ELIMINATING ON-CHIP TRAFFIC WASTE: ARE WE THERE YET?

BY

ROBERT SMOLINSKI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Professor Sarita V. Adve

ABSTRACT

As technology continues to scale, the memory hierarchy in processors is predicted to be a major component of the overall system energy budget. This has led many researchers into focusing on 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.

To answer these questions, this thesis systematically analyzes the on-chip traffic for six applications. For this study, we use an alternative hardware-software coherence protocol, DeNovo, and apply several optimizations to DeNovo to quantitatively show the traffic inefficiencies of a directory-based MESI protocol. With a fully optimized DeNovo protocol, we can remove most of the traffic inefficiencies caused by poor spatial locality, fetch-on-write write policy, poor L2 reuse, and MESI protocol overheads. In our discussion of these improvements, we highlight the software causes of these traffic overheads in order to generalize the results.

The final DeNovo protocol with all optimizations applied is able to reduce network traffic by an average of 39.5

For my family

ACKNOWLEDGMENTS

I would like to thank my advisor, Sarita, for her support, advice, and patience throughout my graduate schooling. I am also grateful for the guidance and camaraderie of the RSIM group members: Pradeep Ramachandran, Siva Kumar Sastry Hari, Byn Choi, Hyojin Sung, Rakesh Komuravelli, Matt Sinclair, and Radha Venkatagiri. Lastly, I am thankful for the support that my family and friends have shown me.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Computer performance is currently limited by energy-efficiency [5, 16, 10, 19]. As such, future GPGPUs and embedded devices are expected to need a 10x energy-efficiency improvement [16, 9], and exascale-level supercomputers are predicted to need a 100x energy-efficiency improvement over 2008 technology levels [19].

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 these future energy goals [5, 19, 16]. Current predictions expect that the energy cost of moving a single bit of data from main memory to a core will be as much as a double-precision fused-multiply add operation [19]. Even moving cache lines across the processor will require a significant amount of energy [16]. As these predictions already account for near-future technology improvements, improving the energy-efficiency of the memory system will rely on design improvements that effectively reduce data movement.

Towards this end, this thesis focuses on determining how much of the on-chip network traffic is unnecessary and how it can be removed. The starting point of this study is a standard multicore system that uses a directory-based MESI protocol. 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 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 an inclusive last-level cache, and traffic used to maintain protocol correctness (e.g. invalidation and "directory unblock" messages).

In order to show the network traffic reductions that are possible, we study an alternative hardware-software protocol named DeNovo [8] and several optimizations applied to DeNovo. The primary difference between DeNovo and MESI is DeNovo's use of software-level information and word-level coherence. These properties allow for DeNovo to be a simpler and more extensible protocol 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. To target other forms of waste, DeNovo can be extended with several optimizations. The optimizations that we apply in this work include: the Flex optimization introduced in [8] to reduce traffic for applications with poor spatial locality, 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 last-level cache pollution, and a novel 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 no additional protocol complexity.

Lastly, we apply one additional optimization to both MESI and DeNovo. This optimization sends data directly from the on-chip memory controller to the requesting L1 cache instead of sending it to the last-level cache first. This change improves performance because it reduces the response latency of memory requests, and it can also be used to reduce traffic for MESI since less data needs to be sent from the L1 to the last-level cache. This optimization, and not the others, is applied to MESI as well because it can be done without adding additional protocol complexity.

To show the benefits of these changes, our results section provides a detailed analysis of the network traffic for six benchmarks. First, we define a profiling methodology to classify data movement as belonging to one of several waste categories. These categories help us isolate the data movement that was unuseful and provide a reason for why it was unuseful. Secondly, we create graphs for each of main components of the network traffic (load, store, writeback, and overhead) and discuss the effects that our protocol changes have on each of these components. Lastly, we present graphs that show how many words are brought into each level of the memory hierarchy. These graphs are used to focus on data movement without concern for the network topology. Along with these graphs, we also include explanations for why we are unable to continue to reduce the amount of data movement.

Overall, the DeNovo protocol with all optimizations applied is able to reduce the amount of on-chip

traffic by an average of 39.5% and the number of words fetched from memory by an average of 7.6% (or 36.8% if we are able to selectively fetch data from memory) over the baseline MESI protocol. In addition to these bandwidth savings, we can reduce execution time by an average of 10.5%.

1.1 Contributions

The specific contributions this thesis makes are:

- We provide a comprehensive analysis of the on-chip network traffic for six benchmarks running on a multicore processor, identifying several sources of waste and overhead in a standard MESI protocol implementation.
- We show that Flex can provide significant traffic savings for two of our benchmarks as a result of sending application-specific portions of the cache lines.
- We show that some benchmarks have poor reuse of the data placed in the last-level cache. In these benchmarks, a significant number of cache lines are not reused because they are either overwritten by a writeback, or evicted as a result of the working set being too large. Using our last-level cache bypass optimization, we are able to drastically reduce the number of cache lines that are placed into the last-level cache unnecessarily.
- 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.

CHAPTER 2

BACKGROUND

The DeNovo coherence protocol [8] is a new hardware-software co-designed coherence protocol. Unlike other hardware coherence protocols, DeNovo maintains coherence at word-level instead of line-level. In the initial work on DeNovo, DeNovo requires the software to be data-race free and use only fork-join parallelism. These design choices allow for sharers-lists and invalidation messages to be replaced with self-invalidation instructions that invalidate stale data at barriers. Additionally, DeNovo uses the last-level cache to store per-word ownership (referred to as registration) so that false sharing can be removed entirely. Altogether, these properties allow the baseline DeNovo protocol design to be drastically simpler to verify and extend than other directory-based protocols since it lacks transient states.

To give a brief overview of the design points of the protocol, below is a summary of the relevant points presented in [8].

- Program data can be partitioned into regions which can be used to make self-invalidations more precise. Based on support that the language and compiler can offer, as demonstrated in the Deterministic Parallel Java project [4, 15], program data can be labeled as belonging to one of a small number of regions. These regions can be used to label all loads and stores with the region that it belongs to, and the hardware can store the region numbers along with the data in the L1 cache. With these regions, self-invalidations can invalidate just the data that is in a specific region, instead of all the data in the cache.

- Private caches need to register every word they write with the last-level cache. For the correct data to always be returned, the last-level cache needs to know where to find the most up-to-date version of an address. In the DeNovo protocol, this requirement is satisfied by having the private caches send an registration request to the last-level cache for every address that the core writes. 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 a directory.

- The coherence granularity is decoupled from the data transfer granularity. In traditional protocol designs, data is moved in cache line increments of 64 or 128 bytes, generally. 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 include words from different cache lines into the same response message. This benefit of this feature is that it allows the hardware to reduce the amount of on-chip traffic and improve performance.

Flexible Communication Granularity (Flex): With the above properties, Choi, et al. [8] introduced an optimization named "Flexible communication granularity", abbreviated as Flex. The aim of Flex is to allow the hardware to provide the performance and energy benefits that a software-level array-of-structs to a struct-of-arrays transformation would provide. Many applications can benefit from this optimization because compilers and programmers will forgo performing this transformation if it is too difficult or would impact maintainability.

