Proximity Coherence for Chip Multiprocessors

Nick Barrow-Williams
Computer Laboratory
University of Cambridge
Cambridge CB3 0FD, UK
npb28@cl.cam.ac.uk

Christian Fensch
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, UK
c.fensch@ed.ac.uk

Simon Moore
Computer Laboratory
University of Cambridge
Cambridge CB3 0FD, UK
swm11@cl.cam.ac.uk

ABSTRACT

Many-core architectures provide an efficient way of harnessing the increasing numbers of transistors available in modern fabrication processes. While they are similar to multi-node systems, they exhibit different communication latency and storage characteristics, providing new design opportunities that were previously not feasible. Traditional cache coherence protocols, although often used in many-core designs, have been developed in the context of multi-node systems. As such, they seldom take advantage of the new possibilities that many-core architectures offer.

We propose Proximity Coherence, a scheme in which L1 load misses are optimistically forwarded to nearby caches via new dedicated links rather than always being indirected via a directory structure. Such an optimization is made possible by the comparable cost of local cache accesses with the use of on-chip network resources. Coherency is maintained using lightweight graph structures embedded in the L1 caches. We compare our Proximity Coherence protocol to an existing directory-based MESI protocol using full-system simulations of a 32 core system. Our extension lowers the latency of L1 cache load misses by up to 32% while reducing the bytes transferred on the global on-chip interconnect by up to 19% for a range of parallel benchmarks. Employing Proximity Coherence provides execution time improvements of up to 13%, reduces cache hierarchy energy consumption by up to 30% and delivers a more efficient solution to the challenge of coherence in chip multiprocessors.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*cache memories*;
C.1.2 [**Computer Systems Organization**]: Multiprocessors—*interconnection architectures*

General Terms

Design, Performance

Keywords

Proximity Coherence, CMP, cache design, network-on-chip

PACT'10, September 11–15, 2010, Vienna, Austria.

1. INTRODUCTION

To effectively utilize the increasing number of transistors available in modern fabrication technologies, the semiconductor industry is moving to many-core architectures [2, 20, 23]. Such architectures provide better scalability than monolithic single core superscalar architectures. While a many-core processor behaves much like a multi-node system implemented on a single chip, important differences exist: storage available on-chip is much more restricted, while communication latencies are considerably lower. Furthermore, the close proximity of processing and storage elements allows for optimizations that were previously unattractive in a multi-node system. Many-core processors are unconstrained by the packaging and interconnect latencies of larger multi-node machines, suggesting many possible architectural advances.

This paper investigates Proximity Coherence, a protocol in which the private caches of neighboring cores are probed upon a cache miss. Instead of immediately sending a message to the directory, a core first asks neighboring caches for a copy of the required line. The core sends a request to the directory only if all neighboring caches reply that they do not have a copy of the data. Implementing this scheme in a multi-node system would be impractical, as the latencies to snoop another cache would be of the same magnitude as going immediately to the directory. Moreover, in the case that no neighboring cache can provide the data, the request must still be sent to the directory, drastically increasing the service time.

However, in a many-core system, the communication costs are different. Messages can be carried between neighboring cores using dedicated point-to-point links, minimizing both latency and energy costs. The overhead of probing a neighboring cache then becomes a matter of a few cycles. This delay is insignificant compared to the service time of a request that is routed to and serviced by a directory.

In this paper, we present a novel extension to a standard MESI cache protocol [18] that implements the snooping mechanism described and provides lower cache miss latencies. We introduce the concept of a proximity cache hit, where data is provided by a neighboring cache without involving the directory. Additionally, we propose the use of lightweight graph structure embedded into the private cache lines to maintain coherence despite the lack of global knowledge at the directory. All Proximity Coherence messages are carried to neighboring cores on new, dedicated, point-to-point links – an implementation made possible by the close proximity of processing elements and the abundance of wires available in many-core architectures.

