PC value are associated and will generate prefetches (activate) when one SPE with that PC activates. We define that SPEs with the same PC and confidence no less than *CONFTHRESH* belong to a *group*. In Figure 2, SPEs for tiles $1, .., 9$ belong to the same group. SPEs join or create a group when their confidence is increased to no less than *CONFTHRESH*. They get evicted when they are evicted from the RPT or their confidence falls below *CONFTHRESH*. SPEs update their confidence in a similar manner to the strided prefetcher. It is by forming groups that LLCP detects and exploits data-parallel access patterns.

C. Architecture

LLCP's internal architecture is shown in Figure 4. Similar to the strided prefetcher [26], SPEs contain a base address, core identifier, radix (R), degree (D) and confidence. When a memory request arrives to the prefetcher, the hash function is used to index the RPT. The RPT returns a SPE that was created *by the same core* as the incoming request.

SPEs in the same group are not constrained to belong to the same RPT set (line), if allowed by the hash function. In our implementation, the hash function is simply $(\frac{PC_{request} \% NumLines_{RPT}}{4} + Core_{ID}) \% 4$. With this hash function, all SPEs with the same PC value have to be in one of four RPT lines (hence the modulo four in the hash function) such that the way associativity of the RPT can be four times less than the maximum group size. Otherwise, groups of maximum size cannot be formed because SPEs of the group will keep evicting other SPEs of the same group in the RPT.

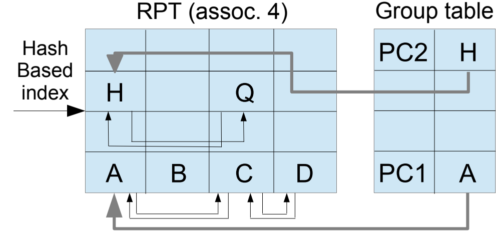

Figure 4: LLCP is based on the strided prefetcher. Thus, it also contains a RPT with associativity (four in this Figure). LLCP extends each SPE by two pointers such that a group is a double-linked list. In this example, SPEs A, C, and D are in the same group and thus linked to each other. B is in the same set as A, C, and D as determined by the hash function, but is not in the same group due to a different PC or low confidence value. H and Q are in a second group. A separate group table holds a pointer to the SPE of each group with the lowest base address.

D. Prefetch Predictions

The decision tree for LLCP is shown in Figure 5. When a memory request arrives to the prefetcher, the RPT and the group table are accessed in parallel. If an existing SPE with the incoming request's core and PC values does not exist in the RPT, the prefetcher behaves exactly as the strided prefetcher [26] by creating a new SPE with the initial confidence (*CONFINIT*) and the request's address as base. This requires finding a free location in the set dictated by the indexing hash function, and potentially finding an eviction candidate [26]. If a SPE is found in the RPT but a group with the request's PC does not exist (the group table contains no such entry), LLCP uses the SPE from the RPT in the same manner as the strided prefetcher to issue prefetches. Therefore, in applications that do not exhibit data-parallel memory access patterns, LLCP operates similarly to the strided prefetcher because no groups are formed. Even if a group activates, the confidence, base address and stride S of the SPE retrieved from the RPT is updated similarly to the strided prefetcher. If a group does exist, that group is used to generate prefetches. Confidence values of other SPEs in the group are not updated.

If the group exists, the prefetcher issues one prefetch per SPE in an interleaved manner by base address order. As

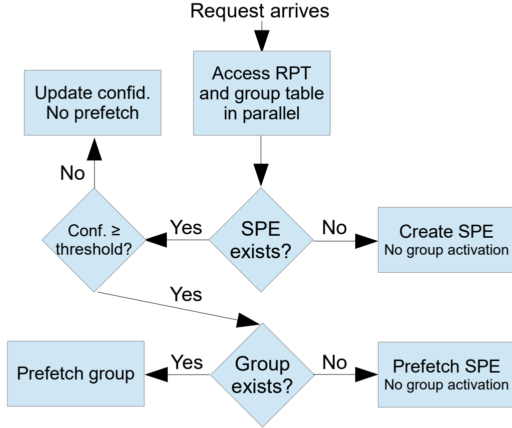

Figure 5: Decision flow chart for when LLCP observes a new memory request.

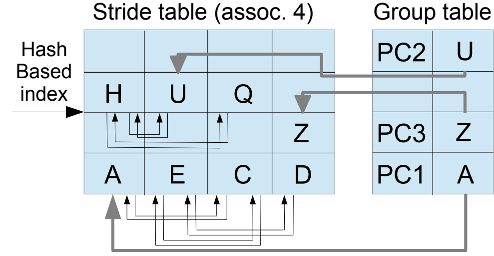

Figure 6: In Figure 4, lets assume that E joins the group with the same PC as A, C, and D ($PC1$). B is evicted by the RPT's replacement policy. If the base address of E is between C and D, the double linked list will be manipulated to result in a group order of A, C, E, D. In addition, U with $PC2$ joins the H, Q group and has the lowest base address. Therefore, the group table updates to point to U instead of H. Finally, Z is created by another memory request with $PC3$, but finds no entry in the group table. Thus, it allocates a new one, essentially starting a new group.

explained in Section III-E, SPEs in groups are kept in a double-linked list by base address order. In Figure 6, the base address order sequence is "A", "C", "E", "D" for that group. Therefore, the linked list is accessed starting from the lowest base address SPE found in the group table ("A" in the example), and then uses the list's pointers to find the next SPE in sequence until all SPEs are accessed. Once all SPEs in the group generate one prefetch request, the traversal repeats for a total number of D linked list sweeps. Therefore, the prefetch stream is that of Equation 1 for N SPEs in the group. Because groups are ordered by base address, this results in a prefetch request stream ordered by memory address. This implies that D and S are the same for all SPEs in the group, which is true for mappings that map tiles to memory the same way for all tiles.