With the Flex optimization, the software supplies information about the program's data structures, such as the size of its regions and which regions are used together to the hardware. This information is known as a program's communication regions. These communication regions are stored in a small hardware table at every cache controller. Using these tables, a cache controller can respond to a request for a region with just the data that is specified by its communication region. Using this feature allows the hardware to reduce the number of unused words that are sent in a reply, and it can also improve the program's performance when communication regions span multiple cache lines.

CHAPTER 3

PROTOCOL ENHANCEMENTS

3.1 Enhancements to DeNovo

For this study, we apply five new optimizations to the DeNovo line protocol presented in [8]. Each optimization targets a mostly disjoint portion of on-chip traffic. The implementation for each optimization does not introduce any additional protocol complexity (e.g. transient states) but does require small hardware changes and additional data storage. Since we use a two-level cache hierarchy in our simulations, we describe these optimizations using L2 to be the shared last-level cache and L1 to be the per-core private caches.

L2 Write-Validate: The original DeNovo protocol chose to implement a write-validate write policy [14] 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 memory. For this optimization, we change the L2 write policy to also be write-validate. With this change, the amount of on-chip and off-chip traffic spent moving data that is overwritten or unused can be reduced. Compared to the baseline DeNovo protocol, this change requires an additional coherence bit for each word in the L2 as we now need to track whether the word is invalid.

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 change requires no additional L2 storage overhead as DeNovo already has per-word dirty bits. However, the difficulty with this optimization is that current DRAM designs do not support writing subsets of a cache line to DRAM. Future memory systems could be designed to support partial loads and stores using approaches similar to the one described in [31]. For our simulations, we assume we have support for partial writes to memory.

Memory Controller to L1 Transfer: For L2 misses, the baseline 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 data is sent to both the L1 and L2 in parallel. The main benefit of this optimization is to reduce the memory hit latency, as it does not guarantee that the total number of flit-hops will be smaller.

To maintain coherence, this modification requires that the memory controller not send any out-of-date data to the requesting core that the core might use. To prevent this from happening, we attach a bit vector to each request from the L2 to the memory controller that marks which words are dirty and should not be returned from memory. The memory controller uses this information to filter invalid words from the data it fetches, and sends the remaining data to both the L1 and L2. A bit vector is also included in the response so that the receiving caches can determine which words are contained in the message. These bit-vectors are small enough to be included inside of the control flit, so the total message size remains the same.

L2 Flex: In [8], the Flex optimization (Section 2) was only applied to requests that hit in a on-chip cache. Any response that came from memory would return the normal cache line. By extending Flex to main memory, we can prevent non-communication region data from being returned from memory and also prefetch communication region data that is on a different cache line.

For our evaluation, we use a conventional DRAM protocol that only allows for cache line sized reads. As such, we have the memory controller use the Flex information to determine which cache lines to fetch from memory and only include the words that are specified by the communication region in the response message. Additionally, we only request lines that lie in the same DRAM row as the critical address because row activation is an expensive operation. With proper DRAM support, such as the design presented in [31], the amount of data brought in from memory could also be reduced. However, adding this support is beyond the scope of this study.

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 wastes the energy moving it to the L2, the energy required to insert it into the cache, and the energy spent evicting other data to make room for the line. For these reasons, it is useful to be able to prevent lines with poor reuse from being sent to the L2 cache.

Our approach to this problem is to allow the programmer or compiler to specify which regions of data should bypass the L2. We found that this ability 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. Removing the first type of access pattern reduces the amount of data going to the L2 that would be overwritten by a processor before its next use. Avoiding the second type of access pattern is good for the cache lines that have long reuse times. In our implementation, we have the programmer tell the hardware which regions should be bypassed. The hardware stores this information in a table that is queried when a load request misses in the L1 cache. Other implementations for deciding what and when to bypass are also possible.

L2 Request Bypass: As a result of the "L2 Response Bypass" optimization, requests for regions with poor L2 locality and L2 bypassing will often miss in the L2. As such, 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.

With the L1 Bloom filters maintained properly, every L1 load miss for a bypassed region will query the L1 Bloom filter to see if the 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 for us to send the request directly to the memory controller. This is sufficient for DeNovo, and not MESI, because: the program is guaranteed to be data-race free, self-invalidations will clear any data that is written in the current phase, and Bloom filters will not return false negatives.

We present the results of this optimization to study how far we can continue to reduce traffic by tackling load request traffic directly. With the current implementation, most Bloom filters with a sufficiently low false positive rate would be fairly large, roughly 32KBs per L1, making it the least desirable of the opti-

mizations presented in this thesis. The parameters and design details of the Bloom filters used in this study are described further in Section 4.4.

3.2 DeNovo Protocols Studied

To provide a baseline for DeNovo we include two protocols included in Choi, et. al. [8]:

DeNovo: The baseline DeNovo line protocol from [8].

DFlexL1: DeNovo with the Flex protocol optimization. Flex is only used for load responses that are from either the L1 or L2 caches.

The following are the protocols that use the new optimizations and extensions to the above baseline DeNovo protocols. To keep the number of protocols low we only use a subset of the combinations of the above optimizations.

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

DMemL1: DValidateL2 with the "Memory Controller to L1 Transfer" optimization.

DFlexL2: DMemL1 with Flex support similar to DFlexL1 and "L2 Flex".

DBypL2: DFlexL2 with "L2 Response Bypass" support.

DBypFull: DBypL2 with "L2 Request Bypass" support.

3.3 MESI Protocols Studied

In this paper, we include two versions of a directory-based MESI protocol. The first is a baseline directory-based MESI protocol included with the Wisconsin GEMS Simulator [24]. The second extends MESI with an optimization that is similar to the "Memory Controller to L1 Transfer" optimization. This protocol, named MMemL1, takes advantage of the blocking directory implementation of the baseline MESI protocol to reduce traffic and improve performance without increasing protocol complexity.

In a blocking directory implementation of MESI, all requests for a line undergoing a transition will be held back or NACKed until the on-going transition completes. This simplifies the protocol at the cost of requiring the L1 caches to send "directory unblock" messages to the L2 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 a L1 writeback will always overwrite this data. Altogether, these changes can reduce memory stall time and on-chip traffic with no additional complexity.

Other optimizations exist for MESI that can provide the same benefits that the Flex and "L2 Response Bypass" optimizations provide for DeNovo. We chose not include them in this study as their improvements would be similar to what the DeNovo optimizations provide, and they increase the protocol complexity of MESI.

CHAPTER 4

METHODOLOGY

4.1 Detailed Waste Characterization

To classify data traffic, we classify each word being moved into one of six categories: *Used*, *Write*, *Fetch*, *Invalidate*, *Evict*, and *Unevicted*. Below, we describe each category and then show the decision diagrams for how the data being fetched into the L1 and L2 caches and from memory is categorized.

- *Used*: a word which was useful to the program because its value was read, or useful to the L2 because it was returned as part of a L2 response.
- *Write*: a word that was brought into a cache but overwritten before being *Used*.
- *Fetch*: a word brought into a cache where the word was already present. This happens when the cache has the word as dirty from a previous write, or when the word was brought in on a previous fetch.
- *Invalidate*: a word that was invalidated by the coherence protocol before being *Used*.
- *Evict*: a word that was brought into the cache and 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.

