7-8-2016

Methods for Efficient Data Access and Communication in Many-core Architectures

Farrukh Hijaz
University of Connecticut - Storrs, hijaz.farrukh@gmail.com

Methods for Efficient Data Access and Communication in Many-core Architectures

Farrukh Hijaz, Ph.D.

University of Connecticut, 2016

ABSTRACT

The trend of increasing processor performance by boosting frequency has been halted due to excessive power dissipation. However, transistor density has continued to grow which has enabled integration of many cores on a single chip to meet the performance requirements of future applications. Scaling to hundreds of cores on a single chip present a number of challenges, mainly efficient data access and on-chip communication. Near-threshold voltage (NTV) operation has been identified as the most energy efficient region to operate in. Running at NTV can facilitate efficient data access, however, it introduces bit-cell faults in the SRAMs which needs to be dealt with. Another avenue to extract data access efficiency is by improving on-chip data locality. Shared memory abstraction dominates the traditional small computer and embedded space due to its ease of programming. For efficiency, shared memory is often implemented with hardware support for synchronization and cache coherence among the cores. However, accesses to shared data with frequent writes results in wasteful invalidations, synchronous write-backs, and cache line ping-pong leading to low spatio-temporal locality. Moreover, communication through coherent caches and shared memory primitives is inefficient because it can take many instructions to coordinate between cores.

This thesis focuses on mitigating the effects of the data access and communication challenges

and make architectural contributions to enable efficient and scalable many-core processors. The main idea is to minimize data movement and make each necessary data access more efficient. In this regard, a novel private level-1 cache architecture is presented to enable efficient and fault-free operation at near-threshold voltages. To better exploit data locality, a last-level cache (LLC) data replication scheme is proposed that co-optimizes data locality and off-chip miss rate. It utilizes an in-hardware predictive mechanism to classify data and only replicate high reuse data in the local LLC bank. Finally, a hybrid shared memory, explicit messaging architecture is proposed to enable efficient on-chip communication. In this architecture the shared memory model is retained, however, a set of lightweight in-hardware explicit message passing style instructions are introduced in the instruction set architecture (ISA) that enable efficient movement of computation to where data is located.

Methods for Efficient Data Access and Communication in Many-core Architectures

Farrukh Hijaz

B.E., National University of Sciences and Technology, Islamabad, Pakistan, 2010

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut

2016

APPROVAL PAGE

Doctor of Philosophy Dissertation

Methods for Efficient Data Access and Communication in Many-core Architectures

Presented by

Farrukh Hijaz, B.E.

Major Advisor _____
Omer Khan

Associate Advisor _____
John Chandy

Associate Advisor _____
Marten Van Dijk

University of Connecticut

2016

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my advisor, Prof. Omer Khan, for his support, guidance, and encouragement throughout my PhD studies. I owe him a great deal for the knowledge and invaluable experiences I have gained during the last 5 years. He taught me how to perform research and how to effectively communicate it to others. His feedback and insightful discussions have strengthened my work and have helped me improve as a researcher. I thank him and my group mates for making my PhD experience an enriching and pleasant one.

I would also like to thank my family for their unconditional support and love throughout my life. I would not have been able to complete my PhD without their prayers and encouragement. I owe all my successes to my parents, Hijaz-ul-Mulk and Raeesa Hijaz, and I cannot express with words how grateful I am for all the sacrifices they have made for me and my siblings. Finally, I am grateful to my loving wife, Rabia, for her patience and unlimited support during the all-important last stretch of my PhD studies.

Contents

List of Figures

Page

List of Tables

Page

Chapter 1

Introduction

The trend of increasing performance by boosting frequency has been halted due to excessive power dissipation. However, transistor density has continued to grow which has enabled integration of many cores on a single chip to meet the performance requirements of future applications. Scaling to hundreds of cores on a single chip present a number of challenges, such as scalability and energy efficiency. Tackling these challenges will require rethinking of how processors are designed and programmed.

To enable scaling to a large number of cores on a single chip, two main challenges arise, namely 1) data access, and 2) on-chip communication. Both of these challenges must be tackled to keep the energy consumption under a reasonable limit and enable performance scaling. Shared memory abstraction dominates the traditional small computer and embedded space. Designing a parallel application under shared memory is attractive from the perspective of ease of programming. Furthermore, cache coherence is implemented in the hardware to enable efficient on-chip data access without programmer intervention. However, the non-uniform nature of a data access in a distributed single chip many-core processor makes efficient data access challenging. Moreover,

communication through coherent caches and shared memory primitives are inefficient because they can take many instructions to coordinate between cores. In this thesis I propose novel architectural methods to improve data access and on-chip communication.

1.1 Efficient Data Access

Future processors are expected to operate on massive amounts of data. These many-core processors will operate on application data structures with varying degree of spatio-temporal locality. Moving data around on a chip under conventional cache hierarchies is inefficient, resulting in high memory access latency and energy consumption [1, 2]. Therefore, future processors will be constrained not by their computation capabilities but rather by their *data access efficiency*. Efficient data access encompasses intelligent management of on-chip cache and network resources at nominal as well as near-threshold voltages.

Extreme voltage scaling increases the energy efficiency of future processors since energy scales quadratically with voltage [3]. Operation at NTV can deliver up to 10× reduction in energy [2]. However, the clock frequency of the processor must be reduced, otherwise the hardware structures may fail due to extremely tight timing guard-bands. Logic elements have been shown to be resilient to timing variations at NTV [2]. However, SRAM memory bit-cells that are used to design fast on-chip caches are most vulnerable to failure due to their tight functionality margins [3].

To make private caches resilient to hard-faults, a non-uniform access private L1 cache architecture (`NUCA-L1`) is proposed. The key idea is to access fault-free cache lines with no additional hit latency. For cache lines with one bit-cell fault, the fault site is corrected by implementing an error correcting mechanism. The trade-off is the additional hit latency for accesses to such cache lines. Finally, cache lines with faults that cannot be corrected (e.g., two or more bit-cell faults) are

disabled. By having an architecture that optimizes for the common case of L1 hit while recovering cache capacity improves data locality.

The *data access efficiency* challenge is further exasperated at higher core counts due to two main reasons. (1) The diameter of on-chip networks increases with core count, making the cost of moving data expensive. Furthermore, emerging technologies such as 3D die stacking [4] will make the "distance" to access data even more non-uniform. (2) At sub-$22nm$, on-chip wires are not scaling at the same rate as transistors. While compute energy is projected to scale down by $6\times$ from $45nm$ to $7nm$, interconnect energy only scales down by $1.6\times$ [2]. Similar trends are projected for transistor and wire delay scaling. Hence, unnecessary data movement not only impacts memory access latency, but also incurs wasteful energy consumption of network and cache resources.

Motivated by the fact that cache lines exhibit varying degrees of reuse at the last level cache (LLC), the locality-aware data replication scheme is proposed. A low-overhead yet highly accurate hardware-only predictive mechanism is proposed to track and classify the *reuse* of each cache line in the LLC. A runtime classifier only allows replicating those cache lines that demonstrate *reuse* at the LLC while bypassing replication for others. This improves data access locality by minimizing the number of accesses to remote locations in a single chip many-core processor.

1.2 Efficient On-chip Communication

Designing a parallel application under shared memory is attractive from the perspective of ease of programming. Cache coherence is implemented in the hardware to enable efficient on-chip data access without programmer intervention. However, communication between cores through coherent caches and shared memory primitives is inefficient and can potentially take hundreds of cycles to complete. This can prove to be very detrimental to performance for accesses to contended shared

data in an application because it produces high on-chip traffic due to cache line ping-pong and cache coherence effects. This leads to higher memory access latency and energy consumption.

To mitigate the inefficient communication bottleneck, a novel architectural mechanism to explicitly manage communication is proposed. As moving data to computation can become inefficient in widely shared read-write data, explicit communication is exploited to move computation to data. This way the data remains in one place while different operations are being performed on it, eliminating unnecessary data movement. The idea of explicit management is also applied to accelerating shared memory synchronization primitives. To enable explicit communication, I propose to retain the shared memory programming model, however, introduce a set of lightweight in-hardware explicit messaging style send and receive instructions in the instruction set architecture (ISA). These instructions are implemented in hardware to enable efficient core-to-core communication. The advantage is that the cache coherence related indirections of shared memory protocols can be augmented by point-to-point explicit communication over the on-chip network, leading to a significant reduction in data access movement and a consequent lowering of energy usage. Furthermore, the proposed explicit communication ISA extension enables flexible but powerful programming and communication models. These include; overlapping communication with computation, making communication concurrent, atomic execution of functions, avoiding coherence traffic by pushing data, and moving computation to where data is located. All these techniques help parallel workloads to overcome shared memory induced bottlenecks and scale better.

The rest of the thesis is organized as follows. Chapter 2 talks about `NUCA-L1` architecture in detail. Locality-aware data replication is discussed in Chapter 3. Scaling shared memory many-core architectures using explicit messaging is discussed in Chapter 4. Chapter 5 concludes the thesis.

Chapter 2

NUCA-L1: A Non-Uniform Access Latency Level-1 Cache Architecture for Multicores Operating at Near-Threshold Voltages

Research has shown that operating in near-threshold region is expected to provide up to 10× energy-efficiency for future processors. However, reliable operation below a minimum voltage (Vccmin) cannot be guaranteed due to process variations. Because SRAM margins can easily be violated at near-threshold voltages, their bit-cell failure rates are expected to rise steeply. Multicore processors rely on fast private L1 caches to exploit data locality and achieve high performance. In the presence of high bit-cell fault rates, traditionally an L1 cache either sacrifices capacity or incurs additional latency to correct the faults. It is observed that L1 cache sensitivity to hit latency offers a design tradeoff between capacity and latency. When fault rate is high at extreme Vccmin, it is beneficial to recover L1 cache capacity, even if it comes at the cost of additional latency. However, at low fault rates, the additional constant latency to recover cache capacity degrades performance. With this tradeoff in mind, I propose a Non-Uniform Cache Access L1 architecture (NUCA-L1) [5] that

avoids additional latency on accesses to fault-free cache lines. To mitigate the capacity bottleneck, it deploys a correction mechanism to recover capacity at the cost of additional latency. Using extensive simulations of a 64-core multicore, it is demonstrated that at various bit-cell fault rates the proposed private `NUCA-L1` cache architecture performs better than state-of-the-art schemes, along with a significant reduction in energy consumption.

2.1 Introduction

Complex uniprocessors have hit the "power wall", and multicores with simpler cores have emerged as the alternative. Multicores exploit concurrency to achieve performance, and rely on the simplicity of design and operation of each core to achieve energy efficiency. However, with the integration of many cores on a single die, future multicores will still be constrained by their energy efficiency [6]. As energy quadratically scales with voltage, extreme voltage scaling can deliver energy efficient processors [3]. However, reliable circuit operation cannot be guaranteed below a minimum voltage, `Vccmin`, as the effects of process, voltage and temperature (PVT) variations become predominant and the hardware components may start to fail. Despite the reliability issue, operating at near-threshold voltage (NTV) is an attractive solution since it has the potential to deliver up to 10× energy reduction, and has been proven to be the most energy efficient region to operate in [2, 4]. Logic elements are somewhat resilient to PVT variations because they can compensate across long logical paths. On the other hand, SRAM memory elements pose a critical limitation in low-voltage operating conditions, as their functionality margins are lower because of their aggressively sized transistors and architectural requirements to maximize array size for area efficiency [3]. Therefore, SRAM bit-cells are especially vulnerable to the PVT variations at NTV conditions, causing their margins to be easily violated.

The higher demand for on-chip multicore cache has been steadily increasing to alleviate expensive off-chip accesses. A private last-level cache (LLC) organization (e.g., [7]) has low hit latency due to high data locality. However, its off-chip miss rate is high in workloads with large private working set and/or high degree of data sharing. A popular cache organization is to implement per-core fast private caches backed by a logically shared (physically distributed) last-level cache to minimize the off-chip miss rate [8]. The varying latency to access the shared LLC naturally gives rise to non-uniform cache access (NUCA) [8]. Although the shared-LLC organization enables large on-chip cache capacity, the average LLC access latency is considerably higher than a private-LLC system. Therefore, tiled multicores rely heavily on their *low-latency* private caches for common case optimizations, and are relatively insensitive to the LLC latency [9, 10].

In the context of NTV operation, error correcting codes (ECC) have been proposed to protect the LLC. However, private caches have been left unprotected, limiting their operation at lower voltages [2]. Since future multicores will need to operate at NTV for energy-efficiency, protecting private caches against the SRAM bit-cell faults is increasingly becoming critical. An experiment was conducted to quantify the performance sensitivity to variations in L1 and L2 cache access latencies in a 64-core private-L1, shared-L2 cache organization. The results show a $>$10% performance loss when L1 hit latency is increased from one to two or more cycles. On the other hand, when L1 hit latency is kept constant at 1-cycle and the latency of LLC slice (L2 cache) is increased, the performance only degrades by $<$2%. Motivated by this, I propose to focus on the latency sensitive private-L1 cache operation at NTV. It is assumed that each LLC slice is protected using a scheme that sacrifices its access latency (e.g., by correcting bit-cell faults or combining multiple data blocks to recover the cache capacity [11, 12]).

NTV proposals that set frequency and voltage such that the processor has zero-faults have been explored. This approach delivers reliable operation without increasing hardware complexity. However, it either operates considerably above the near-threshold voltage [13], hence does not fully

exploit the energy efficiency, or run at a low enough frequency to ensure zero-faults [4], degrading performance substantially. Another widely explored approach is to allow bit-cell faults to exist at NTV and fix them during design time or at runtime [3, 11, 12]. This approach ensures higher energy efficiency by operating near the threshold voltage and acceptable performance by setting the frequency higher than a safe frequency. Because the system now operates at a higher than safe frequency in the NTV region, the timing margins of SRAM bit-cells may be violated.

To mitigate the effect of rising SRAM bit-cell faults at near-threshold voltages, a private-L1 cache can implement three possible mechanisms. The first category relies on circuit level techniques that up-size the transistors or implement a more robust SRAM cell (e.g., 8-T or 10-T SRAM cell instead of the traditional 6-T version) [14, 15, 16, 17, 18, 19]. This results in an increase in area (33% to 100%) of the SRAM cell, reducing the effective cache capacity within a given area budget. The second category deals with the high number of random bit-cell faults using error-correcting (ECC) techniques [20, 21, 22, 23, 24, 11]. These techniques increase the available cache capacity by correcting one or more bit-cell faults but suffer from a constant latency overhead of one or more cycles for error detection and correction. As previously discussed, the private-L1 cache performance is sensitive to its access latency, therefore, traditional ECC based techniques can quickly become ineffective for private caches at near-threshold voltages. The third category consists of architectural techniques [25, 26, 27, 28, 12]. These techniques patch-up and recover cache capacity at the fine granularity of sub-cache-lines. These techniques add a constant latency overhead of up to three cycles, in addition to disabling a portion of cache. Cache line disabling [29, 9, 11] is a technique that does not incur additional latency but is limited by the amount of capacity it can recover. Word-level disabling [27] works at a finer granularity, resulting in high available capacity. However, it evicts the cache line on an access to a faulty word. This generates excessive network traffic, resulting in significant loss in performance and energy efficiency.

The persistent bit-cell faults at nominal down to near-threshold voltages can be classified using

memory built-in self-test (MBIST) mechanism commonly deployed in commercial processors [30]. Utilizing this a priori knowledge of bit-cell faults at NTV, it is possible to encode the number of faults per cache line (or even per word), and capture this information in the cache's tag array. It is proposed to deploy this mechanism and design a non-uniform access latency private cache architecture (`NUCA-L1`). The key idea is to access fault-free cache lines with no additional hit latency. For cache lines with one bit-cell fault, it is proposed to correct the fault site by implementing an error correcting mechanism. The trade-off is the additional hit latency for accesses to such cache lines. Finally, for cache lines with faults that cannot be corrected (e.g., two or more bit-cell faults), it is proposed to disable them by disallowing allocation of such cache lines. `NUCA-L1` cache variants are also proposed that allow fine-grain correction and/or disabling capabilities.

`NUCA-L1` cache architecture and its variants are quantitatively compared to bit-fix [25], word-disable [25, 26], Archipelago [12], cache-line-disable, single error correction double error detection (SECDED) with cache-line-disable, double error correction triple error detection (DECTED) with cache-line-disable, word-level disabling [27], and SECDED with word-level disabling. It is shown that the proposed `NUCA-L1` architecture not only exploits the latency and capacity tradeoff but also adapts seamlessly to varying bit-cell fault rates. To the best of my knowledge, this is the first proposal that combines the latency and capacity tradeoffs for a private L1 cache operating at nominal voltage all the way down to the NTV region.

The novel contributions of this work are:

1. A variable access latency private L1 cache architecture that exploits the latency and capacity tradeoffs by avoiding additional hit latency for fault-free cache lines, while only applying error correction/patch-up-and-recovery mechanism to faulty but correctable cache lines. The cache lines that are not correctable are disabled for allocation, resulting in an effective architecture for NTV operation.

2. The NUCA-L1 architecture seamlessly adapts to NTV conditions with different bit-cell fault rates. It effectively exploits the available capacity to deliver performance and energy close to the ideal fault-free baseline by keeping the hit rate high and hit latency low.
3. The proposed architecture performs 4% and 11% better than the best performing ECC mechanism at high and low fault rates respectively. It also performs on-par with state-of-the-art Archipelago scheme at high fault rate, however it performs 11% better at low fault rate.
4. The proposed architecture's energy is within 5% and 16% of the fault-free baseline system at low and high fault rates respectively. It also consumes 14.5% and 3% lesser energy than Archipelago at low and high fault rates respectively.

2.2 Background

The earlier approaches to improve cache reliability have used true row/column level redundancy by adding spare rows/columns to the cache array [31]. These approaches remap a row/column with bit defects to a functional spare row/column. While this coarse-grain approach is effective at voltages with low fault rates, albeit with significant overhead, it cannot deal with thousands of random bit faults. Operating in the NTV region requires more robust solutions that are capable of dealing with thousands of random faults. These existing solutions are classified in the following categories.

2.2.1 Circuit Level Solutions

Several circuit level solutions have been proposed to operate reliably at low voltages. These solutions span two different dimensions. The first dimension covers designing SRAM bit-cells by up-sizing transistors to achieve higher functionality margins [14, 15, 16, 17]. The second dimension

covers designing more robust SRAM cells (e.g., Schmitt Trigger (ST) SRAM cell [19]) to ensure improved performance at low fault rates. These approaches are more tolerant to different sources of parameter variations compared to the conventional 6T cell and allow reliable operation of the cache at lower voltages. However, larger SRAM cells result in higher leakage current in both low-voltage and high-voltage operation of the cache, as it is directly proportional to the number of transistors. Furthermore, these approaches incur significant area overhead (e.g., 100% area overhead for the ST SRAM cell). Therefore, the effective cache space available is reduced at a given area budget.

2.2.2 Error Correcting Codes Based Solutions

Correcting persistent faults using error detection and correction schemes [20] is a popular mechanism to enable NTV operation. Parity is the most popular design choice for L1 caches to detect bit errors. This scheme is easy to implement and is useful to detect an occasional soft error at high-voltages [32]. However, it does not have any correction capability and is not suitable for caches operating at near-threshold voltages with many permanent bit-cell failures. Similarly, conventional SECDED provides single-bit correction capability but cannot operate at high fault rates with lots of multi-bit faults. Caches with conventional SECDED need to be augmented with cache line disabling to make it work at high bit fault rates [9]. Increasing the strength of ECC (DECTED and higher) can provide better correction capability, enabling more aggressive voltage scaling. However, this comes at a price of higher area and latency overhead.

Two-dimensional ECC schemes have been proposed previously to correct faults [33]. Kim et al. proposed one such two-dimensional ECC scheme to effectively correct clustered multi-bit errors [21]. The scheme works well for protection against soft-errors but is not effective for situations with persistent faults, as it has a high overhead for every cache write. Yoon et al. proposed a 2-tier ECC scheme for the last-level cache [22]. Tier-2 ECC check bits are stored in cacheable DRAM

memory space and is only fetched from DRAM if there is a bit error in a dirty line, which makes it very efficient for protection against soft-errors. However, it cannot tolerate process variation induced high bit-error rate at near-threshold voltages due to excessive cache pollution and off-chip communication. Chishti, et al. proposed a multi-bit segmented ECC (MS-ECC) scheme, capable of correcting bit errors at a sub-cache line granularity [23]. This scheme sacrifices 50% of area to store ECC check bits which makes it ineffective for NTV operation. Alameldeen et al. presented a cache architecture based on variable-strength ECC (VS-ECC) [11]. Their scheme uses SECDED in the normal case, and stronger ECC for cache lines with multi-bit errors. The resources for multi-bit correction are shared among a set, thereby reducing the area overhead. The higher latency of stronger ECC makes this scheme prohibitive for use in L1 caches, especially at near-threshold voltage with high bit error rate. All ECC based schemes add constant latency to the L1 cache that limit their use at near-threshold voltages. SECDED is the simplest ECC scheme and incurs one additional cycle on top of L1 hit latency. The closest more complex scheme is DECTED which requires two additional cycles to correct errors. It is shown in Sec 2.5.3 that although DECTED recovers more capacity, the overall performance still degrades because of its higher access latency. Therefore, the above mentioned schemes that require even higher latency to correct bit-cell faults would not be effective at NTV for L1 caches.

2.2.3 Architectural Patch-up-and-Recovery Solutions

Techniques that leverage disabling portions of a cache at a coarse granularity to save power have been studied before [34, 35, 36]. These coarse-grain techniques disable ways or sets (or a combination of both), which makes it impractical for NTV operation, as the high number of random faults could result in all of the cache being disabled.

Wilkerson et al. proposed two fine-grain sub-cache line level disabling techniques for reliable

low voltage operation of L1 caches, namely bit-fix and word-disable [25]. Bit-fix sacrifices 25% of the cache capacity to correct bit faults in rest of the lines. This scheme incurs 3 cycle additional latency to correct persistent faults, which is too excessive to even expect reasonable performance in the context of private L1 cache. Word-disable on the other hand sacrifices 50% of its capacity to correct persistent faults in rest of the cache. This scheme incurs 1 cycle additional latency. Both these schemes can only operate at low fault rates and cannot tolerate high fault rates. The reason being that it might not have enough fault-free sub-blocks to recover the rest of the ways. Roberts et al. also proposed a technique similar to word-disable [26].

Abella et al. proposed a sub-block level disabling scheme that disables faulty words within a cache line [27]. Accesses to these faulty words are treated as misses and the cache line is evicted, writing back only the valid words in that cache line. Accesses to fault-free words go through normally. When implemented on top of a parity protected cache, this scheme results in a high number of evictions, degrading the performance substantially. Implementing the scheme on an ECC protected cache can recover most of the capacity, however, it suffers from the constant latency overhead of the ECC scheme. In addition, ECC needs to be implemented on a word level, resulting in a high storage overhead.

Ansari et al. proposed a fine-grain patch-up-and-recovery mechanism to enable NTV operation of private caches [12]. In this multi-bank cache technique, collision-free groups (islands) of varying sizes can be formed using with-in cache flexible group formation. Each group contains one sacrificial set which is used to correct the other sets in that group. This scheme loses capacity due to sacrificial sets. It can also increase the contention on some of the cache lines because it remaps the sacrificial sets to other working sets, resulting in performance loss. The cache access in this technique can be divided into three steps. The memory map is read to figure out the index for accessing the cache. After that, the cache and the fault map are accessed in parallel. Finally, the multiplexing layer assemble the requested data based on the information from the fault map. The L1 cache needs at

least one additional cycle to complete these three steps.

2.2.4 Motivation for `NUCA-L1` Architecture

Multicore processors rely on the low latency private L1 caches for performance. The L1 cache has a high hit rate and any increase in latency degrades performance significantly. An ideal solution would keep the latency of the private L1 caches as low as possible. Multicores also rely heavily on the capacity of the private L1 caches to exploit spatio-temporal locality. When the active working set of a workload is large, the limited L1 cache size increases its miss rate. This becomes even more pronounced at high fault rates, when the available capacity is low. Conventional correction/patch-up-and-recovery techniques can help increase the available capacity. However, performance suffers because of the additional latency for error correction/recovery. Cache line/word-level disabling optimizes for latency but negatively impacts the L1 cache hit rate.

Keeping the importance of both lower latency and higher available capacity in mind, `NUCA-L1` cache architecture is proposed. The proposed scheme relies on the a priori knowledge of faulty cache lines at a given operating voltage. The intuition of applying correction or patch-up-and-recovery mechanism to only faulty cache lines is that fault-free lines need not incur additional latency. This way a majority of the cache accesses occur with lowest hit latency. Furthermore, a correction or patch-up-and-recovery scheme is applied to the faulty cache lines to recover cache capacity. Although this recovered capacity comes at a higher latency, it is beneficial at higher fault rates.

2.3 NUCA-L1 Architecture

2.3.1 Proposed Architecture

The main idea behind the proposed private L1 cache architecture (NUCA-L1 [5]) is to optimize for lower latency, and incur no additional latency for fault-free cache lines. To remove the capacity bottleneck, a correction scheme is applied to the faulty cache lines to enable higher available capacity. Two disable bits are added to the tag entry for each cache line to enable the identification of fault-free, single-bit, and multi-bit faults. All cache lines identified as containing faults that cannot be corrected are permanently disabled and not allocated in the L1 cache. To avoid any additional latency for fault-free cache lines, the correction logic is bypassed. Cache lines that can be corrected are dealt with using a correction or patch-up-and-recovery mechanism. It is assumed that the tag array is protected using circuit techniques discussed in Section 2.2.1.

SECDED is implemented as the correction scheme. The cache accesses to faulty cache lines take two hit cycles, and all faulty cache lines that cannot be corrected incur private cache miss latency. In the first cycle, NUCA-L1 reads the tag array to evaluate a hit and read the disable bits. If it is a hit and the disable bits identify the cache line as fault-free, the cache line is accessed in a normal way. If it is a hit but the access is to a cache line with single-bit fault, the SECDED scheme is used in the second cycle to correct the faulty bit. The hit path is modified to enable selective multiplexing of the SECDED logic. In Section 2.3.4 possible extensions of the proposed architecture to support other bit-cell correction schemes are discussed.

Permanently disabled cache lines are not allocated by the replacement/allocation logic. Sets with at least one working way are allocated as usual. The probability of all ways in a set being permanently disabled is very low but that scenario could arise at high fault rates. If this scenario does arise, the cache line is stored in a special structure, called the resilient buffer. The resilient

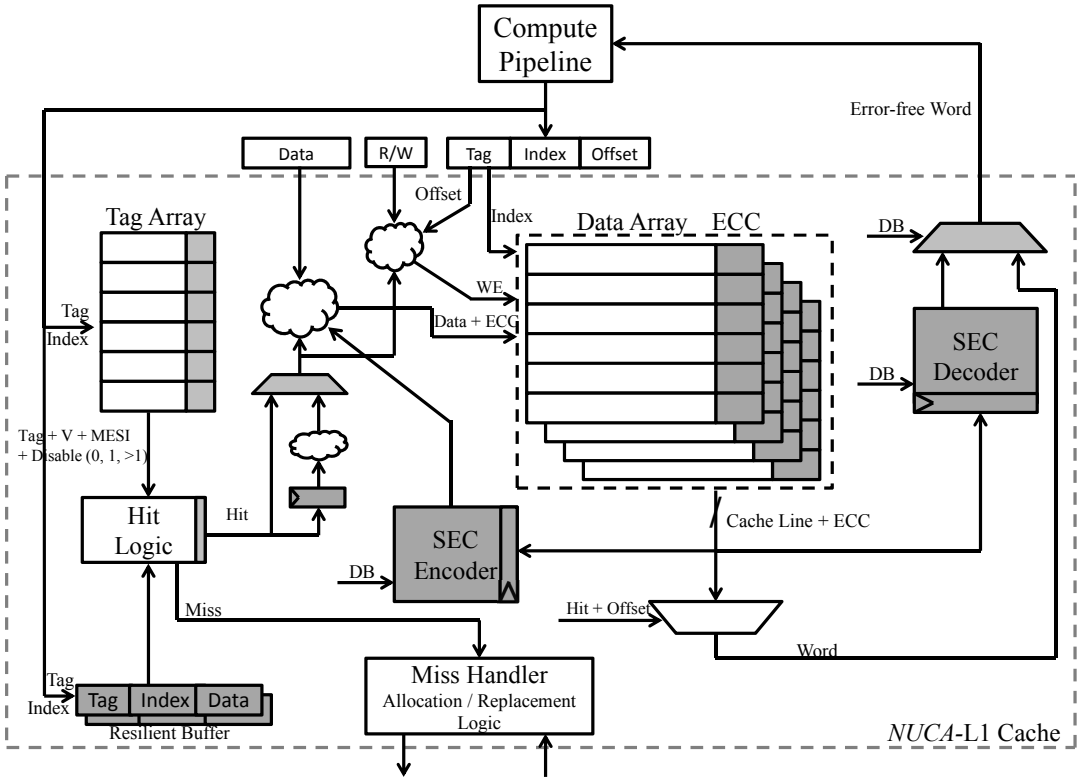


Figure 2.3.1: The microarchitecture diagram of the proposed NUCA-L1 architecture with support for cache line disabling. The additional components required for the architecture are shaded grey. The tag array holds the additional disable bits while the ECC check bits (per cache line) are stored in the data array. The SECDED encoder and decoder are gated with the disable bit (DB). The cache hit logic can operate in one or two cycle mode, depending on the disable bits for each cache line. Additional storage elements are introduced to provide the information such as the hit result for the second cycle. The resilient buffer is a fully associative structure that holds the cache lines that are mapped to permanently disabled sets in the cache.

buffer is a fully associative structure, in which each entry can store a cache line. Resilient buffer is accessed in parallel to the NUCA-L1 cache, and stores data for sets that have all their ways permanently disabled. More details about this buffer are discussed in Section 2.3.3.

As an alternative, a fine-grain disabling mechanism at the word-level can be deployed [27]. A single bit per word (8 bits per cache line) is now required to classify each word within the cache line as disabled or not. This increases the storage overhead but can result in higher available NUCA-L1 capacity. Word-level disable mechanism does not require any changes to the replacement/allocation

logic. However, an access to a word with bit-cell fault(s) is treated as a miss, and eviction is forced on the associated cache line to bring the requested word from the lower level L2 cache. Moreover, this mechanism complicates the L2 cache since word-level write enables are required for write backs from the L1 cache. The default `NUCA-L1` architecture assumes cache line disabling, however variations that use word-level disabling are discussed in Section 2.3.4.

2.3.2 Architecture Operation - Functional Cache Lines

Functional cache lines are defined as those which are fault-free or contain a single-bit fault. When a core makes a request, the tag array is looked up to evaluate a hit and the disable bits are read out. In the following subsections, I describe how a read and a write request is completed in different scenarios in the proposed `NUCA-L1` architecture (cf. figure 2.3.1).

Write Request to a Fault-Free Cache Line

In the case where the write request results in a hit and the disable bits identify the cache line as fault-free, the word is written to the cache line. The bypass multiplexers introduced in the hit path make sure that the access goes through the normal path. The disable bit (DB) is used as a select line for the multiplexers to route data through the normal hit path. Therefore, write requests to fault-free cache lines do not incur any additional latency on a cache hit.

In case of a miss, the cache line is brought in from a lower level L2 cache and inserted in the `NUCA-L1` cache. The word is then written to it through the normal data-path.

Read Request to a Fault-Free Cache Line

In the case where the read request results in a hit and the disable bits identify the cache line as fault-free, the requested word is read from the cache and returned to the core. The bypass multiplexers

introduced in the hit path make sure that the access goes through the normal path. Therefore, a read request to fault-free cache lines does not incur any additional latency.

In case of a miss, the cache line is brought in from a lower level L2 cache and inserted in the `NUCA-L1` cache. The requested word is also returned to the core through the normal data-path.

Write Request to a Cache Line with Single-bit Fault

If the write access is a hit and the disable bits identify the cache line as containing single-bit fault, the whole cache line is read out of the cache. The select line (the disable bit) to the bypass multiplexing logic is now set to one. This ensures that the cache line is forwarded to the SECDED encoder. The disable bit is also used as the enable signal for the SECDED encoder, therefore, a value of one enables the module. The SECDED encoder computes the check bits and writes back the new word and the check bits. The SECDED encoder takes an additional cycle to compute the check bits and write it back to the cache, hence, the access takes two cycles to complete. The output from the hit logic is latched, so that it can be used in the second cycle.

If the access is a miss and the disable bits identify the cache line as containing single-bit fault, the request is sent to the lower level cache. When the lower level cache returns the cache line, the cache allocation logic finds a suitable way for the cache line. In this step, the allocation/replacement logic takes into account the LRU bits and also the fact that all cache lines may not be available. The allocation/replacement logic gives priority to fault-free cache lines over cache lines with single-bit fault, when both the cache lines are invalid. The reason being that fault-free cache lines incur no additional latency while cache lines with single-bit faults incur additional latency. However, if the choice is between two valid cache lines, LRU replacement scheme is used. The cache line is written to the cache as described previously.

The multiplexing logic used to route the data to the SECDED encoder or the cache is gated

with the disable bits. If the disable bits identify the cache line as fault-free, the multiplexing logic routes the word directly to the cache. If the disable bits identify the cache line as single-bit fault, the multiplexing logic routes the word and the read cache line to the SECDED encoder.

Read Request to a Cache Line with Single-bit Fault

If the read access is a hit and the disable bits identify the cache line as containing single-bit fault, the whole cache line is read out of the cache. The disable bit value of one ensures that the cache line is routed to the SECDED decoder and that the SECDED decoder is enabled. The SECDED decoder locates and corrects the faulty bit and then forwards the requested word to the core. The SECDED decoder takes an additional cycle to correct the faulty bit, hence, the access takes two cycles to complete.