SPE base addresses update when the same instruction (same PC) from the same core that created them issues a subsequent request, similar to the strided prefetcher. However, when a SPE activates as part of a group, it still contains an old base address. In Figure 3, the first core that requests the bottom half (green) of its tile (or an entirely new distributed array), will activate all other SPEs of the same group for other tiles. To prevent having all other SPEs fetch the top half of their tile *again* using their old base address, LLCP calculates the difference between the address of the new memory request that is triggering the prefetch and the base address of the SPE with the same PC and core identifier, found in the RPT. This difference (adjustment factor) is applied to all prefetches by SPEs in the same group. In the example of Figure 3, this means that the first request that arrives for the bottom (green) part of a tile, activates all SPEs in the group but for the bottom part of their corresponding tiles, by adding the adjustment factor to the base address of SPEs in the group.

However, to prevent fetching the bottom half of tiles again when another SPE in the group activates due to its own core's memory requests, each SPE is extended to record the base address (adjusted by the adjustment factor) the last time it activated. If a SPE would activate again with the same base address and adjustment factor, it is skipped instead.

In all cases, if a single SPE would cross page boundaries, any further prefetches are suppressed. This determination is done for each SPE individually and does not affect other SPEs in the group. This is similar to the strided prefetcher without physical to virtual address translation. However, because a group can contain multiple SPEs and each SPE can point to a different memory page, a single group activation can fetch from each memory page that SPEs in the group have their base addresses set to. Essentially, this means LLCP prefetches from multiple memory pages from a single prefetch, without address translation.

E. Forming Groups

A SPE joins or creates a collective group when its confidence reaches *CONFTHRESH*. This can happen during SPE creation if *CONFINIT* is no less than *CONFTHRESH*, or at the time memory accesses arrive and are used to update SPE confidence values in the RPT. SPEs leave a collective group when their confidence falls below *CONFTHRESH* or when they are evicted from the RPT by the RPT's replacement policy. A SPE with both of its pointers set to null does not belong to a collective group. This process is illustrated in Figure 7.

For a SPE to join a group, LLCP performs a lookup in the group table. If an entry with the same PC is found, the SPE joins that group. In that case, the SPE traverses the linked list of the group to find the existing SPE with the largest base address that is still smaller that the newly-joining SPE's base address (the largest smaller value). It then manipulates the pointers appropriately to insert the newly-joining SPE

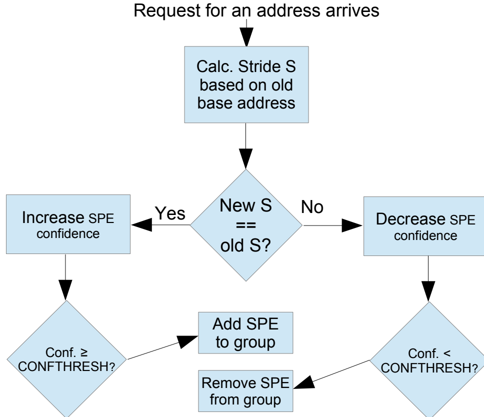

Figure 7: Decision flow chart to add or remove an SPE from a collective group when its confidence updates.

at that location. This serves to keep the group sorted by base address. If an entry with the same PC is not found but a free slot in the group table exists, the SPE creates an entry, essentially starting a new group. Examples are shown in Figure 6.

If a newly-created SPE does not find a group with the same PC and no free entry exists in the group table, it looks for group entries with only one SPE (marked with a flag in the group table). Among them, it replaces the LRU group entry because the oldest group that still contains a single SPE is less likely to get new SPEs in the future. Inactive or imprecise groups eventually get disbanded by either low confidence or RPT evictions. SPEs that do not find space to form a group will retry when they are activated next.

F. Complexity and Cost

A SPE in LLCP contains a base address, stride S, degree D, confidence, PC, two additional pointers to other SPEs to implement the double-linked list, and an additional field to record the base address when the SPE last generated prefetches. Compared to the strided prefetcher, only the additional field and two pointers to other SPEs are added. With 64-bit addresses, each SPE in LLCP requires no more than 64 bytes in the RPT, compared to approximately 52 bytes for the strided prefetcher. LLCP also adds a group table, which is significantly smaller than the RPT that the strided prefetcher also has. The RPT must fit a sufficient number of SPEs such that a sufficient number of groups can form. For instance, to allow ten 64-SPE groups to form, the RPT requires 640 entries minimum. In that case, the group table requires ten entries, with each entry containing just a PC value and a pointer to the SPE of each group with the lowest base address.

Modern prefetchers have tables or similar structures to implement their prediction algorithms [32], [33]. For LLCP, similar to strided, the largest and most energy-consuming structure is the RPT. Inserting a SPE to an existing group requires a linear traversal of the group to locate the appropriate location in the linked list such as to maintain SPE ordering by base address within a group. At every step during the traversal, the base address of that SPE is compared against the new SPE's base address. Each step is better performed in a separate cycle, resulting in multi-cycle insertions. However, this is off the critical path of the prefetcher. Even if the group activates in the meantime, it can ignore the newly-inserted SPE if the insertion is pending.

Activating a group to generate prefetches also requires a linear traversal. In order to avoid accessing the RPT $D \times N$ times as per equation 1, we provide a set of N registers (N is the maximum group size) to act as temporary storage for SPEs. This way, each SPE in the RPT is accessed once and transferred to a register.

For removal, a pointer to the removed SPE is readily available either because we just decremented its confidence or evicted it from the RPT. These are the two scenarios that cause SPEs to be removed. Removing a SPE from a group uses the two double-linked list pointers contained in the SPE to update the pointers of the adjacent SPEs in the linked list, and no traversal or re-sorting is required.