Figure 4.1 shows the decision diagram for data that is brought into the L1 cache. In this diagram, load and store actions occur when a processor issues a load or store instruction for that address. The evict action occurs when the word is deallocated from the cache to make room for incoming data, and the invalidate

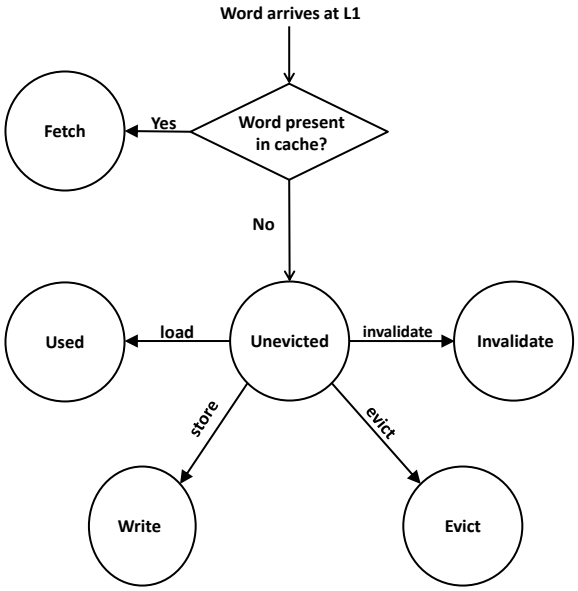


Figure 4.1 Finite state machine for L1 waste profiling.

transitions occur on protocol-specific invalidation actions (e.g. self-invalidation for DeNovo and invalidation messages for MESI).

Figure 4.2 shows the decision diagram for data brought into the L2 cache. For the L2, data becomes *Used* once it is returned to a L1 cache as part of a load or store response, *Evict* when the data is replaced from the L2 cache, and *Write* when it is overwritten by the data included in an L1 writeback.

Figure 4.3 shows the profiling decision diagram for words that are fetched from memory. This metric provides a global view of how much data was brought into the processor from memory and how useful each word was. To track this data, every word fetched from memory is paired with a unique identifier so that it can be profiled separately from words with the same address, internally this is represented as $(address, identifier)$. Keeping track of these separate instances of the word is important for DeNovo because it uses a non-inclusive last-level cache and can have copies of the same word from different requests to memory on-chip at the same time. The edges in Figure 4.3 include the $(address, identifier)$ pair to indicate which reference it is profiling. For stores, we profile all words with $address$ as being *Write* waste when any L1 issues a write because the coherence mechanisms would eventually invalidate or evict all other words with that address that are on-chip.

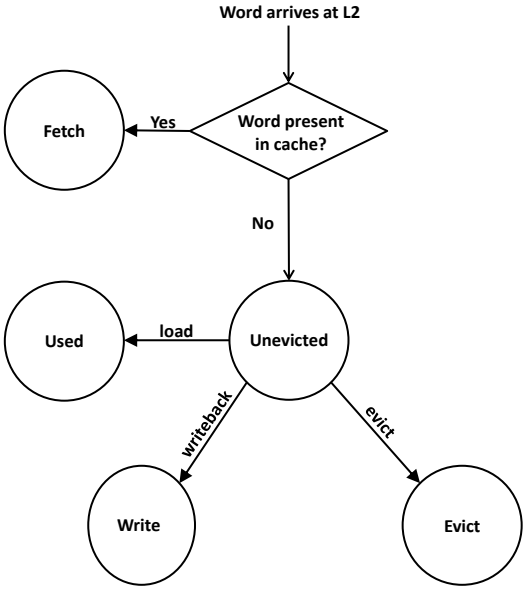


Figure 4.2 Finite state machine for L2 waste profiling.

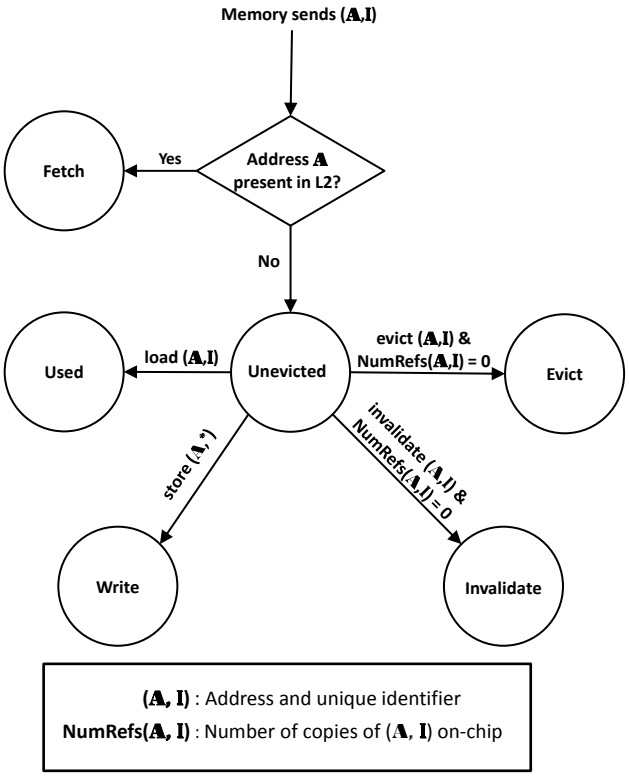


Figure 4.3 Finite state machine for memory waste profiling.

4.2 Simulation Infrastructure and Environment

The simulation infrastructure used in this paper is a combination of the SIMICS full-system simulator [23] used to model each core, the GEMS [24] memory system simulator to model the on-chip memory hierarchy and coherence protocols, Garnet [2] to model the on-chip network, and DRAMSim2 [28] 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.

The MESI protocols are from the GEMS project and have been modified to support non-blocking writes. In these simulations, we allow for there to be up to 32 pending write requests per core.

For our baseline DeNovo protocol we use an implementation that is identical to the design detailed in Choi, et. al. [8] except that it has been extended with a write-combining optimization. The write-combining optimization tries to batch pending registration requests for the same cache line into a single message instead of issuing separate requests for each word that is written. The write-combining optimization is similar to a non-blocking write table where each line with a pending registration request has an entry in a table. We allow for there to be up to 32 entries for these experiments. Each entry has a bit vector to mark which words in the cache line have an registration request pending. Requests are then held in the table until one of the following events occurs: the entire cache line has been written, a 10,000 cycle timeout has been reached, a release instruction has been issued, or the cache line has been evicted from the L1 cache. If a cache line is evicted with outstanding registration requests pending, we send two messages. One message is for words that only need to be written back, and the other message is a combined writeback and register message for words that have not yet been registered with the L2.

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. Each corner tile has a memory controller connected to a single channel DIMM. The tiles are connected by an on-chip mesh network 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 4.1 lists the specific design parameters used for each of these components.

Component	Parameters
Core	2Ghz, in-order
L1D Cache (private)	32KB, 8-way set associative, 64 byte cache lines
L2 Cache (shared)	256KB slices (4MB total), 16-way set associative, 64 byte cache lines
Network	Mesh network, 16 byte links, 3 cycle link latency
Memory Controller	FR-FCFS scheduling, open page policy
DRAM	DDR3-1066, 8 banks, 2 ranks

Table 4.1 Simulated system parameters.

4.3 Benchmarks and Simulation Methodology

The benchmarks in this work are FFT, LU, radix, and Barnes-Hut from SPLASH-2 [30], fluidanimate from Parsec [3], and a parallel kD-tree construction algorithm (kD-tree) [7]. Table 4.2 shows the input sizes used in each benchmark. As the DeNovo protocols in this work do not have support for mutexes, the Barnes-Hut tree building phase was sequentialized and fluidanimate was modified to use the ghost cell pattern [18] to share data between threads. Additionally, we use the aligned version of LU to remove the false sharing in LU.

