Eliminating On-Chip Traffic Waste: Are We There Yet?

Robert Smolinski, Rakesh Komuravelli, Hyojin Sung, and Sarita V. Adve
University of Illinois at Urbana-Champaign, denovo@cs.illinois.edu

***Abstract*—The memory hierarchy is predicted to be a major consumer of on-chip energy, leading researchers to develop techniques that minimize the amount of data moved and the distance that it is moved. While many techniques have been shown to be successful at reducing the amount of on-chip network traffic, no studies have shown how close a combined approach would come to eliminating all unnecessary data traffic, nor have any studies provided insight into where the remaining challenges are. This paper systematically analyzes the traffic inefficiencies of a directory-based MESI protocol and a more efficient hardware-software co-designed protocol, DeNovo. DeNovo provides techniques to address network traffic waste, but only at the L1 cache. In this paper, we categorize data waste into various categories and explore several simple optimizations extending DeNovo with the aim of eliminating all of the on-chip network traffic waste. Specifically, we focus on mitigating the network traffic waste that is destined to and originating from the last level cache. With all the proposed optimizations, we are able to completely eliminate (100%) on-chip network traffic waste at L2 for some of the applications (70.8% on average) compared to the original DeNovo protocol. As a result, on average, 8.8% of the remaining total traffic (L1 and LLC) is spent fetching non-useful data. Using only our optimizations, we found that reducing this non-useful data movement further would not be possible without losing performance because of irregular access patterns in the applications.**

I. Introduction

Computer performance is currently limited by energy efficiency. Due to the growing energy cost of data movement relative to computation, improving the energy efficiency of the memory hierarchy is an important part of reaching the future energy goals [4], [18], [15].

There are several sources of energy inefficiency in today's memory hierarchies. In this paper, we specifically focus on the implications of data that is never used. Transferring data that is never used from one point to another in the memory hierarchy incurs unnecessary network traffic which directly contributes to wasted energy in the system. Once such data is transferred to a destination, it is stored in on-chip memory organization units (e.g., caches, scratchpads, etc.) resulting in less available space for useful data and hence indirectly contributes to energy wastage.

We begin our analysis by providing a classification of all data into six different categories. One of the categories is for used words and five are for words that are never used before getting evicted or overwritten (wasted). We limit our study to focus on quantifying unnecessary network traffic caused by each of the waste categories and proposing simple optimizations to nearly eliminate each of these wastes.

This work is supported in part by Intel Corporation and Microsoft Corporation through the Universal Parallel Computing Research Center (UPCRC) at Illinois, by Intel through the Illinois/Intel Parallelism Center at Illinois, by the National Science Foundation under grants CCF-1018796 and CCF-1302641, and by the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. Rakesh Komuravelli was also supported by a Qualcomm Innovation Fellowship.

Towards this end, we chose a directory-based MESI protocol on a standard multicore system as a baseline. MESI is used as our starting point because it has been the *de facto* standard for research into scalable processors that provide a shared-memory abstraction. However, there are several well-known traffic overheads that make MESI less than ideal for energy-efficiency. Some of these overheads are a result of using fixed cache line sizes for coherence and data transfer. For example, the use of fixed cache lines can cause excessive traffic for applications that have false sharing, poor spatial locality, or data that is overwritten before it is read. Other overheads include the traffic required for maintaining protocol correctness (e.g. invalidation and "directory unblock" messages).

Given the inherent overheads with the MESI protocol, we study an alternative hardware-software co-designed protocol named DeNovo [7]. DeNovo leverages software-level information to replace writer-initiated invalidation for cached copies with self-invalidation, which makes it much more efficient than MESI. In terms of network traffic improvements, a baseline implementation of DeNovo eliminates many of the messages used in MESI to maintain coherence, such as invalidation and "directory unblock" messages, and it also eliminates false sharing by keeping coherence at finer granularity.

The DeNovo protocol in [7] already successfully reduces unnecessary network traffic by introducing a flexible communication granularity optimization (Flex) which fetches only "useful" data in contrast to traditional fixed cache line granularity data transfers. However, it focused only on L1 cache related network traffic waste. It did not analyze this or any other sources of remaining waste.

Our goal in this paper is to eliminate every bit of waste in on-chip traffic. We analyze the effectiveness of the DeNovo protocol and its Flex optimization in eliminating waste at L1. We then quantify the amount of waste at L2 and explore several simple optimizations that extend the DeNovo protocol to eliminate different types of waste at L2. Specifically, we applied a write-validate write policy to reduce the amount of data that is overwritten before it is read, a last-level cache bypass optimization that reduces traffic to the last-level cache, and a mechanism that can safely send load requests directly to memory if it is likely to be a last-level cache miss. These optimizations not only reduce network traffic and execution time, but can also be implemented with little additional protocol complexity.

Compared to the original DeNovo protocol with Flex, we reduce on average 93.5% (up to 100% for two applications) of the total waste at L2 for applications with predictable regular access patterns. This has direct implications on the network traffic waste destined to or originating from L2 (reduced by 94.5% on average). For applications that have a lot of unpredictable data access patterns, we see an average reduction of only 19% in total L2 waste compared to the DeNovo protocol with Flex optimization. The remaining

L2 waste is hard to eliminate without either increasing the complexity of hardware or having a negative impact on the execution time. In addition to reducing network waste, these optimizations also reduce total network traffic and execution time (more details in Section VI).

Our specific contributions are as follows:

- We provide a classification of data waste.
- We quantify and analyze the amount of network traffic waste eliminated by the DeNovo protocol and its Flex optimization.
- We explore several simple optimizations on top of the DeNovo protocol without resulting in any additional protocol complexity that eliminate most of the on-chip waste, at least for applications with little or no unpredictable access patterns.
- With all the proposed optimizations together, we are able to completely eliminate the on-chip network traffic waste at L2 for some of the applications (70.8% on average) compared to the original DeNovo protocol.
- Our final DeNovo protocol that has all optimizations applied wastes only 8.8% of its traffic. We describe the underlying causes of this remaining waste and explain why it is difficult to remove without losing performance.

II. Background

DeNovo [7] is a hardware-software co-designed coherence protocol that exploits software information for better performance-, power-, and complexity-scalability. These design choices allow for sharers-lists and invalidation messages in today's directory based coherence protocols to be replaced with self-invalidation instructions that invalidate stale data at phase barriers. The storage saved by eliminating sharers-lists in turn enabled DeNovo to maintain coherence at finer granularity (word) than traditional cache line. DeNovo uses the last-level cache to store per-word ownership (referred to as registration) which results in complete elimination of false sharing. Finally, assuming a disciplined software model that provides data-race-free guarantee, DeNovo completely eliminates transient states from the hardware implementation of the protocol. Together, these properties allow the DeNovo protocol to be drastically simpler to verify and extend than other directory-based protocols.