Our results show that we are able to reduce the latency of load misses by up to 33%, and 17% on average, resulting in overall execution time improvements of up to 13% for a significant subset of benchmarks. In addition to providing performance benefits, Prox-

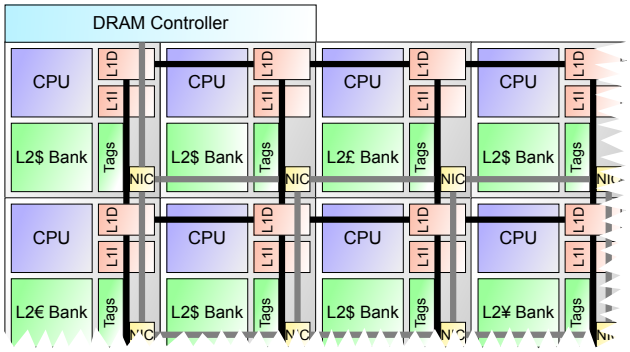


Figure 1: Top left corner of a tiled many-core processor. Gray connections show the global on-chip interconnect. Black connections show the proximity links that link the L1D caches.

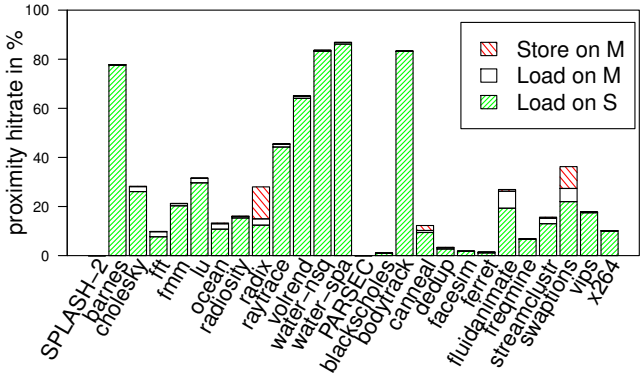


Figure 2: Results of the oracle study to investigate the limits of Proximity Coherence in a 32 core system. Whenever an L1 cache miss occurs, we check all L1s for data that could be forwarded.

imity Coherence also reduces network-on-chip traffic by 19% and cache hierarchy energy consumption by up to 30%.

2. INITIAL STUDY AND MOTIVATION

Proximity Coherence exploits the principle that data may be available in other private caches in the system upon a miss in a processor's local private cache. We have carried out experiments using the Splash-2 [24] and Parsec [5] benchmark suites to gauge the benefits of Proximity Coherence.

In our studies, we use a tiled many-core architecture. Figure 1 shows a corner of the processor, composed of 32 processing tiles arranged on an 8x4 grid. Each tile consists of a processing core, a private L1 cache, a single bank of the interleaved, shared L2 cache, and a network interface that connects the tile to the global on-chip network. Four memory controllers are placed in the corners of the chip. The L2 cache contains a directory that uses a MESI protocol to maintain coherence across all private L1 caches in the system.

2.1 Proximity Hits

When a memory access misses in the cache of a traditional chip-multiprocessor, the request is forwarded to a directory structure. In some cases, the data is already present in a different private cache in the system. The baseline MESI protocol deals with this scenario in one of two ways, depending on whether a private cache has exclusive ownership (states **E** or **M**) of the line. In the first case, providing that no private cache has exclusive ownership for that line, the data is returned from the L2 to the original requester. In the second case, the directory sends a request to the exclusive cache, instructing it to send the data to the requesting cache. In both situations, it is possible to bypass the indirection to the directory and ask private caches already containing the line to provide the data immediately.

We propose a scheme in which cache lines are requested directly from other private caches without contacting the directory, avoiding the aforementioned indirection. We refer to this process as *snooping* another private cache. A situation where a processor misses in its local private cache but receives at least one copy of the requested data directly from another private cache is declared a *proximity hit*. There are three possible ways in which this can occur:

Load on S The requester performs a load operation and snoops a cache that has the data available in state **S**hared. The data can be forwarded to the requester.