IV. Evaluation

A. Methodology

We perform full system simulations using the GEM5 simulator and the classic memory model [42], configured for a CMP with 64 in-order 2GHz Alpha cores, a crossbar, 64KB 2-way set associative private L1 caches, and four 32MB (2MB per core) 8-way set associative shared L2s each co-located with a memory controller. Caches use MESI coherence. Memory controllers have a single DDR3-1600 x64 channel with one command and address bus, based on Micron MT41J512M8. Memory addresses are mapped first to rows, then columns, then ranks, then banks and finally channels. We select representative benchmarks from Parsec [43], Rodinia [44], and Parboil [45] with their medium-size inputs where possible. The benchmarks we choose create a variety of patterns to illustrate LLCP's functionality, including benchmarks that do not benefit from LLCP compared to strided (e.g., Heartwall) as well as benchmarks where LLCP is not the winner. Backprop is an unstructured grid pattern recognition application. Ferret is pipelined instead of data-parallel and performs similarity search. Myocyte is a structured grid biological simulation application. Stencil performs an iterative Jacobi stencil operation on a regular 3D grid [9]. Streamcluster performs data-parallel data mining. Cutcp computes the short-range component of Coulombic potential over a 3D grid. Heartwall performs medical imaging processing. Finally, Fluidanimate computes fluid dynamics using smoothes particle hydrodynamics (SPH). These benchmarks have a varying degree of data sharing [46], [43], [44]. Specifically, Streamcluster and

Stencil having low degrees of sharing and the rest moderate to high.

We compare against the implementation of the strided prefetcher [26] found in GEM5 [42], and optimized implementations of GHB [33] and SMS [32] found in the data prefetching competitions of [47] and [48] respectively. We choose these competitors because they are well-respected state of the art prefetchers that target access patterns similar to LLCP. In Section VI, we discuss alternatives that differ compared to LLCP in scope, software assistance, or are otherwise not directly comparable. All prefetchers are implemented in GEM5 behaviorally and placed at the LLC, but we also report results with L1 cache prefetchers. LLCP and the strided prefetcher each have an RPT with 64 entries, one for each core. Each entry has 16 sets (lines) for LLCP and 18 sets for the strided prefetcher, to normalize for area since LLCP's SPEs are larger. LLCP's group table has 32 entries. The maximum group size is 64 to match the number of cores. The LRU replacement policy is used.

SPEs have a degree (D) of eight in both the strided prefetcher and LLCP. This value of D is chosen based on the number of cores (i.e., maximum SPEs in a group) and the size of the LLC. The minimum confidence (*MINCONF*) is zero, maximum (*MAXCONF*) is seven, and starting and threshold confidences (*INITCONF* and *CONFTHRESH* respectively) are both four. These values were derived to control how conservative LLCP is in creating and activating groups. *INCCONF* and *DECCONF* are both set to one. The SMS prefetcher has 16 miss status handling registers (MSHR) entries, a pattern history table with 1024 lines of 8 entries each, an active generation table of 16 lines of 16 entries each, a block size of 64 bytes, and 512-byte regions. GHB has 1024 entries in its history buffer, 256 eight-way entries in its index table, and a maximum prefetch degree of 8. These sizes were chosen to equalize the area occupied by each prefetcher. No prefetcher performs virtual to physical translations. Memory page size is 4KB.

B. Performance and Accuracy

Figure 8 shows execution time results. For all applications except Backprop and Stencil, LLCP offers the highest speedup. The highest speedup for LLCP compared to no prefetching is 24% for Myocyte and the average among all applications is 9%. Compared to the best-performing of the competition for each benchmark, LLCP offers a 5.5% improvement by average and 10% at maximum for Myocyte.

To understand the performance results, we first look at the LLC miss rate for each benchmark and prefetcher in Figure 9. LLCP has a 56% lower miss rate compared to SMS, 44% compared to GHB, and 63% compared to baseline. LLCP has a comparable miss rate with strided, but DRAM bandwidth, coverage, and timeliness—explained below— contribute to LLCP's superior performance.

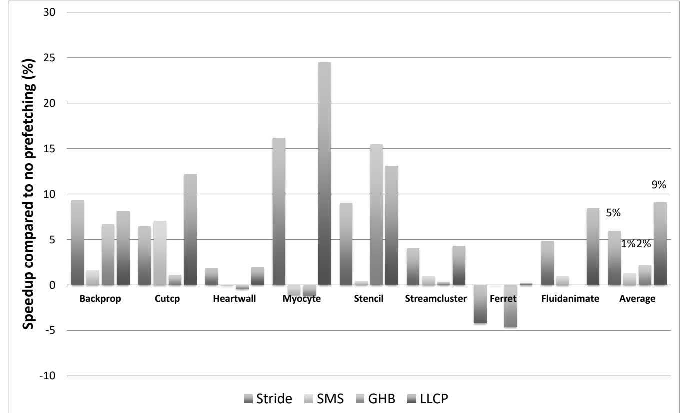

Figure 8: Speedup results compared to no prefetchers (baseline). Negative speedup indicates a slowdown.

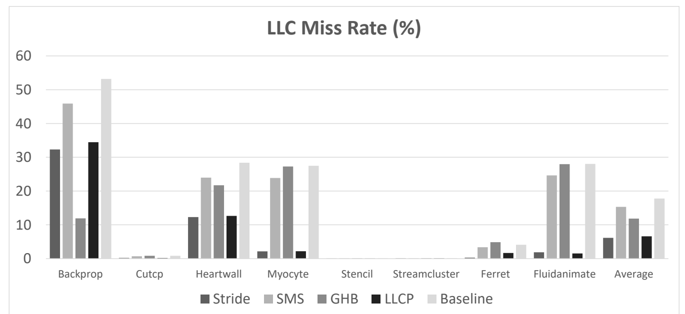

Figure 9: LLC (L2) miss rates (%).