Based on the language and compiler support as demonstrated in the Deterministic Parallel Java project [3], [14], DeNovo assumes metadata that efficiently summarizes the memory regions that are read or written in each phase (whose boundaries are marked by barriers). This information can be conservative; e.g., if such metadata is not available, all memory is assumed to be read and written as a default.

DeNovo uses such software information to provide coherence and consistency in hardware. First, to make sure a read never sees stale data in its private cache, compiler-inserted self-invalidation instructions use the regions to selectively self-invalidate potentially stale data at phase boundaries. Second, to locate an up-to-date copy of the data in case of read misses, the private caches send a registration request to the last-level cache for every address that the core writes in DeNovo. When the last-level cache receives the registration request it stores the new registration information and sends an invalidation message to the old registrant, if one exists, to prevent that core from using stale data.

The lack of sharers-list allows valid data to be sent from any cache to any other cache without sending additional messages to the directory.

The coherence granularity is decoupled from the data transfer granularity. In traditional protocol designs, data is moved in fixed cache line sizes. As DeNovo is built on word-level coherence and has no sharers lists, the hardware is able to respond with subsets of a cache line, and it can also prefetch words from different cache lines into the same response message. This optimization is called as the "flexible communication granularity (Flex)". The benefit of this feature is that it allows the hardware to reduce the amount of unnecessary on-chip traffic and improve performance.

III. Waste Characterization

We separately analyze network traffic for loads, stores, and writebacks. For loads and stores, we distinguish between traffic destined to the L1 and the L2 caches (without loss of generality, we assume a two level cache hierarchy with private L1 and shared L2 data caches). To understand the sources of waste for load and store traffic, we classify data resident in a cache into six categories described below.

Used: For the L1 cache, a word that is read by its core; for an L2 cache, a word resident in the cache that is sent in response to an L1 request.[1]
Write Waste: a word that is overwritten before being profiled as *Used*.
Fetch Waste: a word brought into a cache where the word is already present as either dirty or was brought in by a previous request.
Invalidate Waste: a word that is invalidated by the coherence protocol before being profiled as *Used*.
Evict Waste: a word that is evicted before being profiled as *Used* or *Write*.
Unevicted: a word that is present in a cache at the end of the simulation and has not been classified. As many of the applications in this study use shortened simulation runs, it is unknown whether the words in this category would have been used if the simulation had continued.

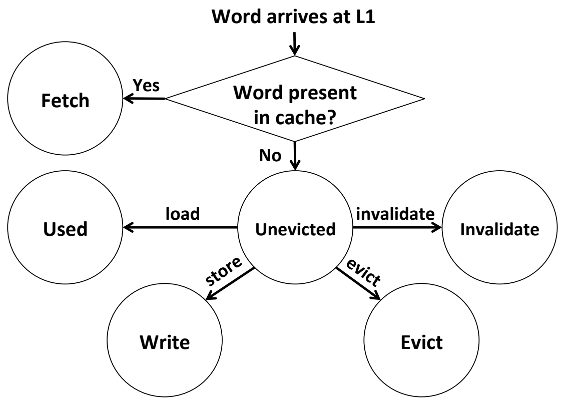


Fig. 1: Finite state machine for waste profiling at the L1 cache.

Figure 1 shows the decision diagram for how data sent to the L1 is categorized. The load and store actions in the figure occur when the L1's core issues a load or store

[1]For the L2, words brought in response to a miss are not (yet) considered *Used*. Such words may be useful to send to the L1, but sending them to the L2 (from the memory controller) is useful only if they will be accessed again at the L2.

for that address. An evict action occurs when the word is deallocated from the cache to make room for incoming data. The invalidate transitions occur on protocol-specific invalidation actions (e.g., self-invalidation for DeNovo and invalidation messages for MESI). The decision diagram at the L2 is analogous to that of the L1 (there is no invalidation action at the L2 and the store action is replaced with "L1 writeback" instead).

A load or a store related network data flit is categorized as above depending on how the data it carries is categorized. Note that in many cases, this categorization can be known only after the data has stayed in the destination cache for some period of time. For writeback (data) flits, analogous to loads and stores, we separate writeback (data) traffic coming from L1 or L2. We categorize a writeback data flit as **Used** or **Waste** based on whether the corresponding word is dirty or not.

IV. Simple Optimizations to Reduce Network Traffic Waste

DeNovo's Flex optimization primarily targets network traffic waste destined for the L1 cache. Motivated by our waste analysis results (Section VI-A), this section discusses several simple optimizations for the DeNovo line protocol [7] that mitigate the network traffic waste related to the L2 cache. As described below, the optimizations do not introduce additional protocol complexity (e.g., transient states) to DeNovo but do require small hardware changes; in contrast, similar optimizations for MESI would require significant changes to the protocol and would be a lot more difficult to implement.

L2 Write-Validate: The original DeNovo protocol chose to implement a write-validate write policy [13] at the L1 and a fetch-on-write write policy at the L2. This means that a write miss at the L1 would not fetch the rest of the cache line, and a write miss at the L2 would fetch the entire line from main memory. Fetching the entire line on an L2 write miss will introduce *Fetch* waste for the critical word (the word that was explicitly requested by the core) and may also introduce *Write* and *Evict* waste for the remaining words in the line. In this optimization, we change the L2 write policy to also be write-validate. This change reduces the amount of on-chip (and off-chip) traffic that is spent moving overwritten or unused data. Overhead: this optimization requires an additional valid bit for each word in the L2 (3% increase in storage).

L2 Dirty-Words-Only Writeback: This optimization uses the L2 per-word dirty bit information so that writebacks to memory from the L2 will only include dirty data, saving the unnecessary traffic required for data that is unchanged. This directly addresses the source of waste in L2 writeback traffic (L1 writebacks already perform this optimization). This change requires no additional L2 storage overhead as DeNovo already has per-word dirty bits. It, however, requires DRAM designs that support writing subsets of a cache line to DRAM. Although this is not currently supported by commercial DRAMs, recent research proposes solutions to this problem (e.g., [32]). We assume such support in this paper.

Memory Controller to L1 Transfer: For L2 misses, the original DeNovo protocol sends data from the memory controller to the L2, and then the L2 forwards the data to the requesting L1. This optimization modifies the protocol so that the memory controller sends the data to both the L1 and the L2 in parallel. The main benefit of this optimization is to reduce the memory hit latency (and not network waste); however, we include it here as a stepping stone to the "L2 Response Bypass" optimization below. The optimization needs some care since the response from memory contains the entire cache line even though some of the words in the line may have been modified on-chip. To maintain coherence in this situation, the L2 attaches a bit vector to each request that goes from the L2 to the memory controller – this vector marks which words are dirty and should not be returned to the requesting L1. The memory controller uses this information to filter invalid words from the data it fetches from memory, and sends the remaining data to both the L1 and the L2. A bit vector is also included in the response so that the receiving caches can determine which words are contained in the message. It is possible that some other core wrote to some of the words while the memory controller was waiting for the response from memory. We do not need to update the bit vector at the memory controller as data-race freedom guarantees us that the requesting core will not access these modified words in the current phase (these words get self-invalidated before the next phase begins). Overhead: this optimization requires an additional bit vector in the control message (2B for a 64B cache-line) and at the memory controller.