Load on M The requester performs a load operation and snoops a cache that has the data available in state **E**xclusive or **M**odified. The data can be forwarded to the requester. However, in order to maintain coherency, the snooped cache can no longer write to its cache line without invalidating the requester's copy first.

Store on M The requester performs a store operation and snoops a cache that has the data available in state **E**xclusive or **M**odified. The data and write permissions can be forwarded to the requester. However, the snooped cache can no longer read or write its cache line without getting an up-to-date copy back first.

To evaluate the potential of this technique, we record memory access traces of all programs in the Splash-2 [24] and Parsec [5] benchmark suite using the Virtutech Simics [15] full-system simulator. This study focuses on memory accesses caused by the benchmark program. Hence, we remove all memory accesses caused by the OS or other processes. We pass the traces through a functional simulator of a MESI cache coherence protocol.

Whenever a processor encounters a cache miss the simulator probes all other L1 caches to find a copy suitable for forwarding. Figure 2 shows that all programs exhibit at least some degree of sharing that can be exploited by Proximity Coherence, with many showing considerable potential. For example, *barnes*, *bodytrack*, *volrend*, *water-nsquared* and *water-spatial* all exhibit proximity hit rates between 65% and 87%. Across all benchmarks, *Load on S* and *Load on M* events cover over 95% of all proximity hits. *Store on M* events occur infrequently, as shared data is almost always read before it is overwritten. *Radix* and *swaptions* are the only exceptions, exhibiting a significant fraction of *Store on M* events due to false sharing. In light of these results, our implementation of Proximity Coherence supports only *Load on S* and *Load on M* forwarding. However, it is necessary to check every other cache in the system after each local cache miss to achieve such hit rates.

2.2 Snoop Width

Due to constraints on wiring resources and limited cache ports, any implementation of Proximity Coherence must select a sub-set of processors in which to snoop for data. We refer to the size of this subset as the *snoop width*.

To investigate the effect of limiting the snoop width, we perform a two phase study. As expected, reducing the snoop width will always reduce the proximity hit rate. For this reason, we only include benchmarks that exhibit proximity hit rates of at least 20%