To gather our simulation results, we run each application for a warm-up period and then report the results of a measurement period. For the iterative algorithms (Barnes-Hut, Radix, fluidanimate, and kD-tree), we run one iteration to warm up the caches and measure 1 iteration (3 iterations for kD-tree). As LU and FFT are not iterative, each program has one core read all of the major data structures for the warm-up.

Application	Input Size
fluidanimate	simmedium
LU	512x512 matrix, 16x16 blocks
FFT	256K points
radix	4 million keys, 1024 radix
Barnes-Hut	16K bodies
kD-Tree	bunny

Table 4.2 Application Input Sizes

4.4 L2 Request Bypass Bloom Filters

Our current design for the "L2 Request Bypass" optimization requires the use of two sets of Bloom filters; one set at the L1s, and another set at the L2s. 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. The resulting structure is similar to a cache where the cache line address is used to hash to a specific Bloom filter and used again to perform the normal Bloom filter operations.

For these experiments, 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. The 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.

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. We use per Bloom filter valid bits to track which Bloom filters have been copied. Copying single Bloom filters on-demand, instead of all of them at once, prevents a burst in on-chip traffic that would otherwise occur when the L1 Bloom filters are cleared at barriers.

CHAPTER 5

RESULTS

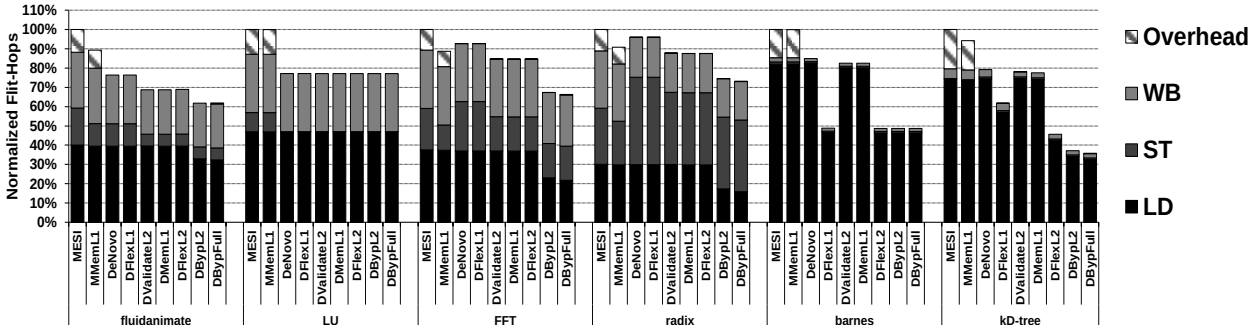
5.1 Overall Results

Figures 5.1a and 5.2 provide a high-level view of the amount of network traffic and execution time for all protocols studied (Section 3), normalized to MESI.

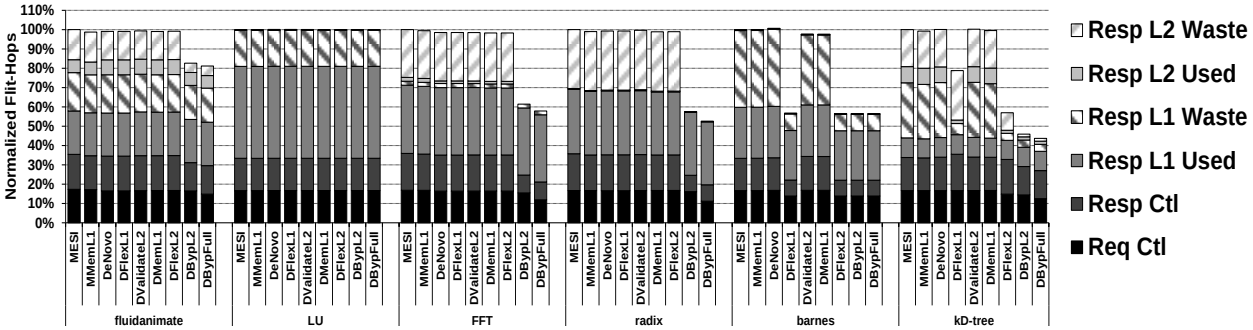
For our network traffic graph (Figure 5.1a), we categorize traffic, measured as flit-hops on the y-axis, into four categories: loads, stores, writebacks, and overhead. The first three categories only contain the traffic for the necessary request and response to complete. For MESI, the overhead category includes all messages that are used to maintain coherence such as invalidations, acknowledgments, directory unblock messages, and NACKs. For DeNovo, the overhead category includes just NACKs and bloom filter request and responses.

Figure 5.2 divides the execution time into CPU busy time (*Busy*), 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*.

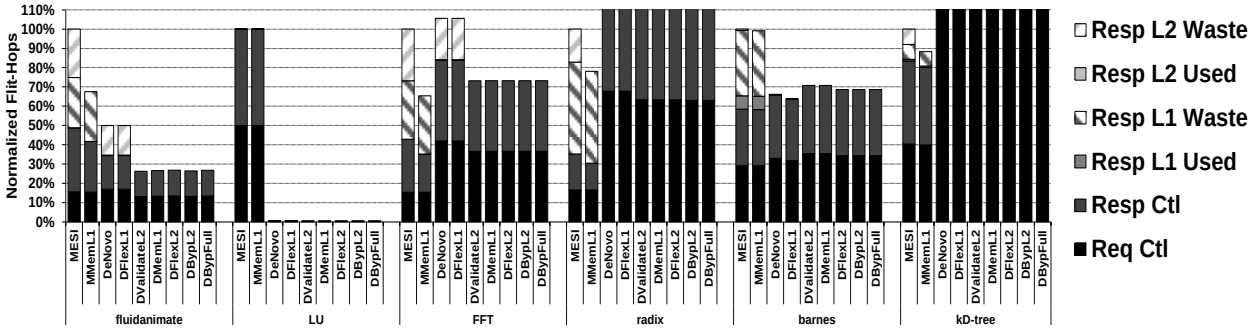
Overall, the figures show that the optimizations included in this work are highly effective at reducing network traffic without losing performance. Specifically, the fully optimized protocol (DBypFull) shows an average reduction of 39.5% (ranging from 22.9% to 64.2%) in network traffic relative to MESI and an average reduction of 35.2% (ranging from 19.4% to 62.0%) relative to MMemL1. Compared to the state-of-the-art DeNovo protocol, DFlexL1, the DBypFull reduces traffic by an average of 18.9% (ranging from



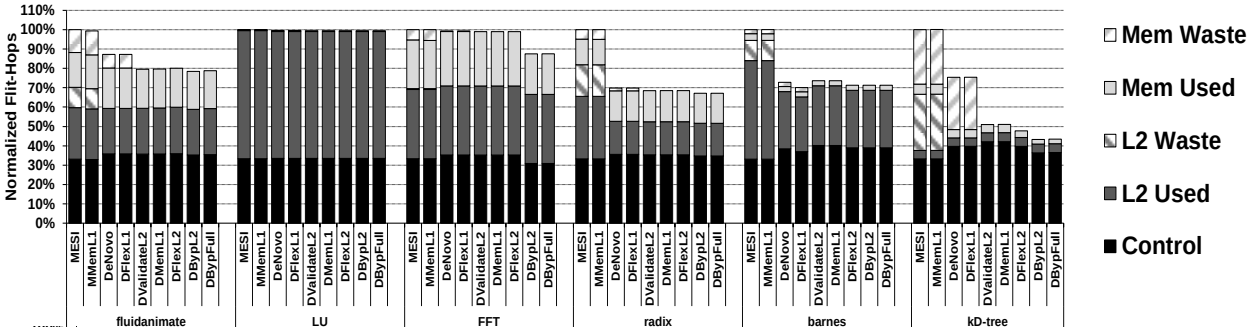
(a) Overall network traffic



(b) LD network traffic breakdown



(c) ST network traffic breakdown



(d) WB network traffic breakdown

Figure 5.1 Network traffic broken down by load/store/writeback and further broken down by control and data traffic. All bars are normalized to MESI.