L2 Flex: The Flex optimization (Section II) was originally applied to requests that hit an on-chip (L2/remote L1) cache, it therefore reduced waste in traffic to the L1. Responses that came from memory and directed to the L2 would return the normal cache line. By extending Flex into the L2-main memory interface, we can prevent useless data from being returned from memory, thereby reducing *Evict* waste traffic.

For our evaluation, we conservatively use a conventional DRAM protocol that only allows for cache line sized reads. Instead, the memory controller uses the Flex information on the cache line(s) received from DRAM and only forwards the needed words in the response message to the on-chip cache(s). Flex also allows prefetching communication region data that spans multiple cache lines. We restricted this to only request lines that lie in the same DRAM row as the critical (requested) address because row activation is an expensive operation. With proper DRAM support, such as the design presented in [32], the amount of data brought in from memory could also be reduced. Overhead: this optimization extends the memory controller to maintain the Flex information of a given request.

L2 Response Bypass: A well-known problem with large data set sizes is that L2 cache lines generally have poor L2 reuse. Without reuse, caching the line at the L2 incurs energy wastage for moving data to the L2, inserting it into the cache, and potentially evicting other data to make space for the line. Specifically, for our purposes, moving such data from the memory controller to the L2 is wasteful of network traffic, potentially incurring *Evict* waste. We therefore explore an optimization that prevents lines with poor reuse from being sent to the L2 cache. We found that this optimizations was useful for targeting two types of access patterns: (1) the region is read and then overwritten by the same core, or (2) the amount of data in a region exceeds the L2 cache size and is read only once in the current phase of the application. To identify these access patterns, we rely on the programmer or compiler to specify which regions of data should bypass the L2 (using annotations similar to the ones used in [7]). Overhead: the memory controller maintains a bit per request to indicate whether to bypass L2 or not and the request message to the memory controller needs a bit to communicate the same (1 bit for a 64B cache line).

Other optimizations: We explored two other optimizations that provided limited benefit. In the first, **L2 Request Bypass**, even the request for an L1 miss bypasses the L2 (in addition to the L2 Response Bypass). This optimization required significant hardware complexity and did not give commensurate benefit over L2 Response Bypass since the request packet length is small relative to that of a response. This optimization does eliminate an unnecessary probe in the L2 potentially providing energy savings from a source not considered here.

The second, **MMemL1**, attempts to provide some of the benefits of the Memory to L1 and the L2 Response Bypass optimizations to the MESI protocol. It observes that a small part of the benefits of these optimizations could be obtained for MESI with little added complexity. Specifically, the memory controller sends the response directly to the L1 requestor, which then forwards it to the L2, piggybacked on its ack to unblock the directory. For stores, no data need be sent to the L2, saving some network waste for MESI. We found that this optimization had only a small impact (Section VI-C).

We explain the above optimizations and their impact in more detail in Section VI-C, but focus the rest of the paper on the others.

Finally, except for *MMemL1*, we did not incorporate any optimization in MESI due to the significant added complexity to the coherence protocol. If we were able to add these optimizations, we expect their improvements in terms of network waste savings would be analogous to those for DeNovo. Previous work already shows that DeNovo significantly reduces network traffic relative to MESI [7] as well as quantifies the high complexity of MESI relative to DeNovo [19]; therefore, we do not study variations of MESI. (except briefly the straightforward MMemL1).

V. Methodology

A. Protocols Studied

We studied the following protocols for our evaluations. To keep the number of protocols small, we only used a subset of the combinations of the optimizations introduced in Section IV.

MESI: The MESI protocol included with the GEMS Simulator [23] (line granularity tags and coherence state). We modified it to support non-blocking writes (up to 32 pending write requests per core).

DeNovo: The baseline DeNovo line protocol [7] that keeps tags at line granularity but maintains coherence at word granularity. It is extended with a simple optimization of write combining (previously reported in [29], [30]).

DFlexL1: $DeNovo$ with the Flex optimization [7]. Flex is only used for load responses from either the L2 or a remote L1 to the requesting L1 cache.

DValidateL2: $DeNovo$ with support for "L2 Write-Validate" and "Dirty-Words-Only Writebacks."

DFlexL2: $DValidateL2$ with support for "Memory Controller to L1 Transfer" and "L2 Flex." We found that just adding "Memory Controller to L1 Transfer" to $DValidateL2$ had negligible impact, and do not discuss it further.

DBypL2: $DFlexL2$ with "L2 Response Bypass" support.

B. Simulation Infrastructure

We used the SIMICS full-system simulator to model each core, the GEMS memory system simulator to model the on-chip memory hierarchy and coherence protocols, Garnet to model the on-chip network, and DRAMSim2 to model DRAM timing. Since our focus is on the memory system in this work, we use a simple, in-order core model from Simics that completes all non-memory instructions in 1 cycle (similar to other studies).

For our simulations, we model a tiled processor with 16 tiles. Each tile has a single core, a 32KB private L1 cache, and a 256 KB slice of the shared L2.[2] Each corner tile has a memory controller connected to a single channel DIMM. The tiles are connected by an on-chip mesh network[3] with a link width of 16 bytes. Packet sizes are limited to at most one control flit and four data flits. A maximum of four data flits means that at most 64 bytes of data can be included in any message. Table I lists the specific design parameters used for each of these components. [4]

Component	Parameters
Processor	16 cores; each is 2GHz, in-order
L1D (private)	32KB, 8-way, 64B cache lines
L2 (shared), 16 banks	4MB, 16-way, 64B cache lines
Network (Mesh)	16B links, 3 cycle link latency [5]
Memory Controller	4, FR-FCFS, open page policy
DRAM, 8 banks, 2 ranks	DDR3-1066

TABLE I: Simulated system parameters.

C. Benchmarks

We use six of the seven benchmarks used in [7] (input size specified in parenthesis) – FFT (m=16), LU (512x512 and 16x16 blocks), radix (4M integers and 1024 radix), and Barnes-Hut (16K bodies) are from SPLASH-2,fluidanimate (simmedium) is from Parsec, and a parallel kD-tree construction algorithm (kD-tree) (bunny) from [6]. As the DeNovo protocols in this work do not support mutexes, the Barnes-Hut tree building phase was sequentialized and fluidanimate was modified to use the ghost cell pattern [17] to share data between threads. We use the aligned version of LU to remove false sharing.