If the access is a miss and the disable bits identify the cache line as containing single-bit fault, the request is sent to the lower level cache. When the lower level cache returns the cache line, the cache allocation logic finds a suitable way for the cache line. The allocation logic works the same way as described in the previous section. The requested word is also returned to the core through the normal path, as the word is brought in from the lower level cache and hence is fault-free.

The multiplexing logic used to route the data to the SECDED decoder or the core is gated with the disable bits. If the disable bits identify the cache line as fault-free, the multiplexing logic routes the word directly to the core. If the disable bits identify the cache line as single-bit fault, the multiplexing logic routes the code word to the SECDED decoder.

2.3.3 Architecture Operation - Permanently Disabled Cache Lines

The cache lines that cannot be corrected by the employed correction scheme are classified as permanently disabled cache lines. The following discussion is based on SECDED correction

scheme, hence, cache lines with multi-bit faults are considered to be permanently disabled. The same description can be generalized to stronger correction schemes as well. In the following subsections, I describe how a read and a write request is completed, when one or more ways are permanently disabled, in the proposed NUCA−L1 architecture (cf. figure 2.3.1).

Set has At Least One Way Available

If a hit is evaluated on an access to a cache line, the request is completed through the process explained in Section 2.3.2. On a miss, the data is retrieved from the lower level cache and inserted in the cache. During the insertion process, the replacement/allocation logic is invoked. The replacement/allocation logic is modified to incorporate the fact that permanently disabled cache lines can be present in a set. In such sets, the cache line is allocated to one of the available ways.

Set has All Ways Permanently Disabled

The case in which all ways in a set are permanently disabled, i.e. all the cache lines have multi-bit faults, that set has a very low probability of occurrence. Analysis of the bit-cell fault distribution shows that on average there is one such set per cache at the highest fault rate evaluated. At the lowest fault rate evaluated, there are no such sets. It is observed through experiments that even a single highly used set can impact performance significantly. Therefore, to deal with this case a fully associative structure, called the "resilient buffer", is proposed. Each entry in this buffer stores the tag, index and the associated bits, as well as the cache line data.

If all the ways in a set are identified as permanently disabled, the resilient buffer is used to store the cache line. This resilient buffer is accessed in parallel to the NUCA−L1 cache. Accessing this buffer is exactly the same as accessing a normal fault-free cache line. If the buffer is completely filled, any new cache line allocation evicts one of the currently valid lines. LRU replacement policy

is used for the resilient buffer. The area overhead of this structure is low and is accounted for while calculating overheads in Section 2.3.5.

2.3.4 NUCA-L1 Architectural Extensibility

The proposed NUCA-L1 architecture can be extended in two dimensions i.e. the correction strength and the disabling granularity. In this section three variations of the proposed NUCA-L1 architecture along these two dimensions are presented.

NUCA-L1$_{DEC}$

A variation of the architecture with stronger ECC correction capability is explored. This variation, called NUCA-L1$_{DEC}$, has the capability to correct up to 2-bit faults. All cache lines that are fault-free, incur no additional latency. Cache lines with 1-bit faults are corrected using the SECDED decoder logic and thus need an extra cycle to complete. Cache lines with 2-bit faults are corrected using the DECTED decoder logic and thus need 2 extra cycles to complete. All cache lines with 3-bit or more faults are permanently disabled and are not allocated. The stronger fault correction capability comes at a cost of higher hardware complexity. The area overhead for this variation is almost prohibitive for an L1 cache. However, this results in performance improvement over the SECDED version, at higher fault rates.

It is possible to use even stronger ECC to make the system work at extreme fault rates. However, the potential benefits may not amortize the additional complexity of ECC. For example, the opportunity to correct more cache lines decline as there are a small number of cache lines with more than 2-bit faults. In addition to that, the latency overhead of stronger ECC quickly approaches that of the L2 cache hit latency.

NUCA−L1$_{\text{CL−SEC_WLD}}$ (Cache Line SECDED with Word Level Disabling)

The NUCA−L1 architecture is modified by introducing word-level disabling scheme for cache lines with multi-bit faults [27]. Fault-free cache lines are accessed without any latency overhead, while cache lines with single-bit faults are accessed with additional one cycle latency for fault correction. Cache lines with multi-bit faults are accessed using word-level disabling. In this case, all accesses to fault-free words go through normally, while accesses to faulty words force a miss, and the cache line is evicted. This enables higher available capacity, since words from otherwise permanently disabled cache line can now be utilized. The higher available capacity is beneficial to the overall performance, especially at high bit-cell fault rates (cf. Section 2.5.1). However, this comes at a cost of higher storage overhead, as word-level disable bits are now required in each tag entry. Moreover, it also complicates the hit path and write back of evicted data to L2.

The modifications on top of the NUCA−L1 architecture are as follows. To treat accesses to words with multi-bit faults as misses, the hit logic is modified. The cache controller evicts the cache line on such an access, and writes back only the valid (fault-free) words to the lower level L2 cache. The L2 cache controller is modified to write back only valid words, instead of the evicted cache line. The allocation/replacement logic is the same as a fault-free baseline, with the capability of updating the LRU bits on forced evictions. This is needed to ensure that the evicted cache line is brought in and inserted in a different cache way [27].

NUCA−L1$_{\text{WL−SEC_WLD}}$ (Word Level SECDED with Word Level Disabling)

SECDED at word-level is considered, accompanied by word-level disabling for words that cannot be corrected. Fault-free words are accessed without any latency overhead, and *words* with single-bit faults are accessed with additional one cycle latency for fault correction. Accesses to words with multi-bit faults are treated as misses and the associated cache lines are evicted. As the correction

scheme is applied at the word level, this variation of the architecture can correct majority of the bit-cell faults, resulting in high available cache capacity (even at high bit-cell fault rates). However, the word-level SECDED comes with a prohibitively high storage overhead (64 bits per cache line for word-level SECDED).

This variation also requires the changes discussed in Section 2.3.4. Moreover, further changes are made to implement the word-level SECDED. Word-level SECDED implementation either incurs high area overhead or high latency overhead. The reason being that when a cache line is brought in from a low level cache and inserted in L1 cache, the ECC check bits are to be computed for each word. In a low area overhead approach, this computation is serialized with seven additional cycles. In a low latency approach, eight parallel SECDED encoders are needed to complete the process in one cycle. The first approach may cause excessive latency overhead if a set with multiple disabled words is highly utilized, as the accesses to these words will result in eviction of the cache line. The low latency approach is modeled, incurring high logic overhead, to achieve the best performance.

2.3.5 Overhead Analysis

In this section the area and latency overheads of the proposed `NUCA-L1` architecture and its variations are calculated.

`NUCA-L1`

Storage The proposed architecture requires 2 bits per cache line to classify the cache line as fault-free, 1-bit fault and 2-bit fault. SECDED correction requires additional 11 bits for each cache line to store the ECC check bits. The following calculations are for one core but they are applicable for the entire system since all cores are identical. The storage overhead shown is for

L1-D cache, L1-I cache has similar storage overheads. Assuming $32KB$ L1 size, the L1-D cache needs $2 \times \frac{32KB}{64B} = 128B$ for storing the disable bits. The storage overhead for storing ECC bits is $11 \times \frac{32KB}{64B} = 704B$. The single entry resilient buffer results in an overhead of $72B$. Therefore, the `NUCA-L1` architecture uses $904B$ more storage per L1 cache.

Logic The logic overhead associated with SECDED encoder and decoder is described below. The logic structure of the encoder and the first step of the decoder is the same (cf. Section 2.3.6). Since either encoding or decoding is done at the L1 cache at any given time, the XOR trees of the encoder are reused for the first step of the decoder. This requires 2k XOR gates and a latency of 8 XORs [11]. Following [11], steps 2 and 3 require 512 XOR and 9.2k AND gates, respectively. The latency of step 2 is 1 XOR gate and that of step 3 is 5 AND gates. Based on these latencies, the SECDED encoder and decoder implementation takes an additional clock cycle each. Considering each AND gate equals two SRAM bit-cells and each XOR gate equals four SRAM bit-cells, this translate into $3.55KB$ overhead per SECDED controller.

Summary The storage and logic overheads for SECDED per L1 cache requires $4.43KB$. Considering an inclusive L1-L2 cache hierarchy, a shared SECDED controller per core suffice. However, each L1 cache needs the $904B$ overhead for storage. Using $32KB$ L1-I, $32KB$ L1-D, and $256KB$ L2 cache, the SECDED area overhead comes out to $\sim 1.64\%$ per core. If an independent SECDED controller is assumed for each L1 cache, the area overhead would be $\sim 2.7\%$ per core.

$\textbf{NUCA-L1}_{\text{CL-SEC\_WLD}}$

The overhead analysis from Section 2.3.5 holds for this architecture variation as well. This implementation adds an additional one bit per word to classify it as either disabled or not (a total

of 8 bits per cache line). This comes out to $512B$ per L1 cache, an increase of $\sim 57\%$ in storage overhead on top of the NUCA−L1 architecture. The logic overhead also increases over NUCA−L1 because word-level disabling increases the hardware complexity of the L1 cache and requires modifications to lower level cache controller as well.

NUCA−L1$_{\text{WL-SEC_WLD}}$

NUCA−L1$_{\text{WL-SEC_WLD}}$ requires 8 bits per word to store the ECC check bits and one bit per word to store the word disable status. This results in a storage overhead of 72 bits per cache line, which is $\sim 5\times$ storage overhead on top of NUCA−L1. Moreover, this architecture variation also suffers from higher hardware complexity due to modifications required to the lower level cache controller.

2.3.6 Discussion

In this section, discussions on how an ECC based scheme is implemented and what goes into the encoder and decoder module are included. The implications of L1 cache and core pipelining, and soft-error mitigation mechanism for the proposed NUCA−L1 architecture are also discussed.

ECC

Binary BCH codes, a class of linear cyclic block codes, are the most popular choice for correcting random bit-errors in memories [37]. A binary BCH code is defined over a finite Galois Field $GF(2^m)$. BCH code words are produced using a generating matrix based on a set of generator polynomials. Checking a code word for errors involve using a parity matrix to obtain the syndromes [37] [38]. The generated syndrome is then checked to determine whether there are any errors in the code word.

Hamming code is a special type of BCH code which can correct one random bit-error [20]. Prior work has proposed parallel designs for simple codes, such as SECDED and DECTED [39] [40], which are faster than iterative designs [40]. These parallel implementations are significantly faster but incur a larger area overhead. The working of the two components of the ECC error correction logic is explained ahead.

Encoder The encoder takes a cache line as input and computes the ECC check bits for it. In a completely bit-parallel implementation, the encoder is made up of XOR trees, each tree computing a single check bit [41]. The data bits are concatenated with the check bits to obtain the final code word. The storage and logic overheads of the SECDED encoder are discussed in Section 2.3.5.

Decoder The decoder detects and corrects all the errors in a stored code word. The error-correcting logic can pinpoint all the bit-errors and then correct them. The decoder operation can be divided into 3 steps [37]. (1) In the first step, the syndrome for the stored code word is calculated. The hardware for this stage is very similar to the encoder, i.e. each syndrome bit is obtained by an XOR tree. If the calculated syndrome is equal to zero, it means the cache line is error free. A non-zero syndrome indicates the occurrence of one or more bit-errors. (2) If the syndrome is non-zero, the error locater polynomial is determined from the syndrome calculated in the previous step [42] [43]. (3) In the final step the error locater polynomial is solved and the roots are determined. Correction is done by flipping the faulty bits using XOR gates. To keep the SECDED decoder latency as low as possible, a parallel SECDED decoder [40, 44] is implemented. As SECDED is the simplest BCH code, the hardware complexity for a parallel version is low and does not result in a steep increase in area.

Pipelining L1 Cache

To keep the hardware complexity of the L1 cache low, a single read/write shared port is implemented. A multi-port cache increases its complexity dramatically. In the single-port cache implementation, all accesses to the L1 cache are serialized. An access to a cache line with a single-bit error takes two cycles to complete. If a pipelined cache is assumed, the two cycle latency can be hidden. It is argued that this is not possible on each cache access. The write accesses to a faulty cache line requires a read-modify-write operation over two cycles. As the cache has only a single read/write shared port, a second access cannot enter the cache. On a read access, however, the cache line is read out and sent to the SECDED decoder for correction. Hence, a second access can commence during the second cycle.

Compute Core Pipeline

The proposed `NUCA-L1` architecture is evaluated using single issue, in-order cores. In such a core, the pipeline stalls if it is waiting on a memory access to return the requested word. In normal operation, the memory access can result in either an "L1 hit", "Local L2 hit", "Remote L2 hit" or an "L2 miss". These scenarios result in different access latencies, ranging from one cycle to 10s or even 100s of cycles. To support the proposed `NUCA-L1` cache, the modifications needed to baseline compute pipeline are limited to the instruction stall logic. This adds additional latency to the compute pipeline, however `NUCA-L1` minimizes this overhead by not incurring additional latency for fault-free cache lines.

Out-of-order processors are more tolerant to unexpected delays as compared to in-order processors. The reason is that an out-of-order processor can continue execution on independent instructions while the outstanding loads are being serviced. In contrast, an in-order processor stalls and waits for the load to complete before it can proceed. In this respect, the presented implementation provides

an insight into the scenario inclining towards the worst case.

Soft-Error Mitigation

The proposed architecture efficiently deals with persistent errors caused by PVT variations at near-threshold voltages. However, random soft-errors caused by alpha-particle strikes can still be a problem. If these errors are not dealt with, the reliable operation of the system cannot be ensured. There are several mechanisms one can use for soft error protection. However, since the focus of this chapter is on more common process variation induced hard errors at NTV, soft-error support is discussed briefly.

The proposed `NUCA-L1` architecture's ECC scheme can be utilized to operate the cache in "soft-error protection mode". However, this results in a constant latency overhead to access the L1 cache, degrading performance. As an alternative, a word-level parity based scheme can be deployed in parallel to the proposed `NUCA-L1` architecture to detect soft-errors.

If a soft-error is detected, the cache line is handled in two different ways, based on the state of the cache line. If the cache line is clean, the copy in L1 cache is discarded and a miss is forced to bring in the cache line from the lower level cache. If the cache line is dirty, the copy in L1 cache cannot be discarded. In this scenario, a roll back mechanism is invoked through the operating system to revert to a previously known good state [45].

2.4 Evaluation Methodology

A 64-core shared memory multicore is evaluated. The default architectural parameters used for evaluation are shown in Table 4.6.1. The baseline system is a tiled multicore with an electrical 2-D mesh interconnection network. Each core consists of a compute pipeline, private L1 instruction

Architectural Parameter	**Value**
Number of Cores	64 @ 1 GHz
Compute Pipeline per Core	In-Order, Single-Issue
Physical Address Length	48 bits
Memory Subsystem	
L1-I Cache per core	32 KB, 4-way Assoc., 1 cycle
L1-D Cache per core	32 KB, 4-way Assoc., 1 cycle
L2 Cache per core	256 KB, 8-way Assoc. 6 cycles, Inclusive, R-NUCA
Cache Line Size	64 bytes
Directory Protocol	Invalidation-based MESI
Num. of Memory Controllers	8
DRAM Bandwidth	5 GBps per Controller
DRAM Latency	100 ns
Electrical 2-D Mesh with XY Routing	
Hop Latency	2 cycles (1-router, 1-link)
Contention Model	Only link contention (Infinite input buffers)
Flit Width	64 bits

Table 2.4.1: Architectural parameters for evaluation.

and data caches, a physically distributed shared LLC cache with integrated directory, a SECDED encoder and decoder, and a network router. The coherence directory is integrated with the LLC slices by extending the tag arrays (in-cache directory organization [46]) and tracks the sharing status of the cache lines in the per-core private L1 caches. The private L1 caches are kept coherent using the ACKwise limited directory-based coherence protocol [47]. Some cores have a connection to a memory controller as well. Reactive-NUCA's data placement, replication and migration mechanisms to manage the LLC [48] are used. Private data is placed at the LLC slice of the requesting core, shared data is address interleaved across all LLC slices, and instructions are replicated at a single LLC slice for every cluster of 4 cores using a rotational interleaving mechanism.

2.4.1 Performance Models

All experiments are performed using the core, cache hierarchy, coherence protocol, memory system and on-chip interconnection network models implemented within the Graphite multicore simulator [49]. The Graphite simulator requires the memory system (including the cache hierarchy) to be functionally correct to complete simulation.

2.4.2 Energy Models

For energy evaluations of on-chip electrical network routers and links, the DSENT tool [50] is used. Energy estimates for the L1-I, L1-D, L2 (with integrated directory) caches are obtained using the McPAT tool [51]. The ECC encoder/decoder energy numbers are estimated from [41].

The energy evaluation is performed at the $11nm$ technology node to account for future technology trends. Models for a tri-gate $11nm$ electrical technology node using the virtual-source transport models of [52] and the parasitic capacitance model of [53] are derived. These models are used to obtain electrical technology parameters (Table 3.4.2) used by both McPAT and DSENT. The static energy (subthreshold and gate leakage) is projected to be the dominant component of the overall energy at NTV [2]. Therefore, in addition to dynamic energy, static energy is also modeled for the NTV evaluation.

Parameter	**Value**
Near Threshold Voltage (V_{NTV})	0.45V
Gate Length	14 nm
Contacted Gate Pitch	44 nm
Gate Cap / Width	2.420 fF/μm
Drain Cap / Width	1.150 fF/μm
Effective On Current / Width (N/P)	739/668 μA/μm
Off Current / Width	1 nA/μm

Table 2.4.2: Projected Transistor Parameters for $11nm$ Tri-Gate

The overall tool flow is as follows. Graphite runs a benchmark for the chosen cache configuration, producing event counters and performance results. The specified cache and network configurations are also fed into McPAT and DSENT to obtain dynamic per-event energies as well as static energy for each component. Event counters and completion time output from Graphite are then combined with per-event energies to obtain the overall energy usage of the benchmark.

2.4.3 Simulated Private L1 Cache Configurations

The following fault-tolerant mechanisms for the private L1 caches are evaluated.

Baselines

1. **Ideal fault-free baseline** implements the L1 cache with 1 cycle hit latency and 100% available capacity at all voltages.

2. **Archipelago** [12] (AP) is an architectural proposal to operate caches at NTV. The available cache capacity depends on the bit-fault rate and ranges from 92% to 99%. The L1 cache incurs one extra hit cycle (total of 2 cycles) to access the cache lines (cf. Sec 2.2.3). Because AP accesses two cache banks for each access, each bank containing multiple ways, the dynamic energy spent for an L1 access is twice that of the baseline system. The energy consumption of the fault map and the multiplexing layer is ignored.

3. **Word-dis** [25] patches up half of the cache capacity (50%) at the expense of 1 extra cycle latency (total of 2 cycles). It should be noted that this scheme cannot operate at high fault rates. The result is presented just as a comparison point.

4. **Bit-fix** [25] is able to recover 75% of cache capacity while incurring 3 extra cycles (total of 4 cycles). This scheme also cannot operate at high fault rates.

5. **Cache-line-disable** (CLD) delivers a 1 cycle L1 cache hit latency. However, due to cache line level disabling, the available cache capacity depends on the single- and multi-bit fault rate (or the operating voltage).

6. **SECDED with cache-line-disable** (SEC_{CL}) corrects cache lines with single-bit faults and incurs a 2 cycle L1 cache hit latency. The available cache capacity depends on the bit-fault rate.

7. **DECTED with cache-line-disable** (DEC_{CL}) can correct up to 2 faults at the expense of additional 1 cycle (total of 2 cycles) for encoder and 2 cycle for decoder (total of 3 cycles).

8. **Word-level disabling** [27] (WLD) works at a fine granularity of word level. It accesses fault-free words with 1 cycle hit latency. It treats accesses to faulty words as misses and evicts those cache lines to reload the fault-free word from lower level cache.

9. **SECDED with word-level disabling** [27] (SEC_{WL}) implements word-level SECDED on top of word-level disabling system. It incurs constant 1 cycle latency overhead on hits to correctable words. Access to a word with multi-bit fault is treated as a miss and is evicted.

Proposed Architecture and Variations

1. **`NUCA-L1`** **[5]** is the proposed architecture. It can operate without any latency overhead on fault-free cache lines. It incurs a single cycle additional latency for cache lines with single-bit faults. Cache lines with multi-bit faults are permanently disabled.

2. $\texttt{NUCA-L1}_{\texttt{CL-SEC\_WLD}}$ is a variation of the proposed architecture with word-level disabling. It operates without any latency overhead on fault-free cache lines, and incurs a single cycle overhead on accesses to cache lines with single-bit faults. Word-level disabling is used for

cache lines with multi-bit faults. In these cache lines, access to a word with multi-bit fault is treated as a miss and is evicted.

3. **NUCA−L1$_{\texttt{WL-SEC_WLD}}$** is a variation of the proposed architecture with word-level SECDED for correction built on top of word-level disabling. It operates without any latency overhead on fault-free words, and incurs a single cycle overhead on accesses to words with single-bit faults. Access to a word with multi-bit fault is treated as a miss and is evicted.

4. **NUCA−L1$_{\texttt{DEC}}$** is a variation of the proposed architecture. It operates without any latency overhead on fault-free cache lines. It incurs a single cycle additional latency for cache lines with single-bit faults. Cache lines with two-bit faults incur additional latency of one cycle for encoder and two cycles for decoder and cache lines with more than 2-bit faults are permanently disabled.

2.4.4 NTV Model

Memory built-in self-test (MBIST) is a popular mechanism used to detect memory faults at runtime. Most state-of-the-art processors incorporate MBIST technology to detect SRAM bit-cell faults. MBIST is deployed to test the integrity of on-chip caches and identify faulty cache lines. To track whether a cache line has zero-, single- or multi-bit faults, two bits per cache line are added. MBIST is run during the boot-up process of the system at various operating voltages. It identifies all faulty bits at the initialized voltage and constructs a bit mask of the disable bits for each cache line accordingly. These disable bit-mask vectors are stored in main memory and are only accessible to the operating system software.

Once the system is up and running, the operating system loads the appropriate disable bit-mask vector in the L1 caches and the processor executes user applications. Any significant change in

voltage can result in a different disable bit-mask vector, necessitating the operating system to context switch. In this work rapid dynamic adjustments to the operating voltage are not allowed. One way to make a significant change in voltage is to wait for the processor utilization to change and remain steady for a certain time duration. When that happens, the operating system is invoked to populate the new disable bit-mask vector and resume normal operation at the new voltage. Another approach could be to populate the disable bit-mask vector of the lowest voltage in a range of voltages, and then let the system dynamically adjust voltage within that range.

2.4.5 Bit Fault Masks for L1 Caches at NTV

Near-threshold voltage depends on the process technology, and can vary within and across generations. At a given NTV operating point, each bit-cell can be modeled as operational or not. Bit-cell fault probabilities have been shown to correlate with NTV [54, 11]. Furthermore, these probabilities exhibit normal distribution and random occurrence [19].

A range of possible NTV operating points are considered that are captured as a separate bit-fault rate for each L1 cache in a multicore. These are *Low* (0.05% probability of a bit-cell fault), *Medium* (0.1% probability of a bit-cell fault), *High* (0.2% probability of a bit-cell fault), and *Very High* (0.3% probability of a bit-cell fault).

The disable bit-mask represents zero-, single-, and multi-bit faults per cache line/word and is loaded in the tag array of the L1 cache when the processor is initialized to run at NTV. Figure 2.4.1a shows the average L1 cache capacity that is available with zero-bit, single-bit, two-bit, and more than two-bit faults per cache line. At *low fault rate*, each L1 cache has on average 76% of the cache lines with no faults, $\sim 21\%$ with single-bit faults, $\sim 2\%$ with two-bit faults, and only $\sim 0.2\%$ with more than two-bit faults. On the other hand, at *very high fault rate*, only 29.2% of the cache lines have no faults, $\sim 39\%$ with single-bit faults, 18.1% with two-bit faults, and $\sim 13.6\%$ of the cache

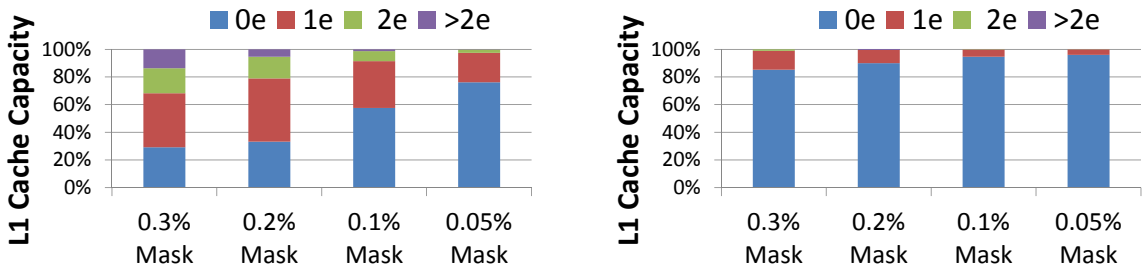


Figure 2.4.1: Based on the probability of a bit-cell fault at the NTV condition, each cache line/word in the private L1 cache is marked with 0, 1, 2 or >2 bit-cell faults. (a) For low to high fault rates, the number of cache lines with single- and multi-bit faults increases from 24% to 71% of the cache capacity. (b) Similarly, the number of words with single- and multi-bit faults increases from 4% to 15% of the cache capacity.

lines have more than two-bit faults. Similarly, figure 2.4.1b shows the average L1 cache capacity that is available with zero-bit, single-bit, two-bit, and more than two-bit faults per word. At the fine granularity of word-level, each L1 cache has on average $\sim$96% of the words with no faults, $\sim$3.9% with single-bit faults, and $\sim$0.1% with two-bit faults at *low fault rate*. This changes to $\sim$85.2% of the words with no faults, $\sim$13.7% with single-bit faults, and $\sim$1.1% with two-bit faults at *very high fault rate*.

2.4.6 Benchmarks and Evaluation Metrics

Twelve SPLASH-2 [55] benchmarks (RADIX, FFT, LU_CONTIGUOUS, CHOLESKY, BARNES, FMM, OCEAN_CONTIGUOUS, WATER-SPATIAL, WATER-NSQUARED, RAYTRACE, VOLREND, and, RADIOSITY), seven PARSEC [56] benchmarks (BLACKSCHOLES, SWAPTIONS, FLUIDANIMATE, CANNEAL, STREAMCLUSTER, DEDUP, and FERRET), two Parallel-MI-Bench [57] benchmarks (DIJKSTRA, and PATRICIA), and two graph benchmarks (CONNECTED-COMPONENTS, STATIC-COMMUNITY) [58] are simulated. The graph benchmarks model social networking based applications.

Each application is run to completion using the medium or large input sets. For each simulation run, the *Completion Time* is measured, i.e., the time in parallel region of the benchmark. The access latency is broken down into six components.

1. **Compute latency** is the processing delay in compute pipeline including the private L1 hit latency.
2. **L1 to L2 cache latency** is the time spent accessing the shared L2 cache including the round-trip time on the network. At the L2 cache, a cache line access can incur additional latency due to coherence overhead.
3. **L2 cache waiting time** is the queuing delay incurred because requests to the same cache line must be serialized to ensure memory consistency.
4. **L2 cache to sharers latency** is the round-trip time needed to invalidate private sharers and receive their acknowledgments. This also includes time spent requesting and receiving synchronous write-backs.
5. **L2 cache to off-chip memory latency** is the time spent accessing memory including the time spent communicating with the memory controller and the queuing delay incurred due to finite off-chip bandwidth.
6. **Synchronization latency** is the time spent waiting due to application synchronization operations such as acquiring locks or barriers.

The energy consumption of the memory system which includes the L1-I cache, L1-D cache, L2 cache, directory, network routers, network links, and ECC logic is also measured. The goal with the `NUCA-L1` architecture is to minimize *Completion Time* to deliver high performance while operating at an energy efficient voltage.

2.5 Results

A detailed per-benchmark analysis of the proposed architecture is performed and the performance and energy consumption results are compared to the ideal fault-free baseline . The systems are evaluated at four Vccmin points, resulting in four different bit-fault rates. The voltage point being used is abstracted out, as it is technology dependent, and present results for the four bit-fault rates ranging from low (0.05%) to very high (0.3%). The results are organized as follows.

The results and analysis of the proposed architecture compared to Archipelago and SECDED with word-level disabling are presented in Section 2.5.1. In Section 2.5.2 the proposed NUCA−L1 architecture performance is compared with the different patch-up-and-recovery schemes available. Similarly, Section 2.5.3 presents the comparison with the different ECC based techniques. Section 2.5.4 shows the comparison between NUCA−L1, $\text{NUCA−L1}_{\text{DEC}}$, $\text{NUCA−L1}_{\text{CL−SEC\_WLD}}$, and $\text{NUCA−L1}_{\text{WL−SEC\_WLD}}$, along with the different tradeoffs involved. Finally, Section 2.5.5 presents a study on the sensitivity of performance to the L1 cache size.

2.5.1 Comparison with the Best Performing ECC and Patch-up-and-recovery Scheme

Different schemes were evaluated and SECDED with word-level disabling and AP were found to be the best of ECC (cf. 2.5.3) and patch-up-and-recovery (cf. 2.5.2) mechanisms, respectively. In this section the per-benchmark completion time and energy results for the proposed architecture are presented, compared to SECDED with word-level disabling and state-of-the-art Archipelago systems. Results for $\text{NUCA−L1}_{\text{CL−SEC\_WLD}}$ and $\text{NUCA−L1}_{\text{WL−SEC\_WLD}}$ are also presented for comparison.

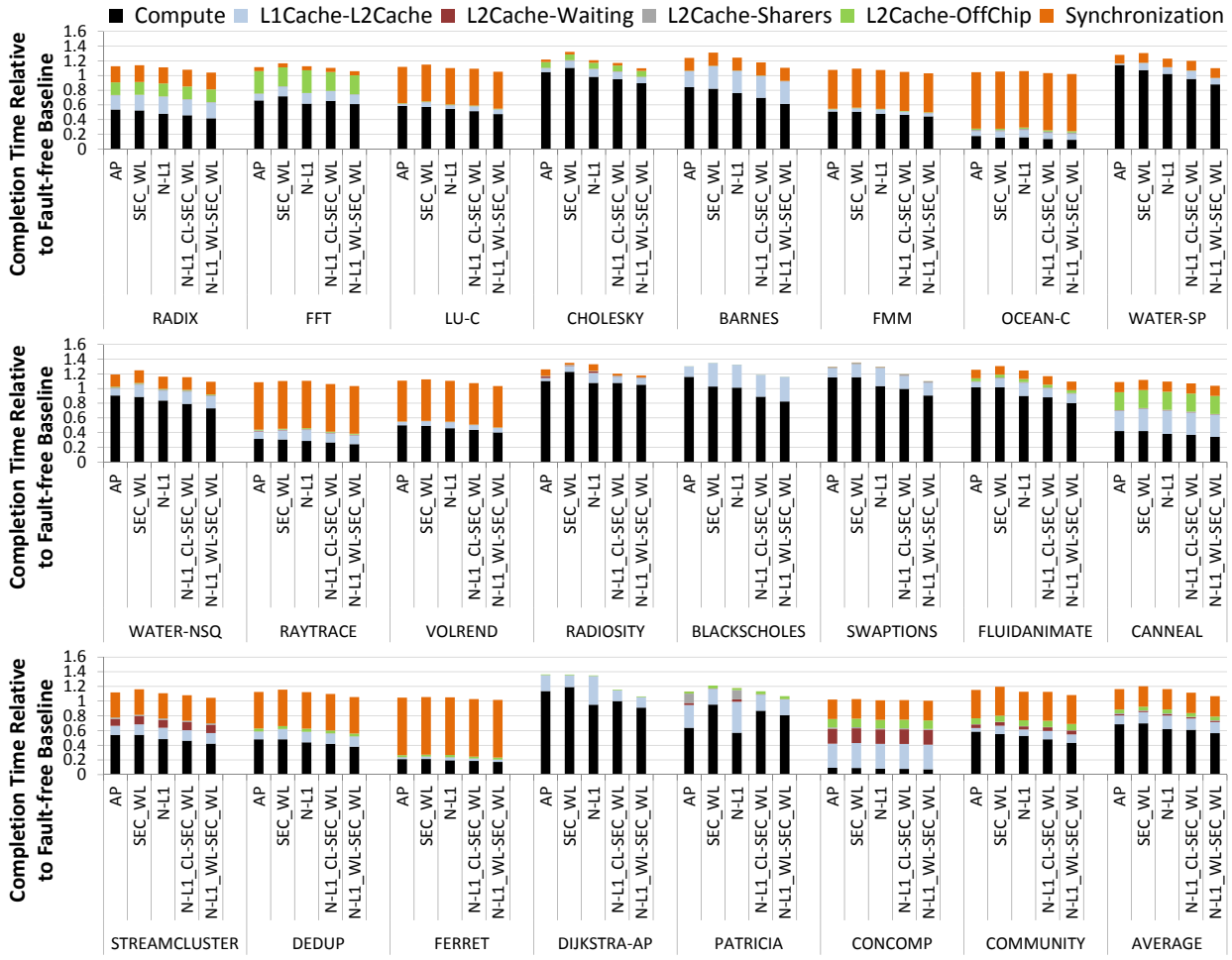


Figure 2.5.1: Completion time results at 0.3% fault rate for AP, SECDED with word-level disabling, NUCA−L1, NUCA−L1$_{\text{CL-SEC_WLD}}$, and NUCA−L1$_{\text{WL-SEC_WLD}}$. Results are normalized to the ideal fault-free baseline.

0.3% fault Rate

0.3% is the highest fault rate evaluated in the performed experiments. This corresponds to an extreme near-threshold voltage.