For instance, Figure 10 shows that LLCP produces an average 9% higher DRAM read bandwidth compared to strided, 14% compared to GHB and SMS, and 18% compared to baseline. This is a crucial factor for memory bandwidth-constrained applications and is caused by LLCP's in-order memory access, and not just prefetching more bytes or mispredicting and thus fetching useless data. This is evident by the average 5% higher row buffer hit rate of LLCP compared to competitors, reduced DRAM access latency, comparable number of bytes fetched compared to

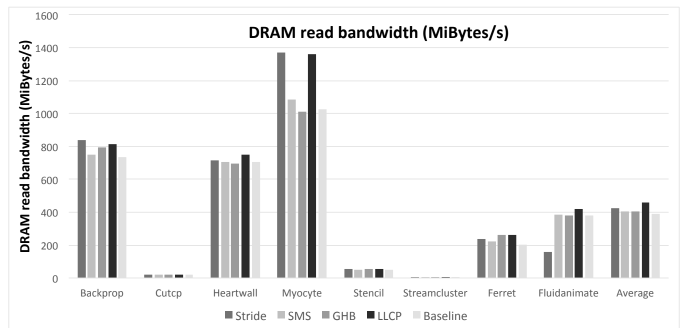

Figure 10: DRAM read bandwidth (MBytes per second).

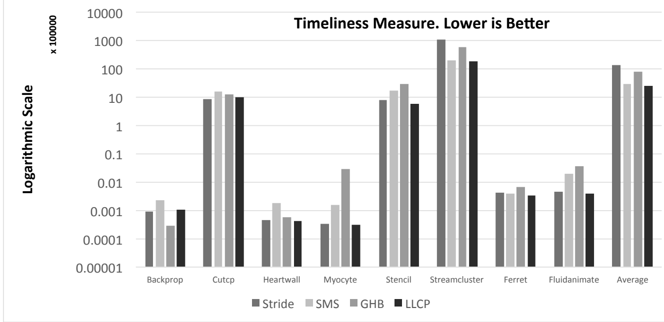

Figure 11: Number of prefetches that were squashed by a demand missed (late prefetches), divided by the total number of prefetchers. Lower is better.

strided and SMS (later explained), and that with LLCP DRAM rows retrieve an average 7% to 11% more bytes each time they activate.

Due to accessing memory in address order, LLCP reduces LLC read miss latency by an average of 15% compared to competition, and the DRAM read latency by 4%.

Compared to all three competitors, LLCP generates more timely prefetches because it can prefetch from multiple memory pages, thus can prefetch more data from a single activation. In addition, another reason LLCP improves timeliness compared to strided is that even though both prefetchers eventually activate the same SPEs, LLCP activates multiple SPEs at once whereas strided has to wait for requests from each core individually. As a measure of timeliness, we record the number of prefetches that were squashed by a demand miss at the LLC, which indicates the number of prefetches that were issued too late. We then divide this number by the overall number of prefetches issued. As shown in Figure 11, by average across benchmarks LLCP improves on this metric by 82% compared to strided, 17% compared to SMS, and 80% compared to GHB.

More read bytes per row activation means fewer row activations if the number of retrieved bytes for LLCP is no greater than competition. This is in fact the case with LLCP causing a comparable number of bytes to be read from DRAM compared to SMS, 3% more than strided, and 14% more than GHB. These differences are slightly magnified if we count the number of cache lines requested by the prefetcher. Fewer row activations result in an average of 5% lower row activation energy in the DRAM for LLCP compared to competition, and a 6% lower overall energy per rank in the DRAM.

The extra bytes LLCP retrieves from DRAM compared to competition are due to cache lines that were fetched and evicted without being referenced because the application previously generated data-parallel access patterns but that is no longer the case, and LLCP has not adapted yet. With applications that frequently change access patterns (a challenge for many prefetchers), LLCP can prefetch useless data which may evict useful data from the LLC; this is the reason for LLCP's lower performance in Backprop compared to strided in Figure 8. For applications that never generate data-parallel access patterns, LLCP behaves similarly to the strided prefetcher, as is the case with Heartwall. However, we notice that the majority of our application benchmarks consistently generate data-parallel access patterns, because LLCP forms a minimum of 126 groups with 64 SPEs for Ferret indicating that 64 cores participate in a data-parallel access pattern, maximum of 3473 64-SPE groups for Myocyte, and an average of 2077. Groups activate a few million times during the entire course of an application.

Finally, we look at accuracy and coverage [20], [21], [22]. Accuracy is the ratio of prefetched data that was referenced over the number of generated prefetches. Coverage is the ratio of prefetched data that was referenced over the number of cache misses. We notice that LLCP has in fact 8% lower accuracy than strided, 2% lower than GHB, and 23% higher than SMS by average. This result is intuitive because LLCP uses SPEs the same way as strided and therefore in the course of an application all SPEs will activate and prefetch the same data, but LLCP risks erroneously activating other SPEs for non-data parallel access patterns. However, it is the *timeliness* of LLCP's prefetches and the in-order memory accesses that generate the performance advantages. Still, compared to SMS, LLCP improves accuracy on top of the other factors. The trend reverses when examining coverage since LLCP has 25% higher coverage by average than strided, 4× compared to SMS, and 5.8× compared to GHB.

Experiments with fewer cores show smaller differences between prefetchers, partly because the stress on the memory is smaller.

C. OOO Cores, LLC Size, L1 Prefetchers

We repeat our experiments with out of order (OOO) cores, different LLC sizes, and L1 cache prefetchers. With OOO cores, for Backprop, Heartwall, and Myocyte LLCP provides smaller gains compared to GHB and SMS and no gain compared to strided. This is the effect of OOO cores being more latency tolerant making timeliness a smaller factor. However, for the rest of our applications LLCP actually increases its performance gains compared to the highest performing competitor, with an average of 8% and maximum of 12% for Fluidanimate. Primarily in-order cores benefit from increasing the cache hit rate and timeliness, whereas OOO cores primarily care about DRAM bandwidth and coverage. Therefore, replacing in-order cores with OOO cores does not universally penalize or favor LLCP.