VI. Results

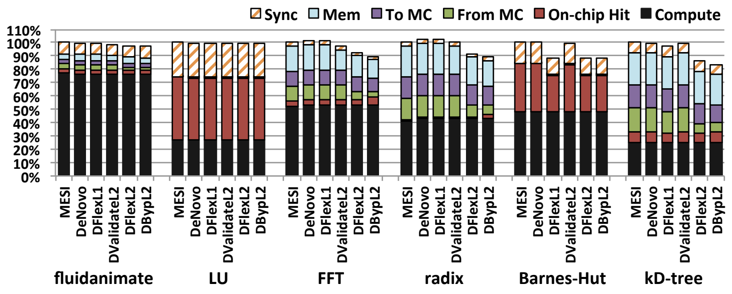
A. Overall Results

Figure 2 provides a high-level overview of our experimental results. It shows the network traffic (part (a)) and execution time (part (b)), normalized to $MESI$, for the protocols studied. The network traffic (Figure 2b) is measured

[2]We did not model an L3 level cache (common in today's microprocessors) as our benchmarks have relatively small input sizes to enable reasonable simulation times.

[3]Although several current processors use rings, we chose a mesh network because it is considered more scalable than a ring (e.g., Tilera [31]). We believe our analysis and results will hold qualitatively on other topologies as well although quantitative benefits may differ.

[4]We adjusted component latencies in our system to adhere to latencies similar to Intel's Nehalem [9]. We do not believe that changing these latencies would significantly affect our results since our focus is largely on network traffic waste.


(a) Execution time

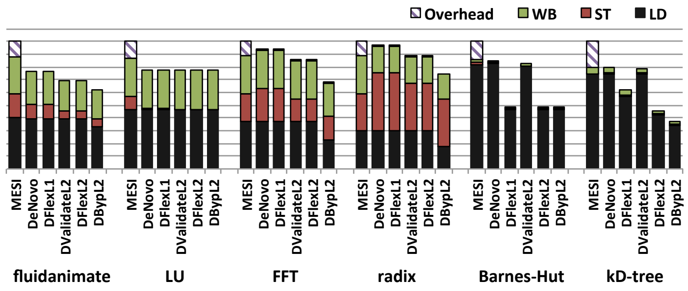

(b) Overall network traffic

Fig. 2: Network traffic and execution time of simulated protocols. All bars are normalized to MESI.

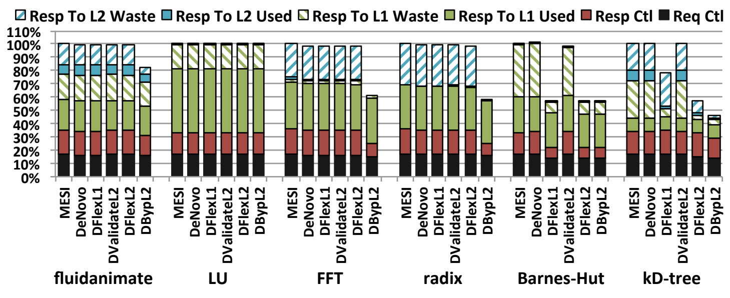
as flit-hops and is divided into four categories: loads, stores, writebacks, and overhead. The first three categories contain only the traffic necessary for the request and the response to complete. The overhead category includes all the traffic required to maintain coherence such as invalidation, ACK, directory unblock, and NACK messages for MESI, and just NACK for DeNovo. Figure 2a divides the execution time into CPU busy or compute time (*Compute*), time spent stalled on hits in the L2 cache or an on-chip remote L1 cache (*On-chip*), time spent stalled on memory hits further categorized as discussed below, and time spent stalled on synchronization (*Sync*). The memory hit time for a request is further split into the time it takes to reach the memory controller from the requesting L1 cache (*ToMC*), the time spent at the memory controller waiting for the DRAM request to complete (*Mem*), and the time from the memory controller back to the L1 cache (via L2 in many cases) labeled as *FromMC*.

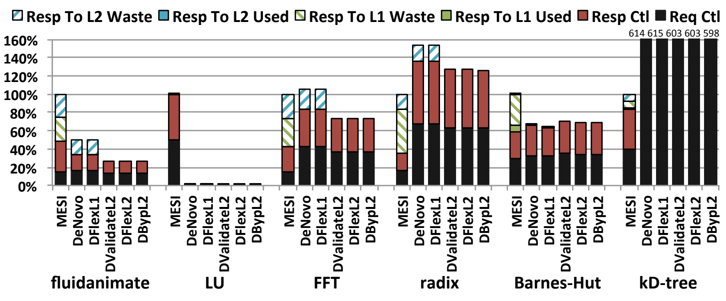
Overall, the results show that, relative to the previous state-of-the-art of $DFlexL1$, the optimizations explored in this paper together are effective at reducing network traffic with the same or improved execution times. $DBypL2$ shows an average 18.2% (range of 0% to 39%) reduction in network traffic relative to $DFlexL1$. The execution time reduction ranges from 0% to 14.5% (6.8% on average). The next section relates these overall results to a more detailed waste analysis.

B. Network Traffic Waste Analysis

Figure 3 shows the network traffic (in flit-hops) separated as load, store, and writeback traffic in parts (a)-(c) respectively, all normalized to *MESI*. For each part, we partition the traffic into flit-hops spent for the control and data portion of the messages. For the load and store traffic graphs, we partition the control traffic further into the request message (*Req Ctl*) and the header of the response message (*Resp Ctl*). For writeback traffic, we merge flit-hops spent for the control portion of the request and response messages into a single category (*Control*). For the data portion, we break the flit-hops spent moving data into several categories based on its message type, i.e., response or writeback data, its source/destination and its usefulness, and use the following labeling: $\{Resp|WB\}$ $\{From\ (Source)|To\ (Destination)\}$ $\{L1|L2\}$ $\{Used|Waste\}$. For easy visual reference, the *Waste* categories are shaded (hatched). As the flits may contain some *Used* and some *Waste* data, we assign fractional flits to the appropriate categories.

Figure 4 reports the various categories of waste from the perspective of the data arriving into the L1 cache (part (a)) and the L2 cache (part (b)) due to loads and stores (as in Section III). The magnitude of the waste at the L1 shown here


(a) LD network traffic breakdown


(b) ST network traffic breakdown

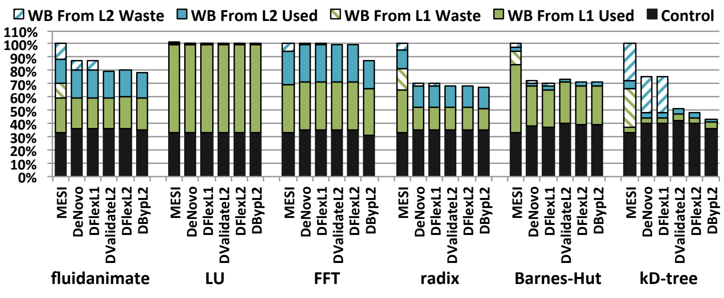

(c) WB network traffic breakdown

Fig. 3: Network traffic broken down by load/store/writeback and further broken down by control and data traffic. All bars are normalized to MESI.

does not correspond exactly to the sum of the appropriate L1 wastes in Figure 3 because the former reports waste in terms of words while the latter reports in terms of flit-hops. Both metrics have value, so we provide them both.