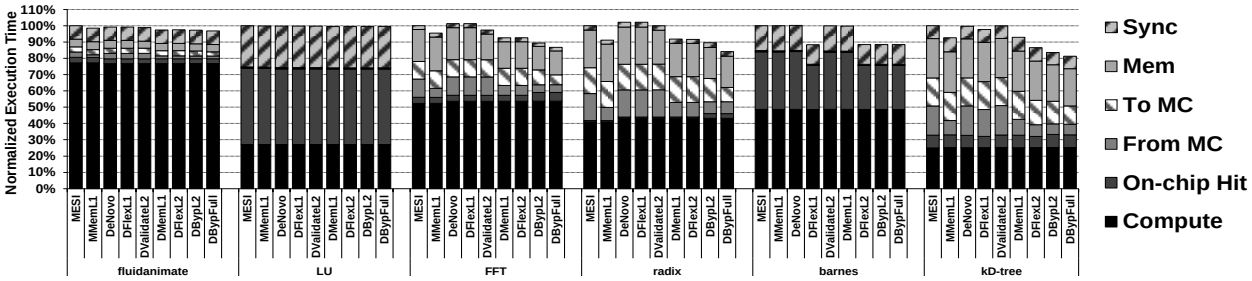
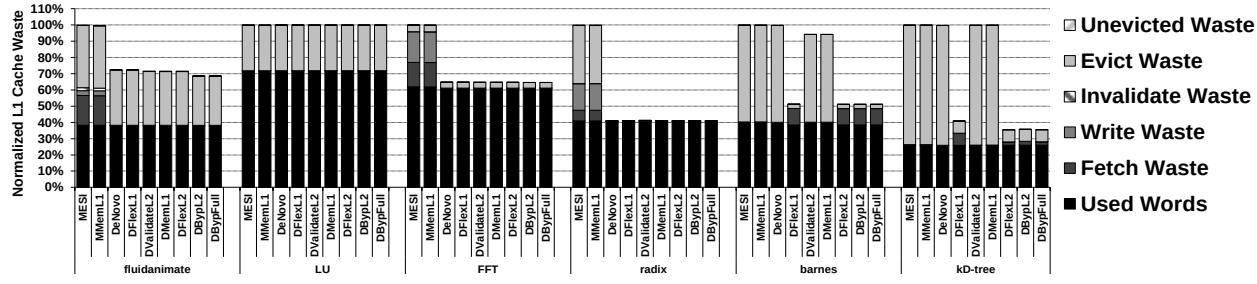
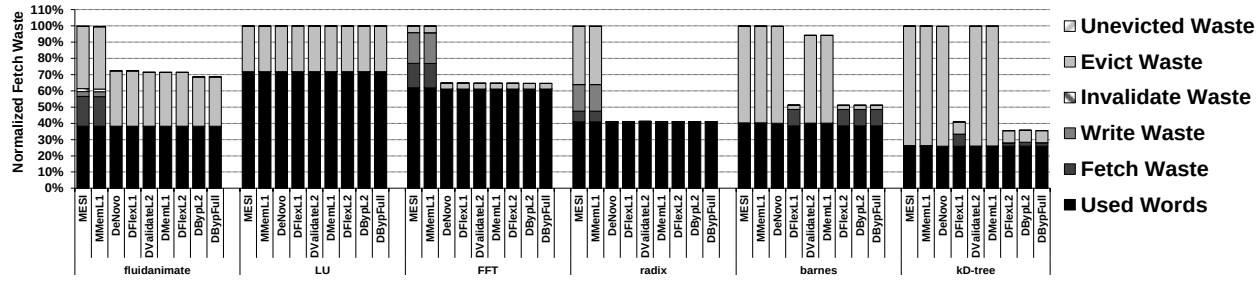


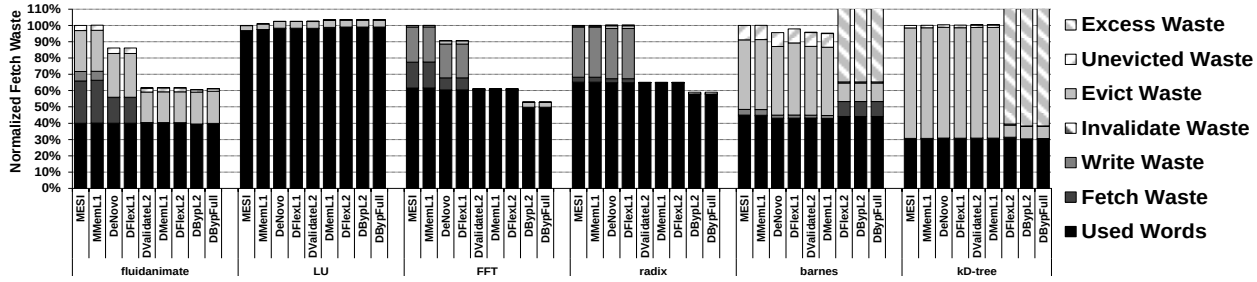
Figure 5.2 Execution Time.



(a) L1 Fetch Waste



(b) L2 Fetch Waste



(c) Memory Fetch Waste

Figure 5.3 Breakdown of the words fetched into the L1/L2 caches or from Memory partitioned by the waste categories presented in Section 4.1. All bars are normalized to MESI.

0.0% to 42.0%).

The "L2 Response Bypass" optimization, in particular, provides significant benefits for four of the six applications. Relative to DFlexL2, DBypL2 reduces traffic by an average of 16.1% for applications that can be bypassed– fluidanimate, FFT, radix, and kD-tree. The "L2 Request Bypass" optimization reduces traffic and execution time slightly. The traffic benefit that DBypFull provides is smaller than the benefit that DBypL2 provides because it is only applicable to a subset of memory requests and it only saves one control sized message. However, DBypFull provides additional L2 access energy savings that are not quantified in these results. The MemL1 optimization does not affect traffic for DeNovo, but the equivalent optimization in MMemL1 reduces the amount of network traffic by an average of 6.2% compared to MESI.

In terms of execution time, the DBypFull protocol reduces execution time by an average of 8.6% relative to DFlexL1. For MESI, MMemL1 provides an average execution time reduction of 3.8% over MESI. Comparing the MESI and DeNovo configurations, the DBypFull protocol reduces the average execution time by 10.5% and 7.1% over MESI and MMemL1, respectively.