Per-Benchmark Completion Time Figure 2.5.1 shows the per-benchmark completion time results for 0.3% fault rate. At this extreme fault rate, the usable capacity (cache lines with zero-

and single-bit faults) for the proposed architecture and SECDED is ∼68.3% while that for AP is ∼92%. The lower capacity available for the proposed architecture, in comparison to AP, results in higher L1 miss rate. However, the lower average hit latency compensates for the higher miss rate. There is a tradeoff in play which dictates the performance of a benchmark based on its access pattern and its active working set. Benchmarks with large active working set tend to put high stress on the L1 caches, resulting in higher capacity misses at lower available capacity. On the other hand, benchmarks that have a smaller active working set take advantage of the lower average L1 hit latency.

SWAPTIONS is a benchmark that shows the lower average hit latency and capacity tradeoff. The proposed architecture reduces the average L1 hit latency significantly (shown by "Compute" in fig. 2.5.1). However, the reduced available capacity impacts the L1 miss rate by 1.2%, enough to offset any gains. Similar trend of lower average hit latency and higher miss rate is observed in several benchmarks such as FFT, BARNES, BLACKSHOLES, FLUIDANIMATE, and DIJKSTRA.

The active working set and access pattern of WATER-SPATIAL is such that the L1 capacity miss rate is very low. Although the L1 capacity miss rate increases for the proposed `NUCA-L1` architecture over AP, it is only ∼0.5% increase. This small change in L1 miss rate does not impact the performance in a big way. However, the performance improvement gained by lowering the average L1 hit latency is significant. This results in an overall improvement in completion time for WATER-SPATIAL by 5% in comparison to the baseline. Similar behavior can be noticed in WATER-NSQUARED, DIJKSTRA and STATIC-COMMUNITY.

CONNECTED-COMPONENTS exhibits high L1 miss rate. Due to this high L1 miss rate, the completion time is dominated by the time the system spend in servicing L1 misses. The `NUCA-L1` architecture does improve the average L1 hit latency but the L1 miss rate is rather high and dominates the overall completion time. This high miss rate is due to the access pattern of the workload, as the miss rate does not increase on the `NUCA-L1` architecture.

The proposed NUCA−L1 architecture improves the average L1 hit latency in RADIOSITY. However, the capacity becomes a major bottleneck. The lower available capacity results in an increase in L1 miss rate, enough to overcome any gain due to lowering of average L1 hit latency. The resulting overall performance for AP is lower by 7% than NUCA−L1, as it has more available capacity to work with. Other benchmarks showing the same trend, but with lower degradation in performance, include OCEAN_CONTIGUOUS, RAYTRACE, BLACKSHOLES, and PATRICIA.

The available capacity for SECDED with word-level disabling is very high (∼99%), which results in a very low L1 miss rate. However, the constant latency overhead and forced evictions due to accesses to disabled words degrade the performance significantly. The lower average L1 hit latency becomes the major contributing factor in the difference in completion time for the NUCA−L1 architecture. This can be seen in RADIOSITY, SWAPTIONS, FLUIDANIMATE, DIJKSTRA, FFT, WATER-SPATIAL, and WATER-NSQUARED (Fig. 2.5.1).

$\text{NUCA−L1}_{\text{CL−SEC\_WLD}}$ and $\text{NUCA−L1}_{\text{WL−SEC\_WLD}}$ perform better than NUCA−L1 in all benchmarks. The reason being the higher available capacity in both the configurations. However, it should be noted that these systems come with additional storage overhead, along with increase in hardware complexity, as discussed in Sec 2.3.4.

$\text{NUCA−L1}_{\text{WL−SEC\_WLD}}$ performs better than $\text{NUCA−L1}_{\text{CL−SEC\_WLD}}$ because it selectively applies SECDED at a finer granularity of word level. This effectively reduces the number of requests to the ECC encoder and decoder, as only faulty words in the cache line need to be corrected, whereas each access to such a cache line incurs additional latency in $\text{NUCA−L1}_{\text{CL−SEC\_WLD}}$ configuration.

The proposed NUCA−L1 architecture performs within 16% of the ideal fault-free baseline, on-par with AP. SECDED with word-level disabling performs 20% worse than the ideal fault-free baseline. $\text{NUCA−L1}_{\text{WL−SEC\_WLD}}$ performs only 6.7% worse than the ideal fault-free baseline, however, it comes with additional overhead and complexity on top of NUCA−L1.

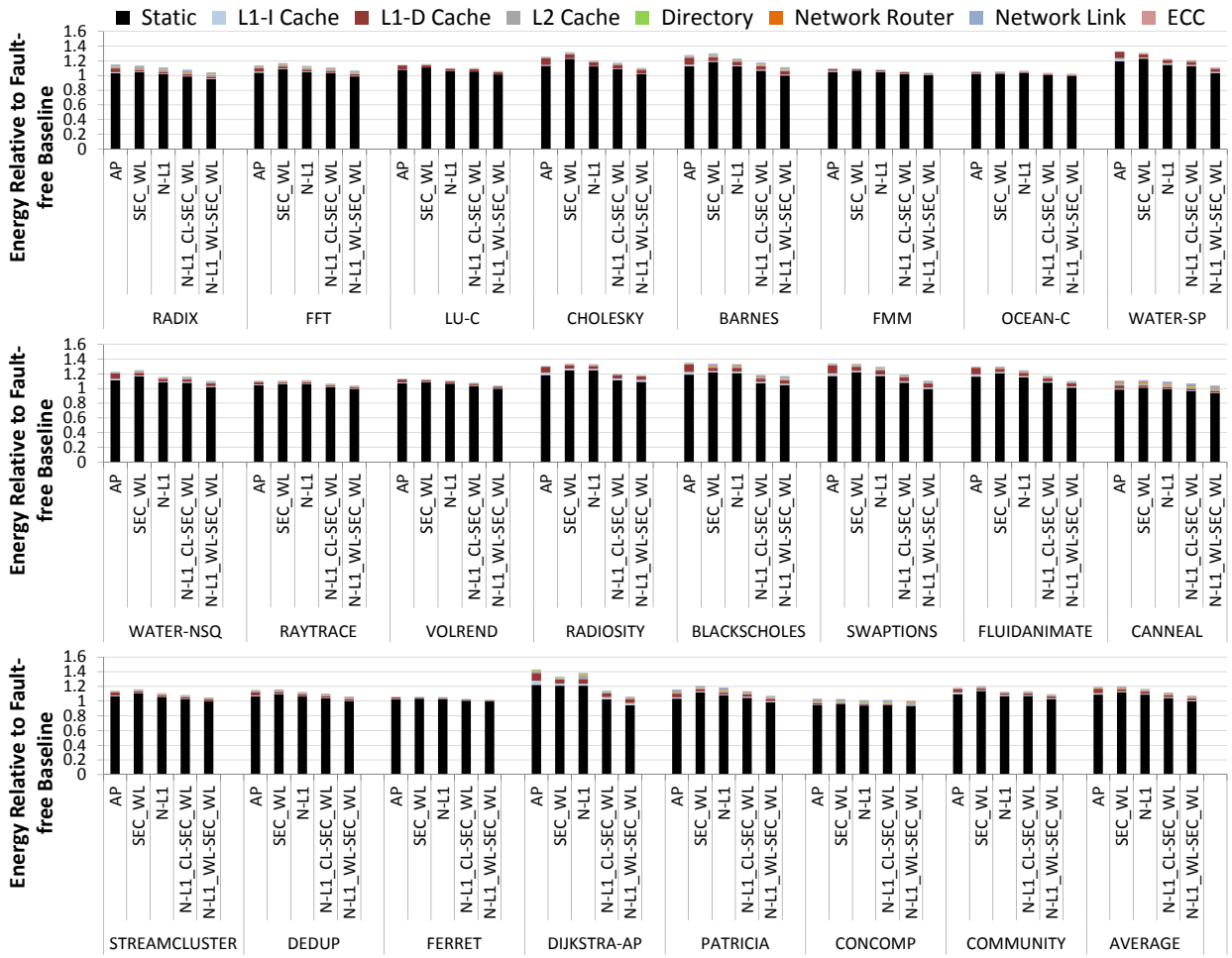


Figure 2.5.2: Energy results at 0.3% fault rate for AP, SECDED with word-level disabling, NUCA−L1, NUCA−L1$_{\text{CL-SEC_WLD}}$, and NUCA−L1$_{\text{WL-SEC_WLD}}$. Results are normalized to the ideal fault-free baseline.

Per-Benchmark Energy The per-benchmark energy results are shown in figure 2.5.2. As static energy is the biggest component of the overall energy, it dictates the energy trends observed. The static energy tracks the completion time closely and the systems that reduce completion time end up reducing static energy as well. As NUCA−L1 and AP have similar completion time, their static energy consumption is also very close. NUCA−L1$_{\text{CL-SEC_WLD}}$, and NUCA−L1$_{\text{WL-SEC_WLD}}$ reduce the completion time, hence reducing the static energy as well.

It is observed that the dynamic energy for AP system is considerably higher as compared to

the other systems. Although it improves on L2 cache, network router, and network link dynamic energy and overall static energy, the impact of increase in dynamic energy in L1 caches is high. The reason for such an increase in L1 cache dynamic energy is the way AP accesses the cache. For each access to the L1 cache it accesses two cache banks, each with multiple ways in it[1]. This doubles the dynamic energy spent on each L1 cache access.

AP decreases L1 miss rate in most of the benchmarks, as discussed previously. This in turn helps decrease the L2 cache, network router, and network link dynamic energy by $\sim 1\%$. Moreover, the static energy consumption of AP is on-par with NUCA−L1. However, the dynamic energy of L1 caches increase by $\sim 4\%$, resulting in an overall increase in energy of 3% over NUCA−L1.

As the L1 miss rate for SECDED with word-level disabling is very low compared to NUCA−L1, the resulting dynamic energy consumption is also very low. SECDED with word-level disabling improves on L1 data cache, L2 cache, and the network dynamic energy but spends more dynamic energy on ECC. Furthermore, it spends significantly higher static energy than NUCA−L1.

$\text{NUCA−L1}_{\text{CL−SEC\_WLD}}$ and $\text{NUCA−L1}_{\text{WL−SEC\_WLD}}$ are helped by their high available capacity and result in lower overall energy consumption overhead of 12% and 11%, respectively. NUCA−L1 architecture spends 16% more energy over the ideal fault-free baseline. In comparison, AP spends $>19\%$ more energy than the ideal fault-free baseline.

0.05% fault Rate

Per-Benchmark Completion Time 0.05% is the lowest fault rate considered in the evaluation. At this fault rate, the usable capacity is ∼97.5% for NUCA−L1 and ∼99.9% for SECDED with word-level disabling. Majority of the accesses, are made to fault-free cache lines, resulting in

[1]The banks can be accessed in a more intelligent way to avoid looking up multiple banks on each access. However, the system is modeled as proposed in the paper [12].

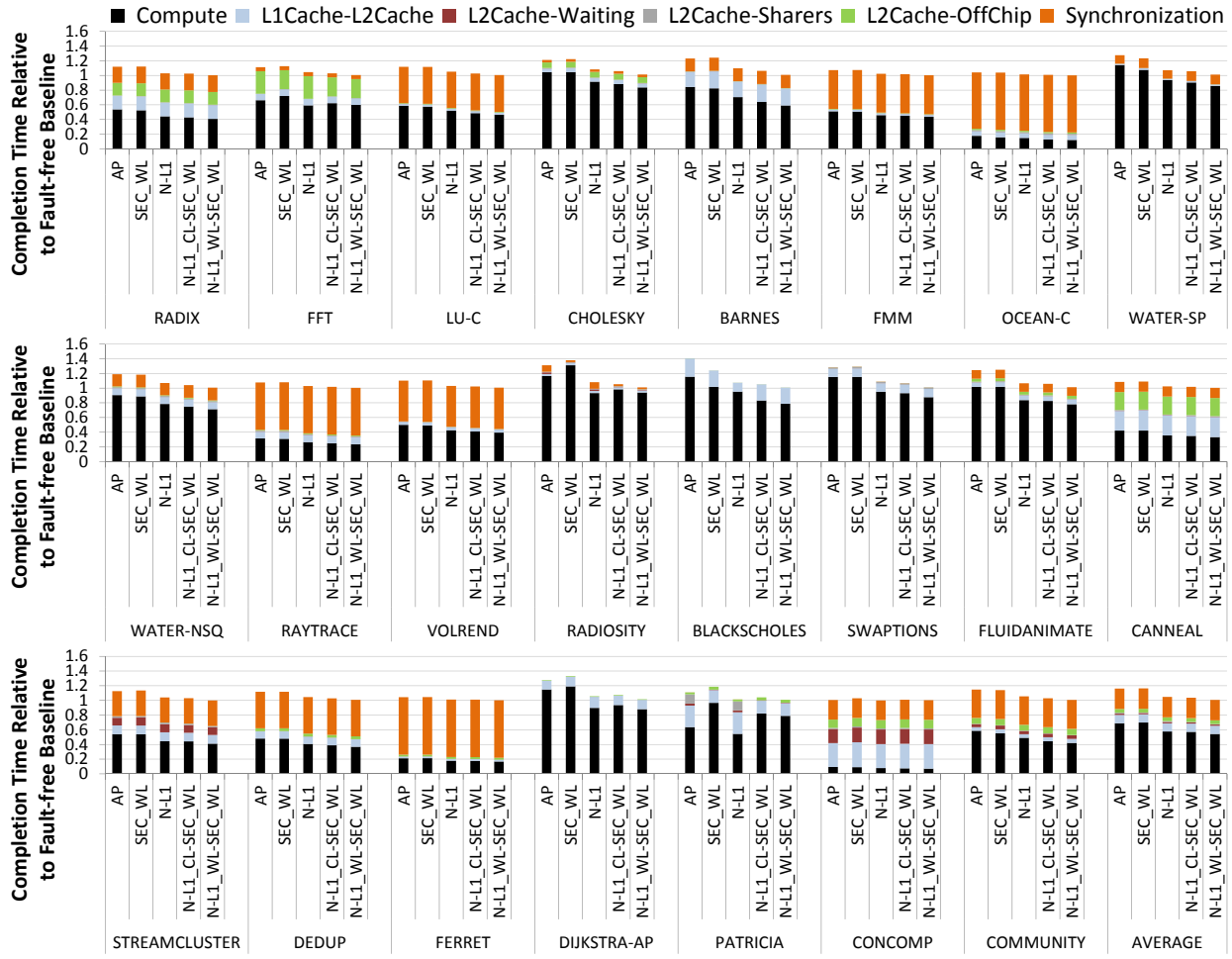


Figure 2.5.3: Completion time results at 0.05% fault rate for AP, SECDED with word-level disabling, NUCA−L1, NUCA−L1$_{\text{CL-SEC_WLD}}$, and NUCA−L1$_{\text{WL-SEC_WLD}}$. Results are normalized to the ideal fault-free baseline.

optimal average hit latency in NUCA−L1.

The miss rate of BARNES increases by 30% over the ideal fault-free baseline's miss rate for NUCA−L1, resulting in a performance degradation of 9.7%. NUCA−L1$_{\text{WL-SEC_WLD}}$ has a similar miss rate. However, it improves greatly on L1 cache access latency, as 96% of the words are accessed without any additional latency and 3.9% word accesses incur a single cycle latency overhead. In comparison, NUCA−L1 access 76% of the cache lines without any latency overhead and 21% cache line accesses incur additional one cycle latency. Similar behavior is observed in

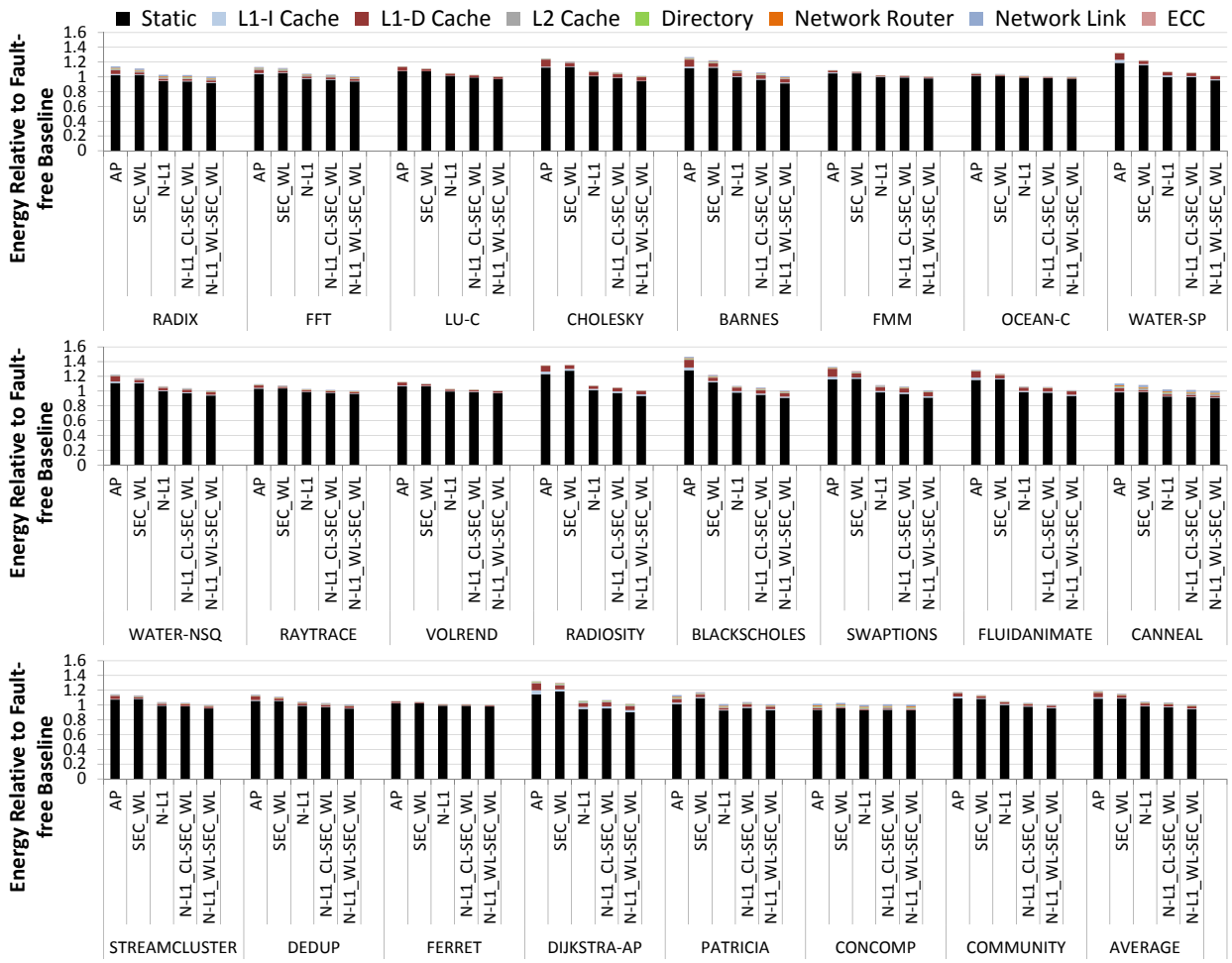


Figure 2.5.4: Energy results at 0.05% fault rate for AP, SECDED with word-level disabling, NUCA−L1, NUCA−L1$_{\text{CL-SEC_WLD}}$, and NUCA−L1$_{\text{WL-SEC_WLD}}$. Results are normalized to the ideal fault-free baseline.

CHOLESKY, WATER-SPATIAL, WATER-NSQUARED, RADIOSITY, BLACKSCHOLES, SWAPTIONS, and FLUIDANIMATE.

At 0.05% bit-cell fault rate, the average hit latency of NUCA−L1 architecture is very close to the ideal fault-free baseline. NUCA−L1 performs within 5% of the ideal fault-free baseline (fig. 2.5.3). The average completion time is >10% better, relative to baseline, than AP and SECDED.

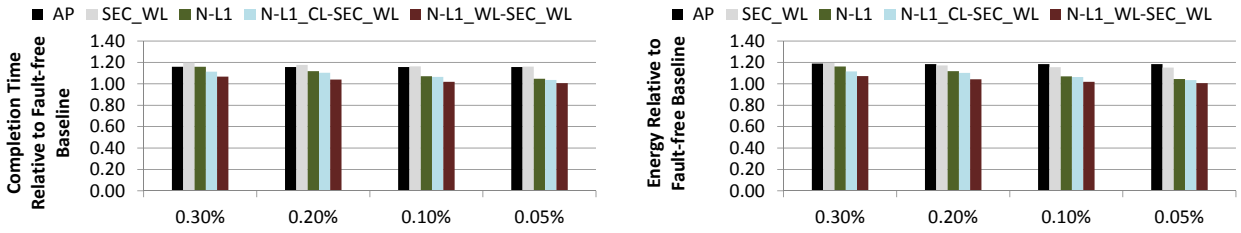


Figure 2.5.5: (a) Completion time and (b) energy consumption sensitivity to different fault rates for AP, SECDED with word-level disabling, NUCA−L1, NUCA−L1$_{\text{CL-SEC_WLD}}$, and NUCA−L1$_{\text{WL-SEC_WLD}}$. Results are normalized to the ideal fault-free baseline.

Per-Benchmark Energy The energy result also shows a similar improvement (fig. 2.5.4) as completion time. A similar trend is seen in static energy consumption as 0.3% fault rate. The systems that improve completion time also end up reducing static energy.

Most of the L1 misses due to capacity are converted in to L1 hits. This in turn reduces the L2 cache, network router, and network link dynamic energy and the static energy. The NUCA−L1 consumes 4.5% more energy than the ideal fault-free baseline.

The energy overhead over fault-free baseline decreases from 16% to 4.5% when the bit-fault rate is decreased from 0.3% to 0.05%. From these results it can be seen that the dynamic energy consumption of NUCA−L1 is heavily impacted by the number of L1 misses. Hence, improving the L1 hit rate results in better performance, as well as lower dynamic energy for NUCA−L1. This can be seen in NUCA−L1$_{\text{WL-SEC_WLD}}$, where the available capacity is $\sim 99.9\%$. The high available capacity results in energy consumption on-par (only 0.7% more) with the ideal fault-free baseline.

Sensitivity to Different Fault Rates

From figures 2.5.5a and 2.5.5b, it is observed that as the bit-fault rate is decreased from 0.3% to 0.05%, NUCA−L1 adapts and exploits the increased usable capacity. Its performance degradation of 16% at the highest fault rate is reduced to $< 5\%$ at the lowest fault rate. It also reduces the

energy consumption overhead from 16% down to $< 5\%$. As usable capacity for AP is high at all bit-fault rates, no significant performance improvement is observed. The energy consumption improves by a small amount and is 19% higher with respect to fault-free baseline. SECDED with word-level disabling slightly improves the completion time (from 20% to 16%) as the bit-cell fault rate goes down, along with decrease in energy consumption from 20% to 15%. Similar improvements in completion time and energy consumption are observed in $\texttt{NUCA-L1}_{\texttt{CL-SEC\_WLD}}$ and $\texttt{NUCA-L1}_{\texttt{WL-SEC\_WLD}}$.

2.5.2 Comparison with Patch-Up-and-Recovery Schemes

In this section the completion time results of NUCA-L1 are compared to systems based on architectural innovations to deal with NTV operation. The systems compared include bit-fix, word-disable, cache-line-disable, word-level disabling and AP. NUCA-L1 is able to outperform all systems at all fault rates (fig. 2.5.6a). It is observed that cache line disable performs poorly at high fault rates but improves and performs on-par with NUCA-L1 at 0.05% fault rate. The reason for this behavior is the higher available capacity at 0.05% fault rate. One can argue that NUCA-L1 should perform better than cache-line-disable, as it has more usable capacity. Although NUCA-L1 has 21% higher usable capacity than cache-line-disable, this capacity incurs an extra cycle for each hit. The result here clearly shows the dependence of overall performance on capacity at higher fault rates, and latency at lower fault rates.

2.5.3 Comparison with ECC based Schemes

In this section the completion time results for NUCA-L1 compared to SECDED with cache-line-disable, DECTED with cache-line-disable, and SECDED with word-level disabling are presented. It is observed that correction strength above SECDED is an overkill, and does not improve the system

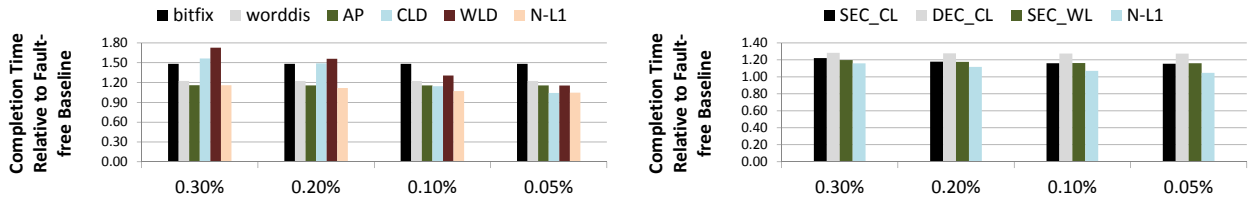


Figure 2.5.6: Geometric mean of normalized completion time of (a) patch-up-and-recovery and (b) ECC based schemes.

performance. It can be seen from fig. 2.5.6b that SECDED performs better than DECTED at all fault rates. This shows that any increase in L1 hit latency degrades the overall performance significantly. It is qualitatively argued that any scheme stronger than DECTED would be suboptimal because of the excessive increase in L1 hit latency. It can also be seen that SECDED with word-level disabling performs better than SECDED with cache-line-disable at higher fault rates. The reason being the higher available capacity. However, this slight increase in performance comes at a high cost ($\sim 6\times$ increase in storage overhead).

2.5.4 Architecture Variations for NUCA-L1

Figure 2.5.7a shows the normalized geometric mean of the completion time for NUCA-L1, $\text{NUCA-L1}_{\text{DEC}}$, $\text{NUCA-L1}_{\text{CL-SEC\_WLD}}$, and $\text{NUCA-L1}_{\text{WL-SEC\_WLD}}$ at different fault rates. The performance of $\text{NUCA-L1}_{\text{DEC}}$ is 3% better than NUCA-L1 with SCEDED correction capability at 0.3% fault rate. This improvement diminishes as the fault rates goes down, and performance is on-par with NUCA-L1 at 0.05% fault rate. The reason for this behavior is that at higher fault rates the DECTED correction capability can correct more cache lines and hence the available capacity is higher. This higher available capacity translates to lower L1 miss rate. However, at low fault rates the opportunity to recover more capacity is low.

$\text{NUCA-L1}_{\text{CL-SEC\_WLD}}$ performs better than NUCA-L1 at all fault rates, however, the difference in

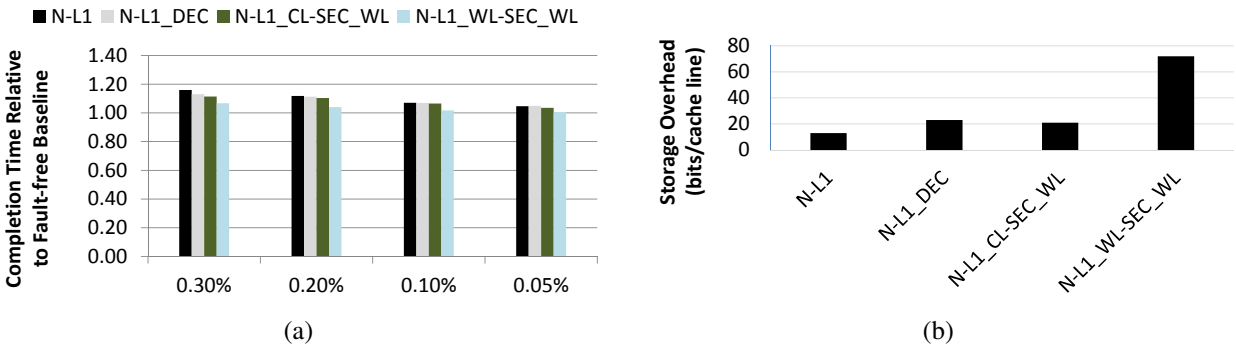
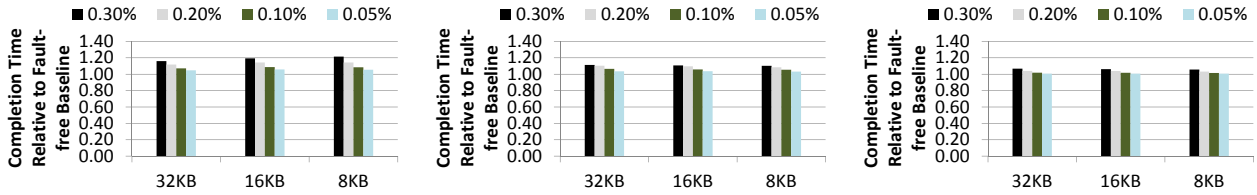


Figure 2.5.7: (a) Geometric mean of normalized completion time and (b) associated storage overhead of NUCA−L1, NUCA−L1$_{\text{CL-SEC_WLD}}$, and NUCA−L1$_{\text{WL-SEC_WLD}}$.

performance decreases as the fault rate goes down and is $<1\%$ at the lowest fault rate. It performs better than NUCA−L1$_{\text{DEC}}$ and has lower storage overhead. However, it has higher hardware complexity, as the L2 cache controller needs to be modified as well for it to work (cf. Section 2.3.4). Taking full advantage of the high available capacity, NUCA−L1$_{\text{WL-SEC_WLD}}$ performs $>4\%$ better than NUCA−L1 at all fault rates. These variations improve the overall performance, however, they come with a high storage overhead and/or increase in hardware complexity (fig. 2.5.7 (right)). If high performance is the ultimate goal and the overheads are not a problem, one can opt for NUCA−L1$_{\text{WL-SEC_WLD}}$. NUCA−L1$_{\text{CL-SEC_WLD}}$ provides a balance between the overheads and the performance improvement. However, the performance improvement is not significant enough at lower fault rates. Similarly, NUCA−L1$_{\text{DEC}}$ comes with a small overhead but the performance is not worth the extra logic and storage overhead. NUCA−L1 is a practical architecture that delivers high performance while keeping the overheads low.

2.5.5 L1 Cache Size Sensitivity

Figure 2.5.8 shows the sensitivity of the proposed NUCA−L1 architecture to three different L1 cache sizes. It is observed that at the highest fault rate, NUCA−L1 performance worsens from $\sim 16\%$ at



(a) NUCA−L1 (b) NUCA−L1$_{CL-SEC_WLD}$ (c) NUCA−L1$_{WL-SEC_WLD}$

Figure 2.5.8: Geometric mean of normalized completion time at private L1 cache sizes of $32KB$, $16KB$, and $8KB$ for NUCA−L1 variants.

$32KB$ L1 cache size to $\sim$21% at $8KB$ L1 cache size. However, the other architecture variants do not degrade because they are able to recover more cache capacity at the highest fault rate. As fault rates drop, all cache sizes evaluated show a decreasing trend in degradation of the completion time. It is noted that at low fault rates the performance dependence on cache size minimizes since the available capacity approaches the fault-free baseline.

2.6 Conclusion

In the imminent era of large-scale shared memory multicores, energy efficient operation will be critical. To enable operation at near-threshold voltages, such processors will need to handle the persistent faults due to PVT variations. Multicore processors rely on fast private L1 caches to achieve high performance. In the presence of bit-cell faults at NTV conditions, an L1 cache can either sacrifice capacity or incur additional latency to correct the faults. With this tradeoff in mind, I have proposed a novel private NUCA−L1 architecture that balances performance and energy consumption at NTV. The proposed NUCA−L1 architecture's performance and energy degradation is as low as 4.5% and 2%, respectively, in comparison to the fault-free baseline. It performs better than state-of-the-art Archipelago and SECDED with cache-line-disable by up to 11% relative to baseline. It also consume lower energy than AP by up to 14.5%.

Chapter 3

Locality-Aware Data Replication in the Last-Level Cache for Large Scale Multicores

Next generation *large* single-chip multicores will process massive data with varying degree of locality. Harnessing on-chip data locality to optimize the utilization of on-chip cache and network resources is of fundamental importance. I propose a locality-aware selective data replication protocol for the last-level cache (LLC) [59]. The goal is to lower memory access latency and energy by only replicating cache lines with *high reuse* in the LLC slice of the requesting core, while simultaneously keep the off-chip miss rate low. The approach relies on low overhead yet highly accurate in-hardware runtime cache line level classifier that only allows replication of cache lines with high reuse. Furthermore, a classifier captures the LLC pressure at the existing replica locations and adapts its replication decision accordingly.

On a set of parallel benchmarks, the proposed protocol reduces the overall energy by 14.7%, 10.7%, 10.5%, and 16.7% and the completion time by 2.5%, 6.5%, 4.5%, and 9.5% when compared

to the previously proposed Victim Replication, Adaptive Selective Replication, Reactive-NUCA, and Static-NUCA LLC management schemes. An efficient classifier implementation is evaluated with an overhead of $5.44KB$, which translates to only 1.58% on top of the Static-NUCA baseline's cache related per-core storage.

3.1 Introduction

Computing trends indicate integration of a large number of cores on a single chip. Since the diameter of on-chip networks increases with core count, the cost of moving data on-chip is becoming expensive. Furthermore, emerging technologies such as 3D die stacking [4] make the "distance" to access data even more non-uniform and costly. On the technology front, on-chip wires are not scaling at the same rate as transistors. While compute energy is projected to scale down by $6\times$ from $45nm$ to $7nm$, interconnect energy only scales down by $1.6\times$ [2]. Similar trends are projected for transistor and wire delay scaling. The many-core processors built from these technologies will execute emerging big data and streaming parallel applications that are expected to process many data structures with varying degrees of locality and reuse. Current multicore caches either fail to capture temporal locality when the reuse distance is too large, or they are simply inefficient for data structures with no or little reuse. Hence, unnecessary data movement not only impacts memory access latency, but also incurs wasteful energy consumption of the networks-on-chip and cache resources [2, 1].