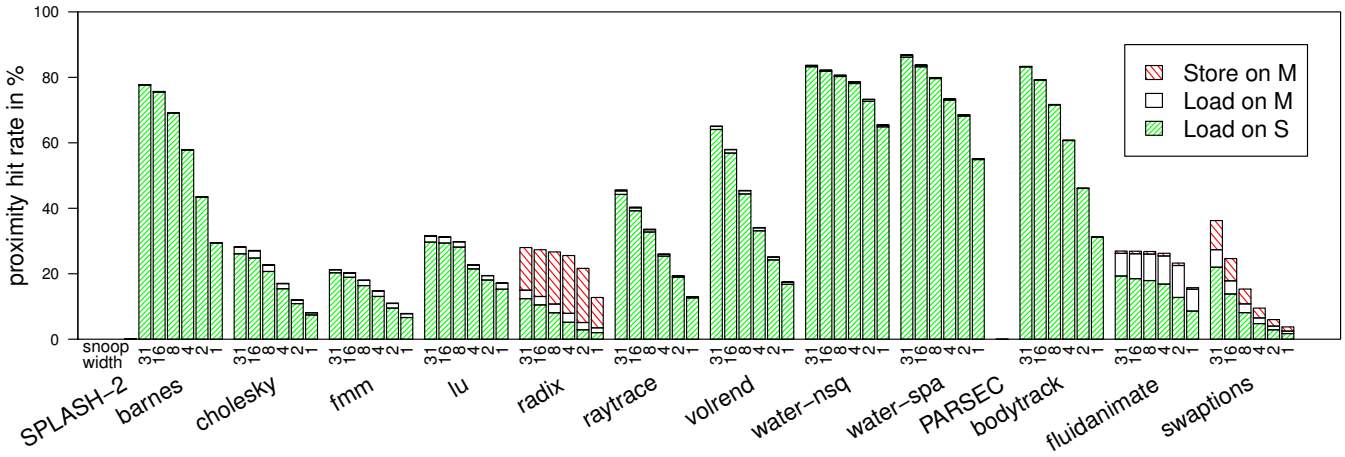


Figure 3: Impact on the proximity hit rate, when the number of cores snooped is reduced. In this study, we limit snooping to the n best neighbors – a good neighbor is a core that is more likely to be able to forward data to the requester.

in the previous oracle study. Using results from the experiments in which all caches are snooped, we generate ordered lists of "preferred neighbors" for each core in the system; a preferred neighbor is a cache that is more likely to return a proximity hit when snooped. We use these lists in the second phase of the study to determine proximity hit rates when snooping only the first 1, 2, 4, 8, 16 or 31 caches, as shown in figure 3.

Due to the tiled layout of our baseline architecture (see figure 1), we are most interested in the proximity hit rate when snooping only four neighboring caches. Our results show that, even with this reduced snoop width, it is possible to capture the majority of all proximity hits. In some cases, snooping just four caches captures up to 90% of all possible hits. For the selection of benchmarks shown in figure 3, the average proximity hit rate is 37%. This suggests that the parallel benchmarks examined have stable sets in which data is shared, allowing for good proximity hit performance through the use of correct thread mappings and network topologies.

2.3 Concurrent Proximity Requests

The forwarding of cache misses to adjacent processors increases the strain placed on the read ports of private caches. Although the probability of generating a proximity message requiring read port access is low, a single cache could be expected to serve up to the four concurrent requests from adjacent tiles. Our trace analysis shows that the likelihood of this is extremely low – averaged across all benchmarks, 99.39% of proximity messages encountered no contention from others. 0.6% of messages encountered contention from a single concurrent message, with three-way and four-way contention making up the final 0.01%. Such a low probability of contention permits the reuse of existing cache read ports and a simple arbitration mechanism with no fear of degrading performance through the stalling of proximity messages.

2.4 Energy Considerations

Research in the network-on-chip field [22] has shown that the energy cost of network routers will inevitably comprise a significant portion of total system demand. As a consequence, schemes that reduce network hop traversals are becoming increasingly attractive. Additionally, advanced on-chip router energy consumption is now comparable to an L1 cache access [4, 14, 21]. Importantly, this validates the use of the additional L1 cache accesses generated by Proximity Coherence to reduce network utilization. If a sufficient number of proximity hits are delivered, we can reduce memory access latency and lower energy consumption.

2.5 Summary

Increasing communication costs and the demand for high performance data sharing motivates the extension of existing cache coherency protocols to exploit the physical locality of shared data. The results generated from our functional cache coherence simulator show that a significant proportion (11 of 24) of benchmarks can benefit from Proximity Coherence extensions.