The next section covers the savings in network traffic in-depth by breaking apart the traffic results into each of its component categories and describing how each optimization affects traffic. The following section provides a look at the number of words fetched by each level of the memory hierarchy and explains the reasons why traffic waste is still a problem.

5.2 Network Traffic Breakdown

To break network traffic down further we discuss each of the major categories separately. For the load and store traffic graphs we partition the traffic into the flit-hops spent on for the request message (*Req Ctl*), the flit-hops spent for the header of the response message (*Resp Ctl*), and breakdown flit-hops spent moving data into several additional categories based on its destination and its usefulness. For data traffic we use the following labeling: *Resp* $\{$*Destination = L1 | L2* $\}$ $\{$*Used | Waste*$\}$.

For example, *Resp L1 Used* represents data flit-hops destined for a L1 cache and the word was *Used* (as determined by the methodology in Section 4.1). Traffic marked as *Waste* is for words that were profiled as belonging to one of the other categories introduced in Section 4.1. For easy visual reference, the *Waste* categories are shaded. As the flits may contain some *Used* and some *Waste* data, we assign fractional flits to the appropriate categories.

For writeback traffic, we merge flit-hops spent for the control portion of the request and response messages into a single category (*Control*). For data, we break the flit-hops into categories based on its destination $\{ L2 \mid Mem \}$ and whether it contained dirty data (*Used*) or the data was unmodified (*Waste*). Fractional flits are again placed into the appropriate *Used* or *Waste* categories.

We do not breakdown protocol overhead traffic any further. Furthermore, in DeNovo it is possible for messages to not have enough data to fill the last data flit (e.g. the last flit only has 1 word out of 4). In these cases, we include the remaining unfilled fraction of the flit into the *Resp Ctl* category for load and store traffic and the *Control* traffic for writeback traffic.

5.2.1 Load Traffic

Figure 5.1b shows the breakdown of the flit-hops spent on sending load requests and receiving load responses. The major improvements in load traffic are a result of the Flex, L2 Bypass Response, and the L2 Bypass Request optimizations. With all three optimizations, the DBypFull protocol generates on average 34.3% less load traffic than MMemL1, 34.4% less than DeNovo, and 25.2% than DFlexL1.

Flex:

The benefit of the Flex optimization is that we are able to prevent unused structure fields from being returned in a response, and we can combine multiple requests and responses for communication regions that span multiple cache lines. Of the benchmarks in this work, Flex is only applicable to Barnes-Hut and kD-tree as their data structure fields are smaller than a cache line, and the applications mix useful and useless words in the same cache line. For these two applications, the load traffic for DFlexL1 and DFlexL2 is reduced by an average of 32.4% and 43.5%, relative to DeNovo. DFlexL2 only shows additional benefits for kD-tree as kD-tree has a significantly higher number of L2 misses than Barnes-Hut. Below is a description of the properties in each benchmark that Flex can take advantage of.

In Barnes-Hut, the main data structures represent the particles in the system and the nodes in its oct-tree. Each structure contains a large number of fields that are used only during the oct-tree construction phase and also has several bytes of compiler-inserted padding to align the double-words. Due to the data layout, many of these useless words share a cache line with words that are useful for the majority of the application. On top of this, the structures are not padded to be a multiple of a cache line size. This causes the useful data in some objects to be spread across a varying number of cache lines, and causing the number of useless words

that are brought to also increase. With Flex we can avoid both problems as the hardware knows the relative offsets of the words that might be useful to program.

Similarly, kD-tree also benefits from Flex due to small subsets of kD-tree's main data structures being used in the same phase of the application. The two main data structures are an array of triangles that represent a mesh of the scene and an array of edges representing the cubes that would circumscribe each triangle. In the algorithm, the array of edges is accessed in a streaming fashion and the array of triangles is randomly accessed. With Flex, the hardware can avoid sending the unused fields in both data structures from the on-chip caches and memory. For performance, Flex can also be used to prefetch the data in the edges array as it has a predictable access pattern.

L2 Response Bypass:

The "L2 Response Bypass" optimization was designed to minimize the impact that data with poor L2 reuse has by having it bypass the L2. Of these benchmarks, this optimization was applicable to only fluidanimate, FFT, radix, and kD-tree because their data sets greatly exceeded the size of the L2. For these applications, DBypL2 reduces the amount of load traffic by an average of 37.5% compared to MESI and 28.8% compared to DFlexL2.

The primary benefit of this optimization is that a smaller amount of data with low to no reuse is inserted into the L2. The first type of region we bypassed were regions that are read and then overwritten. This benefited fluidanimate and FFT because their algorithms frequently read and then write the same address. The second type of region that we bypass is data that is read once in the current phase. FFT and radix benefited from this because of their respective transpose and permutation operations which read each element of a source array and never read it again. kD-tree also benefited from bypassing its array of Edges because it is large and it behaves like streaming data.

The secondary benefit of bypassing data is that it increases the probability of L2 reuse for data that is not bypassed. For example, FFT and radix create a destination array during their respective transpose and permutation operations and then use the array in the next phase. In kD-tree, its array of Triangles is randomly accessed for most of the application. By bypassing kD-tree's array of Edges, the randomly accessed array has more L2 cache space to be held in and thus a greater chance of reuse.

L2 Request Bypass:

The last optimization that improves load traffic is our novel "L2 Request Bypass" optimization. This

optimization reduces the amount of request control traffic by sending requests for regions being bypassed with the "L2 Response Bypass" optimization directly to the memory controller. With this optimization, DBypFull shows an average reduction of 5.2% in load traffic relative to DBypL2 for applications with bypass applied (fluidanimate, FFT, radix, and kD-tree).

The traffic benefits of this optimization are smaller than "L2 Response Bypass" as it relies on being able to accurately predict whether it is safe to have the request skip the L2. Additionally, the optimization saves only one flit when it is applicable, instead of the five that "L2 Response Bypass" can save. Improving these numbers further requires reducing the false positive rate of the L1 Bloom filters. This could be achieved by either increasing the size of each Bloom filter, the number of Bloom filters per L2, by copying the state of the L2 cache more frequently, or changing the approach entirely.

5.2.2 Store Traffic

Figure 5.1c shows the store traffic broken down into its component parts. This section covers the benefits that the MMemL1 and write-validate protocols offer in reducing store traffic, and it also explains why DeNovo has higher store control traffic for some benchmarks.

MemL1:

With the MMemL1 protocol, MESI no longer needs to send data to the L2 that is guaranteed to be overwritten. Specifically, this prevents the data that is returned on a L2 write miss from going to the L2. Using this optimization eliminates "Resp L2 Waste" from MESI's store traffic, for an average savings of 16.9%.

Write-validate:

For these protocols, we have three versions of write-policy: the MESI protocols which have fetch-on-write throughout, DeNovo and DFlexL1 have write-validate for L1 writes and fetch-on-write for L2 writes, and the remaining DeNovo protocols use a write-validate throughout. As Figure 5.1c shows, the clear benefit of the write-validate write policy is that "Resp L1 Waste" and "Resp L1 Used" are eliminated with L1 write-validate enabled, and "Resp L2 Waste" and "Resp L2 Used" are eliminated with L2 write-validate enabled.