3.1.1 Data Access in the Multicore Last-Level Cache

A large monolithic on-chip cache that holds the application working set, does not scale beyond a small number of cores, and the only practical option is to physically distribute on-chip memory

in pieces so that every core is near some portion of the cache [60]. In theory this provides a large amount of aggregate cache capacity and fast private memory for each core. Unfortunately, it is difficult to manage the distributed cache and network resources effectively since they require architectural support for cache coherence and consistency under the ubiquitous shared memory model.

Popular directory-based protocols enable fast local caching to exploit data locality, but scale poorly with increasing core counts [61, 62]. Many recent proposals have addressed directory scalability in single-chip multicores using sharer compression techniques or limited directories [63, 64, 65, 66]. Yet, fast private caches still suffer from two major problems: (1) due to capacity constraints, they cannot hold the working set of applications that operate on massive data, and (2) due to frequent communication between cores, data is often displaced from them [67]. This leads to increased network traffic and request rate to the last-level cache.

Last-level cache (LLC) organizations offer trade-offs between on-chip data locality and off-chip miss rate. While private LLC organizations (e.g., [7]) have low hit latencies, their off-chip miss rates are high in applications that have uneven distributions of working sets or exhibit high degrees of sharing (due to cache line replication). Shared LLC organizations (e.g., [68]), on the other hand, lead to non-uniform cache access (NUCA) [8] that hurts on-chip locality, but their off-chip miss rates are low since cache lines are not replicated.

Several proposals have explored the idea of hybrid LLC organizations [69, 70, 71, 72, 48, 73, 74]. These proposals attempt to combine the good characteristics of private and shared LLC organizations by relying on either dynamic data placement, data replication, or both. These proposals either do not quickly adapt their policies to dynamic program changes, or replicate cache lines without paying attention to their locality. In addition, some of them either add significant hardware overheads, or complicate coherence and do not scale to large core counts.

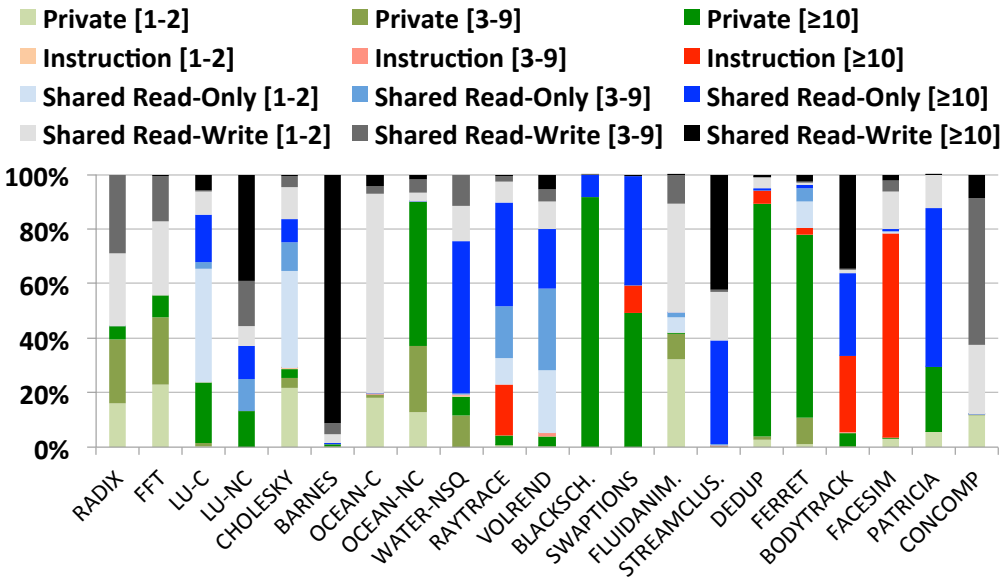


Figure 3.1.1: Distribution of instructions, private data, shared read-only data, and shared read-write data accesses to the LLC as a function of *run-length*. The classification is done at the cache line granularity for the evaluated benchmarks, using the methodology described in Section 3.4.

3.1.2 Motivation for Locality-Aware Replication in LLC

Static data placement optimally utilizes the LLC capacity. However, it suffers from sub-optimal placement of *all* data, thus results in increased on-chip network traffic and high LLC access latency. On the other hand, some key challenges with dynamic data placement schemes are: (1) the sub-optimality of shared data placement, (2) the potential over-subscription of private data at the local LLC slice, and (3) false sharing of cache lines due to misclassification of OS pages. If cache line replication is enabled in the local LLC slice of the requesting core, these challenges can be mitigated, resulting in improved data locality that lowers the LLC access latency, on-chip traffic, and off-chip miss rate.

The utility of data replication in the LLC can be understood by measuring cache line reuse. Figure 3.1.1 plots the distribution of the number of accesses to cache lines in the LLC as a function of run-length. *Run-length* is defined as the number of accesses to a cache line (at the LLC) from

a particular core before a conflicting access by another core or before it is evicted. Cache line accesses from multiple cores are conflicting if at least one of them is a write. For example, in BARNES benchmark, over 90% of the accesses to the LLC occur to shared (read-write) data that has a *run-length* of 10 or more. Greater the number of accesses with higher *run-length*, greater is the benefit of replicating the cache line in the requester's LLC slice. Hence, BARNES would benefit from replicating shared (read-write) data. Similarly, FACESIM would benefit from replicating instructions and PATRICIA would benefit from replicating shared (read-only) data. On the other hand, FLUIDANIMATE and OCEAN-C would not benefit since most cache lines experience just 1 or 2 accesses to them before a conflicting access or an eviction. For such cases, replication would increase the LLC pollution without improving data locality. Furthermore, BLACKSCHOLES would benefit because the misclassified private data would now be replicated.

3.1.3 Proposed Idea of Locality-Aware Replication in LLC

A low-overhead yet highly accurate hardware-only predictive mechanism is proposed to track and classify the *reuse* of each cache line in the LLC. A runtime classifier only allows replicating those cache lines that demonstrate *reuse* at the LLC while bypassing replication for others. When a cache line replica is evicted or invalidated, the classifier adapts by adjusting its future replication decision accordingly. This reuse tracking mechanism is decoupled from the sharer tracking structures that cause scalability concerns in traditional cache coherence protocols. Further optimizations are proposed to the reuse tracking hardware to lower the storage overheads. For this purpose a limited number of cache lines and a limited number of sharers for each tracked cache line are used to make the LLC replication decisions. The proposed scheme improves over static and dynamic NUCA baselines by intelligently replicating *all* types of cache lines in the requester's local LLC slice. The evaluation using a 64-core multicore shows an improvement of 14.7%, 10.7%, 10.5%, and 16.7% in

energy and 2.5%, 6.5%, 4.5%, and 9.5% in completion time over Victim Replication (VR) [70], Adaptive Selective Replication (ASR) [71], Reactive-NUCA (R-NUCA) [48], and Static-NUCA (S-NUCA) [8] LLC management schemes, respectively. The locality-aware protocol is advantageous because it:

1. Enables lower memory access latency and energy by selectively replicating cache lines that show *high reuse* in the LLC slice of the requesting core.

2. Better exploits the LLC by balancing the off-chip miss rate and on-chip locality using a classifier that adapts to the runtime reuse at the granularity of cache lines.

3. Mitigates the necessity of complex dynamic data placement schemes by demonstrating efficient implementations of locality-aware replication in LLC on top of both dynamic and static NUCA baselines.

4. Allows coherence complexity almost identical to that of a traditional non-hierarchical coherence protocol, since replicas are only allowed to be placed at the LLC slice of the requesting core.

5. Incurs low area overhead by tracking a limited number of cache lines and a limited number of sharers for each tracked cache line. This results in a storage overhead of $5.44KB$ per core, on top of the S-NUCA's on-chip cache capacity. However, the overall completion time is improved by 9.5% and dynamic energy consumption by 16.7%.

The rest of the chapter is organized as follows. Section 3.2 presents the details of the baseline multicore, and outlines the related work for the data placement and replication in the last-level cache (LLC). Section 3.3 presents the protocol operations as well as the architecture requirements and description of the proposed locality-aware replication in LLC. A discussion section outlines

the replica creation strategy, coherence complexity, replication at the cluster granularity, and the storage efficient classifier organization. Moreover, the overheads section presents a detailed bit storage level breakdown of the proposed classifier. In addition, the protocol overheads are discussed. Section 3.4 describes the modeling strategy for performance and dynamic energy, as well as the parallel benchmarks and LLC schemes used for evaluation. Section 3.5 evaluates the proposed locality-aware replication in LLC scheme on top of S-NUCA and R-NUCA baselines, and compares it to four popular state-of-the-art LLC management schemes. Sensitivity studies are also presented to evaluate the proposed organization of the locality classifier, replication threshold, impact of LLC size variations, as well as the impact of using an out-of-order core type to implement the multicore. Section 3.6 concludes the chapter.

3.2 Background & Related Work

3.2.1 Baseline System

The baseline system is a tiled multicore with an electrical 2-D mesh interconnection network. Each core consists of a compute pipeline, private L1 instruction and data caches, a physically distributed shared LLC cache with integrated directory, and a network router. The coherence is maintained using a MESI protocol. The coherence directory is integrated with the LLC slices by extending the tag arrays (in-cache directory organization [75, 46]), and tracks the sharing status of the cache lines in the per-core private L1 caches. The private L1 caches are kept coherent using the ACKwise limited directory-based coherence protocol [47]. Some cores have a connection to a memory controller as well. The memory controllers are placed in a way that minimizes the average distance from the tiles. The electrical mesh network uses dimension-order X-Y routing and wormhole flow control.

The ACKwise protocol maintains a limited set of hardware pointers (p) to track the sharers of a cache line. It operates like a full-map protocol when the number of sharers is less than or equal to the number of hardware pointers. When the number of sharers exceeds the number of hardware pointers, the ACKwise$_p$ protocol does not track the identities of the sharers anymore. Instead, it tracks the *number* of sharers and performs a broadcast invalidate on an exclusive request. However, acknowledgments need to be received from only the actual sharers of the data. In conjunction with a broadcast network, the ACKwise protocol has been shown to scale to large number of cores [47]. The 2-D mesh network is also augmented with broadcast support. Each router selectively replicates a broadcasted message on its output links such that all cores are reached with a single injection.

3.2.2 Data Placement

This section describes state-of-the-art static-NUCA and dynamic-NUCA data placement schemes.

Static-NUCA

The application's address space is divided in subsets and each subset is mapped to an LLC slice, known as the "home" for that subset of addresses. In case of a private cache miss, a request is sent to the home LLC slice based on a simple address based decode. Static interleaving at the fine granularity of cache lines fully exploits the on-chip LLC capacity. However, majority of the accesses are to remote LLC slices, resulting in an increase in on-chip network traffic. In short, it utilizes the on-chip capacity efficiently, however, it hurts the locality of private data. This scheme is straightforward to implement and does not come with any area overhead.

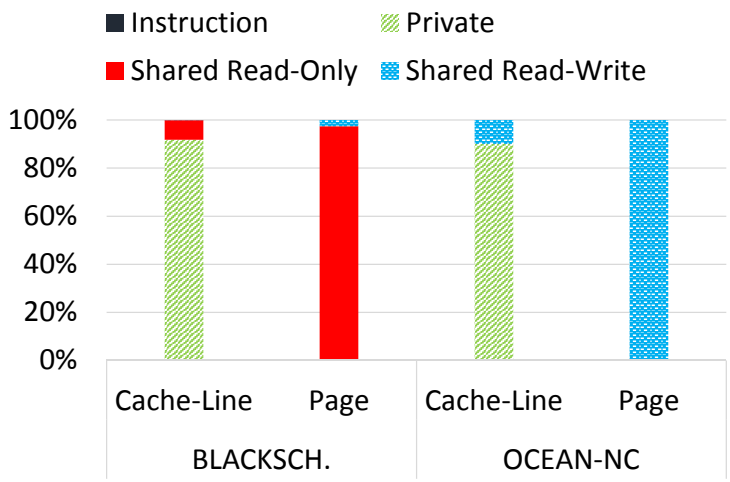


Figure 3.2.1: Distribution of instructions, private data, shared read-only data, and shared read-write data accesses to the LLC for BLACKSCHOLES and OCEAN-NC. The classification is done at the cache line and page granularity, highlighting false sharing under a page classification mechanism.

Dynamic-NUCA

Reactive-NUCA (R-NUCA) [48] classifies data as private or shared at page granularity using the existing operating system virtual memory management mechanism. All the cache lines of a private page are placed at the requesting core's LLC slice. Shared data tends to exhibit limited affinity to a particular core since it is accessed by multiple cores over time. Therefore, cache lines of shared pages are assigned a home core based on static address interleaving. This allows R-NUCA to exploit locality on private data and enable low average access latency/energy for private data. However, it still suffers from high average access latency/energy for shared data access.

A drawback of R-NUCA is that it requires modifications in the virtual memory system, which has several serious ramifications. Firstly, the OS needs to be modified to take into account the new information in the page-table and Translation Lookaside Buffer (TLB). Secondly, the page table and TLB structures are modified and the hardware now needs to support migration of data pages at runtime. The OS support for page classification extends each TLB entry by 1 bit and each

page table entry by an additional $\log_2$(n) bits, where *n* is the number of cores. Additionally, in workloads with primarily private data, the on-chip cache capacity is underutilized and can result in higher off-chip miss rate. Similar underutilization can occur in a multi-program setup. Finally, the page-level classification under certain circumstances results in false classification of private data as shared. This is evident in figure 3.2.1 which plots the distribution of the number of accesses in the LLC to instructions, private data, shared read-only data, and shared read-write data at the cache line and page level separately. Due to false sharing, almost all true private data in the BLACKSCHOLES and OCEAN_NON_CONTIGUOUS benchmarks is classified as shared at the page-level.

Other Related Work

Cho et. al. [76] propose an OS-assisted data allocation scheme using *S-NUCA* organization and page coloring. It uses the first-touch color assignment, i.e., a page is assigned a color that places it in the LLC slice nearest to the core that first touches that page. This approach potentially leads to inefficient use of on-chip cache capacity, and puts pressure on some heavily used LLC slices. To mitigate this challenge, it proposes to augment first-touch with page-spreading policies. Therefore, subsequent page requests from that core are assigned colors that map the pages to neighboring tile's LLC slices. Awasthi et. al. [77] and Chaudhuri et. al. [72] build on top of first-touch page coloring to solve the challenge of sub-optimal home location for shared data. They propose a further level of indirection to migrate pages in order to improve the locality of shared data. Although a step in the right direction, they introduce several non-trivial overheads in the process. R-NUCA is a relatively simpler dynamic-NUCA scheme that avoids adding complexity to the underlying coherence protocol. Therefore, R-NUCA is chosen as the representative dynamic data placement scheme.

3.2.3 Data Replication

This section describes the data replication schemes used in evaluation, as well as other related work to complete the discussion. Cache line replication in the LLC can be deployed to improve locality for instructions, shared, and private data. Furthermore, misclassified private data, as seen in figure 3.2.1, can also be replicated in the local LLC slice of the requester. In order to enable data replication at the cache line granularity, the baseline lookup strategy for the LLC is slightly modified for all schemes. The local LLC slice is always looked up on an L1 cache miss or eviction. Additionally, both the L1 cache and LLC slice are probed on every asynchronous coherence request (i.e., *invalidate*, *downgrade*, *flush* or *write-back*). This is needed because the directory only has a single pointer to track the local cache hierarchy of each core. This method also allows the coherence complexity to be similar to that of a non-hierarchical (flat) coherence protocol.

Victim Replication

In *Victim Replication (VR)* [70], a cache line is replicated in the local LLC slice upon eviction from the L1 cache if either, (1) an invalid cache line exists, (2) a home cache line with no sharers exists, or (3) another replica exists in the target set. *VR* implements a random replacement policy within a candidate group, with priority in the order listed above. If no such cache line exists, the evicted cache line is communicated back to its home core. Upon a subsequent access resulting in a hit, the replica is invalidated in the local LLC slice and moved to the L1 cache. Because of this exclusive relationship, clean cache lines that would otherwise be simply invalidated are sent back to the local LLC upon eviction. This results in unnecessary movement of data between the caches, increasing the cache energy consumption.

Adaptive Selective Replication

Adaptive Selective Replication (ASR) [71] replicates cache lines in the requester's local LLC slice on an L1 eviction. However, it only allows LLC replication for cache lines that are classified as shared read-only and does not exploit locality for other types of data. ASR implements hardware monitoring circuits that quantify the replication effectiveness and then probabilistically decides whether to replicate or not on a per-core basis. ASR assumes an 8-core processor and the LLC lookup mechanism that needs to search all LLC slices does not scale to large core counts. Furthermore, the hardware monitoring circuits add a storage overhead of more than $18KB$ per core.

Locality-Aware Data Replication

Motivated by the fact that cache lines exhibit varying degrees of reuse at the LLC, as discussed earlier in figure 3.1.1, the locality-aware data replication scheme is proposed [78]. This scheme replicates cache lines in the local LLC slice based on their reuse captured using an in-hardware cache line level classifier. A cache-line level classifier is introduced at the LLC to distinguish between low and high reuse cache lines. Cache lines that exhibit high reuse are allowed to be replicated in the local LLC slice of the requesting core, while bypassing replication in the local LLC slice for the others.

The overhead of tracking the reuse information for all cache line sharers and for every cache line in the LLC (called "Complete classifier"), is prohibitively high (cf. Section 3.3.5). To mitigate this serious bottleneck, a horizontal reduction is proposed by tracking only a few sharers for each cache line and taking a majority vote for other sharers, this is called the *Limited*$_k$ classifier, where 'k' represents the number of sharers tracked for each cache line. To make the replication scheme even more practical, a vertical reduction is proposed by tracking only a subset of the LLC cache lines, this is called the *Limited*$_{k-m}$ classifier, where 'm' represents the number of cache lines tracked

in each LLC slice. The locality information for these limited number of cache lines is stored in a self-standing structure, called "locality table". This organization completely decouples the locality tracking structure from the directory. The locality tracking does not interfere with the underlying coherence protocol in any manner. However, a parallel lookup is now required to extract the locality information. This does not impact performance, as the lookup is not in the critical path and is hidden by the parallel LLC slice lookup. The locality-aware replication scheme and its optimizations are discussed in detail in Section 3.3.

Other Related Work

CMP-NuRAPID [69] decouples tag and data arrays. It maintains private per-core tag arrays and a shared data array divided into multiple distance groups. It replicates shared read-only cache lines in the cache bank closest to the requesting core on its second access. CMP-NuRAPID does not allow replication of shared read-write data and only one copy can exist in the cache. Also, it forces write-through for such cache lines to maintain coherence using an additional Communication (C) coherence state. As each private per-core tag array potentially has to store pointers to the entire data array, CMP-NuRAPID does not scale with the number of cores. In addition, CMP-NuRAPID requires snooping coherence to broadcast invalidate replicas, and complicates the coherence protocol significantly by introducing many additional race conditions that arise from the tag and data array decoupling.

Proposals that start with a private-L1, private LLC organizations have been proposed [79, 80, 81, 82, 83]. These proposals attempt to improve the negative feature of private LLC i.e. the low hit rates, at the same time try to capitalize on the favorable property i.e. low LLC access latency. They start with a private-private organization and then either spill data into other private caches, or reconfigure the private cache size to reduce the off-chip misses and efficiently utilize the on-chip

capacity. However, keeping track of and figuring out where to place the spilled data is a challenge by itself. These schemes generally rely on coarse-grain information that are not very accurate. Furthermore, they all suffer from the complex hardware structures required to keep the private caches coherent.

The locality-aware replication in LLC protocol starts with a shared LLC organization. However, it allows replicating cache lines in the LLC based on reuse information gathered at a fine granularity of cache line.

3.3 Locality-Aware Data Replication in the Last-Level Cache

Locality-aware data replication protocol for the LLC [59] starts off with the static data placement scheme (S-NUCA). It then captures the reuse of cache lines using an in-hardware cache line level classifier. If enough reuse is observed for a certain sharer of a cache line, it replicates that cache line in that sharer's local LLC slice. For sharers that exhibit only low reuse, it disallows replication in the local LLC slice. This section first describes the protocol operation, classifier organization and architecture, and finally the overheads.

3.3.1 Protocol Operation

The four essential components of data replication in LLC are: (1) choosing which cache lines to replicate, (2) determining where to place a replica, (3) how to lookup a replica, and (4) how to maintain coherence for replicas. Lets define a few terms to facilitate describing the protocol.

- **Home Location**: The core where all requests for a cache line are serialized for maintaining coherence.

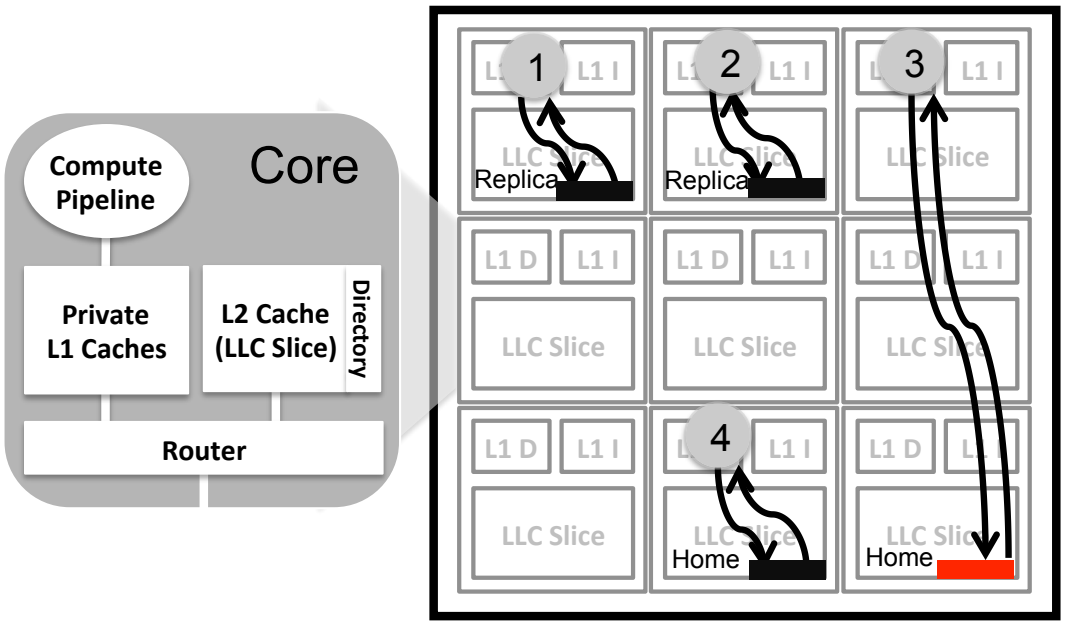


Figure 3.3.1: textcircled1 – ④ are mockup requests showing the *locality-aware LLC replication* protocol. The *black* data block has high reuse and a local LLC replica is allowed that services requests from ① and ②. The low-reuse *red* data block is not allowed to be replicated at the LLC, and the request from ③ that misses in the L1, must access the LLC slice at its home core. The home core for each data block can also service local private cache misses (④).

- **Replica Sharer**: A core that is granted a replica of a cache line in its LLC slice.
- **Non-Replica Sharer**: A core that is *NOT* granted a replica of a cache line in its LLC slice.
- **Replica Reuse**: The number of times an LLC replica is accessed before it is invalidated or evicted.
- **Home Reuse**: The number of times a cache line is accessed at the LLC slice in its home location before a conflicting write or eviction.
- **Replication Threshold (RT)**: The reuse above or equal to which a replica is created.

Note that for a cache line, one core can be a replica sharer while another can be a non-replica sharer. The protocol starts out as a conventional directory protocol and initializes all cores as

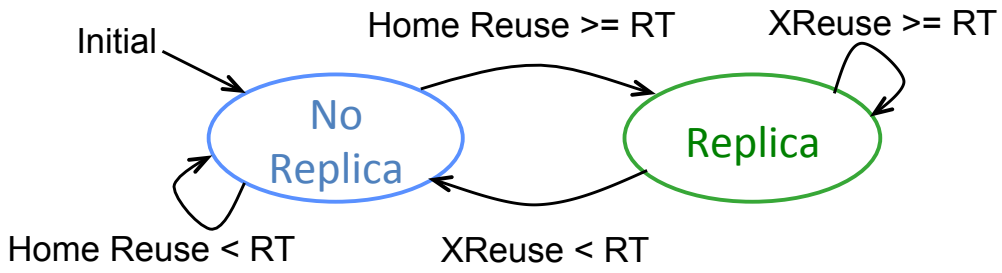


Figure 3.3.2: Each directory entry is extended with *replication mode* bits to classify the usefulness of LLC replication. Each cache line is initialized to non-replica mode with respect to all cores. Based on the reuse counters (at the home as well as the replica location) and the parameter RT, the cores are transitioned between replica and non-replica modes. Here XReuse is (Replica + Home) Reuse on an invalidation and Replica Reuse on an eviction.

non-replica sharers of all cache lines (as shown by Initial in Figure 3.3.2). Let us understand the handling of read requests, write requests, evictions, invalidations, and downgrades as well as cache replacement policies under this protocol.

Read Requests

On an L1 cache read miss, the core first looks up its local LLC slice for a replica. If a replica is found, the cache line is inserted at the private L1 cache. In addition, a *Replica Reuse* counter (as shown in Figure 3.3.3) at the LLC directory entry is incremented. The replica reuse counter is a saturating counter used to capture reuse information. It is initialized to '1' on replica creation and incremented on every replica hit.

On the other hand, if a replica is not found, the request is forwarded to the LLC home location. If the cache line is not found there, it is either brought in from the off-chip memory or the underlying coherence protocol takes the necessary actions to obtain the most recent copy of the cache line. The directory entry is augmented with additional bits as shown in Figure 3.3.3. These bits include (a) *Replication Mode* bit and (b) *Home Reuse* saturating counter for each core in the system. Note that adding several bits for tracking the locality of each core in the system does not scale with the

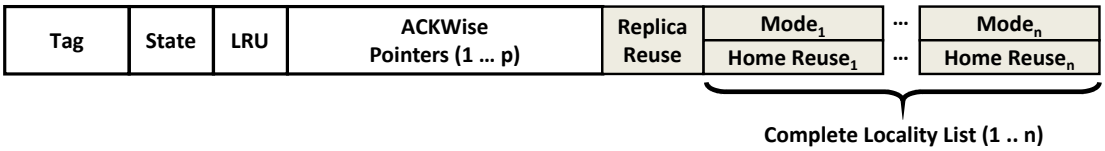


Figure 3.3.3: ACKwise$_p$-*Complete* locality classifier LLC tag entry. It contains the tag, LRU bits and directory entry. The directory entry contains the state, ACKwise$_p$ pointers, a *Replica reuse* counter as well as *Replication mode* bits and *Home reuse* counters for every core in the system.

number of cores, therefore, cost-efficient classifier implementations are presented in Sections 3.3.2 and 3.3.3. The replication mode bit is used to identify whether a replica is allowed to be created for the particular core. The home reuse counter is used to track the number of times the cache line is accessed at the home location by the particular core. This counter is initialized to '0' and incremented on every hit at the LLC home location.

If the replication mode bit is set to *true*, the cache line is inserted in the requester's LLC slice and the private L1 cache. Otherwise, the home reuse counter is incremented. If this counter has reached the *Replication Threshold (RT)*, the requesting core is "promoted" (the replication mode bit is set to *true*) and the cache line is inserted in its LLC slice and private L1 cache. If the home reuse counter is still less than RT, a replica is not created. The cache line is only inserted in the requester's private L1 cache.

If the LLC home location is at the requesting core, the read request is handled directly at the LLC home. Even if the classifier directs to create a replica, the cache line is just inserted at the private L1 cache.

Write Requests

On an L1 cache write miss for an exclusive copy of a cache line, the protocol checks the local LLC slice for a replica. If a replica exists in the *Modified(M)* or *Exclusive(E)* state, the cache line is inserted at the private L1 cache. In addition, the *Replica Reuse* counter is incremented.

If a replica is not found or exists in the *Shared(S)* state, the request is forwarded to the LLC home location. The directory invalidates all the LLC replicas and L1 cache copies of the cache line, thereby maintaining the single-writer multiple-reader invariant [84]. The acknowledgments received are processed as described in Section 3.3.1. After all such acknowledgments are processed, the *Home Reuse* counters of all non-replica sharers other than the writer are reset to '0'. This has to be done since these sharers have not shown enough reuse to be "promoted".

If the writer is a non-replica sharer, its home reuse counter is modified as follows. If the writer is the only sharer (replica or non-replica), its home reuse counter is incremented, else it is reset to '1'. This enables the replication of migratory shared data at the writer, while avoiding it if the replica is likely to be downgraded due to conflicting requests by other cores.

Evictions and Invalidations

On an invalidation request, both the LLC slice and L1 cache on a core are probed and invalidated. If a valid cache line is found in either caches, an acknowledgment is sent to the LLC home location. In addition, if a valid LLC replica exists, the replica reuse counter is communicated back with the acknowledgment. The locality classifier uses this information along with the home reuse counter to determine whether the core stays as a replica sharer. If the *(replica + home)* reuse is $\geq RT$, the core maintains replica status, else it is demoted to non-replica status (as shown in Figure 3.3.2). The two reuse counters have to be added since this is the total reuse that the core exhibited for the cache line between successive writes.

When an L1 cache line is evicted, the LLC replica location is probed for the same address. If a replica is found, the dirty data in the L1 cache line is merged with it, else an acknowledgment is sent to the LLC home location. However, when an LLC replica is evicted, the L1 cache is probed for the same address and invalidated. An acknowledgment message containing the replica reuse

counter is sent back to the LLC home location. The replica reuse counter is used by the locality classifier as follows. If the *replica* reuse is $\geq RT$, the core maintains replica status, else it is demoted to non-replica status. Only the replica reuse counter has to be used for this decision since it captures the reuse of the cache line at the LLC replica location.

After the acknowledgment corresponding to an eviction or invalidation of the LLC replica is received at the home, the locality classifier sets the home reuse counter of the corresponding core to '0' for the next round of classification.

The eviction of an LLC replica back-invalidates the L1 cache (as described earlier). A possibly more optimal strategy is to maintain the validity of the L1 cache line. This requires two message types as well as two messages, one to communicate back the reuse counter on the LLC replica eviction and another to communicate the acknowledgment when the L1 cache line is finally invalidated or evicted. Back-invalidation was chosen for two reasons: (1) to maintain the simplicity of the coherence protocol, and (2) the energy and performance improvements of the more optimal strategy are negligible since (a) the LLC is more than $4\times$ larger than the L1 cache, thereby keeping the probability of evicted LLC lines having an L1 copy extremely low, and (b) the LLC replacement policy prioritizes retaining cache lines that have L1 cache copies.

LLC Replacement Policy

Traditional LLC replacement policies use the least recently used (LRU) policy. One reason why this is sub-optimal is that the LRU information cannot be fully captured at the LLC because the L1 cache filters out a large fraction of accesses that hit within it. In order to be cognizant of this, the replacement policy should prioritize retaining cache lines that have L1 cache sharers. Some proposals in literature accomplish this by sending periodic *Temporal Locality Hint* messages from the L1 cache to the LLC [85]. However, this incurs additional network traffic.

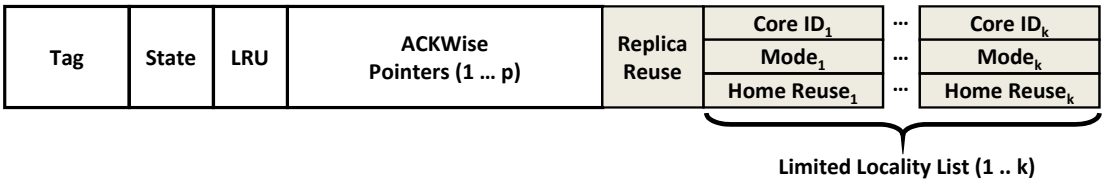


Figure 3.3.4: ACKwise$_p$-*Limited*$_k$ locality classifier LLC tag entry. It contains the tag, LRU bits and directory entry. The directory entry contains the state, ACKwise$_p$ pointers, a *Replica reuse* counter as well as the Limited$_k$ classifier. The Limited$_k$ classifier contains a *Replication mode* bit and *Home reuse* counter for a limited number of cores. A majority vote of the modes of tracked cores is used to classify new cores as replicas or non-replicas.

The proposed replacement policy accomplishes the same using a much simpler scheme. It first selects cache lines with the least number of L1 cache copies and then chooses the least recently used among them. The number of L1 cache copies is readily available since the directory is integrated within the LLC tags ("in-cache" directory). This reduces back invalidations to a negligible amount. It should be noted that a replica is treated the same as a home cache line with no sharers. Furthermore, a replica insertion can also evict a home cache line, which can have some undesirable effects (e.g. too many dirty home evictions) in some benchmarks.

3.3.2 *Limited*$_k$ Locality Classifier

The classifier described so far keeps track of the locality information for all cores in the directory entry. It is termed the *Complete* locality classifier, and has a storage overhead of $\sim$30% (calculated in Section 3.3.5) at 64 cores and over 5$\times$ at 1024 cores. In order to mitigate this overhead, a space efficient classifier is proposed that maintains locality information for a limited number of cores, and classifies the other cores as replica or non-replica sharers based on this information.

The locality information for each core consists of (1) the core ID, (2) the replication mode bit and (3) the home reuse counter. The classifier that maintains a list of this information for a limited number of cores (k) is termed the Limited$_k$ classifier. Figure 3.3.4 shows the information that is

tracked by this classifier. The sharer list of the ACKwise limited directory entry cannot be reused for tracking locality information because of its different functionality. While the hardware pointers of ACKwise are used to maintain coherence, the limited locality list serves to classify cores as replica or non-replica sharers. Decoupling in this manner also enables the *locality-aware* protocol to be implemented efficiently on top of other scalable directory organizations. The working of the limited locality classifier will now be described.