We then repeat our experiments with half the original LLC size, as well as one quarter. For both half the size and one quarter the size, LLCP remains the highest performer but with a smaller margin (3% and 2% respectively), while still reducing DRAM energy and increasing bandwidth. As

expected, smaller LLCs magnify the adversary effects of prefetching wrong data which LLCP is more prone to, because that useless data is more likely to evict useful data. However, smaller LLCs also increase the miss rate which makes accurate and timely prefetching more important.

Finally, we repeat our experiments after adding strided prefetchers to L1 caches. LLCP still outperforms competition in four benchmarks (Backprop, Heartwall, Fluidanimate, and Streamcluster) by up to 5%. It also performs comparably in all other benchmarks to the highest performing competitor, except for Stencil where it shows a 3% drop compared to strided. The average speedup for LLCP compared to baseline with L1 prefetchers is 4%. L1 prefetchers cannot replicate LLCP's functionality because they prefetch independently and thus create an out-of-order access stream to the memory, which hurts performance and power. LLCP is synergistic to L1 prefetchers because L1 prefetchers can move the data from the LLC to each core's level 1 cache, without LLCP having to push data to lower-level caches to achieve the same result.

D. Energy and Area

For the configuration used in our experiments, each instance of LLCP occupies 65,536 bytes. This is approximately equal to SMS, GHB, and the strided prefetcher. LLCP occupies an insignificant 1% of each of the L2 LLCs in our CMP.

As previously stated, fewer row activations due to in-order memory accesses result in an average 5% lower row activation energy in the DRAM for LLCP compared to competition and a 6% lower overall energy per rank in the DRAM. The power consumed by LLCP is dominated by the RPT because that is its largest structure. For the given configuration, the total dynamic read energy per access to a L2 cache requires 17× more energy than a similar access to LLCP's RPT. In fact, the total dynamic energy for LLCP is 95% lower than for each L2 cache, measured by recording the number of accesses to the RPT and cache, by average across benchmarks. Compared to the other prefetchers, LLCP consumes 2× to 3× more dynamic energy, but that is still a marginal increase (e.g., 1%) compared to just a single L2 cache. The cache also has 50× higher leakage power compared to LLCP. Energy was estimated for a 32nm technology using Cacti 6.5 [49].

Finally, the cycle time of LLCP is comparable to the strided prefetcher. The logic complexity of using and updating SPEs, processing incoming requests, and accessing the RPT are comparable. In addition, the latency to generate prefetches is the same because the group table is accessed in parallel to the RPT (which the strided prefetcher also accesses) for each incoming memory request. Actions that take additional steps, such as adding a SPE to a group, are not time sensitive and thus are off the critical path.

V. Discussion

Similar to any prefetcher, LLCP can potentially prefetch useless data and pollute the LLC. This can happen if multiple cores generate memory accesses from instructions with the same PC, but the requested addresses are unpredictable and do not follow data parallelism. Setting *INITCONF* to be much smaller than *CONFTHRESH* makes LLCP more conservative by having SPEs only produce prefetches if their prediction has already been confirmed multiple times. In fact, we can extend LLCP to dynamically adjust *CONFTHRESH* as well as the maximum group size based on run-time prediction quality feedback. In memory access patterns that are caused by instructions with different PCs such as irregular access patterns, LLCP forms no groups and thus behaves similarly to the strided prefetcher. In divergent memory accesses where the access pattern follows data parallelism but the number of participating cores changes, a high *CONFTHRESH* makes LLCP more conservative and lets LLCP adjust quickly by removing inactive SPEs from collective groups; this is preferable in memory bandwidth constrained systems.

Applications with low degrees of data sharing between cores favor LLCP because with high sharing other prefetchers are more likely to fetch data that are useful to more than one core. Still, even with high sharing all data are not typically shared by all cores which means LLCP will tend to generate more timely prefetches compared to competition. In addition, in the case where some cores are slower than others, LLCP will still prefetch the data for each individual core's next iteration regardless of the iteration count for every group activation, because of how the adjustment factor is calculated (Section III-D). Multiple threads on the same core generating requests from the same PC can be confusing for many prefetchers, but this can be alleviated by making LLCP thread-aware or allowing threads to run long enough to re-train LLCP.

Furthermore, because some memory requests may be satisfied by lower-level caches, the LLC risks not observing the constant-stride sequences applications create but instead consider them as variable-stride patterns. However, this is not predominant in data-parallel applications where at every iteration of the algorithm each core fetches a new tile from a new distributed array. Thus, no on-chip cache contains that data [15], [8]. Therefore, the LLC observes the application's unfiltered access stream.

Because SPEs are in the physical address space, it is possible that by the time LLCP activates SPEs in a group, some of those SPEs refer to address memory pages that have been released by the operating system. This, however, is unlikely to happen in data-parallel applications where distributed arrays are typically allocated and released as a whole. Furthermore, in some modern processors, the LLC only observes cache line addresses instead of memory

addresses. In such cases, strides in SPEs can be based on cache lines.

In its current form, LLCP fetches data to the LLC and does not push to lower-level caches to avoid modifications to current cache coherency protocols that do not support receiving data they did not request. Such modifications lead to complex verification [50]. Given that the latency and energy required to fetch data from an on-chip location compared to off-chip is an order of magnitude lower, fetching data to the LLC provides a good trade-off of benefit versus complexity [2].

A variety of past prefetchers also make use of the PC at the LLC, including the strided prefetcher [33], [26]. Accessing PC information at the LLC is already part of a variety of processors [51], [52]. However, some architectures do not expose the PC to the LLC. In such cases, associating SPEs into groups can be performed by other information such as virtual addresses or more complex access stream pattern matching. LLCP thus would be similar to cache block-based stride prefetchers [53].