Overall, Figure 3 shows that the optimizations studied virtually eliminate all sources of waste related to the L2 cache (relative to $DFlexL1$). Figure 4 shows that they do this by addressing all forms of waste. For waste related to L1, $DFlexL1$ is very effective for the two applications that it is applicable to, but leaves some L1 waste on the table.

The following discusses each traffic type and optimization in more detail.

1) Load Traffic: As seen in Figure 3a, the major improvements in load traffic come from the Flex and the L2 Bypass Response optimizations. The "FlexL1" optimization, as previously proposed in [7], focuses on eliminating *Resp To L1 Waste*, but it shows mixed performance with the benchmarks. It is highly effective for *Barnes-Hut* and *kD-tree*, but not applicable to the others. On the other hand, "FlexL2" and "L2 Bypass Response" optimizations successfully eliminate *Resp To L2 Waste* for all benchmarks. With both the optimizations, the $DBypL2$ protocol generates on average 23.1% less load traffic than $DFlexL1$, and 32.7% less than MESI. Figure 3a shows that $DBypL2$ almost completely eliminates the L2 Waste in the load traffic for *radix* and *kD-tree*. The "FlexL2" optimization by itself benefits only *kD-tree* by reducing 27.7% of the load traffic compared to *DFlexL1*. Below, we provide detailed account of how each optimization affects the load traffic in terms of waste reduction.

FlexL1: Flex in $FlexL1$ is the only optimization in this paper that targets reducing network traffic destined to L1. Of the benchmarks in this work, only *Barnes-Hut* and *kD-tree* benefit from the Flex for their particular AoS (Array-of-Structures) access pattern. Their main data structures have multiple fields of which different subsets are used only during a certain phase but not in other phases. These fields are small, and many of the useless words share a cache line with the words that are useful in a phase. This results in bringing in many useless words with the fixed cache-line granularity communication. With the Flex, we can avoid sending the unused fields from the on-chip caches and memory as the hardware knows the relative offsets of the words that might be useful to program. For these two applications, the load traffic for $DFlexL1$ is reduced by an average of 32.4% relative to DeNovo.

Relating the above results to detailed characterization of L1 waste in Figure 4a, we can see that the total waste of *Barnes-Hut* and *kD-tree* is reduced by 78.5% and 79.7% repsectively in $DFlexL1$ compared to $DeNovo$. While Flex significantly reduces *Evict* waste by not bringing in the "useless" words destined to be evicted without being used, it also introduces some *Fetch* waste. This is because Flex aggressively brings in data from other cache lines that may already exist in the cache. The remaining waste in DFlexL1 (other protocols studied here do not affect L1 network traffic waste) show in Figure 4a is caused by less predictable access patterns of the applications; *fluid animate* and *LU* have data structures that are conservatively allocated thus not fully utilized in runtime, which can result in *Evict* waste. The remaining waste in *Barnes-Hut* and *kD-tree* are due to certain data being conditionally used. This type of wasted data is very difficult to eliminate without incurring high software and hardware overheads.

FlexL2: $DFlexL2$ reduces the load traffic in the same way as $DFlexL1$, but for the data brought into the L2 instead of the L1. Figure 3a shows that $DFlexL2$ gives additional traffic reduction for *kD-tree* by 27.6% compared to $DFlexL1$. While the both *Barnes-Hut* and *kD-tree* benefits from $DFlexL2$ in reducing *Resp To L2 Waste*, the L2 read miss rate of *Barnes-Hut* is so low compared to the L1 miss rate that it does not show in the graph. The effect of "Flex L2" on L2 waste reduction can be seen more closely in Figure 4b. The figure shows that $DFlexL2$ reduces the total waste by 39.3% and 65% for the both benchmarks respectively compared to $DFlexL1$. Most of the reduced waste is *Evict* waste, while *Barnes-Hut* introduces some *Fetch* waste as described above.

L2 Response Bypass: This optimization proved effective in minimizing the impact that data with poor L2 reuse has by having it bypass the L2. It was applicable to *fluidanimate, FFT, radix*, and *kD-tree* because their data sets greatly exceed the size of the L2. For these benchmarks, $DBypL2$ reduces the amount of load traffic by an average of 23.1% compared to $DFlexL1$.

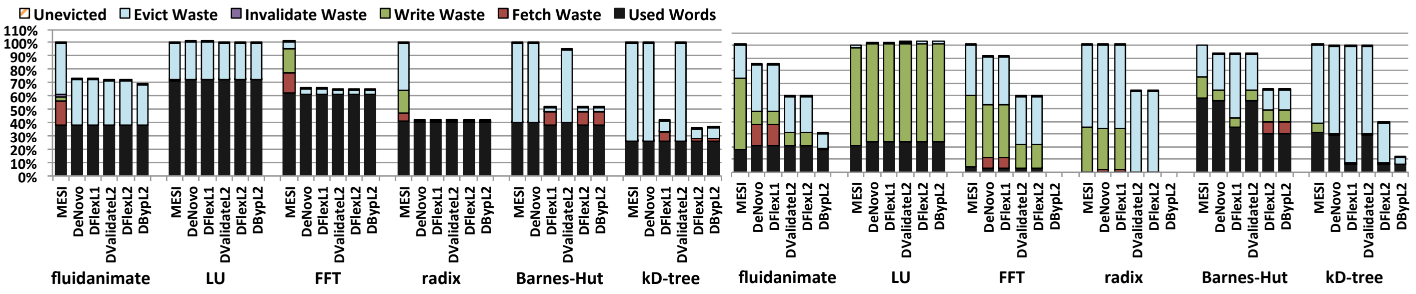
The primary benefit of this optimization is that a smaller amount of data with low to no reuse is inserted into the L2. We observed two types of regions in the benchmarks that can exploit this benefit. The first type are the ones that are read and then overwritten. This benefited *fluidanimate* and *FFT* because their algorithms frequently read and then write the same address, by reducing *Write* waste. The second type of region read only once in the current phase. *FFT*, *radix* can benefit from this because of their matrix operations which do not reuse source arrays, while streaming access pattern of *kD-tree* also benefits from bypassing.