At startup, all entries in the limited locality list are free and this is denoted by marking all core IDs' as *Invalid*. When a core makes a request to the home location, the directory first checks if the core is already being tracked by the limited locality list. If so, the actions described previously are carried out. Else, the directory checks if a free entry exists. If it does exist, it allocates the entry to the core and the same actions are carried out.

Otherwise, the directory checks if a currently tracked core can be replaced. An ideal candidate for replacement is a core that is currently not using the cache line. Such a core is termed an *inactive* sharer and should ideally relinquish its entry to a core in need of it. A replica core becomes inactive on an LLC invalidation or an eviction. A non-replica core becomes inactive on a write by another core. If such a replacement candidate exists, its entry is allocated to the requesting core. The initial replication mode of the core is obtained by taking a majority vote of the modes of the tracked cores. This is done so as to start off the requester in its most probable mode.

Finally, if no replacement candidate exists, the mode for the requesting core is obtained by taking a majority vote of the modes of all the tracked cores. The limited locality list is left unchanged. The storage overhead for the Limited$_k$ classifier is directly proportional to the number of cores (k) for which locality information is tracked. In Section 3.5.2, the storage and accuracy tradeoffs for the *Limited*$_k$ classifier are evaluated.

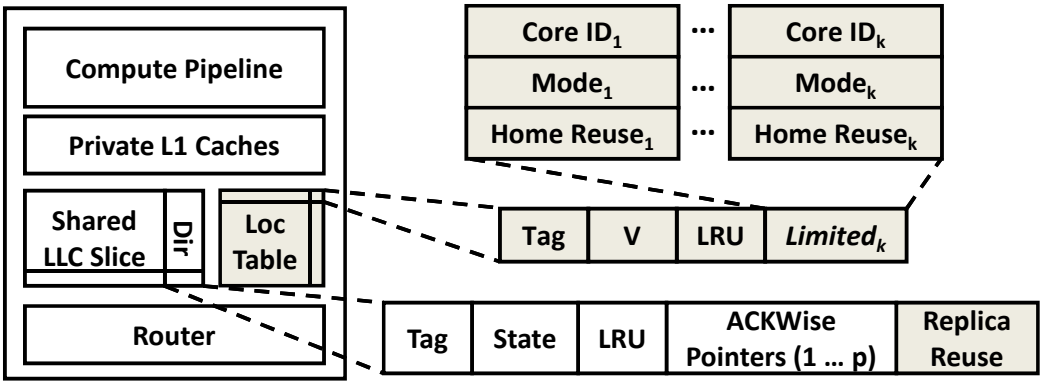


Figure 3.3.5: ACKwise$_p$-*Limited*$_{k-m}$ locality classifier LLC tag entry and the locality table structure. The LLC tag entry contains the tag, LRU bits and directory entry. The directory entry contains the state, ACKwise$_p$ pointers, a *Replica reuse* counter. Each locality table entry contains the tag, valid bit, LRU bits, and the *Limited*$_k$ classifier.

3.3.3 *Limited*$_{k-m}$ Locality Classifier

A completely decoupled locality classifier from the LLC tag array is explored to further reduce the storage overhead. This is achieved by tracking a limited number of cache lines, as opposed to all the cache lines in *Limited*$_k$ classifier. To enable tracking of a limited number of cache lines, a set-associative cache like structure, *locality table*, is introduced that is accessed in parallel with the LLC, shown in figure 3.3.5. Each entry in the locality table stores the *Limited*$_k$ classifier information for one cache line along with the tag, valid bit, and LRU bits. LRU replacement policy is used to determine a replacement candidate within a set. This is termed as a *Limited*$_{k-m}$ classifier, where 'm' represents the number of cache lines being tracked. The working of the locality table will now be described.

At startup, all entries in the locality table are free and this is denoted by marking all valid bits as *Invalid*. When a core makes a request to the home location, the locality table is also looked up in parallel to the LLC slice. If it results in a hit, it means that the cache line is being tracked. At this point, it is checked whether the core is already being tracked by the limited locality list and the actions described in Section 3.3.2 are performed. On the other hand, if the lookup results in a miss,

the classifier information for that cache line is inserted into the locality table, and the cache line is now tracked for reuse classification. Note that the classifier information for the evicted cache line from the table is lost and the classifier needs to re-learn it if the same cache line is inserted again in the locality table at some time in the future.

The eviction candidate is selected using the LRU replacement policy. In the locality table, if a cache line is heavily communicated back and forth between the cache hierarchies, it remains in the locality table and the mode is updated accordingly. On the other hand, if a cache line is used sparingly, its entry in the locality table is eventually taken by a another cache line. The LRU policy captures such access patterns, and retains cache lines with useful reuse behavior in the locality table.

The storage overhead for the Limited$_{k-m}$ classifier is directly proportional to the number of cache lines (m) for which locality information is tracked. In Section 3.5.2, the storage and accuracy tradeoffs for the *Limited*$_{k-m}$ classifier are evaluated. Based on observations, the *Limited*$_{3-512}$ classifier was picked.

3.3.4 Discussion

Replica Creation Strategy

In the locality-aware protocol, replicas are allowed to be created in all valid cache states. A simpler strategy is to create an LLC replica only in the *Shared* cache state. This enables instructions, shared read-only and shared read-write data that exhibit high read run-length to be replicated so as to serve multiple read requests from within the local LLC slice. However, migratory shared data cannot be replicated with this simpler strategy because both read and write requests are made to it in an interleaved manner. Such data patterns can be efficiently handled only if the replica is created in both *Exclusive* and *Modified* states as well. Benchmarks that exhibit both the above access patterns

are observed in the evaluation (cf. Section 3.5.1).

Coherence Complexity

The local LLC slice is always looked up on an L1 cache miss or eviction. Additionally, both the L1 cache and LLC slice is probed on every asynchronous coherence request (i.e., *invalidate*, *downgrade*, *flush* or *write-back*). This is needed because the directory only has a single pointer to track the local cache hierarchy of each core. This method also allows the coherence complexity to be similar to that of a non-hierarchical (flat) coherence protocol.

To avoid the latency and energy overhead of searching the LLC replica, one may want to optimize the handling of asynchronous requests, or decide intelligently whether to lookup the local LLC slice on a cache miss or eviction. In order to enable such optimization, additional sharer tracking bits are needed at the directory and L1 cache. Moreover, additional network message types are needed to relay coherence information between the LLC home and other actors.

In order to evaluate whether this additional coherence complexity is worthwhile, the proposed protocol is compared to a dynamic oracle that has perfect information about whether a cache line is present in the local LLC slice. The dynamic oracle avoids all unnecessary LLC lookups. The completion time and energy difference when compared to the dynamic oracle is less than 1%. Hence, in the interest of avoiding the additional complexity, the LLC replica is always looked up for the above coherence requests.

Cluster-Level Replication

In the locality-aware protocol, the location where a replica is placed is always the LLC slice of the requesting core. An additional method by which one could explore the trade-off between LLC hit latency and LLC miss rate is by replicating at a *cluster*-level. A cluster is defined as a group of

neighboring cores where there is at most one replica for a cache line. Increasing the size of a cluster would increase LLC hit latency and decrease LLC miss rate, and decreasing the cluster size would have the opposite effect. The optimal replication algorithm would optimize the cluster size so as to maximize the performance and energy benefit.

Cluster-level replication is not found to be beneficial [78]. The reasons include: (1) Using clustering increases network serialization delays since multiple locations now need to be searched/invalidated on an L1 cache miss. (2) Cache lines with low degree of sharing do not benefit because clustering just increases the LLC hit latency without reducing the LLC miss rate. (3) The added coherence complexity of clustering increases the design and verification time significantly.

Classifier Organization

The *Complete* locality classifier for the locality-aware protocol is organized using an *in-cache* structure, i.e., the replication mode bits and home reuse counters are maintained for all cache lines in the LLC. However, this is a not an essential requirement. The classifier is logically decoupled from the directory and could be implemented using a *sparse* organization.

Number of Tracked Sharers: *Limited*$_k$ with $k = 64$ corresponds to the *Complete* locality classifier for a 64-core multicore. To reduce the prohibitively high overhead of the *Complete* locality classifier, a limited number of sharers are tracked, as discussed in Section 3.5.2. Although *Limited*$_1$ has the lowest overhead, the classifier is more unstable than the others and performs significantly worse for BARNES and STREAMCLUSTER (cf. Section 3.5.2). *Limited*$_3$ is found to be the optimal classifier with substantially lower overhead and performance close to that of the *Complete* classifier (cf. Section 3.5.2).

Number of Tracked Cache Lines: *Limited*$_{k-m}$ classifier with $m = 4096$ corresponds to the *Limited*$_k$ classifier for a $256KB$ per-core LLC slice. Reducing the number of tracked cache lines

potentially impacts the accuracy of the classifier. On the other hand, not all cache lines are used at all times and tracking only a subset of cache lines can help reduce the associated overheads. The goal is to find a balance between the overhead reduction and the completion time/energy penalty paid for it. In order to quantify this tradeoff, experiments are run with varying number of tracked cache lines. It is observed that tracking 512 cache lines provides completion time/energy approaching that of tracking all cache lines (cf. Section 3.5.2), while reducing the space overhead by more than $2.5\times$ over *Limited*$_3$ classifier.

Locality Table Associativity: To find the right structure of the locality table, experiments are performed with varying associativity of the locality table while keeping the number of tracked cache lines constant. Although no significant variation was observed, associativity of 8 provided the best energy-delay product (cf. Section 3.5.2).

3.3.5 Overheads

Storage

Complete Classifier: The locality-aware protocol requires extra bits at the LLC tag arrays to track cache line reuse information. Each LLC directory entry requires 2 bits for the replica reuse counter (assuming an optimal *replication threshold*, *RT* of 3). Tracking one core requires 2 bits for the home reuse counter, and 1 bit to store the replication mode. Hence the *Complete* classifier, requires 192 (= 64 $\times$ 3) bits of storage per LLC directory entry.

All the following calculations are for one core but they are applicable for the entire processor since all the cores are identical. The sizes of the per-core L1 and LLC caches used in the system are shown in Table 4.6.1. The storage overhead of the replica reuse bit is $\frac{2\times256}{64\times8} = 1KB$. For the

complete classifier, it is $\frac{192\times256}{64\times8} = 96KB$, resulting in a total overhead of $97KB$. The *Complete* classifier uses 28.25% more storage than the S-NUCA baseline's cache related per core storage.

***Limited*$_k$ Classifier:** The *Limited*$_3$ classifier tracks the locality information for three cores. Tracking one core requires 2 bits for the home reuse counter, 1 bit to store the replication mode, and 6 bits to store the core ID (for a 64-core processor). Hence, the *Limited*$_3$ classifier requires an additional 27 (= 3 $\times$ 9) bits of storage per LLC directory entry. The storage overhead of the *Limited*$_3$ classifier is $\frac{27\times256}{64\times8} = 13.5KB$. To track the replica reuse, 2 bits are added to the tag of each cache line, resulting in $\frac{2\times256}{64\times8} = 1KB$ overhead for a $256KB$ LLC slice. The overall storage overhead of the *Limited*$_3$ classifier is $14.5KB$. The *Limited*$_3$ classifier uses 4.22% more storage than the S-NUCA baseline's cache related per core storage.

***Limited*$_{k-m}$ Classifier:** The locality table requires 27 bits of storage per tracked LLC cache line and 2 bits per LLC cache line, similar to *Limited*$_3$ classifier. The reduction comes from tracking limited number of cache lines. Each tracked cache line now needs a tag, a valid bit, and LRU bits along with the *Limited*$_3$ classifier bits. The total overhead for each tracked cache line is $40 + 1 + 3 + 27 = 71$ bits. For 512 tracked cache lines, *Limited*$_{3-512}$ overhead is $1KB + \frac{71\times512}{1024\times8} = 5.44KB$ per core. This is an increase of 1.58% in storage compared to the S-NUCA baseline's cache related per core storage. This also translates to a reduction of $\sim$2.5$\times$ and $\sim$17.5$\times$ over *Limited*$_3$ classifier and the *Complete* classifier respectively.

LLC Tag, Directory, & Classifier Accesses

Updating the replica reuse counter in the local LLC slice requires a read-modify-write operation on each replica hit. However, since the replica reuse counter (being 2 bits) is stored in the LLC tag array that needs to be written on each LLC lookup to update the LRU counters, the proposed protocol does not add any additional tag accesses.

At the home location, the lookup/update of the locality information is performed concurrently with the lookup/update of the sharer list for a cache line. However, the lookup/update of the directory is now more expensive since it includes both sharer list and the locality information in case of the *Limited*$_3$ classifier. For *Limited*$_{3-512}$ classifier, each LLC access is now more expensive in energy because of the additional parallel lookup required for the locality table. This additional expense is accounted for in the evaluation. However, the locality table lookup does not incur any additional latency as it is performed in parallel with LLC tag lookup.

Network Traffic

The locality-aware protocol communicates the replica reuse counter to the LLC home along with the acknowledgment for an invalidation or an eviction. This is accomplished *without* creating additional network flits. For a 48-bit physical address and 64-bit flit size, an invalidation message requires 42 bits for the physical cache line address, 12 bits for the sender and receiver core IDs and 2 bits for the replica reuse counter. The remaining 8 bits suffice for storing the message type.

3.4 Evaluation Methodology

The evaluation is conducted for a 64-core shared memory multicore. The default architectural parameters used for evaluation are shown in Table 4.6.1. Single-issue, in-order compute cores are modeled because the power consumption of complex out-of-order cores can be prohibitively high at large core count. Furthermore, the high concurrency in present and future applications makes the case for using many simple cores, as compared to a lower number of complex out-of-order cores. However, results for an out-of-order based multicore are also presented to demonstrate applicability of the proposed architecture in such configuration.

Architectural Parameter	Value
Number of Cores	64 @ 1 GHz
Compute Pipeline per Core	In-Order, Single-Issue
Physical Address Length	48 bits
Memory Subsystem	
L1-I Cache per core	32 KB, 4-way Assoc., 1 cycle
L1-D Cache per core	32 KB, 4-way Assoc., 1 cycle
L2 Cache per core	256 KB, 8-way Assoc. 2 cycle tag, 6 cycle data Inclusive
Cache Line Size	64 bytes
Directory Protocol	Invalidation-based MESI ACKwise$_4$ [47]
Num. of Memory Controllers	8
DRAM Bandwidth	5 GBps per Controller
DRAM Latency	75 ns
Electrical 2-D Mesh with XY Routing	
Hop Latency	2 cycles (1-router, 1-link)
Contention Model	Only link contention (Infinite input buffers)
Flit Width	64 bits

Table 3.4.1: Architectural parameters for evaluation.

3.4.1 Performance Models

All experiments are performed using the core, cache hierarchy, coherence protocol, memory system and on-chip interconnection network models implemented within the Graphite multicore simulator [49]. All mechanisms and protocol overheads discussed in Sections 3.3 are modeled. The electrical mesh interconnection network uses XY routing. Since modern network-on-chip routers are pipelined [86], and 2- or even 1-cycle per hop router latencies [87] have been demonstrated, a 2-cycle per hop delay is modeled; the appropriate pipeline latencies associated with loading and unloading a packet onto the network are also accounted for. In addition to the fixed per-hop latency, network contention delays are also modeled.

3.4.2 Energy Models

For energy evaluations of on-chip electrical network routers and links, the DSENT tool [50] is used. Energy estimates for the L1-I, L1-D, L2 (with integrated directory) caches, and DRAM are obtained using McPAT [51]. McPAT uses CACTI [88] internally to model storage elements. The energy evaluation is performed at the $11nm$ technology node to account for future technology trends. The models are derived for a tri-gate $11nm$ electrical technology node using the virtual-source transport models of [52] and the parasitic capacitance model of [53]. These models are used to obtain electrical technology parameters (Table 3.4.2) used by both McPAT and DSENT. As clock frequencies are relatively slow, high threshold transistors are assumed for lower leakage.

Parameter	**Value**
Process Supply Voltage (V_{DD})	0.6 V
Gate Length	14 nm
Contacted Gate Pitch	44 nm
Gate Cap / Width	2.420 fF/μm
Drain Cap / Width	1.150 fF/μm
Effective On Current / Width (N/P)	739/668 μA/μm
Off Current / Width	1 nA/μm

Table 3.4.2: Projected Transistor Parameters for $11nm$ Tri-Gate

The overall tool flow for energy modeling is as follows. Graphite simulates a benchmark for the chosen system configuration, producing event counters and performance results. The specified cache and network configurations are also fed into McPAT and DSENT to obtain dynamic per-event energy for each component. Event counters and completion time output from Graphite are then combined with per-event energies to obtain the overall dynamic energy usage of the benchmark.

Application	Problem Size
SPLASH-2 [55]	
RADIX	4M Integers, radix 1024
FFT	4M complex data points
LU_CONTIGUOUS	1024 $\times$ 1024 matrix
LU_NON_CONTIGUOUS	1024 $\times$ 1024 matrix
CHOLESKY	tk29.O
BARNES	64K particles
OCEAN_CONTIGUOUS	1026 $\times$ 1026 ocean
OCEAN_NON_CONTIGUOUS	1026 $\times$ 1026 ocean
WATER-NSQUARED	512 molecules
RAYTRACE	car
VOLREND	head
PARSEC [56]	
BLACKSCHOLES	64K options
SWAPTIONS	64 swaptions, 40,000 sims
FLUIDANIMATE	5 frames, 300,000 particles
STREAMCLUSTER	8192 points per block, 1 block
DEDUP	31 MB data
FERRET	256 queries, 34,973 images
BODYTRACK	4 frames, 4000 particles
FACESIM	1 frame, 372,126 tetrahedrons
OLTP Database Management System [89]	
YCSB	1 GB database
TPCC	1 GB database
Others: Parallel MI Bench [57], UHPC Graph benchmark [58], CRONO [90]	
PATRICIA	10000 IP address queries
ALL-PAIRS-SHORTEST-PATH	Graph with 2^{18} nodes, 16 edges
BETW_CENT	Graph with 2^{18} nodes, 16 edges
CONNECTED-COMPONENTS	Graph with 2^{18} nodes

Table 3.4.3: Problem sizes for the parallel benchmarks.

3.4.3 Benchmarks and Evaluation Metrics

Each multithreaded benchmark is run to completion using the input sets from Table 3.4.3. For performance, the completion time is measured, i.e., the time in the *parallel* region of the benchmark. This includes the compute latency, the memory access latency, and the synchronization latency. The memory access latency is further broken down into:

1. **L1 to LLC replica latency** is the time spent by the L1 cache miss request to the LLC replica location and the corresponding reply from the LLC replica including time spent accessing the LLC.

2. **L1 to LLC home latency** is the time spent by the L1 cache miss request to the LLC home location and the corresponding reply from the LLC home including time spent in the network and first access to the LLC.

3. **LLC home waiting time** is the queueing delay at the LLC home incurred because requests to the same cache line must be serialized to ensure memory consistency.

4. **LLC home to sharers latency** is the round-trip time needed to invalidate sharers and receive their acknowledgments. This also includes time spent requesting and receiving synchronous write-backs.

5. **LLC home to off-chip memory latency** is the time spent accessing memory including the time spent communicating with the memory controller and the queueing delay incurred due to finite off-chip bandwidth.

The cache miss types are tracked to evaluate the proposed protocol. They are as follows:

1. **LLC replica hits** are L1 cache misses that hit at the LLC replica location.

2. **LLC home hits** are L1 cache misses that hit at the LLC home location when routed directly to it or LLC replica misses that hit at the LLC home location.

3. **Off-chip misses** are L1 cache misses that are sent to DRAM because the cache line is not present on-chip.

3.4.4 Simulated Schemes

The following LLC management schemes are implemented and evaluated.

1. **S-NUCA** address interleaves all cache lines among all LLC slices.

2. **R-NUCA** places private data at the requester's LLC slice, replicates instructions in one LLC slice per cluster of 4 cores using rotational interleaving, and address interleaves shared data in among all LLC slices.

3. **VR** starts with S-NUCA and uses the requester's local LLC slice as a victim cache for data that is evicted from the L1 cache. The evicted victims are placed in the local LLC slice only if a line is found that is either invalid, has no sharers, or is a replica itself in the L1 cache.

4. **ASR** also replicates cache lines in the requester's local LLC slice on an L1 eviction. However, it only allows LLC replication for cache lines that are classified as shared read-only. ASR pays attention to the LLC pressure by basing its replication decision on per-core hardware monitoring circuits that quantify the replication effectiveness based on the benefit (lower LLC hit latency) and cost (higher LLC miss latency) of replication. The hardware monitoring circuits or the dynamic adaptation of replication levels are not modeled. Instead, ASR is run at five different replication levels (0, 0.25, 0.5, 0.75, 1) and the setting with the lowest energy-delay product is chosen for each benchmark.

5. ***RT3_3_512*** starts with S-NUCA and replicates a cache line in the local LLC slice if the reuse of that cache line is above the replication threshold (RT). Section 3.5.3 evaluates a sweep study for a range of RT values and chooses '3' for the evaluation. Furthermore, the locality tracking is done using the $Limited_{3-512}$ classifier. This setting is justified in the evaluation Section 3.5.2.

6. ***RT3_3_512_R*** is similar to *RT3_3_512*, however it is built on top of the R-NUCA data placement policy.

3.5 Results

3.5.1 Comparison of LLC Replication Schemes

Completion Time & Energy Tradeoffs

Figures 3.5.1 and 3.5.2 plot the energy and completion time breakdown for the LLC replication schemes evaluated. The energy and completion time trends can be understood based on the following factors: (1) the type of data accessed at the LLC (instruction, private data, shared read-only data and shared read-write data), (2) reuse run-length at the LLC, and (3) working set size of the benchmark. Figure 3.5.3, which plots how L1 cache misses are handled at the LLC, is also instrumental in understanding these trends. The *RT3_3_512* bar correspond to the locality-aware scheme with replication threshold of 3 and $Limited_{3-512}$ locality classifier with S-NUCA data placement. *RT3_3_512_R* is similar to *RT3_3_512* with R-NUCA's data placement.

As a general trend VR shows some benefits in completion time over S-NUCA. However, it exhibits higher LLC energy than the other schemes for two reasons. (1) Its process of creating replicas on all evictions results in the pollution of the LLC, leading to less space for useful replicas

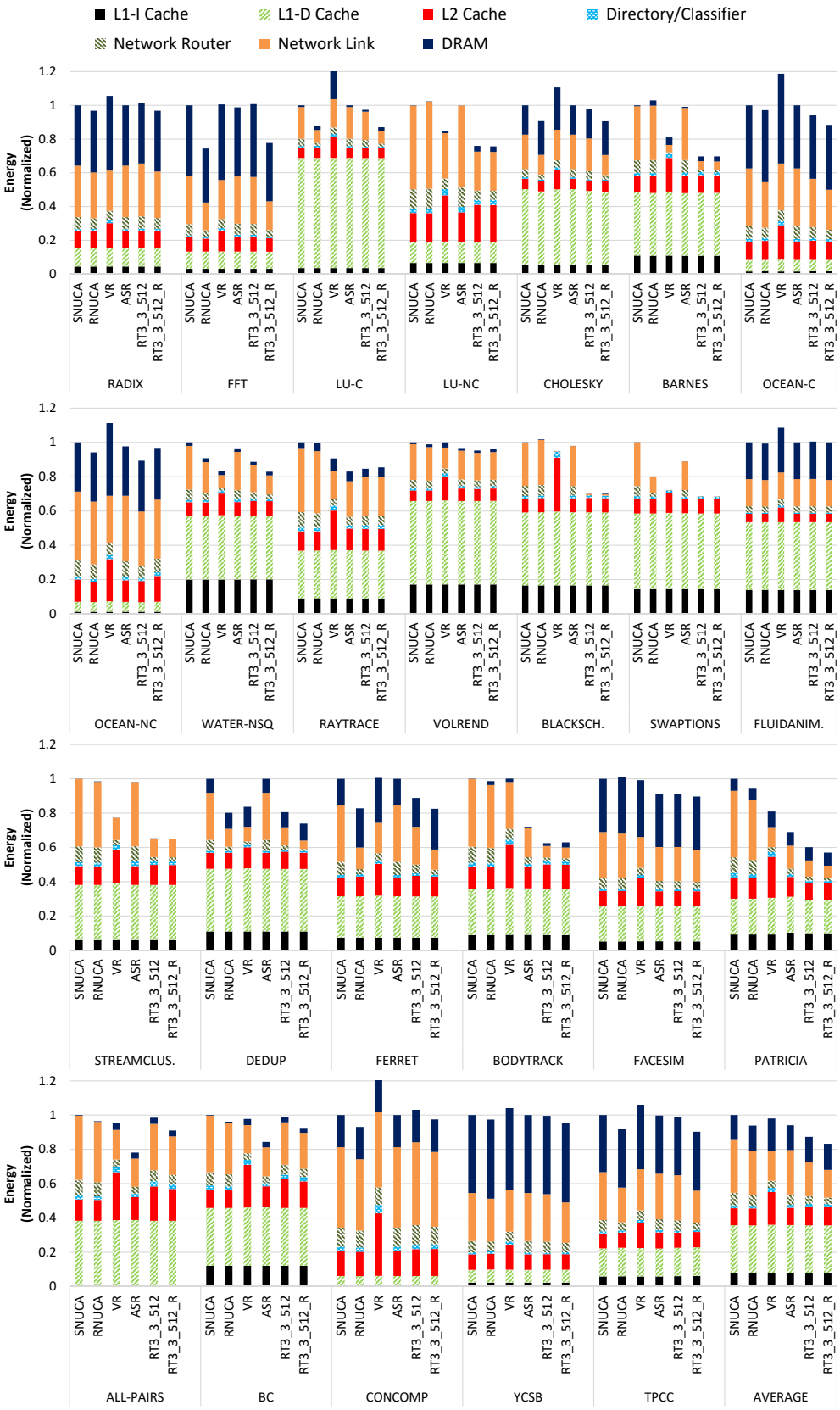


Figure 3.5.1: Energy breakdown for the LLC replication schemes evaluated.

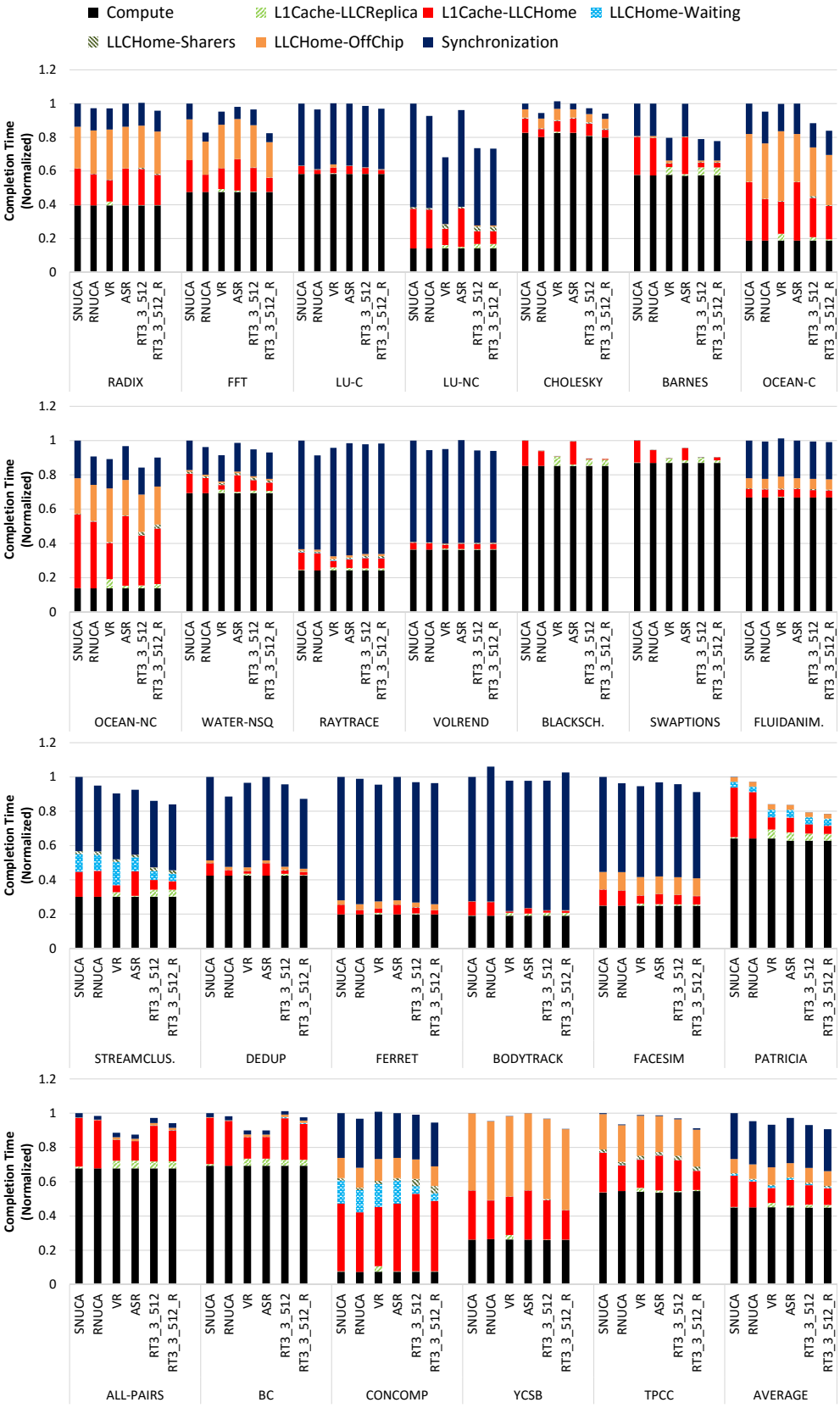


Figure 3.5.2: Completion time breakdown for the LLC replication schemes evaluated.

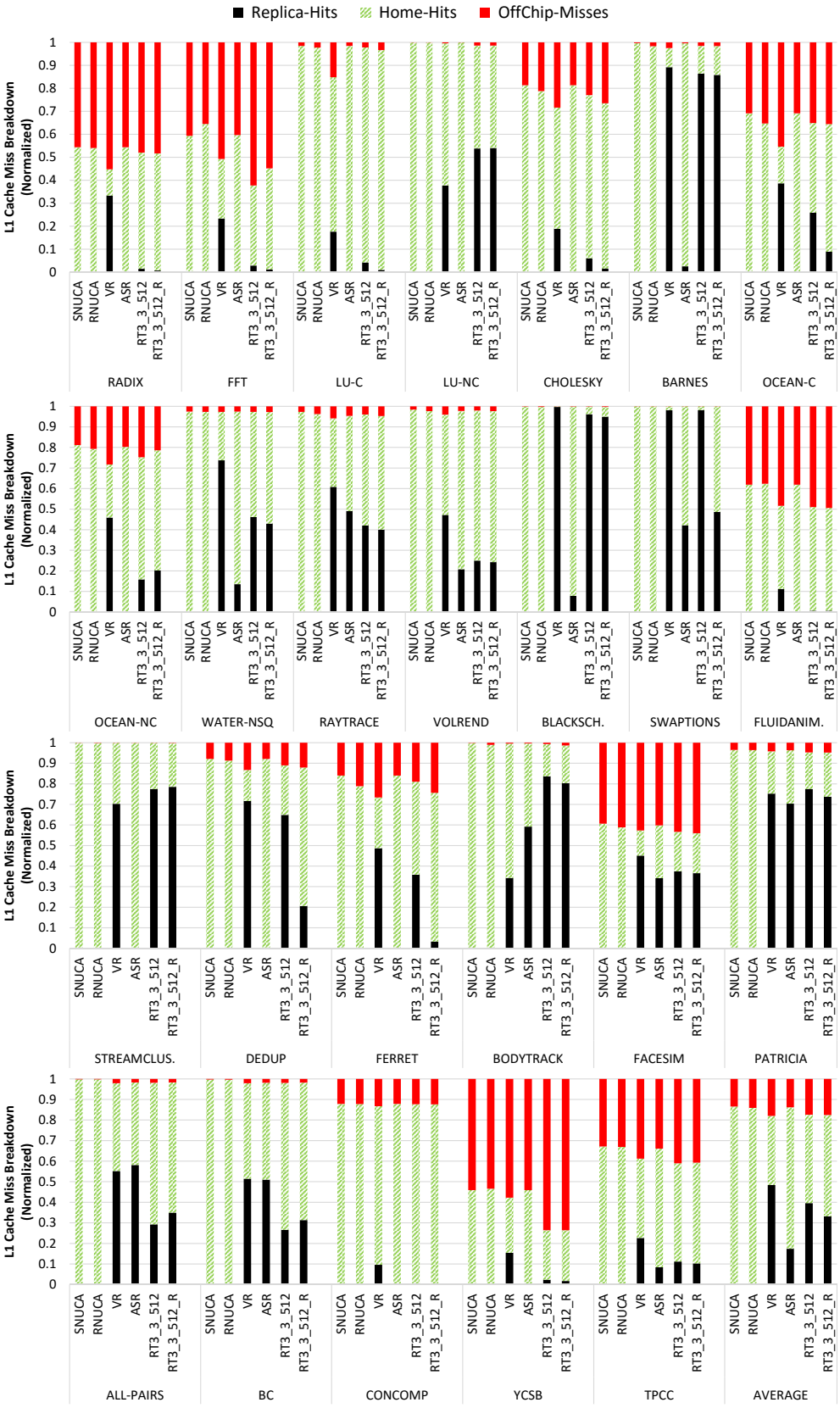


Figure 3.5.3: L1 Cache Miss Type breakdown for the LLC replication schemes evaluated.

and LLC home lines. (2) The exclusive relationship between the L1 cache and the local LLC slice in VR causes a line to be always written back on an eviction even if the line is clean. Hence, in the common case where replication is useful, each hit at the LLC location effectively incurs both a read and a write at the LLC. And a write expends 1.2× more energy than a read. The first factor leads to higher network energy and completion time as well.