In essence, LLCP acts as memory access accelerator for the critical and large class of data-parallel applications. In this sense, LLCP pushes the boundary of specialization of memory accelerators, which is an avenue for continued performance scaling for digital computing.

Our study focuses on DRAM because capacity and cost concerns necessitate that most of the memory in future large-scale systems will remain "conventional" DRAM, including stacked in-package memories such as HBM [14]. 3D stacked memory with abstracted memory interfaces may be affected differently by the order of memory access streams, but prefetching will still be an effective way to reduce data access latency. Even with memory technologies that provide higher bandwidth and lower latency, architectures and applications tend to adapt and eventually become bandwidth or latency limited again. LLCP is also a prime candidate for graphical processor units (GPUs) because many applications with bulk-synchronous data parallelism execute in such platforms. However, as we discuss in Section VI, some cooperative prefetching techniques exist for GPUs but very little similar work is found for CMPs.

VI. Related Work

The SMS [32] and GHB [33] prefetchers are data prefetchers for the LLC that monitor global memory access patterns and exploit regularity of access patterns of data structures. Thus, they have similar goals with LLCP. SMS predicts the blocks in a certain memory range that are spatially correlated potentially beyond a single cache block by recording which blocks are accessed within a predefined time interval into a history table. SMS defines spatial correlation as recurring patterns in relative addresses. Future predictions use a combination of address and PC to retrieve a recorded pattern from the history table. In contrast, GHB identifies strides between addresses of different instructions using the PC and addresses to index into an index table, which contains pointers to a global history table. Each global history table entry stores a global miss address and a link pointer, which chains multiple table entries into address lists. Through forming address lists, a memory access caused by instruction A may trigger a prefetch for instruction B.

Numerous other data prefetchers exploit data locality [21], [20], typically stride-based [54], [26], context-based [55], or a combination [21], [56]. The complex prefetcher uses multiple prediction tables to capture access patterns with multiple strides (address deltas) within the same page [57]. The sandbox prefetcher determines at run-time the appropriate prefetch algorithm [58]. The irregular stream buffer is simultaneously temporally and spatially ordered [59]. The indirect prefetcher captures irregular access patterns generated from indirect patterns of the form $A[B[i]]$ [60]. Similarly, the stateless prefetcher predicts addresses in pointer-intensive applications [61]. These are synergistic approaches that we can apply to LLCP with modifications to handle pointer-intensive applications.

Furthermore, the adaptive stream prefetcher adaptively modulates the aggressiveness of a stream prefetcher to match the workload's spatial locality [62]. Prefetching has also been used to improve row buffer locality and reduce thrashing, but this work exploited the predictability of GPU applications, the specific architecture, and shared some limitations with other work such as prefetching from a single memory page at a time [63]. OWL includes a cooperative prefetching scheme for threads in a GPU, but only prefetches among already active DRAM rows [64]. Other prefetchers adapt their aggressiveness using information collected during program execution [65], [62] or performance gradients [65]. Stealth prefetching requires keeping address-related metadata (hundreds of KBs per core) to decide the subset of a page that should be prefetched [66]. Other context-based prefetchers capture the working set of loop iterations by using code annotations and are based on the observation that code block working sets are highly interdependent across tight loop iterations [56]. Other approaches include delta correlation instead of simple strides [67], defining spatial groups that are simultaneously fetched [54], and using tags instead of cache-line addresses [68].

Other work motivates a heterogeneous interconnect where low-power wires carry prefetch requests [22]. Furthermore, compression in caches can affect the accuracy of prefetchers [69]. In addition, using the same replacement policies for prefetched and demand requests may evict prefetched data before they are used [70]. Pinning data to the LLC and suppressing select prefetch requests may increase prefetch accuracy [39]. Prefetching also applies to TLBs [25]. Related to LLCP, the inter-core cooperative TLB multi-level prefetcher exploits common miss patterns across cores in a CMP [71].

Software-only or hardware/software prefetchers also exist [21], [20]. The key differences are the modifications to the software and hardware to give the compiler or application detailed knowledge of the architecture as well as the ability to pass down information to the hardware, combined with the difficulty in making predictions statically (without run-time data). This is a fundamental tradeoff which makes accurate hardware-only prefetching attractive, and thus the focus of LLCP. Still, software can provide helpful hints to the hardware [72], [73], [74] to change the memory bandwidth available for prefetching [75], manage scratchpad memories [76], or assist with other functions such as dead block prediction [77]. or perform application-specific optimizations [78], [79]. Software prefetching (including direct memory access (DMA)–based techniques) can outperform LLCP, just like other hardware prefetchers as past studies show [21], [20], since they receive all necessary information from the application instead of predicting it [27], [80]. LLCP aims to be hardware-only and thus not receive software hints, thus we compare against other hardware-only prefetchers.

Software prefetchers can adversely affect the training of hardware prefetchers if not designed in tandem [20], [81]. GPUs can use adjacent threads to more efficiently identify address patterns [82], use software hints [83], or use prefetch-aware warp scheduling [84]. The operating system can also provide support in the form of a buffer to mitigate LLC pollution [85]. Other studies examine the interference between hardware prefetchers residing at different levels of the cache hierarchy [86], [87] as well as the effect on memory bandwidth.

Alternative approaches hide or mitigate memory access latency with load value prediction [88], approximation [89], precomputing data with separate threads [90], simultaneous multithreading, and others.