The secondary benefit of bypassing data is that it increases the probability of L2 reuse for data that is not bypassed and that is relatively large and long-lived. For *FFT, radix* and *kD-tree*, bypassing created more space in the L2 for the other data structures, resulting in improved cache hit and reuse rates for them.

Figure 4b shows the waste reduction corresponding to the result in Figure 3a for the four benchmarks. Comparing with $DFlexL2$, $DBypL2$ reduces *Evict* and *Write* waste by 87.8% on average for *fluidanimate, LU, FFT* and *kD-tree*, and up to 100% for *FFT* and *radix*. *FFT* and *radix* do not incur *Write* or *Evict* waste any more by bypassing the L2 for the read-once data and the frequently overwritten data as described above. This shows the value of the optimization for applications with low L2 cache reuse. The streaming access pattern of *kD-tree* also benefits significantly from "L2 Bypass Response" by reducing the *Evict* waste in the L2, while the remaining *Evict* waste is from the randomly accessed array, which makes it virtually impossible to predict the reusability of data. For *fluidanimate*, *Write* waste from overwriting the data without being used is entirely eliminated. The remaining *Evict* waste is due to the unpredictable L2 reuse in *fluidanimate*.

2) Store Traffic: Figure 3b shows that $DBypL2$ successfully eliminates the "Resp To L2 Waste" along with "Resp To L2 Used" traffic for all benchmarks compared to DeNovo or DFlexL1. The improvement is brought out by the "Write Validate" optimization in $DValidateL2$ protocol as described in Section IV. While the MESI protocols suffer from the waste in the store traffic with fetch-on-write write policy as they maintain coherence at cache-line granularity, DeNovo can exploit write-validate write policy for the L1 and L2 caches to eliminate such waste.

Figure 4b shows that $DValidateL2$ reduces the waste for *fluidanimate, FFT* and *radix* by 57.9% on average compared to $DeNovo$. The fetch-on-write write policy is particularly wasteful for benchmarks where either a large amount of data is overwritten or a large amount of spatial data goes



(a) Waste Classification at the L1

(b) Waste Classification at the L2

Fig. 4: Waste breakdown of the words fetched into the L1/L2 caches by the categories presented in Section III. All bars are normalized to MESI.

unused. As described in Section IV, $DValidateL2$ reduces three types of waste: $Evict$, $Fetch$, and $Write$. The major contribution of waste at L2 in *fluidanimate* is *Evict* waste, since the space for unused data slots for its objects will eventually become *Evict* waste as the entire cache line is fetched on a write.

With $DValidateL2$, the *Evict* waste in *fluidanimate* gets reduced to 22.3% compared to $DeNovo$. When issuing a write miss to memory, fetch-on-write write policy will invariably incur *Fetch* waste for the critical word. The "Write-Validate" policy does not fetch the line from memory on a write miss and hence eliminates any such waste. The Figure 4b shows that $DvalidateL2$ completely eliminates the *Fetch* waste. Finally, three benchmarks, *FFT, radix* and *fluidanimate* suffer from high *Write* waste in $DeNovo$, since the spatial data brought in is mostly overwritten without ever being used. Figure 4b shows that $DValidateL2$ reduces *Write* waste by 10%, 54.4%, and 100% for *FFT, radix* and *fluidanimate* respectively by not bringing the data to be wasted. Interesting to note that the remaining *Write* waste is later eliminated by $DBypL2$ for the same benchmark, which let *read* misses to bypass the L2 and not bring in potentially wasteful data to begin with.

Higher store control traffic in DeNovo protocols: We notice that DeNovo protocols show higher store control traffic than MESI for *FFT, radix, Barnes-Hut,* and *kD-tree*. This increase is the result of two design decisions: (1) the lack of an *Exclusive* state in DeNovo, and (2) DeNovo write-combining not being able to combine all registration requests for a single cache line into a single request.

In MESI, the *Exclusive* state allows the cache to silently transition to the *Modified* state on a write, without requiring any extra messages. This silent transition helps in applications like *FFT, Barnes-Hut*, and *kD-tree* where many lines have no other sharers and the lines are read before being written. As DeNovo lacks a similar state, it cannot take advantage of such application behavior and must issue at least one registration request. Secondly, the inability to batch registration requests with the write-combining optimization can also greatly increase the amount of store control traffic. If a program writes to much more different lines in a short period than the write-combining implementation can handle, as in *radix*. the cores have to issue multiple registration requests for the same cache line, in contrast to the single request by MESI.

***LU*'s store traffic anomalies**: The *LU* store traffic shows a couple of oddities: (1) the lack of data traffic for MESI is a result of the majority of store requests being "Upgrade" requests to transition from *Shared* to *Modified* state. (2) DeNovo has a much smaller amount of store control traffic than MESI because many of DeNovo's registration requests are combined with writebacks. This is attributed to the access pattern of *LU* where the L1 working set size is much larger than the L1 size in our experiments resulting in evicting lines soon after writing to them.

3) Writeback Traffic: The "Dirty-Words-Only Writeback" optimization is effective in reducing writeback traffic for DeNovo protocols, as seen in Figure 3c. We see that this optimization applied to L1-to-L2 writebacks ($DeNovo$ and $DFlexL1$) eliminates L2 *Waste* data and when applied to L2-to-memory writebacks ($DValidateL2$ and the rest), eliminates Mem *Waste* data. These two changes reduce writeback traffic by an average of 15.9% and 21.5% relative to $MESI$. It is interesting to note that $DeNovo$ and $DFlexL1$ protocols have lower "Mem Waste" traffic than $MESI$. This is attributed to the extra delay DeNovo has in issuing registration requests to L2 due to write-combining; as the requests can be held for up to 10,000 cycles waiting for more requests to combine, it can cause the lifetime (and the reuse rate) of some lines in the L2 to increase. For MESI, write requests are issued immediately to the L2. In addition, DeNovo uses a non-inclusive L2 cache that helps reduce the number of L1 misses, and therefore, the amount of data replaced to make room for data re-fetched.

4) Overhead Traffic: MESI requires additional network messages to maintain coherence compared to DeNovo, which is categorized as overhead traffic in Figure 2b. These messages account for 13.6% of $MESI$'s traffic on average. For $MESI$'s overhead traffic, 65.3% is spent on directory unblock messages, 26.1% for WB control messages (e.g. clean writebacks), 4.4% on invalidation messages, and 4.3% on acknowledgments. The baseline DeNovo protocol, on the other hand, has a negligible amount of protocol overhead for its only overhead message type, NACKs.

C. Discussion and Future Work

In this section we describe the results of other optimizations mentioned in Section IV and some future directions for our work.