Benchmarks with working set that is smaller than the LLC capacity (e.g. BARNES) tend to benefit from replication. It can be observed from figure 3.1.1 that BARNES benchmark shows a high number of accesses to shared read-write data at the LLC. S-NUCA and ASR do not replicate shared read-write data, hence they do not observe any benefits for BARNES. On the other hand, *RT3_3_512* and VR replicate shared read-write data and deliver ∼20% improvement in completion time and dynamic energy over S-NUCA. This improvement in performance is due to the fact that more LLC accesses hit in the local LLC slice and a round-trip on the network is avoided. The LLC energy is higher in VR because of the reasons discussed previously. However, the network energy is substantially lower than S-NUCA and ASR, due to replication of shared read-write data. Similar trends in VR performance and energy exist in the WATER-NSQUARED, PATRICIA, BODYTRACK, FACESIM, RAYTRACE, SWAPTIONS and BLACKSCHOLES benchmarks.

BLACKSCHOLES and DEDUP benchmarks exhibit a large number of LLC accesses to thread-private data with high reuse (cf. figure 3.1.1). ASR only replicates shared read-only cache lines and identifies these lines by using a per cache line sticky *Shared* bit. Hence, ASR follows the same trends as S-NUCA. Although VR places all data based on static address interleaving, its replication strategy proves effective, and significant completion time benefits are observed. *RT3_3_512* also improves on S-NUCA by replicating private cache lines in the local LLC slice of the requesting core. The LLC energy increases for VR, but it is offset by substantial reductions in the network energy consumption. Dynamic energy consumption closely follows the trend seen in completion time. Same arguments hold for OCEAN_NON_CONTIGUOUS and FERRET but the network energy

benefits for VR are not enough to offset the increase in LLC energy.

RADIX, FFT, LU_CONTIGUOUS, and CHOLESKY have significant number of accesses to thread-private data (cf. figure 3.1.1). ASR, being built on top of S-NUCA, shows the same trends as S-NUCA. VR's replication is not as effective in these benchmarks and only slightly improves over S-NUCA. Same argument is valid for *RT3_3_512* replication as well. These benchmarks benefit more from data placement than from data replication. Similar trends are observed in dynamic energy consumption results. FLUIDANIMATE and CONNECTED-COMPONENTS have working sets that do not fit in the LLC. Therefore, no significant improvement is seen with VR or *RT3_3_512*.

LU_NON_CONTIGUOUS shows significant number of accesses to migratory shared data. Such data exhibits exclusive use (both read and write accesses) by a unique core over a period of time before being handed to its next requester. Since ASR does not replicate shared read-write data, it cannot show benefit for benchmarks with migratory shared data. VR replicates such data almost blindly, and show $> 30\%$ improvement in completion time and energy consumption. Similarly, *RT3_3_512* replicates data and shows strong gains.

S-NUCA enables efficient utilization of on-chip cache capacity. VR's replication displaces home cache lines with no sharers. This result in higher off-chip miss rate component (cf. figure 3.5.3), as the evicted cache lines need to be brought back on-chip on subsequent accesses. This higher off-chip miss rate along with higher LLC energy negates the energy benefits gained through reduction in network energy. *RT3_3_512* intelligently replicates and adapts to the cache pressure and results in off-chip miss rate degradation of only 0.4%.

Data Placement Sensitivity

Figures 3.5.1 and 3.5.2 plot the energy and completion time breakdown for the R-NUCA, and the locality-aware scheme on top of R-NUCA, i.e., *RT3_3_512_R*. Here, benchmarks that show signifi-

cant diversion from the S-NUCA trends are discussed. For example, DEDUP almost exclusively accesses private data (cf. figure 3.1.1), and hence performs optimally with R-NUCA. Cache line replication of *RT3_3_512_R* only slightly improves over R-NUCA, as there is not much opportunity to improve. Even then an energy improvement of $> 5\%$ is achieved. The combined effects of R-NUCA data placement and intelligent data replication in *RT3_3_512_R* improve the completion time and energy by 13% and 26% over baseline S-NUCA system.

RADIX, FFT, LU_CONTIGUOUS, CHOLESKY, and OCEAN_CONTIGUOUS have significant number of accesses to thread-private data (cf. figure 3.1.1). Therefore, R-NUCA shows benefit over S-NUCA, as it places private data in the local LLC slice. Similar benefits can be seen for *RT3_3_512_R*. CONNECTED-COMPONENTS has a working set that does not fit in the LLC. Therefore, both R-NUCA and *RT3_3_512_R* fail to improve completion time significantly. However, R-NUCA's local placement of private data helps decrease its energy consumption compared to the configurations based on S-NUCA data placement.

BLACKSCHOLES exhibits a large number of LLC accesses to private data, and a small number of LLC accesses to shared read-only data (cf. figure 3.1.1). Since R-NUCA places private data in its local LLC slice, one expects it to show some performance and energy benefits over S-NUCA. However, this is not the case since the data classification mechanism of R-NUCA falsely classify private pages as shared (see figure 3.2.1). On the other hand, *RT3_3_512_R* improve over R-NUCA by replicating the falsely classified private cache lines in the local LLC slice of the requesting core. *RT3_3_512_R* shows a reduction of 5% in completion time and $> 30\%$ in energy over R-NUCA.

The migratory data in LU_NON_CONTIGUOUS is classified as shared data by R-NUCA, hence, it is placed in an LLC slice based on static address interleaving. This is why it performs on-par with S-NUCA. *RT3_3_512_R* replicates such data and shows significant gains in both completion time (20%) and energy (26%). TPCC also shows improvement under R-NUCA's data placement. The benefits are further enhanced under *RT3_3_512_R*, as it replicates useful cache lines. An

improvement of $\sim$10% is observed in completion time and energy compared to the S-NUCA baseline. This demonstrates the applicability of the proposed locality aware data replication scheme to server workloads. Similar trends are observed in YCSB.

R-NUCA improves completion time by 5% and energy by 6% over S-NUCA on average. In the presence of data replication, the benefits of R-NUCA data placement are diminished on average. However, some workloads demonstrate greater benefits and make a case for better data placement of data along with intelligent data replication. It should be noted that R-NUCA comes with additional overheads (discussed in Sec 3.2.2) and hence there is a tradeoff between design complexity and derived benefits. *RT3_3_512_R* improve by $\sim$2.5% in completion time and $\sim$4% in energy over *RT3_3_512*.

Summary

R-NUCA performs better than S-NUCA and VR in workloads with mostly private data by placing such data in the local LLC slice. On the other hand, VR performs better than S-NUCA and R-NUCA in workloads with mostly shared data by replicating such data in the local LLC slice. However, it lags behind in energy as it spends more operations on each replica hit. Another reason is that it generally increases the off-chip miss rate, which hurts both performance and energy. ASR only replicates shared read-only data and hence cannot take advantage of opportunities available in replicating other types of data. Overall, ASR performs worse than VR in completion time but improves on energy consumption. Finally, *RT3_3_512* intelligently replicates all types of data and instructions, availing all opportunities to exploit data locality. *RT3_3_512_R* shows further gains by taking advantage of the dynamic data placement policy of R-NUCA.

VR needs 1 bit per tag in the LLC to identify a cache line as replica, resulting in a storage overhead of $0.5KB$ per core. On the other hand, the hardware monitoring circuits in ASR add a

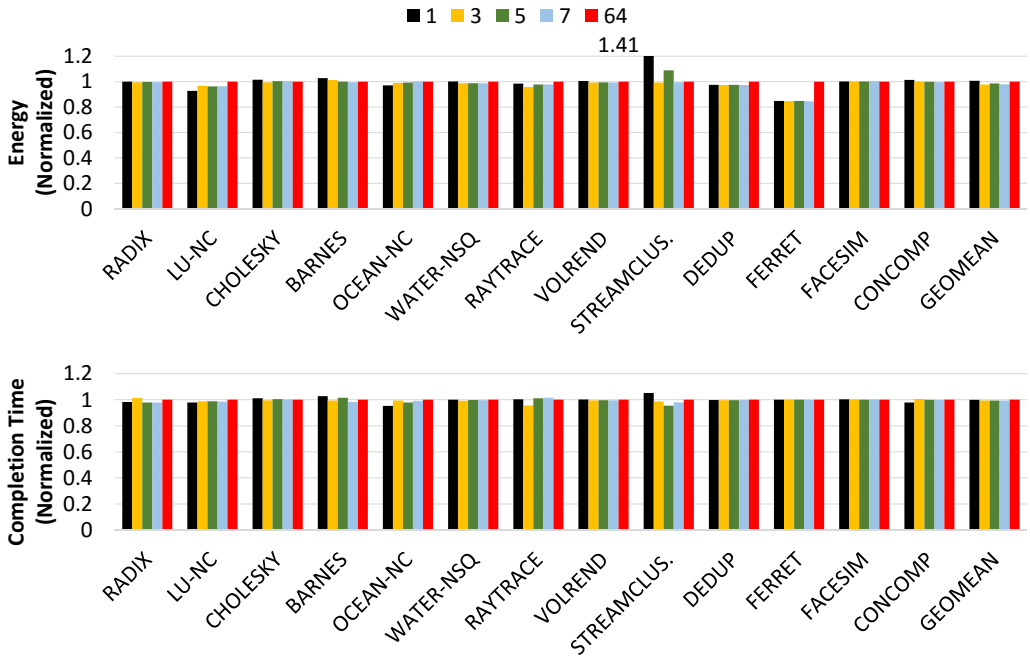


Figure 3.5.4: Energy and completion time results for the Limited$_k$ classifier as a function of number of tracked sharers (k). The results are normalized to that of the Complete (= Limited$_{64}$) classifier.

storage overhead of more than $18KB$ per core. In comparison *RT3_3_512* pays a modest overhead of $5.44KB$ per core to enable the hardware-only predictive LLC replication mechanism. The proposed locality-aware data replication protocol reduces overall energy by 14.7%, 10.7%, 10.5%, and 16.7% and the completion time by 2.5%, 6.5%, 4.5%, and 9.5% when compared to the previously proposed VR, ASR, R-NUCA, and S-NUCA LLC management schemes.

3.5.2 Tuning *Limited*$_{k-m}$ Locality Classifier

Figure 3.5.4 plots the energy and completion time of the benchmarks with the Limited$_k$ classifier when k is varied as $\{1, 3, 5, 7, 64\}$. The configuration with k = 64 corresponds to the Complete classifier. The results are normalized to that of the Complete classifier. The benchmarks that are not shown are identical to DEDUP, i.e., the completion time and energy stay constant as *k* varies. The

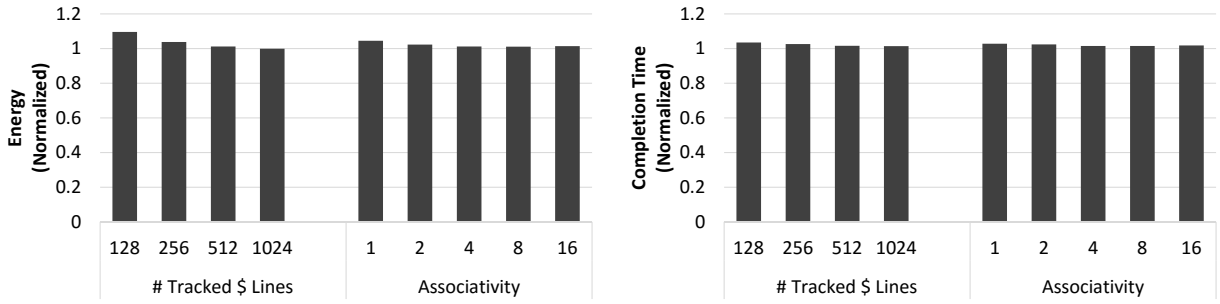


Figure 3.5.5: Energy (left) and completion time (right) results normalized to *RT3_3*. The 'm' parameter is varied from 128 to 1024. The associativity of the locality table is varied from 1-16.

experiments are run with the best *RT* value of 3. The completion time and energy of the Limited$_3$ classifier never exceeds by more than 2% the completion time and energy consumption of the Complete classifier except for STREAMCLUSTER.

With STREAMCLUSTER, the Limited$_3$ classifier starts off new sharers incorrectly in non-replica mode because of the limited number of cores available for taking the majority vote. This results in increased communication between the L1 cache and LLC home location, leading to higher completion time and network energy. The Limited$_5$ classifier, however, performs as well as the complete classifier, but incurs an additional $9KB$ storage overhead per core when compared to the Limited$_3$ classifier. From the previous section, it can be observed that the Limited$_3$ classifier performs better than all the other baselines for STREAMCLUSTER. Hence, to trade-off the storage overhead of the classifier with the energy and completion time improvements, $k = 3$ is chosen as the default for the limited classifier. The Limited$_1$ classifier is more unstable than the other classifiers. While it performs better than the Complete classifier for LU_NON_CONTIGUOUS, it performs worse for the BARNES and STREAMCLUSTER benchmarks. The better energy consumption in LU_NON_CONTIGUOUS is due to the fact that the Limited$_1$ classifier starts off new sharers in replica mode as soon as the first sharer acquires replica status. On the other hand, the Complete classifier has to learn the mode independently for each sharer leading to a longer training period.

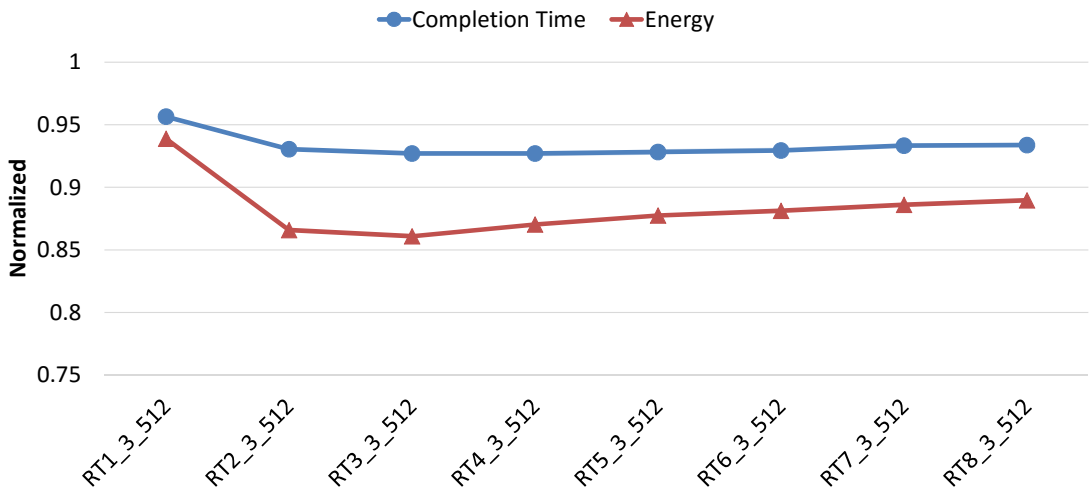


Figure 3.5.6: Geometric mean of completion time and energy results for *Limited*$_{3-512}$ locality classifier at RT values of 1–8, normalized to the S-NUCA baseline.

Figure 3.5.5 plots the energy and completion time of the benchmarks with the *Limited*$_{3-m}$ locality classifier. The number of tracked cache lines parameter, 'm', is varied as {128, 256, 512, 1024}, while the associativity is kept at 8. The associativity of the locality table is varied as {1, 2, 4, 8, 16}, while the number of tracked cache lines is kept at 512. The configuration with 4096 tracked cache lines and associativity of 8 corresponds to the *Limited*$_3$ locality classifier. The results are normalized to that of the *Limited*$_3$ locality classifier (*RT3_3*). Increasing the number of tracked cache lines results in better accuracy albeit higher overhead. Going from 256 tracked cache lines to 512, significant improvement in both completion time and energy is seen, justifying the additional overhead. However, going from 512 to 1024 an improvement of $< 1\%$ in energy-delay product is observed, while paying double the storage overhead for it. Therefore, m = 512 is chosen as the default for the *Limited*$_{3-m}$ locality classifier. Varying the associativity of the locality table had little impact on the energy and completion time as long as the value is 4 or higher. The best energy-delay product is seen at an associativity of 8, therefore, it is chosen as the default for the *Limited*$_{3-512}$ locality classifier.

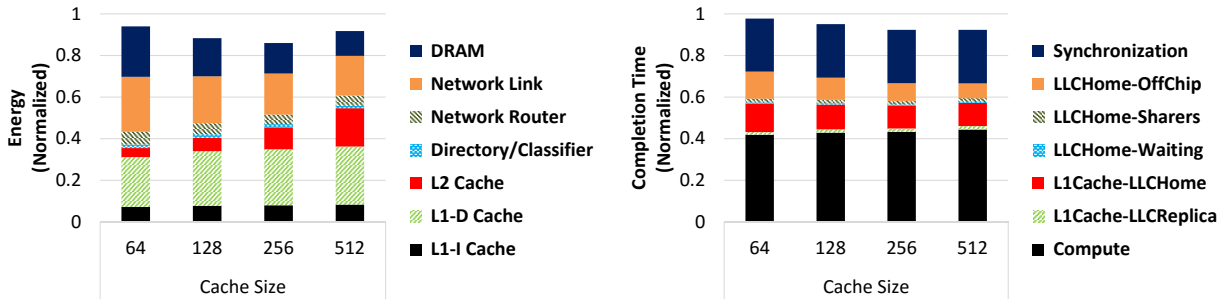


Figure 3.5.7: Completion time and energy results of varying cache sizes normalized to its respective S-NUCA baseline.

3.5.3 Sensitivity – Replication Threshold

Figure 3.5.6 plots the geomean of the energy and completion time of the benchmarks with the *Limited*$_{3-512}$ classifier when the RT value is varied as {1, 2, 3, 4, 5, 6, 7, 8}. The results are normalized to the S-NUCA baseline. RT value of 1 shows the worst performance and energy, as it is too aggressive in replicating data. It basically makes a replica in the local LLC slice on the first access at the LLC home. Increasing RT value results in better performance and energy consumption, as only useful cache lines are being replicated. The performance delta is $< 1\%$ going from RT value of 2 to RT value of 8. The best completion time is observed at at RT value of 3. The spread in energy consumption is $\sim$3% between RT values of 2 and 8. The lowest energy consumption is observed at RT value of 3. The best energy-delay product is seen at the RT value of 3, therefore, it is chosen as the default for the *Limited*$_{3-512}$ locality classifier.

3.5.4 Sensitivity – LLC Cache Size

The advantage of *RT3_3_512* increases as the LLC capacity is increased. This is because more space is available for replicas to be created. This trend holds going from $64KB$ to $128KB$ to $256KB$ per core cache size (cf. figure 3.5.7). However, the increase from $256KB$ to $512KB$ per core cache size does not result in any significant improvement. This is because the active working set of most

workloads fit in $256KB$ LLC slice per core. Furthermore, the useful replicas created also saturate at this particular cache size. The additional capacity available at $512KB$ LLC slice is not utilized effectively and results in no appreciable improvement.

As the cache size increases, each cache event becomes more costly in terms of energy. This is evident from figure 3.5.7 (right), where the "L2 Cache" energy component increases with increase in cache size. However, this increase in L2 energy is offset by reduction in network and off-chip energy, as more and more replicas are created. As discussed previously, at $512KB$ cache slice the capacity is not utilize effectively and it does not result in more useful replicas. This can be verified from the energy numbers as well where the network energy remains the same as $256KB$ LLC slice. At the same time, the increase in size pushes the L2 energy consumption higher. The reduction in off-chip energy is not enough to offset the increase in L2 energy and thus results in higher overall energy.

3.5.5 Sensitivity – Out-of-Order Core Type

Figure 3.5.8 shows the energy and completion time results for a subset of benchmarks in an out-of-order core based multicore processor, using the *RT3_3_512* classifier. The results are normalized to the S-NUCA baseline. The out-of-order processor spends significantly lower time in compute and memory stalls due to its dynamic scheduling. The out-of-order core also executes instructions speculatively, allowing long latency operations (such as private cache misses) to be hidden without stalling the pipeline. This results in a relatively lower benefit in completion time compared to the in-order core type. This trend can be observed in FFT, BARNES, STREAMCLUSTER, and BLACKSCHOLES. However, some benchmarks still find plenty of opportunities to show similar or higher benefits, such as LU_NON_CONTIGUOUS, PATRICIA, FACESIM, and CONNECTED-COMPONENTS. On the other hand, significant gains in energy are observed in most benchmarks, such as LU_NON_CONTIGUOUS, WATER-NSQUARED, RAYTRACE, and DEDUP.

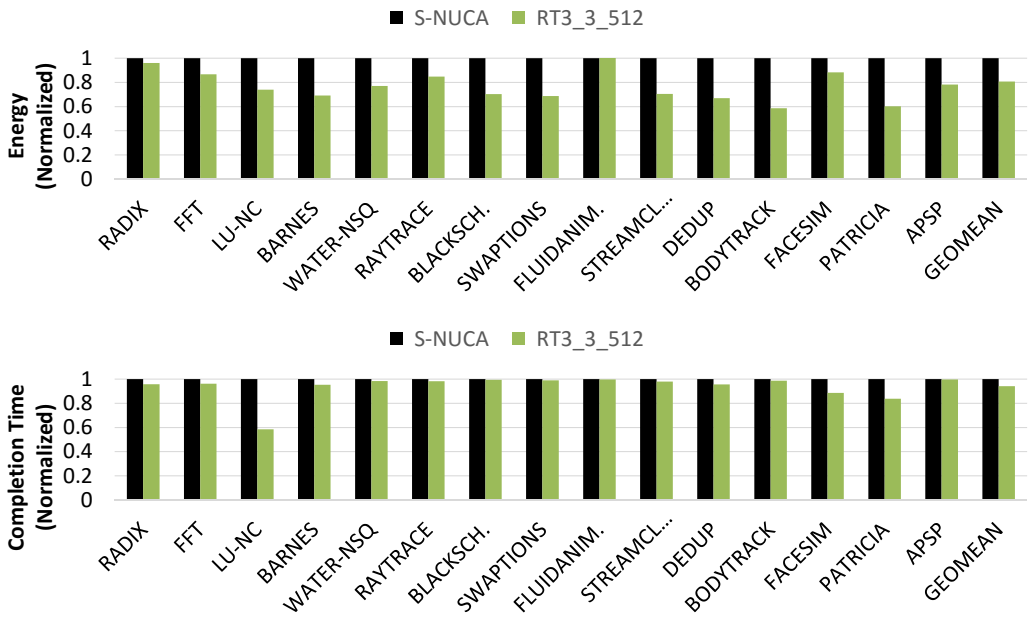


Figure 3.5.8: Energy and completion time for *RT3_3_512* in an out-of-order core setup. Results are normalized to the S-NUCA baseline. Note that *Geometric-Mean* plotted here is across *all* the benchmarks.

3.6 Conclusion

An intelligent locality-aware data replication scheme is proposed for the last-level cache of large-scale multicore processors. The cache line level reuse is profiled at runtime using a low-overhead yet highly accurate in-hardware classifier. On a set of parallel benchmarks, the locality-aware protocol reduces overall energy by 14.7%, 10.7%, 10.5%, and 16.7% and the completion time by 2.5%, 6.5%, 4.5%, and 9.5% when compared to the previously proposed Victim Replication, Adaptive Selective Replication, Reactive-NUCA and Static-NUCA LLC management schemes. The coherence complexity of the proposed protocol is almost identical to that of a traditional non-hierarchical (flat) coherence protocol. This is due to the fact that cache lines replicas are only allowed to be created at the LLC slice of the requesting core. The classifier is implemented with only 1.58% ($5.44KB$) storage overhead on top of the S-NUCA baseline's cache related per-core storage.

Chapter 4

Scaling Shared Memory Many-core Architectures Using Explicit Communication

Shared Memory stands out as a *sine qua non* for parallel programming of many commercial and emerging multicore processors. For efficiency, shared memory is often implemented with hardware support for synchronization and cache coherence among the cores. It optimizes patterns of communication that benefit common programming styles. As parallel programming is now mainstream, those common programming styles are challenged with emerging applications that communicate often and involve large amount of unstructured data. Such applications include graph analytics and machine learning, and this work focuses on these domains. Message passing has been mostly used as a parallel programming model for supercomputers and large clusters. However, this paradigm offers opportunities to explicitly and efficiently manage communication patterns. I propose to retain the shared memory model, however, introduce a set of lightweight in-hardware explicit message passing style instructions in the instruction set architecture (ISA). The explicit messaging

protocol is used to move computation to where data is located, hence eliminating unnecessary movement of data. This in turn is used as an "accelerator for communication" that efficiently supports synchronization primitives and minimize the overheads of multi-party communication in shared memory protocols [91, 92].

4.1 Introduction

Power dissipation and hardware complexity has halted the drive to higher core frequencies and ever-increasing size of brawny cores. However, transistor density has continued to grow, and CPUs with many low-power cores have become the means to obtaining higher performance. Leveraging parallelism using multicores in datacenters [89] and other exascale environments is already underway, and future massive computational models are expected to have multicores with thousands of cores on-chip [1]. This rise in computational and memory concurrency has greatly facilitated usage of data for real world problems, such as predicting weather patterns and synapse deductions from brain graphs. Such data comes from abstract level paradigms, such as road networks [93], social networks [94], computational biology [95], and neuroscience [96]. Together they are processed using emerging data and graph analytic algorithms.

Shared-memory paradigm provides a uniform view of the system to the programmer and pushes the complexity of handling data movement and communication to the hardware. The main focus is on providing ease of programming and reducing programmer overhead. Shared-memory is supported in hardware through cache coherence and consistency protocols. This hardware support is effective in exploiting spatio-temporal locality when accessing primarily private and shared read-only data. On the flip side, accesses to shared data with frequent writes results in wasteful invalidations, synchronous write-backs, and cache line ping-pong leading to low spatio-temporal

locality. Traditionally, accesses to shared data is coordinated using primitives involving communication via coherent caches and coordination via special instructions, such as traditional RISC-based load-with-reservation and conditional store [97]. The downside of this style of communication between two cores is that it can potentially take hundreds of instructions and memory operations to complete. This limits the performance scaling of multi-threaded workloads with contended shared data being managed by shared-memory synchronization primitives. Furthermore, in traditional shared memory systems the data is moved around the chip to perform different operations on it. This can lead to unnecessary and in some cases excessive movement of data leading to higher energy consumption and access latency.

Harnessing parallelism in data and graph analytic algorithms remains a challenging task [98]. Current state-of-the-art algorithms do not scale well on parallel machines, even though parallelism and performance is very much needed in these domains. Bottlenecks such as memory coherence, inter-core communication, and fine-grain data dependencies all limit scalability. Of all these bottlenecks, synchronization is the primary culprit that steals performance gains at high thread counts [99]. Synchronization is a consequence of deploying concurrent threads or processes to perform computations on shared data. This leads to mutual exclusion requirements that handle race conditions occurring on shared data structures, which leads to parallel programming primitives such as locks and barriers [100]. In most parallel algorithms related to data/graph analytics, synchronization is unavoidable since there is always shared data between threads/processes that needs to fulfill memory consistency requirements [101]. The main challenge therefore boils down to dealing efficiently with shared data structures and communication.

To overcome the shortcomings of inefficient communication in shared memory, explicit message passing protocols (e.g., MPI [102]) have been frequently used in high-performance computing. Software based explicit messaging approaches have limited applicability because of high overheads associated with setup and orchestration of send and receive messages among cores. Hardware

based explicit messages have been explored in the past by MIT Alewife machine [103] and Active Messages [104] in the context of multiprocessors. Similar explicit communication paradigms can be leveraged in single-chip multicores to mitigate inefficient communication bottlenecks [92]. This is achieved by eliminating instruction retries and associated memory accesses for lock acquisitions, and excessive cache line movement of shared data.

In this work, a low-overhead, practical, and deadlock-free hybrid architecture is presented that overlays an explicit communication layer on top of a shared memory architecture. The goal is to retain shared memory for backward compatibility and introduce a set of lightweight in-hardware explicit messaging instructions to the instruction set architecture (ISA) to mitigate the inefficient communication bottleneck in shared memory architectures. The explicit messaging layer enables programmable communication models, overlapping of communication with computation, efficient management of data locality, and moving computation to data to reduce excessive data movement. This significantly mitigates the bottlenecks caused by shared memory synchronization, resulting in improved scalability in emerging graph and data analytics workloads. The need to keep shared memory paradigm for reasons other than backward compatibility is also demonstrated.

The contributions made by this work includes:

1. Emerging graph and machine learning workloads are characterized and implemented with explicit messaging to demonstrate the increased scalability achievable using the flexible communication models.
2. The importance of shared memory, beyond backward compatibility, is demonstrated by showing that complex access patterns are either difficult or inefficient to manage using only explicit messaging.
3. The effectiveness of the proposed hybrid architecture is demonstrated by showing scaling trends far superior to traditional shared memory systems. Workloads with contended locks show up to an order of magnitude better scaling compared to shared memory.

4.2 Background

4.2.1 Baseline

The baseline system is a tiled multicore with an electrical 2-D mesh interconnection network. Each core consists of a RISC compute pipeline, private L1 instruction and data caches, a physically distributed shared LLC cache with integrated directory, and a network router. Cache coherence is maintained using a MESI protocol. The coherence directory is integrated with the LLC slices by extending the tag arrays (in-cache directory organization [105, 60]) and tracks the sharing status of the cache lines in the per-core private L1 caches. The private L1 caches are kept coherent using the ACKwise limited directory-based coherence protocol [47]. Some cores have a connection to a memory controller as well. The memory controllers are placed in a way that minimizes the average distance from the tiles. The electrical mesh network-on-chip (NOC) uses dimension-order X-Y routing and wormhole flow control.

4.2.2 Related Work

The idea of hardware-level explicit messages for efficient communication has been explored in the context of multiprocessors [103, 104]. Maintaining cache coherence between chips is generally more complex and expensive. The proposed architecture has similarities in the sense that it attempts to mitigate the inefficient communication bottleneck. However, it differs from these works by exploring the idea of explicit communication in the context of single-chip multicores where the tradeoffs are different. Moreover, the challenge applications are the emerging graph analytic and machine learning. Hence the potential impact of the the proposed architecture is immediate.

The architecture closest to the proposed architecture is that developed and commercialized by Tilera [106]. They have developed a many-core cache coherent processor with message passing

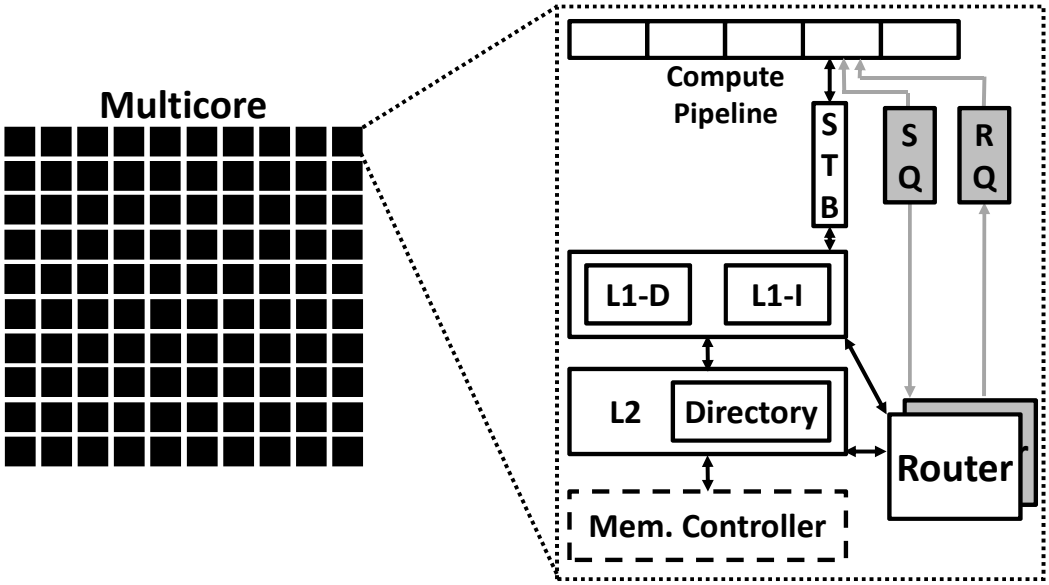


Figure 4.2.1: Tile view of the proposed many-core processor. The shaded blocks are the new components added to the baseline architecture to enable explicit messaging.

on a separate network. This work is different than Tilera chips in some implementation details. For example, using a single per thread capacity counter for flow control, consistency, and context switching is unique to the proposed architecture. Also, the messaging network in Tilera chips (UDN) can be used by only 1 thread on any core. So, if there are multiple threads running on a core, only 1 can perform explicit messaging, limiting functionality.

Tesseract [107] recently showed the benefits of message passing style architecture in processing-in-memory context. They utilize 3D stacked memory and a large number of simple cores located near the memory. Their proposed architecture lacks shared memory support and hence is not backward compatible. In contrast, the proposed architecture leverages similar message passing style communication while retaining shared memory. It is demonstrated that shared memory is needed for efficiency in many of the challenge workloads.

ADM [108] explores point-to-point asynchronous messaging for task scheduling. ADM implements the messaging on top of shared memory, however, it does not utilize it for general purpose

computation. Active Messages (AM) [109] recently showed the applicability of hardware message passing on top of a shared memory architecture. They demonstrate benefits achievable by such a communication model. However, they stop short of fully exploring the practical implementation aspects of the architecture for real workloads. Furthermore, they require a separate context for processing explicit messaging. A lot of similarities can be found between AM and this work. However, this work contributes towards presenting a low overhead and practical architecture. The different deadlock freedom aspects are explored and the overheads quantified. The proposed architecture is also evaluated for emerging graph analytic and machine learning parallel workloads.

4.3 Architecture & Protocol

4.3.1 ISA Extensions

The proposed architecture utilizes a simple RISC style ISA with extension for explicit messaging instructions. There are four basic instruction types being added to support explicit communication. Lets look at what these four instructions are, how they are structured, and implemented.