The fetch-on-write write policy is particularly wasteful for these benchmarks because of the large amount of data that is overwritten (*Write* waste) and the large amount of spatial data that goes unused

(*Evict* waste). *Write* waste is an issue for FFT and radix as both overwrite their destination arrays while performing their respective transpose and permutation phases. *Write* waste is also an issue for fluidanimate as large amounts of data is overwritten during an array-to-array copy and when some accumulators need to be zeroed.

Evict waste becomes a problem for the fetch-on-write policy in radix and fluidanimate. In radix, the permutation phase can randomly write to 1024 different cache lines, which is greater than the L1 cache size. For radix, if a line is evicted before the entire line is overwritten, the line will need to be refetched at least once more. When this occurs, at least one cache line worth of *Evict* waste is generated. For fluidanimate, a significant amount of the waste is the result of its objects being designed to hold up to 16 different particles. As the majority of objects are not fully filled, a dynamic amount of the preallocated space for each field of the object goes unused in the current iteration of the algorithm. As the entire cache line is fetched on a write, the extra unused space is also brought into the caches and will eventually become *Evict* waste.

Increase in DeNovo Store Control Traffic:

The noticeable exception to DeNovo's improvements is the increase in store control traffic for four benchmarks (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. This is a large problem for radix as the current implementation of write-combining is limited to a maximum of 32 registration requests for different lines. In radix's permutation phase, each thread may be write to 1024 different lines before it is finished a line. This mismatch in size causes the core to issue multiple registration requests for the same cache line, instead of the single request that MESI would require to gain ownership for the entire line.

LU Store Control Traffic:

There are several oddities in the LU store traffic for MESI as shown by the lack of data traffic and a large amount of control traffic. The lack of data traffic for MESI is a result of the majority of store requests being "Upgrade" requests that are necessary for the cache line to transition from the *Shared* state to the *Modified* state. Upgrade requests occur more frequently in LU than other applications because the lines that will be written to are not returned in the *Exclusive* state as there is often a previous sharer. Additionally, DeNovo has a much smaller amount of store control traffic than MESI because many of DeNovo's registration requests are combined with writebacks. In the current write-combining implementation, DeNovo sends a combined writeback and registration request when a line is being written back and there are registration requests pending for that line. This combined message is profiled as writeback traffic.

5.2.3 Writeback Traffic

Figure 5.1d shows the writeback traffic for these protocols. The MMemL1 protocol shows little benefit to a baseline MESI protocol as the optimization does not reduce the number of writebacks that will occur. The use of dirty-words-only writebacks for L1 to L2 (DeNovo and DFlexL1) eliminates L2 *Waste* data, and the dirty-words-only writeback for L2 to memory (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.

One interesting difference that the DeNovo and DFlexL1 protocols show compared to MESI is a reduction in "Mem Waste" traffic. This can be attributed to the extra delay that DeNovo has when issuing registration requests to L2 that occurs as a result of the write-combining implementation. For MESI, write requests are issued immediately to the L2. For DeNovo, the request can be held for up to 10,000 cycles as the write-combining unit waits for more requests to combine. This extra delay causes the lifetime of some lines in the L2 to increase. Furthermore, DeNovo uses a non-inclusive L2 which helps reduce the number of L1 misses and, therefore, the amount of data that is replaced to make room for data that is refetched.

5.2.4 Overhead Traffic

One of the differences between MESI and DeNovo is MESI's need for additional messages to maintain coherence. These messages account for 13.6% of MESI's traffic and 12.1% of MMemL1's traffic, on average. For MESI's overhead traffic: 65.3% is spent on directory unblock messages that are used to finish transitions, 26.1% for WB control messages (e.g. clean writebacks), 4.4% on invalidation messages, 4.3%

on acknowledgments. MMemL1 reduces overhead traffic by an average of 15.8% because of the directory unblock messages can be turned into unblock+data messages that are profiled as load traffic. The baseline DeNovo protocol, on the other hand, has a negligible amount of protocol overhead spent on its only overhead message type, NACKs.

The only significant network overhead for the DeNovo protocols is introduced with the "L2 Request Bypass" optimization. For DBypFull, the overhead is the result of needing to copy the L2 bloom filters into the L1. The traffic to request and receive a copy of the Bloom filter is an average of .5% of DBypFull's traffic for fluidanimate, FFT, radix, and kD-tree. However, this increase in overhead is made up for by a larger reduction in load request traffic.

5.3 Fetch Waste Discussion

In this section, we provide three graphs that show the breakdown for the wasted words being fetched by the L1 (Figure 5.3a), L2 (Figure 5.3b), and from memory (Figure 5.3c), and also detail why the remaining wasted words in DBypFull are difficult to remove. The height of each bar is the number of words that each protocol fetches, normalized to MESI. The words in each bar are partitioned into the categories presented in Section 4.1. One additional category, *Excess* waste, is added in Figure 5.3c to show how much data is being dropped at the memory controller as a result of the "L2 Flex" optimization.

L1 Fetch Waste:

As Figure 5.3a shows, DBypFull successfully reduces the number of words being brought into the L1 caches by 39.8% relative to MESI, and 3.14% relative to DFlexL1. The reasons for these gains have already been been described in Section 5.2.

The largest remaining challenge in eliminating wasted traffic in DBypFull is the data that is accessed with irregular access patterns. In fluidanimate, each object can hold 16 particles but most of the objects have far fewer than 16. This results in many cache lines having unused space at the end of the cache lines, causing *Evict* waste. The waste in LU is caused by accessing the upper triangular component of the blocks being operated on. The waste in Barnes-Hut is a combination of *Fetch* and *Evict* waste. The *Fetch* waste is a result of Flex returning data for a different cache line that is already present in the cache. The *Evict* waste is the result of some fields in its data structures being conditionally used. Lastly, in kD-tree one of the data structures contains three pairs of pointers. Which pair, or pairs, of pointers will be used depends on dynamic

conditions. Removing any of these forms of waste by being more conservative with the data that Flex brings in would worsen performance, so we opted to continue to fetch this data.

L2 Fetch Waste:

In Figure 5.3b we show the amount of data being brought into the L2 from memory. The DBypFull protocol reduces the amount of data fetched into the L2 by 63.7% relative to DeNovo, 65.0% relative to MESI, and 61.6% relative to MMemL1. The reasons for these improvements were covered in Section 5.2.

Improving the remaining L2 waste further is difficult because it is determined by unpredictable L2 reuse. In fluidanimate, a stencil computation is performed on a grid being traversed in X-Y-Z order. Since the traversal is not blocked, there is a large disparity in the L2 reuse time for the grid elements. Our Bypass mechanism lacks the fine-grained ability to specify this variance in reuse time. In kD-tree, the unused data is from an array that is randomly accessed, which makes reuse hard to accurately predict. Lastly, the input sizes for LU and Barnes that we use in our experiments have very small L2 working sets which make it difficult to bypass data as there is little opportunity.