MMemL1: In $MMemL1$, we extend $MESI$ with an optimization that is similar to the "Memory Controller to L1 Transfer" optimization. In a blocking directory implementation of $MESI$, all requests for a line undergoing a transition will be held back or NACKed until the ongoing transition completes. This requires the L1 caches to send

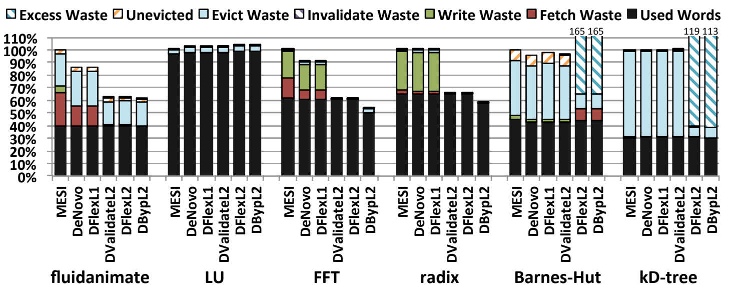


Fig. 5: Breakdown of the words fetched from Memory partitioned by the waste categories presented in Section III. All bars are normalized to MESI.

"unblock" messages to the directory after the L1 finishes its transition. Using this property, it is possible to have the memory controller send the data to the L1 cache first and then have the L1 forward the data to the L2 as a combined "unblock+data" message. Taking this one step further, data that is being fetched from memory on a write does not need to be forwarded to the L2 because an L1 writeback will always overwrite this data. Altogether, these changes can reduce memory stall time and on-chip traffic with little additional protocol complexity. As a result, this optimization reduces the total network traffic by an average 6.2% and the execution time by an average 3.8% compared to $MESI$.

L2 Request Bypass: With the "L2 Response Bypass" optimization, requests for regions with poor L2 locality and L2 bypassing will often miss in the L2. It would be ideal to send requests directly from the L1 to the memory controller, skipping the L2. However, to perform this optimization safely we need to be sure that none of the words that the L1 receives in a response and will use are dirty in another cache.

Our approach is to predict whether the data is dirty on-chip by using two types of Bloom filters. The first type is a counting Bloom filter that is placed at the L2 caches and is used to track the addresses of the cache lines in the L2 cache that have dirty words. The second type of Bloom filter is a non-counting Bloom filter that is placed at each L1 cache. The L1 Bloom filters try to conservatively approximate the state of the L2 caches so it is possible to tell when it is safe to send requests directly to the memory controller. To accomplish this we do the following: (1) clear the L1 Bloom filters at every barrier, (2) copy the L2 Bloom filters into the L1 Bloom filters after the first request that needs the filter, and (3) insert the cache line addresses of every L1 writeback into the L1 Bloom filter. Every L1 load miss for a bypassed region will query the L1 Bloom filter to see if its line address is present. If it is present, then there may be dirty data on-chip and the request needs to be sent to the L2. Otherwise, we know that it is safe to send the request directly to the memory controller. To populate the L1 Bloom filters, we make a Bloom filter copy request to the L2 for the necessary Bloom filter after the first demand miss for that filter. When the response arrives, it is then unioned with the current L1 Bloom filter contents.

As we limit ourselves to sending 64 bytes in a single message, we need to set the maximum size of the Bloom filter to 64 bytes. To reduce the false positive rate, we place multiple Bloom filters at each L2. We set the Bloom filter to be an idealized size to show how effective the technique can be at reducing request traffic. Specifically, we set each Bloom filter to have 512 entries and use one H_3 hash function. Each entry is 1 bit for L1 filters and 8 bits for L2 filters. We set the number of Bloom filters per L2 slice to be 32. Each L1 has Bloom filter storage for all $32 * 16$ L2 Bloom filters. This leads to a storage requirement of $32 * 512 * 16 = 32$KB per L1 cache and $32 * 512 * 8 = 16$KB per L2 slice, for a total of 768KB for our 16 tile processor.

This optimization is only applicable to a subset of memory requests and wherever applicable, only saves one control sized message compared to $DBypL2$. Applying this optimization shows a relatively small average reduction of 5.2% in load traffic for applications with bypassing applied (*fluidanimate, FFT, radix* and *kD-tree*). Improving the accuracy of L1 and L2 Bloom filters can allow more requests to be bypassed, while it may further increase the hardware overhead. However, this optimization provides additional L2 access energy savings that are not quantified in this paper. Quantifying energy benefits of this optimization is part of our future work.

Memory Waste: The graph in Figure 5 shows the effectiveness of all the optimizations in reducing the number of words that need to be fetched from memory. Relative to $MESI$, $DBypL2$ reduces the number of words fetched by 7.7% (or 36.9% if *Excess* Waste is not included). The benefits of $DValidateL2$ and $DBypL2$ are straightforward as their main focus is to reduce the number of cache lines that need to be fetched from off-chip. The increase in memory traffic for protocols with the "L2 Flex" optimization ($DFlexL2$ and $DBypL2$) in some of the applications is caused by the impact that our lack of fine-grained DRAM support has on memory traffic. This results in large *Excess* waste, roughly 60.3% and 66.1% of *Barnes-Hut* and *kD-tree*'s network traffic. This waste is caused by either change of fields that are used from phase to phase (entire cache-line is re-fetched from memory in the new phase) or useful data spanning multiple cache lines does not fit into a single response message (resulting in additional requests to memory). The *Excess* traffic can be reduced if we can incorporate a more sophisticated memory system into our evaluation that supports partial reads similar to the one described in [32] and is part of our future work.

VII. Related Work

In this section, we highlight some of the alternative optimizations that have been proposed towards reducing on-chip network traffic waste. First, we describe alternative MESI-based approaches whose benefits are similar to that of Flex optimization. Next, we describe alternative hardware-only approaches to our "L2 Response Bypass" optimization that have been used to improve L2 reuse. Finally, we list several optimizations that target other types of inefficiencies in the memory hierarchy which are orthogonal to our work.

The main focus of our Flex optimization is to reduce the number of useless words that are returned in response to a cache miss. Others have targeted this form of traffic waste through a variety of hardware and software means. Within hardware-only approaches, previous works [20], [5], [24], [27] have tackled this problem by using predictors that track which parts of a cache line are used before eviction, and would optimistically fetch just those parts on the next cache miss. For these approaches they use a combination of the PC and the cache line offset to index into the predictor table. These approaches are as effective as Flex at reducing the number of unused words, but generally are unable to provide the same prefetching benefits that Flex can provide. Yoon, et al. [32] used the predictor in [5] to reduce both on-chip and off-chip network traffic by adding DRAM support for partial reads and writes. To reduce the impact of partial reads on DRAM throughput and energy, they fetch the entire line when more than half of the line was to be fetched from memory. For future research, addressing the drawbacks in the method proposed by Yoon, et al. [32] may help improve our optimizations to further reduce the amount of data movement.