***send*:** The *send* is a non-blocking explicit communication instruction. The *send* instruction requires a destination address along with the data to be sent. The data for the *send* instruction is setup in the register file explicitly using load/store instructions. A message is composed by reading the register file, and inserting it into the send queue for transmission on the network. The destination tile id is obtained by looking up a table that stores valid address-to-tile mapping. Its a fire-and-forget message from the compute pipeline's point-of-view, *i.e.,* it puts the message into the send queue and retires the instruction in the sender core. It should be noted that a non-blocking *send* can block due to actions of the underlying NOC or messaging flow-control protocol, or the send queue being full.

***recv*:** The *recv* is a blocking instruction. When a new message is received, it is buffered in a queue until handled via a receive instruction. A core's pipeline is stalled if it gets to a *recv* instruction but hasn't received the message yet. The receive instruction loads the message contents into already setup registers. As soon as the message is consumed at the receiver side (*i.e.,* removed from the receive queue), an ACK message is generated and sent back to the original sender.

***sendr*:** The *sendr* is a blocking explicit communication instruction. A return value is required from the receiving core and can be thought of as a form of remote procedure call. *sendr* instruction works similar to a *send* instruction with the exception that it blocks the compute pipeline until an explicit reply is received from the destination thread.

***resumer*:** The *resumer* is a non-blocking explicit communication instruction. It works similar to a *send* instruction. It is used to respond to a *sendr* instruction from a sender. The destination address is set by the preceding "*recv*" instruction. The data carried over can be some computed data or just an acknowledgment, signaling the sending core that it can resume operation.

4.3.2 Protocol Operation

Figure 4.2.1 shows a logical view of a tile within the proposed many-core processor. The shaded modules are introduced to support the explicit messaging protocol on top of shared memory. The compute pipeline is also extended with logic to support the *send* and *receive* instructions. Send and receive queues (SQ and RQ) are introduced to buffer send and receive messages. The NOC is also extended to ensure network level deadlock freedom.

To enable point-to-point communication between cores, the sender needs to know the destination

$CoreID$ (or the target receive queue). This information is setup in software using a data structure that maintains a set of addresses assigned to the receive queues. Consequently, the destination $CoreID$ is determined by performing a lookup in an operating system maintained thread-to-core mapping table. Note that each *send* or *sendr* is preceded by these overhead instructions to setup a message.

Lets make some assumptions about the architecture and then remove them systematically, as the protocol is described. The first assumption is that there is enough capacity in the destination's receive queue. The second assumption is that the receiver does not impose any ordering requirement on arriving packets from different cores; *i.e.,* it does not care about the source of the packet, and has the necessary capability to consume it regardless. The discussion is broken up into two parts; 1) send-recv, and 2) sendr-recv-resumer.

Send-Recv Protocol

Consider two communicating threads, one sending an explicit message to the other. Assume the source thread generates a *send* instruction. A *send* instruction is paired with a *recv* instruction at the destination thread to consume the sent message. Each step in the life of this explicit message is now discussed.

When a *send* instruction is executed, a message is constructed by reading source registers setup by preceding software. The source registers contain the data to be sent to the destination thread. Moreover, the destination $CoreID$ is determined by performing a lookup to the thread-to-core mapping table discussed earlier. The header of the explicit message contains the destination $CoreID$. If the send queue is full, the pipeline stalls. Otherwise, the message is inserted into the send queue and the compute pipeline commits the instruction. Since *send* is non-blocking, the pipeline can go ahead and execute other instructions. Note that this protocol assumes a weak RISC

consistency model. Memory consistency implications of *send* are further discussed in Section 4.5.

The decision to inject the message into the NOC is based on the value of a *"capacity counter"*. If the value is > 0, the message is injected into the network and the *capacity counter* is decremented. On the other hand, if the value is '0', the message is blocked in the send queue. The need for *capacity counter* and how it works is explained in Sec 4.3.3.

The NOC routes the message and delivers it to the destination tile's router. If the destination receive queue has available space, the message is inserted into the queue. Otherwise, the message stalls in the NOC. Note that the NOC implements flow-control that can potentially stall a source core from injecting more messages. Since a received message is always consumed at the receiving core, the machine is always guaranteed to make forward progress.

Upon execution of the *recv* instruction at the destination core, the message is de-allocated from the receive queue and consumed by the pipeline by copying the data into appropriate registers. The *recv* is a blocking instruction, hence, it stalls the pipeline until it can pull the data from the receive queue. On the other hand, if the pipeline hasn't gotten to the *recv* instruction yet, the message sits in the receive queue waiting to be consumed. As soon as the message is de-allocated form the receive queue, an *ACK* is generated and injected into the network. This *ACK* message is routed back to the source core, where it increments the *capacity counter*. The *ACK* message is not necessary to ensure protocol level deadlock freedom. However, it enables an efficient mechanism to track the completion of *send* instructions, as well as manage flow-control for messages between communicating threads.

SendR-Recv-ResumeR Protocol

Now consider the two threads are communicating through a *sendr* instruction. A *sendr* instruction blocks the pipeline and expects an explicit reply from the destination thread. It is paired with a

recv instruction at the destination, which consequently generates a *resumer* instruction to send an explicit reply back to the *sendr* thread. Each step in the life of this message that is different from Sec 4.3.2 is now discussed.

At the sender side, the difference is that the pipeline stalls until it receives an explicit reply from the destination thread. The decision to inject the message into the NOC is exactly the same as described in the previous section. One may argue that there is no need to check the *capacity counter* for *sendr* since there can only be one such outstanding message. However, a *sendr* can be preceded by *send* instructions that can benefit from managed flow-control. At the receiving side, a *recv* instruction consumes the message. Moreover, a *resumer* instruction is executed once the receiving thread is ready to send an explicit reply. The *resumer* is a non-blocking instruction and behaves like a *send* instruction. Therefore, the pipeline moves on after inserting the formed message into the send queue. It is used to convey some results back to the *sendr* thread, or just signal it to resume operation. It follows similar steps to that of a *send* instruction. However, it is implicitly consumed by the the original *sendr* thread's logic, and does not require an explicit *recv*. At this point the *sendr* instruction completes and the pipeline is allowed to proceed.

4.3.3 Deadlock Freedom

The explicit messaging protocol can deadlock for the following reasons: (1) improper use of explicit message instructions at the application level, (2) lack of resources to handle messages at the receiver side, and (3) lack of resources in the NOC. In order to ensure functional correctness, certain limitations are introduced to remove these potential deadlock situations.

Application Level Deadlock Freedom

A shared memory system can easily be deadlocked if the synchronization primitives are not properly used. Similarly, an easy way to deadlock the proposed architecture is by improper use of explicit message instructions. Lets consider a scenario where *thread a* is expecting to receive a message from *thread b*. *Thread a's* pipeline is stalled on the *recv* instruction, while *thread b* does not send any message at all. This is poor programming and will lead to an application level deadlock. Another example is when two threads want to communicate with each other through blocking sends and both execute *sendr*, *recv*, and then a *resumer* instruction, in that order. In this case, none of the threads can make progress as the pipeline is stalled on both cores and they cannot get to the *recv* instruction.

A deadlock situation can occur if order of receiving messages is required for functionality. For example, consider a *thread c* expects to receive message from *thread a* before the message from *thread b*. However, this can not be ensured due to several reasons, including *thread b* getting to its *send* instruction first and the congestion in the interconnect for different routes. If *thread b's* message gets to the destination first, it will result in a deadlock. At best, the destination thread can go ahead and receive the message, however, it will result in incorrect functionality. This deadlock scenario can be resolved by making sure that the architecture implements multiple independent receive queues to buffer the messages. This will ensure that the message from *thread b* is pulled in from the network and inserted into the appropriate receive queue, making it possible to receive the message from *thread a*. To generalize this scenario, *the number of independent receive queues must be equal to the number of concurrent communicating threads if the order of receiving messages is necessary for program functionality*.

If the order of arrival of the explicit messages is not important for program functionality, then this deadlock situation will not occur. In this case the receiving thread always makes progress and receive messages from all communicating threads. Hence a single receive queue per core will

suffice. This offers a tradeoff between programmability and flexibility since the programmer doesn't have to worry about managing multiple receive queues[1]. All the workloads evaluated in this work do not impose any ordering requirement for message arrival. Therefore, the proposed architecture implements a single receive queue per core.

Protocol Level Deadlock Freedom

Lets assume that a thread imposes order of arrival restriction and has enough receive queues to communicate with two other threads. However, both the receive queues have space for holding only one message at a time (the scenario can be generalized to more than one message trivially). Now lets assume that *thread a* sends one message to *thread c* while *thread b* sends two back-to-back messages to *thread c*. It can so happen that both the messages from *thread b* reach *thread c* before that of *thread a*. In this case, the first message from *thread b* is pulled in and inserted into the receive queue. However, the second message cannot be pulled in as the receive queue is full and the compute pipeline is unable to consume the first message. This second message will end up blocking the router at *thread c* and result in a protocol level deadlock.

The deadlock scenario discussed above can be resolved by always replying to the sender either explicitly (through *resumer*) or implicitly (through *ACK* message). This reply message in turn increments a *"capacity counter"* at the sender thread which is used to avoid overflowing the buffering available at the receiver. There is one *capacity counter* per hardware thread. The value of this counter is set though a special instruction in the sender's thread at the beginning of program execution. This counter keeps track of the in-flight messages from the sender's point of view. Each send decrements the counter by 1 and each ACK increments it by 1. A message can only be injected into the network from the send queue if the value is greater than 0. If the value is 0, the send cannot

[1] It should be noted that an application cannot impose ordering requirement on one receive queue while not on another receive queue. Mixing of ordered and unordered requirements can result in a deadlock situation.

proceed and is held in the send queue until an ACK increments the value to 1. For example, if a sender's *capacity counter* is set to 4 and it sends out 4 explicit messages, 1 each to a unique receiver. At this point, the capacity counter for that sender is '0'. So, it cannot inject any more messages into the network until it gets an ACK back from 1 of the 4 receivers.

Note that unordered receive queue based architecture does not require explicit *capacity counter* based flow control for deadlock freedom. However, it is implemented in the protocol for the management of NOC traffic rate, as well as ensuring strict ordering when the sending core needs to ensure all *send* messages have been observed at their destinations (see Section 4.5).

NOC Level Deadlock Freedom

Lets state certain assumptions about the interconnect that are required for the shared memory baseline. Firstly, the NOC guarantee point-to-point ordered delivery of messages without deadlocks. Secondly, the routing algorithm is deadlock-free or can always eventually recover from a deadlock.

A deadlock scenario can arise if the same physical network/virtual channel (referred to as "channel" from here onward) is utilized for both request messages (*i.e., send*, *sendr*) and reply messages (*i.e., ACK*, *resumer*). In this scenario, *thread a* and *thread b* are sending an explicit message to each other. The *ACK* reply is also supposed to flow on the same channel. This will result in a deadlock as both cores have a circular dependency on each other. The network deadlock can be removed by providing an additional NOC channel for the request messages. However, the reply messages are always ensured to complete as long as they do not share the channel for request messages. Therefore, it is proposed to reuse the existing cache coherence reply NOC channel for the reply messages. This is a valid assumption since it has been shown that cache coherence reply NOC channel can carry replies both requesting core's requests as well as directory forwarded requests [110].

4.3.4 Overheads

An additional NOC physical or virtual channel is required for explicit communication requests (*send* and *sendr*). Moreover, at each tile a 4-entry (1 entry = 4 words) send queue is implemented, which equates to an overhead of $4 \times 4 \times 8B = 128B$ per tile. Each receive queue is sized at 128 entries, resulting in an overhead of $128 \times 4 \times 8B = 4KB$ per tile. In addition to the memory structures, gates are also added for extending the core pipeline and implementing the new logic blocks. However, the number logic gates required are negligible and hence not counted as a significant overhead.

4.4 Execution Model

The proposed architecture enables flexibility by exposing the underlying communication hardware to the programmer. One may choose to restrict programming using traditional shared memory communication primitives. However, to benefit from the proposed concurrency controls, an application needs to be designed in a way that it exploits the the explicit messaging capability. An obvious beneficiary of explicit communication is highly contended shared data that causes serious concurrency bottlenecks due to synchronization in shared memory. These inefficiencies are more pronounced as single chip multicores scale to hight core count.

4.4.1 Communication Models

This section demonstrates that shared memory synchronization primitives (e.g., lock or barrier) can be implemented in an efficient way using explicit messages. The key idea is to eschew the need for cache line ping pong for contended synchronization, and utilize the ultra-efficient messaging instructions to perform synchronization functionality. Moreover, explicit messaging enables a paradigm for atomic operation of critical code section when the data/code for it can be pinned on

a single core. The idea is to utilize explicit messages to communicate the intent and the inputs from multiple threads to atomically operate on the critical section. When possible, this paradigm eliminates the need for synchronization, as well as inefficient cache line ping pong for critical section data.

Lets define a couple of terms before getting into the different models.

1. *Worker thread*: A thread that performs main application work.

2. *Services thread*: A thread that provides different services (support functions) to the *worker threads* to help complete the application work.

The next few sections will explain how are the different type of threads utilized to perform application work and how they accelerate synchronization primitives.

Shared Memory Synchronization Primitives

The shared memory synchronization primitives are implemented and optimized using RISC load-link (ll) and store-conditional (sc) instructions. This enables ease in porting applications to the proposed architecture as all shared memory primitives are directly supported. These synchronization primitives are implemented with hardware support, as shown in fig. 4.4.1, and without any involvement from the operating system (OS). This design decision was made because OS is not simulated and can have dynamic effects that can not be captured in the simulator. Moreover, in an accelerator-type setting, the cost of involving the OS in synchronization can be extremely costly.

Lets look at how a lock is implemented using ll and sc instructions. To acquire a lock, the core issues an ll instruction. This instruction translates to an exclusive read in the memory system; i.e. no other core can have the cache line in its private cache. After the load, the value of mutex variable is checked and if it is '1', signifying that the lock is already acquired by some other thread, the

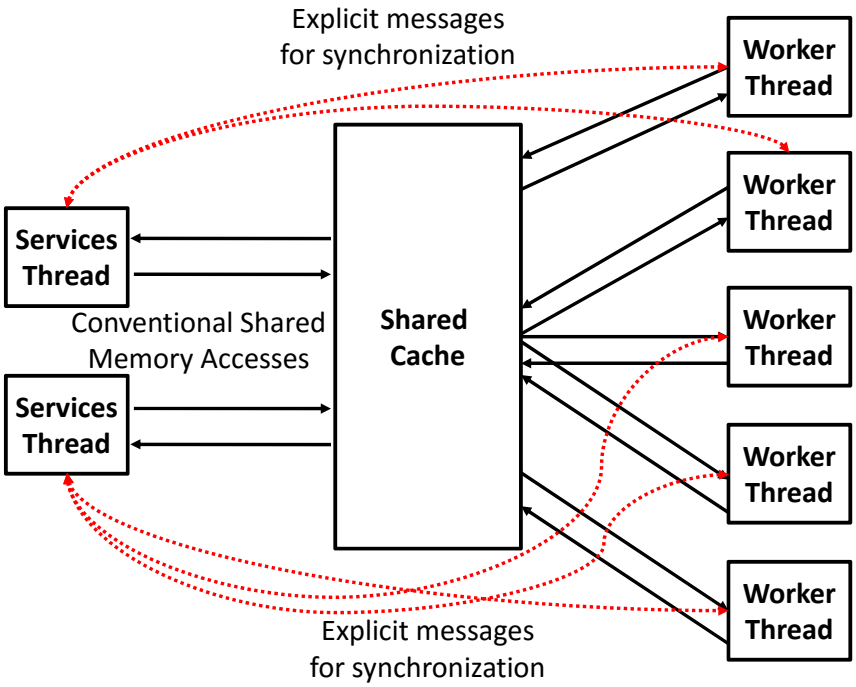


Figure 4.4.1: Threads are divided into two types; 1) *worker threads*, and 2) *services threads*. *Worker threads* perform the main application work. *Services threads* provide different services (support functions) to the *worker threads* to help complete the application work. *worker threads* and *services threads* can be used to implement flexible yet powerful models to accelerate synchronization primitives.

code loops back to the ll instruction to retry lock acquisition. This will continue till the value of the mutex variable is set to '0' by the other thread. Once the value is '1', the core captures the address of the lock variable in a "link register" and can now proceed to executing the sc instruction. The sc instruction writes a '1' to the mutex variable if the cores still holds the reservation for that address. If the reservation is lost, the code will loop back and try the whole sequence again (starting with ll instruction). A contended lock forces retries to acquire the lock, as threads can steal the reservation of a lock from others. This increases the instructions executed and memory accesses performed by the threads.

Accelerating Synchronization Primitives

Explicit point-to-point communication is proposed to accelerate shared memory synchronization. This is achieved by dedicating one or more threads to manage locks, called *services threads*. Figure 4.4.1 shows the logical view of different threads being used as either *worker threads* or *services threads*. Explicit messaging instructions are used to communicate between application threads and the services threads. For example, a *"lock_acquire"* request generates a blocking send message through a *sendr* instruction. The data carried by this message contains the address of the lock. If the requested lock is free, the corresponding *services thread* marks it as locked and send an explicit reply to the requester using the *resumer* instruction. The *resumer* instruction unblocks the requesting core, which enters the critical section. On the other hand, if a lock is already acquired by another thread, the *services thread* adds the requesting core into a waiter list. The services thread only replies back once the requested lock is free. This behavior ensures atomicity, as there is only one thread managing a given lock variable. The race to acquire a lock is now only in the interconnect and all requests get serialized in the receive queue.

Once a core is done executing critical section, it releases the lock by executing *"lock_release"*. A non-blocking send message is generated through a *send* instruction. Upon receiving an unlock request, the *services thread* marks the lock as free. It also checks for any pending lock requests and replies to the waiter core, if there is one. A *"barrier_wait"* is implemented along the same lines using blocking *sendr* instruction. In this case, the *services thread* waits for a message from all threads participating in the barrier before replying and unblocking the sender threads.

The idea of *services threads* is generalized to one or more threads. This helps improve the performance of lock management, as the communication is spread across cores and exploits concurrency. Furthermore, the queuing delay caused by serialization is also spread across cores, improving performance. Similarly, multiple *services threads* are utilized to implement hierarchical

barrier. This eliminates a single serialization point for the barrier, and distributes network congestion as well. Furthermore, the communication with barrier managers occurs concurrently, which results in additional benefits.

Moving Computation To Data

This section explores a model of execution where the proposed architecture enables doing away with synchronization primitive all together, and the critical section is pinned and executed at a dedicated core. Instead of sending a request to acquire a lock, each *worker thread* sends a message to the *services thread* executing the corresponding critical code section. The send message can either be blocking or non-blocking depending on the workload. The *services thread* receives the message and performs the critical section operations atomically and without any interruption. If the *worker thread* expects a reply, the *services thread* sends one and starts waiting for the next message. This way the data itself is pinned to a single location and computation is sent over to that location to be performed on the data.

This approach provides benefits on multiple fronts. First, the core doesn't have to incur stalls due to lock acquisition delays. This is especially beneficial in the case where non-blocking messages from the *worker threads* can be used, as the *worker threads* can send the message and immediately move on. Second, the shared data is pinned at a single core, hence taking advantage of data locality and reduced coherence traffic. Multiple *services threads* provide benefits by distributing the work and overlapping communication with computation. This approach also provides another flexibility which can prove to be highly valuable; *i.e.,* atomic execution of more than one operation.

4.4.2 Graph & Machine Learning Workloads

The applicability of the proposed architecture is illuminated using the emerging graph analytic and machine learning parallel workloads. Graph analytics is well known for its unstructured data and intractable communication patterns. Moreover, machine learning workloads are also known for their massive amount of data communication requirements. A representative graph benchmark, Single Source Shortest Path (SSSP), is taken from the *CRONO* benchmark suite [90], and a popular neural network benchmark, Convolutional Neural Network (CNN), is developed for evaluation purposes.

To better understand communication patterns and synchronization primitive usages, these benchmarks are classified based on their implementation requirements, such as need for locks and barriers. SSSP require the use of both fine-grain locks and barriers. On the other hand, CNN do not require fine-grain locking. However, it requires barrier synchronization to separate layers of learning, and propagate the learned information to subsequent layers. Both evaluated benchmarks are first implemented using the pthread library and then rewritten using the proposed mechanisms discussed in Section 4.4.1. The workload with critical section, SSSP, is implemented using the two methods described in the Sections 4.4.1 and 4.4.1. On the other hand, workload that require only barrier synchronization, CNN, is implemented with a simple barrier realization using explicit messaging (as explained in Sec. 4.4.1).

In the rest of this section, the implementation of the two benchmarks is discussed. SSSP is chosen because it has a large critical section which needs synchronization of data access. This is a representative graph benchmark that is known for its lack of scalability on traditional shared memory. CNN does not have any critical section, however it needs barriers for thread synchronization so learned data propagates to subsequent layers. These characteristics make this representative benchmark require barrier synchronization and producer-consumer communication, as are most machine learning algorithms.

Single Source Shortest Path (SSSP)

SSSP algorithm computes the shortest path in a graph with non-negative edge weights, and is a popular benchmark used in various applications, such as self-driving cars [111]. The algorithm starts from a user defined vertex, and hops over all the vertices in the graph, updating neighboring vertices with lowest path costs from the starting vertex. A distance array is updated with the lowest path costs, depicted by the relax function in Algorithm 1.

Algorithm 1 The relax function in SSSP, using locks

```
1:  Initialize distance_array(D)
2:  (v, u) is a vertex, neighbor pair
3:  << Parallel Relax Function >>
4:  for (each neighbor, u, of v) do
5:      Lock (u)
6:      if D[v] + D[v, u] < D[u] then
7:          D[u] ← D[v] + D[v, u]
8:      UnLock (u)
9:  barrier_wait(barrier)
10: update_range ()
11: barrier_wait(barrier)
```

The algorithm consists of two main loops, an outer loop that visits all the vertices, and the inner loop which visits all the neighboring vertices of a given vertex, each of which can be parallelized. Only the inner loop is shown in Algorithm 1, as it depicts the critical section. The outer loop can be parallelized in a controlled manner, where vertices are scheduled in pareto fronts to update distance costs. [98]. In the traditional shared memory version of SSSP, the outer loop is divided amongst threads dynamically by allocating chunks of pareto fronts amongst threads. Vertex path costs are updated using locks, Algorithm 1 lines 6 and 7, as threads may update distances of vertices with common neighbors. Real world graphs, such as road networks, are known to have a smaller numbers of neighboring vertices, and hence the outer loop parallelization works well in such common cases due to smaller synchronization requirements. In this work, range-based parallelization strategy is

used in which the outer loop is divided among threads depending on the calculated range. The range specifies which vertices can or cannot be relaxed, and is determined based on the input graphs size and connectivity [112]. Since propagations of range values need to be pushed to all threads simultaneously, a master thread updates the global range value via barrier synchronization (Algorithm 1 lines 9-11).

Algorithm 2 The relax function in SSSP, using messages

```
1:  Initialize distance_array(D)
2:  (v, u) is a vertex, neighbor pair
3:
4:  << Worker Thread : Parallel Relax Function >>
5:  update_range () using sendr
6:  for (each neighbor, u, of v) do
7:      send(v, u, tid)
8:
9:  << Service Thread : Parallel Relax Function >>
10: recv(v, u, tid)
11: if D[v] + D[v, u] < D[u] then
12:     D[u] ← D[v] + D[v, u]
```

The hybrid implementation differs from the shared memory version in two ways. First, it uses a controller thread to calculate new pareto fronts, and worker threads send "*work request*" messages to the controller thread to pull parallel work. This eliminates the need for barrier (Algorithm 1, lines 9-11), as the controller thread implicitly prohibits worker threads from pulling unsafe work. Second, it accelerates the critical section usage. The two implementations described in the Sections 4.4.1 and 4.4.1 are used to synchronize access to shared data. With the first method, lock management is parallelized using multiple services threads to manage the fine-grain locks. This distributes synchronization, and worker threads are dispatched to the services threads for lock requests instead of increasing the usual memory coherence traffic by ping-ponging lock cache lines around the chip. Moreover, the second method circumvents locks and atomically perform the critical section work

(i.e., updates distances) within a services thread that is responsible for that vertex. Worker threads send messages to the services threads with the target distance array (Algorithm 2 line 7), after which the services thread locally performs computations and commit the distance values (Algorithm 2 lines 10-12).

Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNNs) are special type of feed-forward neural networks in which each neuron in each convolutional layer is connected to a small region of the input. CNNs are utilized in various areas in machine learning such as image recognition, video analysis, and natural language processing. Specifically, they have accomplished state-of-the-art performance in the visual classification and recognition related applications, namely bank check reading, face recognition, and traffic sign recognition [113] [114] [115].

Algorithm 3 Shared Memory CNN Implementation

```
1:  << Parallel Work >>
2:  ConvolutionLayer(input, conv_out, tid, threads)
3:
4:  input=conv_out
5:  SubSamplingLayer(input, sub_out, tid, threads)
6:  barrier_wait(barrier)
7:
8:  input=sub_out
9:  FullyConnectedLayer(input, fully_out, tid, threads)
10: barrier_wait(barrier)
11:
12: input=fully_out
13: OutputLayer(input, output, tid, threads)
14: barrier_wait(barrier)
```

CNNs generally consist of multiple convolution layers. Each convolution layer contains multiple filters to extract different features from the given input data (input image or output of previous

layer). Convolution layer is often followed by a sub-sampling layer and non-linear function, such as rectified linear unit (ReLU) function or *tanh* function. Generally, one or more fully-connected layers come after these layers to be utilized as classifier layers, and generally a softmax layer is used as output layer to generate probability distribution of the classes [116].

Algorithm 4 Hybrid CNN Implementation

```
1:  << Worker Threads Parallel CNN >>
2:  ConvolutionLayer(input, conv_out, tid, threads)
3:
4:  input=conv_out
5:  SubSamplingLayer(input, sub_out, tid, threads)
6:  sendr(MasterThread, barrier)
7:
8:  input=sub_out
9:  FullyConnectedLayer(input, fully_out, tid, threads)
10: sendr(MasterThread, barrier)
11:
12: input=fully_out
13: OutputLayer(input, output, tid, threads)
14: sendr(MasterThread, barrier)
15:
16: << Master Thread >>
17: for ( each worker thread ) do
18:     recvmsg(&addr[thread], &msg)
19:
20: for ( each worker thread ) do
21:     resumer(addr[thread], continue)
```

Algorithm 3 depicts basic parallelization strategy for the convolution and sub-sampling layers of CNN under shared memory. As it is shown in the algorithm, the neurons are tiled and tiles are divided among threads. When the neurons are properly tiled in both convolution and sub-sampling layers, there is no need for synchronization between these layers. Synchronization is only needed after the sub-sampling layer before moving on to the next layer. Similarly, neuron-level parallelization is deployed in fully-connected layers. Since neurons in the fully-connected layers

are obliged to access all the outputs from previous layer neurons, thread synchronization is required after each fully-connected layer.

While traditional shared memory implementation exploits traditional spin-barriers under RISC-based systems to synchronize the threads, the hybrid approach implements barriers using explicit messages. As seen in Algorithm 4, each thread sends a blocking *sendr* message to the controller thread after they are done with their work. The controller thread receives messages from all the *worker threads*, and then sends back a *resumer* message to each *worker thread*. Unlike the traditional barrier implementation, the hybrid barrier eliminates the cache line ping-pong among threads and eliminates the extra test instructions. As CNN does not have a critical section and only uses barrier synchronization, remote function execution does not make sense for this type of workloads. If one tries to implement CNN with remote function execution model, it will look exactly the same as the version described above. The reason is that using a blocking send message is a must in implementing a barrier primitive.

4.5 Discussion

4.5.1 Memory Consistency Implications

In order to ensure that all explicit messages have been observed at their destinations, the RISC fence instruction is extended for explicit messages. A barrier in the proposed architecture is implemented using the blocking *sendr* instruction, which provides synchronization guarantee but not data consistency. Therefore, a fence instruction may be required before the *sendr* to ensure data consistency. A fence instruction must ensure that all pending messages in the *send queue* are pushed into the network and observed at the receiving side. The condition for observability at the receiving side is ensured by monitoring the *capacity counter* since it tracks all in-flight messages

whose *ACKs* haven't been observed yet. Once the *capacity counter* reaches its initialized value, all sent messages have been seen at their destinations. At this point, the core moves on and commits the fence instruction.

4.5.2 Asynchronous Communication

The proposed hybrid architecture supports synchronous transfer of data using the *recv* instruction. Another option could have been to get rid of this instruction and interrupt the core on a message arrival, similar to prior work [108, 107]. The core can then perform a context switch and execute a service routine to handle the message. This approach can cause continuous context switching if the rate of arrival of messages is high. However, if one wishes to implement such asynchronous communication, it is possible to do so in the proposed architecture. For that, a thread can spin on a *recv* instruction, going to sleep if there is no data to consume. The thread is woken up once a message arrives and can then perform the desired operation. This is the approach taken to implement *services threads* in a multi-threaded core setup.

4.5.3 Context Switching & Thread Migration

Supporting context switching and thread migration is a necessity for a general purpose processor. However, the proposed architecture can deadlock if in-flight explicit messages are not dealt with properly. This can happen because an in-flight message can be delivered to a core where the thread is not running any more. To properly handle this situation, a clean-up mechanism is required to ensure all in-flight messages are delivered before thread migration can occur. The clean-up mechanism works as follows. The OS halts all the threads from injecting any more messages into the network. After that, the OS monitors the *capacity counter* of each thread and waits for them to get back to

their initialized values. This signifies that all the threads have received *ACKs* for all their in-flight messages and there is no explicit message in the network. Therefore, the OS only allows context switches when the whole system is in a clean state. At this point, there are no valid entries in the the send and receive queues and the OS can perform the context switch. It also updates the thread-to-core mapping so that future messages can get to their destination properly. If a thread tries to send a message to a thread that in not currently active, the OS will first switch the required thread in and then allow the send to be inserted in the send queue (eventually making it to the desired thread).

4.5.4 Pushing Data Model

Maintaining cache coherence for shared read-write data can result in generating additional on-chip messages. These messages can negatively impact the performance in a workload that consists of mainly shared read-write data. Furthermore, a core that needs some data, *pulls* it in by actively requesting it. In a producer-consumer setting with multiple producers and consumers, accesses to the shared data is synchronized through a *barrier*. One can implement the strategy discussed in Sec 4.4.1 to improve the performance of barriers. However, the workload can be redesigned to take advantage of the explicit communication. In this model, the producers *push* their data towards the consumers through explicit communication. The consumers can only use that data once it is pushed towards them, hence, eliminating the need for a barrier.

This approach eliminates the coherence indirections, resulting in lower on-chip traffic. The removal of barrier further improves performance by increasing resource utilization (due to lower wait times). However, a producer thread now needs to send all the data explicitly by executing a special software routine that generates explicit send instructions. This can quickly become a bottleneck of its own and supersede any benefits achieved. The described behavior is prominent in

workloads with one-to-all type of communication (cf. Sec 4.4.2). This behavior is a use-case that underlines the necessity of retaining shared memory. Furthermore, the system is prone to deadlock if a finite send capacity counter is used. This can easily happen in an all-to-all communication pattern where every thread needs to send a certain number of explicit messages before it can start receiving messages. In this case, if all threads stop sending messages because they run out of send capacity, the system can no longer make forward progress and deadlock. The only way to resolve the deadlock situation is to keep a very high send capacity counter. However, that can become impractical as the buffering required at the receiver can be prohibitively high (cf. Sec 4.4.2).

4.6 Evaluation Methodology

4.6.1 Software Tool-chain

An LLVM-based compiler has been developed that supports the proposed ISA extensions. An application must be re-compiled using the compiler. The compiler itself does not inherently understand explicit messaging. Instead, the send and receive instructions are simply wrapped within assembly blocks, using the gcc extended asm block syntax to instruct the compiler as to what registers are inputs or outputs. This allows the compiler to allocate registers properly and schedule the code.

4.6.2 Simulator Setup

The proposed architecture has been implemented using an in-house industry-class simulator and associated tool-chain for the LLVM-based compiler. A compiled application is fed to an Architecture Description Language (ADL) [117] front-end, which in turn drives the performance models. A

Architectural Parameter	**Value**
Number of Cores	256 @ 1 GHz
Compute Pipeline per Core	In-Order, Single-Issue
Physical Address Length	48 bits
Memory Subsystem	
L1-I Cache per core	16 KB, 4-way Assoc., 1 cycle
L1-D Cache per core	32 KB, 4-way Assoc., 1 cycle
L2 Cache per core	256 KB, 8-way Assoc. 2 cycle tag, 4 cycle data Inclusive
Cache Line Size	64 bytes
Directory Protocol	Invalidation-based MESI ACKwise$_4$ [47]
Num. of Memory Controllers	8
DRAM Bandwidth	10 GBps per Controller
DRAM Latency	100 ns
Explicit Communication	
Send queue per core	4 entry
Receive queue per core	128 entry
Electrical 2-D Mesh with XY Routing	
Hop Latency	2 cycles (1-router, 1-link)
Contention Model	Only link contention (Infinite input buffers)
Flit Width	64 bits

Table 4.6.1: Architectural parameters for evaluation.

futuristic 256-core tiled many-core processor with a two-level private L1, shared L2 cache hierarchy per core is evaluated. The default architectural parameters used for evaluation are shown in Table 4.6.1. Single-issue, in-order compute cores are modeled because the power consumption of complex out-of-order cores can be prohibitively high at large core count. Furthermore, the high concurrency in present and future applications makes the case for using many simple cores, as compared to a lower number of complex out-of-order cores.