VII. Conclusion

We present LLCP, a collective hardware LLC data prefetcher that exploits access patterns generated by data-parallel algorithms by using one core's memory access stream to predict data for other cores with no software assistance [15], [34], [91]. LLCP can prefetch from multiple pages from a single triggering memory request without the need for expensive translations [25]. To maximize memory bandwidth and reduce power, LLCP issues prefetch requests to memory on behalf of multiple cores *in memory address order* [27], [28], [29]. Both in-order memory accesses and prefetching from multiple memory pages are important advances. For applications that do not exhibit data-parallel access patterns, LLCP behaves as a strided prefetcher [26]. Essentially, LLCP acts as a memory access accelerator for data-parallel applications.

LLCP improves execution time by 5.5% on average (10% maximum), increases DRAM bandwidth by 9% to 18%, decreases DRAM rank energy by 6%, produces 27% more timely prefetches, and increases prefetching coverage by 25% at minimum across applications compared to the best-performing from our state of the art competitors, with comparable complexity, and with just a marginal increase in logic area and power in the CMP.

Acknowledgments

This work was supported by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE- AC02-05CH11231.

References

[1] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, 2011.
[2] J. Shalf, S. S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," ser. VECPAR, vol. 6449. Springer, 2010.
[3] J. Torrellas, "How to build a useful thousand-core manycore system?" *International Parallel and Distributed Processing Symposium*, 2009.
[4] V. Agarwal *et al.*, "Clock rate versus IPC: the end of the road for conventional microarchitectures," ser. ISCA, 2000.
[5] M. Horowitz and W. Dally, "How scaling will change processor architecture," ser. ISSCC, 2004.
[6] S. Borkar, "Thousand core chips: a technology perspective," ser. DAC, 2007.
[7] K. Sudan *et al.*, "Micro-ignorepages: increasing DRAM efficiency with locality-aware data placement," ser. ASPLOS, 2010.
[8] K. Datta *et al.*, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," ser. SC, 2008.
[9] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures," ser. ICS, 2012.
[10] L. Peng *et al.*, "High-order stencil computations on multicore clusters," ser. IPDPS, 2009.
[11] A. Vega *et al.*, "Breaking the bandwidth wall in chip multiprocessors," ser. SAMOS, 2011.
[12] S. Rixner, "A bandwidth-efficient architecture for a streaming media processor," Ph.D. dissertation, MIT, 2001.
[13] B. M. Rogers *et al.*, "Scaling the bandwidth wall: challenges in and avenues for CMP scaling," ser. ISCA, 2009.
[14] D. Jevdjic *et al.*, "Unison cache: A scalable and effective die-stacked dram cache," ser. MICRO, 2014.
[15] Y.-K. Chen *et al.*, "Convergence of recognition, mining, and synthesis workloads and its implications," *Proceedings of the IEEE*, vol. 96, no. 5, 2008.
[16] E. Ebrahimi *et al.*, "Parallel application memory scheduling," ser. MICRO, 2011.
[17] J. Huh, D. Burger, and S. W. Keckler, "Exploring the design space of future CMPs," ser. PACT, 2001.
[18] R. Murphy, "On the effects of memory latency and bandwidth on supercomputer application performance," ser. IISWC, Sept 2007.
[19] L. Subramanian *et al.*, "MISE: Providing performance predictability and improving fairness in shared main memory systems," ser. HPCA, 2013.
[20] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn't, and why," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, 2012.
[21] S. Byna, Y. Chen, and X.-H. Sun, "A taxonomy of data prefetching mechanisms," ser. I-SPAN, May 2008.
[22] A. Flores, J. Aragon, and M. Acacio, "Energy-efficient hardware prefetching for CMPs using heterogeneous interconnects," ser. PDP, Feb 2010.
[23] J. Meng, J. W. Sheaffer, and K. Skadron, "Exploiting inter-thread temporal locality for chip multithreading," ser. IPDPS, April 2010.
[24] J. Weinberg *et al.*, "Quantifying locality in the memory access patterns of HPC applications," ser. SC, 2005.
[25] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level TLBs for chip multiprocessors," ser. HPCA, 2011.