Memory Fetch Waste:

The last graph, Figure 5.3c, tracks the number of words that are fetched from memory. This graph is included to show the benefits that the write-validate, bypass, and Flex optimizations can have in reducing the number of words that need to be fetched from memory. Relative to MESI, DValidateL2 reduces the number of words fetched by 18.9%, DFlexL2 by 4.1%, and DBypL2 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 can be explained by the impact that our lack of fine-grained DRAM support has on memory traffic. With our line-based memory system, roughly 60.3% and 66.1% of Barnes-Hut's and kD-tree's memory traffic is *Excess* waste. For Barnes-Hut, the cause of this waste can be attributed to the change in fields that are used from phase to phase in the application. As the L2 is relatively large, the fields that are unused in one phase of the application are likely to survive in the L2 until the next application phase where it will become useful data. With the "L2 Flex" optimization applied, we need to re-fetch those parts from memory. For kD-tree, the ability to prefetch data with Flex works against itself. As a result of the access pattern for its Edges array and the limit of 64-bytes per network packet, nearly two of every three cache lines that contain data for the Edges

array will be read twice from memory because the useful data that is in all three lines cannot fit into a single response message. Both of these problems could be reduced if we were to incorporate a more sophisticated memory system into our evaluation that supported partial reads similar to the one described in [31].

CHAPTER 6

RELATED WORK

In this section, we highlight some of the alternative optimizations that have been proposed towards reducing network on-chip traffic. First, we describe alternative MESI-based approaches that provide many of the same network traffic savings that our Flex optimization provides. 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 other types of traffic saving optimizations that we have not included in our study.

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, 6, 25] 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. Work by Yoon, et al. [31] used the predictor in [6] to reduce on-chip and off-chip network traffic by adding DRAM support for partial reads and writes. In their work, they took the impact that partial reads will have on DRAM throughput and energy into consideration. This consideration caused Yoon, et al. to use a policy where the entire line would be fetched when throughput or energy were hurt, most notably when more than half of the line was to be fetched from memory. As including a memory system that can perform partial reads and writes is essential to some of our optimizations, future research should continue to remove the drawbacks in method proposed by Yoon, et al. [31] to continue reducing the amount of data movement.

Other software-enabled approaches similar to Flex include [13, 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. Flex can be used to target the same access patterns.

The focus of our "L2 Response Bypass" optimization was 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 [11, 17, 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. One benefit of the hardware predictors over our optimization is that they automatically adapt to working set size changes and high contention in the last-level cache. Further study is necessary to determine how much additional benefit can be gained from hardware-only predictors beyond what is possible with our bypass optimization.

Optimizations that we did not explore in this work generally fall into one of three categories. The first category attempts to optimize the placement of the last-level cache's data. One project, R-NUCA [12], places all cache lines for non-shared virtual pages into the tiles that are adjacent to the core that uses the data. By modifying the placement, the average distance that requests and responses need to travel can be reduced. The second category focuses on reusing cache space that would hold unnecessary data in normal system. Example approaches include [27, 21, 32]. With these optimizations, the focus is to preserve the useful data that is cached by reorganizing either how the data is stored or how it is addressed. Lastly, many approaches, such as [26, 29], have focused on improving the insertion and replacement policies so that lines with low reuse will be evicted earlier. Including one or more of these types of optimizations will be an important part of continuing to reduce DeNovo's traffic as these optimizations all reduce traffic that we classified as *Used*in this work.

CHAPTER 7

CONCLUSION

In this thesis, we focused on reducing on-chip network traffic by analyzing and targeting sources of wasted data movement. The sources of waste in a commonly used directory-based MESI protocol, such as false sharing, poor spatial locality, and protocol overhead, are well-known. However, no study, to the best of our knowledge, has determined which techniques are effective in reducing wasted data movement or how much remaining waste there is.

To answer these questions, we evaluated the network traffic for an alternative coherence protocol, DeNovo, which eliminates several of MESI's weaknesses like false sharing. While the baseline DeNovo protocol reduces the amount of traffic by an average of 13.9% relative to MESI, several additional optimizations can be applied to DeNovo to reduce network traffic further. These changes included: using a write-validate write policy, using the Flex optimization to reduce the amount of useless data sent from caches and memory, and having data bypass the last-level cache. Additionally, we introduced a novel optimization that is able to safely send requests directly to memory instead of the L2, reducing latency and traffic.

Overall, our DeNovo protocol with all optimizations applied (DBypFull) reduces traffic by an average of 39.5% and 35.2% relative to a baseline MESI protocol and an optimized MESI protocol (MMemL1), and 18.9% relative to the previous best DeNovo protocol (DFlexL1). Of the network traffic in DBypFull, an average of 8.8% of the remaining data traffic is unnecessary because it is accessed with an irregular access pattern. Since we were unwilling to sacrifice performance for traffic improvements in this study, we could not remove this data movement with any of the optimizations used in this study.

REFERENCES

[1] Deepak Agarwal, Wanli Liu, and Donald Yeung. Exploiting Application-Level Information to Reduce Memory Bandwidth Consumption. In *Proceedings of the 4th Workshop on Complexity-Effective Design (WCED)*, June 2003.

[2] Niket Agarwal et al. Garnet: A Detailed Interconnection Network Model Inside a Full-System Simulation Framework. Technical report, 2008.

[3] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A Type and Effect System for Deterministic Parallel Java. In *Proceedings of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications*, 2009.

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

[6] Chi F. Chen, Se-Hyun Yang, Babak Falsafi, and Andreas Moshovos. Accurate and Complexity-Effective Spatial Pattern Prediction. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA'04, 2004.

[7] Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L. Bocchino, Sarita V. Adve, and John C. Hart. Parallel SAH k-D Tree Construction. In *High Performance Graphics (HPG)*, 2010.

[8] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compililation Techniques*, 2011.

[9] DARPA. Power Efficiency Revolution For Embedded Computing Technologies (PERFECT), April 2012.

[10] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[11] Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. Bypass and Insertion Algorithms for Exclusive Last-level Caches. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[12] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[13] Teresa L. Johnson, Matthew C. Merten, and Wen-Mei W. Hwu. Run-time Spatial Locality Detection and Optimization. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO)*, 1997.

[14] Norman P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, 1993.

[15] Robert L. Bocchino Jr., Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2011.

[16] Stephen Keckler, William Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, September 2011.

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

[18] Fredrik Berg Kjolstad and Marc Snir. Ghost Cell Pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, 2010.

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

[20] Sanjeev Kumar and Christopher Wilkerson. Exploiting Spatial Locality in Data Caches using Spatial Footprints. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, 1998.

[21] Snehasish Kumar, Hongzhou Zhou, Arrvindh Shriraman, Eric Matthews, Sandhya Dwarkadas, and Lesley Shannon. Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierachy. In *Proceedings of the 45th Annual International Symposium on Microarchitecture (MICRO)*, 2012.

[22] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *Proceedings of the 41st Annual International Symposium on Microarchitecture (MICRO)*, 2008.

[23] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *Computer*, Vol. 35, February 2002.

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

[25] Prateek Pujara and Aneesh Aggarwal. Increasing Cache Efficiency by Eliminating Noise. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture (HPCA)*, 2006.

[26] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[27] Moinuddin K. Qureshi, M. Aater Suleman, and Yale N. Patt. Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[28] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters*, 2011.

[29] Daniel Sanchez and Christos Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[30] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, 1995.

[31] Doe Hyun Yoon, Min Kyu Jeong, Michael B. Sullivan, and Mattan Erez. The Dynamic Granularity Memory System. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[32] Lixin Zhang, Zhen Fang, Mike Parker, Binu K. Mathew, Lambert Schaelicke, John B. Carter, Wilson C. Hsieh, and Sally A. McKee. The Impulse Memory Controller. *IEEE Transactions on Computers*, 2001.