4.6.3 Performance Models

All experiments are performed using the core, cache hierarchy, coherence protocol, memory system, and on-chip interconnection network models implemented within the simulator. These models have been derived from Graphite multicore simulator [49]. The performance models have been extended to accurately account for explicit communication instructions. All mechanisms and protocol overheads discussed in Section 4.3 are modeled. The electrical mesh interconnection network uses XY routing. Since modern network-on-chip routers are pipelined [86], and 2- or even 1-cycle per hop router latencies [87] have been demonstrated, a 2-cycle per hop delay is modeled; the appropriate pipeline latencies associated with loading and unloading a packet onto the network is also accounted for. In addition to the fixed per-hop latency, network contention delays are also modeled (derived from Graphite multicore simulator).

4.6.4 Benchmarks and Evaluation Metrics

Each multi-threaded benchmark is run to completion using NYC Road Network [118] for SSSP and MNIST [119] for CNN as the input. For performance, the completion time is measured, i.e., the time in the *parallel* region of the benchmark. The parallel completion time is broken down into the following categories:

1. *Instructions:* Time spent retiring instructions.
2. *L1-I Fetch Stalls:* Stall time due to instruction cache misses.
3. *Compute Stalls:* Stall time due to waiting for functional unit (ALU, FPU, Multiplier, etc.) results.
4. *Memory Stalls:* Stall time due to load/store queue capacity limits, fences and waiting for load

completion.

5. *Branch Speculation:* Stall time due to mis-predicted branch instructions.

6. *Communication Stalls:* Stall time due to waiting on locks, barriers and condition variables.

The communication latency includes stalls caused by *send* (due to blocking in the send queue, as discussed in Sec 4.3.3), *recv* (waiting for the message to arrive), and *sendr* (round-trip latency to the destination, wait time in the destination's receive queue, and destination's execution latency) instructions.

4.6.5 Configurations

1. **Shared Memory:** This is the baseline system which relies on traditional synchronization primitives to implement locks and barriers.

2. ***Acc:*** This configuration replaces shared memory synchronization with explicit communication. It dedicates a certain number of cores to manage locks and barriers as described in Sec 4.4.1.

3. ***MC:*** Moving computation to data. This configuration removes the need for synchronization through locks completely and pins the critical section at one or more cores (cf. Sec 4.4.1). This approach enables efficient management of data locality by eliminating cache line ping-pong, and also eliminates overhead of round-trip latency for acquiring lock.

4. ***Push:*** This configuration removes the need for barrier synchronization and instead transfer data using explicit communication, as described in Sec 4.5.4.

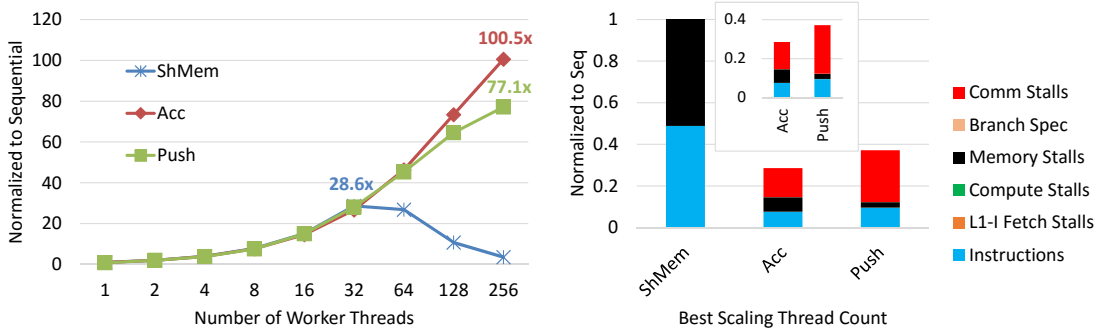


Figure 4.7.1: Completion time results for CNN under shared memory, *Acc*, and *Push*. All results are normalized to sequential baseline.

4.7 Results

4.7.1 The Case For Shared Memory

Shared memory paradigm is prevalent in the computing world because of its ease of use and the fact that the complexity is hidden from the programmer. Shared memory is efficient in exploiting spatio-temporal locality for private and shared read-only data through the underlying coherence and consistency protocols. Tesseract [107] makes a case for implementing a message passing only system and doing away with shared memory. The argument is valid for workloads with fine-grain locks and contended data (such as graph workloads). However, it is not feasible to transfer large amounts of data in a multiple producers, multiple consumers setting efficiently with message passing only. I claim that shared memory is a necessity to enable efficient data transfer and that accelerating the synchronization barrier provides the extra performance. This claim is validated through a machine learning workload, CNN, which needs to shovel massive amounts of data from producers to consumers (cf. Sec 4.4.2).

Figure 4.7.1 shows the scaling trends of CNN at 1 to 256 threads under different configurations. The shared memory version utilizes coherence protocol and barrier synchronization to keep data

consistent. Its clearly visible that the barrier becomes prohibitively expensive at higher thread count because of excessive retries and memory operations. Shared memory CNN only scales till 32 threads to achieve a speedup of $28.6\times$ over sequential and the performance starts deteriorating after that. *Acc* performs the data transfer through shared memory, however, it accelerates the barrier using techniques described in Sec 4.4.1. The barrier implementation in *Acc* eliminates instruction retries and useless memory accesses. *Acc* scales up to 256 threads and achieve a speedup of $3.5\times$ over the best point in shared memory and $100.5\times$ over sequential.

To validate the claim that shared memory is needed for efficient data transfer, experiments are also run with *Push*. In this configuration, synchronization as well as data transfer is performed using explicit communication. As CNN contains all-to-all communication between different layers, using explicit messaging starts becoming expensive real quick. The reason being that the producer thread has to execute 255 *send* instructions (at 256 threads) to transfer data to the consumers. In comparison, data transfer through shared memory only require a single store to a globally shared structure. The consumers threads can then concurrently perform a load to consume the data. As each producer needs to send its data to all other threads, it increases the congestion in the on-chip network. Consequently, same performance gains are not observed as in *Acc*. *Push* provides a speedup of $77.1\times$ over sequential at 256 threads, $\sim 23\%$ slower than *Acc*. It is still better than the shared memory version by $2.7\times$ because it also accelerates the barrier. It should be noted that the experiment for *Push* was run with an infinite send capacity counter, as limiting it deadlocks the system. The max utilization at the receive queues was monitored in order to have a sense of the kind of buffering needed to make it work. Experiments showed a max utilization of 720, prohibitively high overhead for any system.

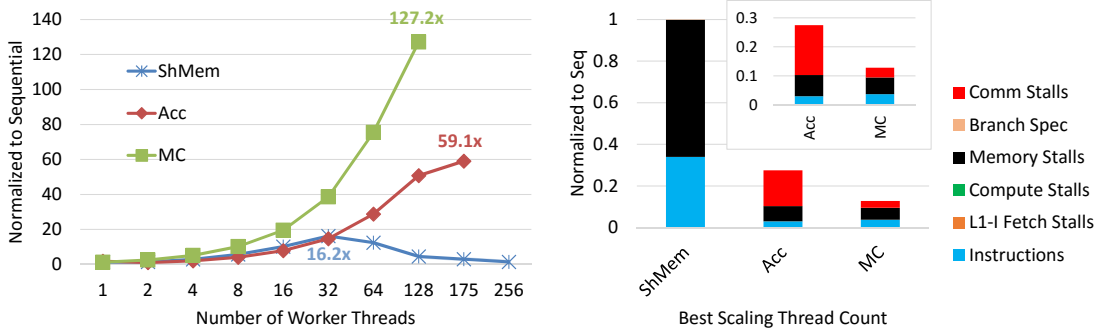


Figure 4.7.2: Completion time results for SSSP under shared memory, *Acc*, and *MC*. All results are normalized to sequential baseline.

4.7.2 The Case For Explicit Communication

Figure 4.7.2 shows the scaling trends of SSSP at 1 to 256 threads under different configurations. As illustrated in the figure, shared memory implementation scales to 32 threads and shows a maximum of $16.2\times$ speedup. The speedup decreases after 32 threads because of the instruction retries to acquire the lock. Instruction retries both increases the instruction count and memory operations drastically which limits the scalability in benchmarks like SSSP due to contended locks. Increasing thread count from 32 to 64 doubles the L1D accesses and boosts the sharing misses by $\sim 3\times$. Even though increasing thread count helps with the compute time, blowup in the memory stall suppresses the benefit and limits the performance. Further increase in the thread count causes instruction count to balloon as well.

Figure 4.7.2 also displays the scalability results for the *Acc* implementation using multiple services threads as lock managers. The Acc implementation improves the scalability up to 256 threads and yields a maximum speedup of $59.1\times$ over sequential and $3.65\times$ over best shared memory speedup. The maximum speedup is obtained using 80 threads as lock managers while the rest as worker threads. By deploying a lock manager the cache line ping-pong and instructions retries are eliminated, and multiple lock managers overlap communication and computation providing

further gains.

MC affixes the critical section at a set of cores, as described in Sec 4.4.1. This improves over *Acc* by eliminating the lock acquisition overhead and pinning the shared data at a single location. As the *worker threads* perform the critical section update using non-blocking *send* messages, more computation is overlapped with communication. The compute pipeline can still stall due to other reasons described in Sec 4.3.3. As a send capacity counter of 4 is used in this experiment, communication stalls do occur and are visible in fig. 4.7.2. SSSP is able to take advantage of *MC* and scales all the way up to 256 threads to achieve a speedup of $127.2\times$ over sequential. This translates to a speedup of $7.8\times$ over shared memory and $2.1\times$ over *Acc*.

4.7.3 SSSP – Services Threads Sweep Study

In order to determined the right balance between worker and services threads in *MC*, a sweep study is performed. Figure 4.7.3 shows the results for this experiment. The number of services threads are varied from 32 to 160 in 32 threads increments while keeping the send capacity counter at 4. With increasing services threads, the number of worker threads goes down with each worker thread doing more application work. This can be seen in fig. 4.7.3 where the number of instructions executed is increasing with increasing services threads. Furthermore, the memory stall also increases, as the same number of accesses are now performed by a smaller number of worker threads. However, as there are more services threads, the computation is being posted to more threads, increasing concurrency. Additionally, the higher number of services threads process the requests faster, enabling the worker threads to not stall due to send capacity bottleneck. This is visible in the figure where the red portion of the bar, "Communication Stalls", goes down with increasing services threads. 128 services threads delivers the best completion time due to the reasons discussed above, delivering $> 3\times$ speedup over 32 services threads configuration. Moving to 160 services threads

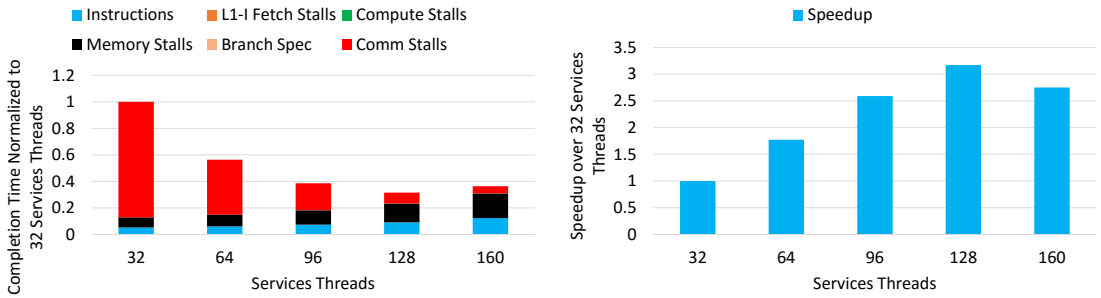


Figure 4.7.3: Completion time (left) and speedup (right) observed at varying services thread count for SSSP under *MC* at send capacity counter of 4. Results are relative to 32 services threads (1st bar).

further reduces the "Communication Stalls", however, number of instructions executed and the memory access latency increases enough to nullify the gains and the overall completion time is higher compared to that of 128 services threads.

4.7.4 SSSP – *Capacity Counter* Sweep Study

A sweep study is conducted to determined the optimal value of send capacity counter in *MC*. An optimal value is defined as one that keeps the max utilization of the receive queues reasonable while delivering high performance. Figure 4.7.4 shows the results for this experiment. The send capacity counter is varied from 2 to 8 in increments of 2 while keeping the services threads count at 128. The increase in send capacity counter affects the communication stalls component of the completion time. Higher the capacity counter, lower the communication stalls. The reason being that a worker thread can have more in-flight messages in the network. However, if the services threads are not fast enough, the messages can build up in the receive queue. This inflates the max utilization of the receive queues, increasing the hardware required to support maximum throughput. The max utilization for each capacity counter data point is mentioned in fig. 4.7.4 (left). It can be seen that going from capacity counter of 2 to 4, there is a big jump in speedup. However, increasing the value further provides diminishing returns. Furthermore, a capacity counter value of 4 results in a

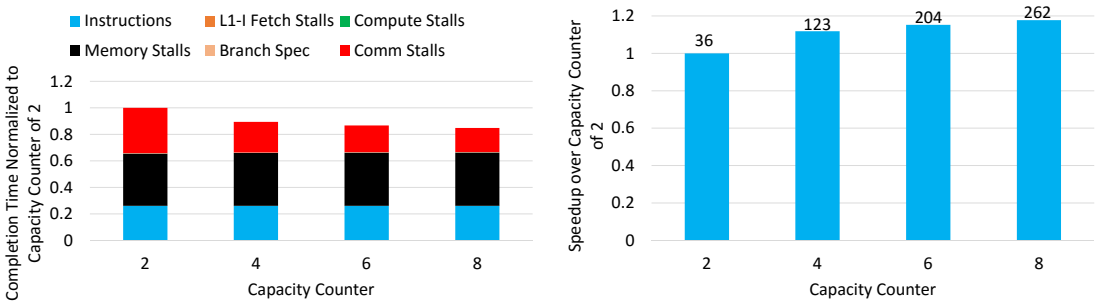


Figure 4.7.4: Completion time (left) and speedup (right) observed at varying send capacity counter for SSSP under *MC* at 128 services threads. Results are relative to a send capacity counter of 2 (1st bar).

max receive queue utilization of 123. This enables us to keep the receive queue size of 128 entries, keeping the overheads small.

4.8 Conclusion

This chapter proposes a hybrid shared memory, explicit messaging many-core architecture that implements instruction-level explicit messaging instructions on top of the shared memory paradigm. The idea is to moved as much computation to data as possible, hence accelerating communication, in future single-chip many-core processors that are expected to execute emerging workloads, such as graph analytic and machine learning. These workloads exhibit fine-grain communication and data sharing for large amounts of data. The communication models introduced by the proposed architecture enable such workloads to substantially improve performance over the traditional shared memory paradigm.

Chapter 5

Conclusion

Many-core processors with hundreds of cores on a single chip are projected to be available within the next decade to meet the performance requirements of future applications. Scaling to hundreds of cores on a single chip present a number of challenges, mainly efficient data access and on-chip communication. This thesis provides practical solutions to challenges with efficient data access and efficient on-chip communication. The contributions made in this thesis enable energy efficient and scalable many-core architectures.

This thesis first discusses energy efficient operation for future many-core processors at nominal as well as near-threshold voltages. Operating at near-threshold voltage renders some of the memory bit-cells faulty due to PVT variations. This work presents a private L1 cache architecture that balances the tradeoff between access latency and cache capacity. NUCA-L1 can be generalized to L2 and lower level caches as well.

The thesis then presents a data replication scheme to improve the utilization of the last level cache and reduce on-chip data movement. The scheme balances the tradeoff between improving on-chip data locality and off-chip miss rate. This is achieved by implementing a low-overhead yet

highly accurate hardware-only predictive mechanism to track and classify the *reuse* of each cache line in the LLC. The proposed replication scheme enables lower memory access latency and energy by selectively replicating cache lines that show *high reuse* in the LLC slice of the requesting core, while bypassing replication for cache lines with *low reuse*. This improves data access locality by minimizing the number of accesses to remote locations in a single chip many-core processor.

Finally, the thesis proposes a low-overhead and deadlock-free shared memory–explicit messaging hybrid architecture. The architecture introduces an explicit messaging layer on top of shared memory that enables programmable communication models, overlapping of communication with computation, efficient management of data locality, and movement of computation to data. The proposed architecture significantly mitigates the bottlenecks caused by shared memory synchronization and inefficient on-chip communication.

Bibliography

[1] Shekhar Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[2] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. Near-threshold voltage (ntv) design: opportunities and challenges. In *Design Automation Conference*, 2012.

[3] A.P. Chandrakasan, D.C. Daly, D.F. Finchelstein, J. Kwong, Y.K. Ramadass, M.E. Sinangil, V. Sze, and N. Verma. Technologies for ultradynamic voltage scaling. *Proceedings of the IEEE*, 98(2):191–214, Feb 2010.

[4] Ronald G. Dreslinski, David Fick, Bharan Giridhar, Gyouho Kim, Sangwon Seo, Matthew Fojtik, Sudhir Satpathy, Yoonmyung Lee, Daeyeon Kim, Nurrachman Liu, Michael Wieckowski, Gregory Chen, Dennis Sylvester, David Blaauw, and Trevor Mudge. Centip3de: A 64-core, 3d stacked near-threshold system. *IEEE Micro*, 33(2):8–16, 2013.

[5] Farrukh Hijaz and Omer Khan. Nuca-l1: A non-uniform access latency level-1 cache architecture for multicores operating at near-threshold voltages. *ACM Trans. Archit. Code Optim.*, 11(3):29:1–29:28, October 2014.

[6] Kathy Yelick. Ten ways to waste a parallel computer. In *Keynote at International Symposium on Computer Architecture*, 2009.

[7] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *Micro, IEEE*, 30(2):16–29, 2010.

[8] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *ASPLOS*, 2002.

[9] Farrukh Hijaz, Qingchuan Shi, and Omer Khan. A private l1 cache architecture to exploit the latency and capacity tradeoffs in multicores operating at near-threshold voltages. In

Proceedings of the 2013 31st IEEE International Conference on Computer Design, ICCD '13, 2013.

[10] Farrukh Hijaz and Omer Khan. Rethinking last-level cache management for multicores operating at near-threshold voltages. In *Second Workshop on Near-threshold Computing*, 2014.

[11] Alaa R. Alameldeen, Ilya Wagner, Zeshan Chishti, Wei Wu, Chris Wilkerson, and Shih-Lien Lu. Energy-efficient cache design using variable-strength error-correcting codes. In *Int'l Symposium on Computer Architecture*, 2011.

[12] A. Ansari, Shuguang Feng, S. Gupta, and S. Mahlke. Archipelago: A polymorphic cache design for enabling robust near-threshold operation. In *Int'l Symposium on High Performance Computer Architecture*, 2011.

[13] Ulya R. Karpuzcu, Abhishek Sinkar, Nam Sung Kim, and Josep Torrellas. Energysmart: Toward energy-efficient manycores for near-threshold computing. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 542–553, Feb 2013.

[14] Zhiyu Liu and V. Kursun. High read stability and low leakage cache memory cell. In *Int'l Symposium on Circuits and Systems*, 2007.

[15] F. Moradi, D.T. Wisland, S. Aunet, H. Mahmoodi, and Tuan-Vu Cao. 65nm sub-threshold 11t-sram for ultra low voltage applications. In *Int'l SOC Conf.*, 2008.

[16] Y. Morita, H. Fujiwara, H. Noguchi, Y. Iguchi, K. Nii, H. Kawaguchi, and M. Yoshimoto. An area-conscious low-voltage-oriented 8t-sram design under dvs environment. In *Symp. on VLSI Circuits*, 2007.

[17] G.K. Chen, D Blaauw, T. Mudge, D Sylvester, and Nam Sung Kim. Yield-driven near-threshold sram design. In *Int'l Conference on Computer-Aided Design*, 2007.

[18] B. Maric, J. Abella, and M. Valero. Apple: Adaptive performance-predictable low-energy caches for reliable hybrid voltage operation. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–8, May 2013.

[19] J.P. Kulkarni, Keejong Kim, and K. Roy. A 160 mv robust schmitt trigger based subthreshold sram. *IEEE Journal of Solid-State Circuits*, 2007.

[20] C.L. Chen and M.Y. Hsiao. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development*, 28(2):124–134, 1984.

[21] Jangwoo Kim, N. Hardavellas, Ken Mai, B. Falsafi, and J.C. Hoe. Multi-bit error tolerant caches using two-dimensional error coding. In *Int'l Symposium on Microarchitecture*, pages 197–209, 2007.

[22] Doe Hyun Yoon and Mattan Erez. Memory mapped ecc: low-cost error protection for last level caches. In *Int'l Symp. on Computer Arch.*, 2009.

[23] Z. Chishti, A.R. Alameldeen, C. Wilkerson, Wei Wu, and Shih-Lien Lu. Improving cache lifetime reliability at ultra-low voltages. In *Int'l Symposium on Microarchitecture*, 2009.

[24] T.N. Miller, R. Thomas, J. Dinan, B. Adcock, and R. Teodorescu. Parichute: Generalized turbocode-based error correction for near-threshold caches. In *Int'l Conf on Microarchitecture*, 2010.

[25] Chris Wilkerson, Hongliang Gao, Alaa R. Alameldeen, Zeshan Chishti, Muhammad Khellah, and Shih-Lien Lu. Trading off cache capacity for reliability to enable low voltage operation. In *Int'l Symposium on Computer Architecture*, 2008.

[26] D. Roberts, Nam Sung Kim, and T. Mudge. On-chip cache device scaling limits and effective fault repair techniques in future nanoscale technology. In *Euromicro Conference*, 2007.

[27] Jaume Abella, Javier Carretero, Pedro Chaparro, Xavier Vera, and Antonio González. Low vccmin fault-tolerant cache with highly predictable performance. In *International Symposium on Microarchitecture*, 2009.

[28] Jaume Abella, Eduardo Quiñones, Francisco J. Cazorla, Yanos Sazeides, and Mateo Valero. Rvc: A mechanism for time-analyzable real-time processors with faulty caches. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 97–106. ACM, 2011.

[29] Farrukh Hijaz, Qingchuan Shi, and Omer Khan. Low-latency mechanisms for near-threshold operation of private caches in shared memory multicores. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture Workshops*, MICROW '12, pages 68–73, Washington, DC, USA, 2012. IEEE Computer Society.

[30] M. Franklin and K.K. Saluja. Built-in self-testing of random-access memories. *Computer*, 23(10):45–56, 1990.

[31] S.E. Schuster. Multiple word/bit line redundancy for semiconductor memories. *IEEE Journal of Solid-State Circuits*, 1978.

[32] R.G. Gallager. Low-density parity-check codes. *Information Theory, IRE Transactions on*, 8(1):21–28, 1962.

[33] Peter Calingaert. Two-dimensional parity checking. *J. ACM*, 8(2):186–200, April 1961.

[34] David H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *International Symposium on Microarchitecture*, 1999.

[35] Se-Hyun Yang, Babak Falsafi, Michael D. Powell, Kaushik Roy, and T. N. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. In *International Symposium on High Performance Computer Architecture*, 2001.

[36] Se-Hyun Yang, Babak Falsafi, Michael D. Powell, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *HPCA*, 2002.

[37] T. R. N. Rao and E. Fujiwara. *Error-control coding for computer systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[38] Y. Kamiyanagi Hideki Imai. A construction method for double error correcting codes for application to main memories. *Transactions of the IECE Japan*, J60-D:861–868, October 1977.

[39] Tomoko K. Matsushima Toshiyasu Matsushima Shigeichi Hirasawa. Parallel encoder and decoder architecture for cyclic codes. *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences Vol.E79-A No.9 pp.1313-1323*, 1996.

[40] D. Strukov. The area and latency tradeoffs of binary bit-parallel bch decoders for prospective nanoelectronic memories. In *Signals, Systems and Computers, 2006. ACSSC '06. Fortieth Asilomar Conference on*, 2006.

[41] Sheng Li, Ke Chen, Ming-Yu Hsieh, N. Muralimanohar, C.D. Kersey, J.B. Brockman, A.F. Rodrigues, and N.P. Jouppi. System implications of memory reliability in exascale computing. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12, 2011.

[42] Elwyn. R. Berlekamp. *Algebraic coding theory*. McGraw-Hill, 1968.

[43] J Massey. Step-by-step decoding of the bose-chaudhuri-hocquenghem codes. *Information Theory, IEEE Transactions on*, 11(4):580–585, 1965.

[44] I.S. Reed and M.T. Shih. Vlsi design of inverse-free berlekamp-massey algorithm. *Computers and Digital Techniques, IEE Proceedings E*, 138(5):295–298, 1991.

[45] J.C. Smolens, B.T. Gold, B. Falsafi, and J.C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 223–234, 2006.

[46] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - processor: A 64-Core SoC with mesh interconnect. In *International Solid-State Circuits Conference*, 2008.

[47] George Kurian, Jason Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, Lionel Kimerling, and Anant Agarwal. ATAC: A 1000-Core Cache-Coherent Processor with On-Chip Optical Network. In *Int'l Conference on Parallel Architectures and Compilation Techniques*, 2010.

[48] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *International Symposium on Computer Architecture*, 2009.

[49] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA*, pages 1–12, 2010.

[50] Chen Sun, Chia-Hsin Owen Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. DSENT - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *Int'l Symposium on Networks-on-Chip*, 2012.

[51] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[52] Khakifirooz, A. and Nayfeh, O.M. and Antoniadis, D. A Simple Semiempirical Short-Channel MOSFET Current-Voltage Model Continuous Across All Regions of Operation and Employing Only Physical Parameters. *Electron Devices, IEEE Transactions on*, 56(8):1674 –1680, aug. 2009.

[53] Lan Wei, F. Boeuf, T. Skotnicki, and H.-S.P. Wong. Parasitic Capacitances: Analytical Models and Impact on Circuit-Level Performance. *Electron Devices, IEEE Transactions on*, 58(5):1361 –1370, may 2011.

[54] M.K. Qureshi and Z. Chishti. Operating secded-based caches at ultra-low voltage with flair. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–11, 2013.

[55] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Conference on Computer Architecture*, 1995.

[56] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[57] S.M.Z. Iqbal, Yuchen Liang, and H. Grahn. ParMiBench - an open-source benchmark for embedded multiprocessor systems. *Computer Architecture Letters*, 2010.

[58] DARPA UHPC Program BAA. https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-10-37/listing.html, March 2010.

[59] Farrukh Hijaz, Qingchuan Shi, George Kurian, Srinivas Devadas, and Omer Khan. Locality-aware data replication in the last-level cache for large scale multicores. *The Journal of Supercomputing*, 72(2):718–752, 2016.

[60] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598, Feb 2008.

[61] Anant Agarwal, Richard Simoni, John L. Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *International Symposium on Computer Architecture*, 1988.

[62] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7), 2012.

[63] Daniel Sanchez and Christos Kozyrakis. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. In *International Symp. on High-Performance Computer Architecture*, 2012.

[64] Hongzhou Zhao, Arrvindh Shriraman, and Sandhya Dwarkadas. SPACE: sharing pattern-based directory coherence for multicore scalability. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 135–146, 2010.

[65] Jason Zebchuk, Vijayalakshmi Srinivasan, Moinuddin K. Qureshi, and Andreas Moshovos. A tagless coherence directory. In *International Symposium on Microarchitecture*, 2009.

[66] Noel Eisley, Li-Shiuan Peh, and Li Shang. In-network cache coherence. In *IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 321–332, 2006.

[67] George Kurian, Omer Khan, and Srinivas Devadas. The locality-aware adaptive cache coherence protocol. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 523–534, New York, NY, USA, 2013. ACM.

[68] First the tick, now the tock: Next generation Intel microarchitecture (Nehalem). White Paper, 2008.

[69] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 357–368, Washington, DC, USA, 2005. IEEE Computer Society.

[70] Michael Zhang and Krste Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Int'l Symposium on Computer Architecture*, 2005.

[71] Bradford M. Beckmann, Michael R. Marty, and David A. Wood. Asr: Adaptive selective replication for cmp caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 443–454, Washington, DC, USA, 2006. IEEE Computer Society.

[72] M. Chaudhuri. PageNUCA: selected policies for page-grain locality management in large shared chip-multiprocessor caches. In *HPCA*, pages 227–238, 2009.

[73] Qingchuan Shi, F. Hijaz, and O. Khan. Towards efficient dynamic data placement in noc-based multicores. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 369–376, Oct 2013.

[74] J. Merino, V. Puente, and J.A. Gregorio. Esp-nuca: A low-cost adaptive non-uniform cache architecture. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–10, Jan 2010.

[75] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput.*, 27(12):1112–1118, December 1978.

[76] Sangyeun Cho and Lei Jin. Managing distributed, shared l2 caches through os-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 455–468, Washington, DC, USA, 2006. IEEE Computer Society.

[77] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 250–261, Feb 2009.

[78] George Kurian, Srinivas Devadas, and Omer Khan. Locality-aware data replication in the last-level cache. In *High Performance Computer Architecture (HPCA2014), 2014 IEEE 120th International Symposium on*, Feb 2014.

[79] Jichuan Chang and G.S. Sohi. Cooperative caching for chip multiprocessors. In *Computer Architecture, 2006. ISCA '06. 33rd International Symposium on*, pages 264–276, 2006.

[80] Enric Herrero, José González, and Ramon Canal. Elastic cooperative caching: An autonomous dynamically adaptive memory hierarchy for chip multiprocessors. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 419–428, New York, NY, USA, 2010. ACM.

[81] M.K. Qureshi. Adaptive spill-receive for robust high-performance caching in cmps. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 45–54, Feb 2009.

[82] S. Srikantaiah, E. Kultursay, Tao Zhang, M. Kandemir, M.J. Irwin, and Yuan Xie. Morphcache: A reconfigurable adaptive multi-level cache hierarchy. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 231–242, Feb 2011.

[83] Hyunjin Lee, Sangyeun Cho, and B.R. Childers. Cloudcache: Expanding and shrinking private caches. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 219–230, Feb 2011.

[84] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures in Computer Architecture, Morgan Claypool Publishers, 2011.

[85] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 151–162, Washington, DC, USA, 2010. IEEE Computer Society.

[86] William J Dally and Brian Towles. *Principles and practices of interconnection networks*. Morgan Kaufmann, 2004.

[87] Sunghyun Park, Tushar Krishna, Chia-Hsin Chen, Bhavya Daya, Anantha Chandrakasan, and Li-Shiuan Peh. Approaching the theoretical limits of a mesh noc with a 16-node chip prototype in 45nm soi. In *Design Automation Conference*, 2012.

[88] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *ISCA*, 2008.

[89] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, November 2014.

[90] M. Ahmad, F. Hijaz, Qingchuan Shi, and O. Khan. Crono: A benchmark suite for multi-threaded graph algorithms executing on futuristic multicores. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 44–55, Oct 2015.

[91] Farrukh Hijaz, Brian Kahne, Peter Wilson, and Omer Khan. Accelerating communication in single-chip shared memory many-core processors. In *Sixth Annual Boston Area Architecture Workshop*, 2015.

[92] F. Hijaz, B. Kahne, P. Wilson, and O. Khan. Efficient parallel packet processing using a shared memory many-core processor with hardware support to accelerate communication. In *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*, pages 122–129, Aug 2015.

[93] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008.

[94] Julian Mcauley and Jure Leskovec. Discovering social circles in ego networks. *ACM Trans. Knowl. Discov. Data*, 8(1):4:1–4:28, February 2014.

[95] A. Kundaje and et. al. Integrative analysis of 111 reference human epigenomes. In *Nature 518*, Jan 2015.

[96] J. W. Lichtman, H. Pfister, and N. Shavit. The big data challenges of connectomics. In *Nature Neuroscience 17*, Sept 2014.

[97] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising c/c++ and power. In *PLDI*, 2012.

[98] Shuai Che, B.M. Beckmann, S.K. Reinhardt, and K. Skadron. Pannotia: Understanding irregular gpgpu graph applications. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 185–195, Sept 2013.

[99] Jin Wang and Sudhakar Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured gpu applications. In *2014 IEEE International Symposium on Workload Characterization*, October 2014.

[100] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. Deterministic galois: On-demand, portable and parameterless. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 499–512, 2014.

[101] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[102] W. D. Gropp and M. Snir. Programming for exascale computers. 2013.

[103] John Kubiatowicz and Anant Agarwal. The anatomy of a message in the alewife multiprocessor. In *ICS*, 1993.

[104] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *ISCA*, 1992.

[105] L.M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *Computers, IEEE Transactions on*, C-27(12):1112–1118, Dec 1978.

[106] http://www.tilera.com. Tile-gx processor family: Product brief. 2011.

[107] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 105–117, New York, NY, USA, 2015. ACM.

[108] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 311–322, New York, NY, USA, 2010. ACM.

[109] R. Harting and William Dally. On-chip active messages for speed, scalability, and efficiency. *TPDS*, 99(PrePrints), 2014.

[110] J Balkind, M McKeown, Y Fu, T Nguyen, Y Zhou, A Lavrov, M Shahrad, A Fuchs, S Payne, X Liang, M Matl, and D Wentzlaff. Openpiton: An open source manycore research framework. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, April 2016.

[111] Google Is Building Its Own Self-Driving Car Prototypes. http://spectrum.ieee.org/cars-that-think/transportation/self-driving/google-is-building-selfdriving-car-prototypes, May, 2014.

[112] Jin Y. Yen. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. Quarterly of Applied Mathematics 27: 526530., November 1970.

[113] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[114] A Frome, G Cheung, A Abdulkader, M Zennaro, B Wu, A Bissacco, H Adam, H Neven, and L Vincent. Large-scale privacy protection in street-level imagery. *ICCV09*, 1(2), 2009.

[115] Pierre Sermanet and Yann LeCun. Traffic sign recognition with multi-scale convolutional networks. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 2809–2813. IEEE, 2011.

[116] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[117] Brian Kahne. FreescaleADL: An Industrial-Strength Architectural Description Language For Programmable Cores. http://opensource.freescale.com/fsl-oss-projects/, June 2013.

[118] 9th DIMACS Implementation Challenge. http://www.dis.uniroma1.it/challenge9/download.shtml, November 2006.

[119] Yann Lecun and Corinna Cortes. The MNIST database of handwritten digits.