[26] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," ser. MICRO, 1992.
[27] G. Michelogiannakis *et al.*, "Collective memory transfers for multi-core chips," ser. ICS, 2014.
[28] A. N. Udipi *et al.*, "Rethinking DRAM design and organization for energy-constrained multi-cores," ser. ISCA, 2010.
[29] S. Rixner *et al.*, "Memory access scheduling," ser. ISCA, 2000.
[30] J. H. Ahn *et al.*, "Future scaling of processor-memory interfaces," ser. SC, 2009.
[31] D. T. Wang, "Memory DRAM memory systems: performance analysis and a high performance, power-constrained DRAM scheduling algorithm," Ph.D. dissertation, University of Maryland, 2005.
[32] S. Somogyi *et al.*, "Spatial memory streaming," ser. ISCA '06, 2006.
[33] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," ser. HPCA, 2004.
[34] S. M. F. Rahman, Q. Yi, and A. Qasem, "Understanding stencil code performance on multicore architectures," ser. CF, 2011.
[35] G. L. Yuan, A. Bakhoda, and T. M. Aamodt, "Complexity effective memory access scheduling for many-core accelerator architectures," ser. MICRO, 2009.
[36] A. Mazouz, S.-A.-A. Touati, and D. Barthou, "Study of variations of native program execution times on multi-core architectures," ser. CISIS, 2010.
[37] D. Xu, C. Wu, and P.-C. Yew, "On mitigating memory bandwidth contention through bandwidth-aware scheduling," ser. PACT, 2010.
[38] Y. Ishii, M. Inaba, and K. Hiraki, "Unified memory optimizing architecture: memory subsystem control with a unified predictor," ser. ICS, 2012.
[39] M. Kandemir, Y. Zhang, and O. Ozturk, "Adaptive prefetching for shared cache based chip multiprocessors," ser. DATE, 2009.
[40] S. Subha, "A set associative cache architecture," ser. ITNG, 2010.
[41] T. Henretty *et al.*, "Data layout transformation for stencil computations on short-vector SIMD architectures," ser. CC/ETAPS, 2011.
[42] N. Binkert *et al.*, "The Gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, 2011.
[43] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.
[44] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," ser. IISWC, 2009.
[45] J. A. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois at Urbana-Champaign, Urbana, Tech. Rep. IMPACT-12-01, Mar. 2012.
[46] H. Luo, X. Xiang, and C. Ding, "Characterizing active data sharing in threaded applications using shared footprint," 2013.
[47] M. Dimitrov and H. Zhou, "Combining local and global history for high performance data prefetching," 2009.
[48] M. Ferdman, S. Somogyi, and B. Falsafi, "Spatial memory streaming with rotated patterns," in *In 1st JILP Data Prefetching Championship*, 2009.
[49] S. Li *et al.*, "CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques," ser. ICCAD, 2011.
[50] R. Komuravelli, S. V. Adve, and C.-T. Chou, "Revisiting the complexity of hardware cache coherence and some implications," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, 2014.
[51] S. Khan *et al.*, "Improving cache performance by exploiting read-write disparity," ser. HPCA, 2014.
[52] S. Ghose, H. Lee, and J. F. Martínez, "Improving memory scheduling via processor-side load criticality information," ser. ISCA, 2013.
[53] T. Sherwood, S. Sair, and B. Calder, "Predictor-directed stream buffers," ser. MICRO, 2000.
[54] C. F. Chen *et al.*, "Accurate and complexity-effective spatial pattern prediction," ser. HPCA '04, 2004.
[55] D. Joseph and D. Grunwald, "Prefetching using markov predictors," ser. ISCA '97, 1997.
[56] A. Fuchs *et al.*, "Loop-aware memory prefetching using code block working sets," ser. MICRO, 2014.
[57] M. Shevgoor *et al.*, "Efficiently prefetching complex address patterns," ser. MICRO, 2015.
[58] S. Pugsley *et al.*, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," ser. HPCA, Feb 2014.
[59] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," ser. MICRO, 2013.
[60] X. Yu *et al.*, "IMP: Indirect memory prefetcher," ser. MICRO, 2015.
[61] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," ser. ASPLOS, 2002.
[62] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," ser. MICRO, 2006.
[63] R. Panda *et al.*, "Prefetching techniques for near-memory throughput processors," ser. ICS, 2016.
[64] A. Jog *et al.*, "OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance," ser. ASPLOS, 2013.
[65] L. M. Ramos *et al.*, "Multi-level adaptive prefetching based on performance gradient tracking," *Journal of Instruction-Level Parallelism*, vol. 13, 2011.
[66] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "Stealth prefetching," ser. ASPLOS, 2006.
[67] K. Nesbit, A. Dhodapkar, and J. Smith, "AC/DC: an adaptive data cache prefetcher," ser. PACT, 2004.
[68] Z. Hu, M. Martonosi, and S. Kaxiras, "TCP: Tag correlating prefetchers," ser. HPCA, 2003.
[69] A. Alameldeen and D. Wood, "Interactions between compression and prefetching in chip multiprocessors," ser. HPCA, Feb 2007.
[70] C.-J. Wu *et al.*, "PACMan: Prefetch-aware cache management for high performance caching," ser. MICRO, 2011.
[71] A. Bhattacharjee and M. Martonosi, "Inter-core cooperative TLB for chip multiprocessors," ser. ASPLOS, 2010.
[72] S. W. Son *et al.*, "A compiler-directed data prefetching scheme for chip multiprocessors," ser. PPoPP, 2009.
[73] Z. Wang *et al.*, "Guided region prefetching: A cooperative hardware/software approach," ser. ISCA, 2003.
[74] V. Papaefstathiou *et al.*, "Prefetching and cache management using task lifetimes," ser. ICS, 2013.
[75] V. Jimenez *et al.*, "Increasing multicore system efficiency through intelligent bandwidth shifting," ser. HPCA, Feb 2015.
[76] L. Alvarez *et al.*, "Runtime-guided management of scratchpad memories in multicore architectures," ser. PACT, 2015.
[77] M. Manivannan *et al.*, "RADAR: Runtime-assisted dead region management for last-level caches," ser. HPCA, 2016.
[78] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," ser. ICS, 2016.
[79] D. Zucker, R. Lee, and M. Flynn, "Hardware and software cache prefetching techniques for MPEG benchmarks," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 10, no. 5, Aug 2000.
[80] S. Saidi *et al.*, "Optimizing explicit data transfers for data parallel applications on the cell architecture," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, Jan. 2012.
[81] S. Mehta *et al.*, "Multi-stage coordinated prefetching for present-day processors," ser. ICS, 2014.
[82] A. Sethia *et al.*, "APOGEE: Adaptive prefetching on GPUs for energy efficiency," ser. PACT, 2013.
[83] J. Lee *et al.*, "Many-thread aware prefetching mechanisms for GPGPU applications," ser. MICRO, Dec 2010.
[84] A. Jog *et al.*, "Orchestrated scheduling and prefetching for GPGPUs," ser. ISCA, 2013.
[85] L. Soares, D. Tam, and M. Stumm, "Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer," ser. MICRO, 2008.
[86] N. Enright Jerger, E. Hill, and M. Lipasti, "Friendly fire: understanding the effects of multiprocessor prefetches," ser. ISPASS, 2006.
[87] F. Liu and Y. Solihin, "Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors," ser. SIGMETRICS, 2011.
[88] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," *SIGOPS Oper. Syst. Rev.*, vol. 30, no. 5, 1996.
[89] J. S. Miguel, M. Badr, and N. E. Jerger, "Load value approximation," ser. MICRO, 2014.
[90] I. Atta *et al.*, "Self-contained, accurate precomputation prefetching," ser. MICRO, 2015.
[91] M. Müller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications," ser. SCA, 2003.