Other software-enabled approaches similar to Flex include [12], [1]. These approaches used annotations made by the programmer or compiler to fetch either a big or small cache lines. Small lines were shown to be useful for data that is accessed with large strides, indirectly, or involves pointer chasing. In contrast, the Flex optimization can do better than these approaches as it not only fetches just the "useful" portions of a cache line but also prefetches "useful" data from other cache lines.

The focus of our "L2 Response Bypass" optimization is to reduce the number of cache lines with low or no reuse that are placed into the L2. Previous hardware-only approaches, such as those in [2], [8], [16], [22], use hardware predictors that correlate access history or trip counters to the usefulness of caching a block at the different cache levels. These predictors can be used to prevent lines with no reuse from being inserted, and they can also remove a line when it is predicted to become dead. These approaches may have better adaptability but come at an additional cost to the hardware. Our "L2 Response Bypass" optimization differs by just exploiting software information to determine which regions of data should bypass the L2. It does not need support for hardware predictors.

There have also been efficient instructions proposed for uncacheable data; e.g., an Intel SSE4 instruction, MOVNTDQA (Load Double Quadword Non-Temporal Aligned Hint), is used for loading streaming data from Uncacheable Write Combining (USWC) memory [11]. The behavior of such instructions resemble that of our "L2 Response Bypass" optimization. But these instructions are for uncacheable data, so they do not have to deal with the coherence protocol. Also the data cannot be cached in the L1 and thus any benefits of temporal locality are lost.

There is a vast body of work that focuses on other types of inefficiencies in the memory hierarchy such as better placement of data [10], better usage of cache space for reuse data [26], [21], [33], and replacement policies [25], [28] that also have some effect on the amount of on-chip network traffic. All of these optimizations reduce traffic that we classified as *Used* and hence are orthogonal to our work as we specifically focus on ways to reduce on-chip network traffic waste.

VIII. Conclusion

This paper focuses on reducing the on-chip network traffic by analyzing sources of wasted data movement and identifying optimizations that reduce such waste in the network. The sources of waste in commonly used directory-based MESI protocols originate not only from applications, such as poor spatial locality and data reuse, but also from the way current hardware is designed; e.g., false sharing from fixed cache line granularity transfers. This study determines how effective different techniques are in reducing wasted data movement and how much remaining waste there is.

Our evaluation showed that the optimizations explored in this paper can significantly reduce the wasted network traffic for the applications with predictable access patterns. With all the optimizations, we eliminated up to 100% of the on-chip network traffic waste at L2, and 70.8% on average compared to DeNovo. These optimizations are simple and don't incur intrusive modifications to the hardware. More importantly, the detailed characterization of the waste in the paper provides an opportunity to understand how the optimizations and the waste interact more precisely, and how, or if, the optimizations can further eliminate the remaining waste. We believe that this study can work as a useful limits analysis and guideline that the potential techniques for waste reduction can rely on.

References

[1] Deepak Agarwal et al. Exploiting Application-Level Information to Reduce Memory Bandwidth Consumption. In *WCED*, June 2003.

[2] Jorge Albericio et al. The reuse cache: Downsizing the shared last-level cache. In *MICRO*, pages 310–321, New York, NY, USA, 2013. ACM.

[3] Robert L. Bocchino, Jr. et al. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.

[4] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5), 2011.

[5] Chi F. Chen et al. Accurate and Complexity-Effective Spatial Pattern Prediction. In *HPCA*, 2004.

[6] Byn Choi et al. Parallel SAH k-D Tree Construction. In *HPG*, 2010.

[7] Byn Choi et al. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *PACT*, 2011.

[8] Jayesh Gaur et al. Bypass and Insertion Algorithms for Exclusive Last-level Caches. In *ISCA*, 2011.

[9] D. Hackenberg et al. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *MICRO*, 2009.

[10] Nikos Hardavellas et al. Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. In *ISCA*, 2009.

[11] Ashish Jha and Darren Yee. Increasing Memory Throughput With Intel Streaming SIMD Extensions 4 (Intel SSE4) Streaming Load. https://software.intel.com/en-us/articles/increasing-memory-throughput-with-intel-streaming-simd-extensions-4-intel-sse4-streaming-load, April 2007.

[12] Teresa L. Johnson et al. Run-time Spatial Locality Detection and Optimization. In *MICRO*, 1997.

[13] Norman P. Jouppi. Cache write policies and performance. In *ISCA*, 1993.

[14] Robert L. Bocchino Jr. et al. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL*, 2011.

[15] Stephen Keckler et al. GPUs and the Future of Parallel Computing. *IEEE Micro*, September 2011.

[16] Mazen Kharbutli and Yan Solihin. Counter-Based Cache Replacement and Bypassing Algorithms. *IEEE Transactions on Computers*, 2008.

[17] Fredrik Berg Kjolstad and Marc Snir. Ghost Cell Pattern. In *ParaPLoP*, 2010.

[18] Peter Kogge et al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. 2008.

[19] Rakesh Komuravelli. Verification and Performance of the DeNovo Cache Coherence Protocol. Master's thesis, UIUC, 2010.

[20] Sanjeev Kumar and Christopher Wilkerson. Exploiting Spatial Locality in Data Caches using Spatial Footprints. In *ISCA*, 1998.

[21] Snehasish Kumar et al. Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierachy. In *MICRO*, 2012.

[22] Haiming Liu et al. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *MICRO*, 2008.

[23] Milo M. K. Martin et al. Multifacet's General Execution-driven Multi-processor Simulator(GEMS) Toolset. *SIGARCH Computer Architecture News*, 2003.

[24] Prateek Pujara and Aneesh Aggarwal. Increasing Cache Efficiency by Eliminating Noise. In *HPCA*, 2006.

[25] Moinuddin K. Qureshi et al. Adaptive Insertion Policies for High Performance Caching. In *ISCA*, 2007.

[26] Moinuddin K. Qureshi et al. Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines. In *HPCA*, 2007.

[27] Minsoo Rhu et al. A locality-aware memory hierarchy for energy-efficient gpu architectures. In *MICRO*. ACM, 2013.

[28] Daniel Sanchez and Christos Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *MICRO*, 2010.

[29] Hyojin Sung et al. DeNovoND: Efficient Hardware Support for Disciplined Non-determinism. In *ASPLOS*, 2013.

[30] Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve. DeNovoND: Efficient Hardware for Disciplined Nondeterminism. *IEEE Micro Top Picks from the Computer Architecture Conferences*, May 2014.

[31] Tilera. Tilera TILEPro64 processor. 2008. http://www.tilera.com/products/processors/TILEPRO64.

[32] Doe Hyun Yoon et al. The Dynamic Granularity Memory System. In *ISCA*, 2011.

[33] Lixin Zhang et al. The Impulse Memory Controller. *IEEE Transactions on Computers*, 2001.