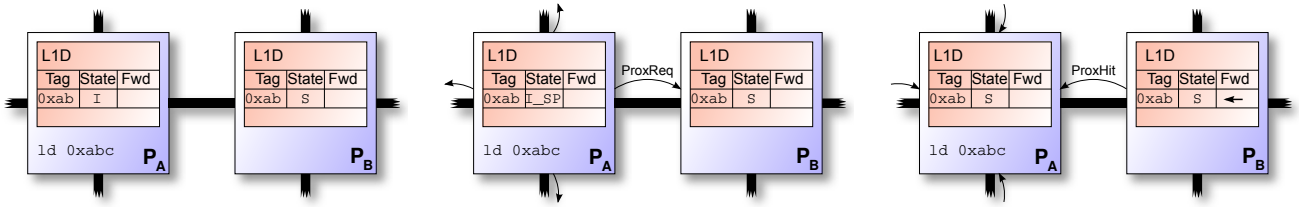
3. PROXIMITY COHERENCE

Proximity Coherence is built around the concept that a core snoops its four neighboring caches before sending a request to the directory. We refer to this request as a *proximity-request*. If a snooped cache can provide the data, it performs a cache-to-cache transfer to the requester and marks it as forwarded. If any neighboring caches supply the requested data, then we classify the original cache miss as a *proximity-hit*. These cache-to-cache transfers use novel point-to-point links between neighboring cores, rather than the packet switched, global on-chip network. Due to the critical nature of proximity requests from adjacent nodes, we prioritize them when arbitrating for cache read ports.

Forwarding data in this way presents design challenges, as the directory is not aware of the additional sharers. In order to maintain coherence, modifying the cache coherency protocol is necessary to provide the following mechanisms:

- When an L1 cache replaces a cache line that has been forwarded, it sends an L1_UPDATE_S (Update Sharer) message to the directory. The message contains a list of the cores to which the replacing cache has forwarded the data. To avoid incoherent data being held in the system, it is necessary for the directory to acknowledge this message. A similar mechanism is already used in the baseline MESI protocol when an L1 cache evicts a dirty cache line.

  Due to silent evictions of shared data, it is possible that the L1_UPDATE_S message will contain cores that no longer hold a copy of the data. This is not an issue, as the MESI baseline protocol dictates that invalidates received for non-present data are immediately acknowledged.



(a) P_A performs a load operation, which misses in its L1 cache. P_B has a copy of this data in its local cache with read permissions.

(b) Instead of contacting the directory, P_A sends PROXREQ messages to neighboring cores. These messages are sent using direct point-to-point links.

(c) P_B can supply the data to P_A and replies with a PROXHIT message. In addition, it records that it has forwarded the data. P_A obtains the data through a *proximity hit*.

Figure 4: Example of a *proximity hit*.

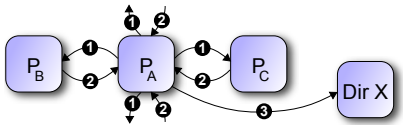


Figure 5: Example of a *proximity miss*. P_A misses in its local cache and sends out 4 PROXREQ messages to its neighboring cores (step ❶). Since none of these tiles can provide the data, they all respond with a PROXMISS message each. We call this situation a *proximity miss* (step ❷). P_A now sends a GETS message to the directory in order to request the data (step ❸).

- When an L1 cache receives an INVALIDATE message it is necessary to propagate this message to any cores to which it has forwarded the cache line. After the cache has received all acknowledgements, it can acknowledge the original INVALIDATE message. As the propagated messages (PROXINV) can only be sent to neighboring cores, they are sent using the same direct links as proximity-requests.

- If a core requires exclusive access to a cache line that it has already forwarded, all forwarded copies must be invalidated and an UPGRADE message sent to the directory. These events can be performed in parallel, speeding up the invalidation process.

3.1 An Example of Proximity Coherence

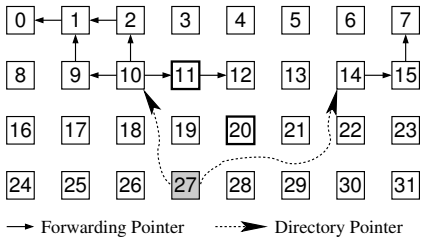
Figure 4 shows the detailed behavior of the Proximity Coherence protocol when a load operation misses in an L1 cache. P_A issues a load to address `0xabc`, but the corresponding line `0xab` is not valid in its cache. P_B has a valid copy of this line in state **S**hared (figure 4a). Instead of sending a request to the directory, P_A sends out four *Prox Requests* to its neighboring cores and moves the line into a transient state (figure 4b), which indicates that the cache is awaiting replies from all proximity requests. Since P_B has a valid copy of line `0xab`, it replies by sending a PROXHIT message containing the data and marks the cache line as forwarded to its left neighboring core (figure 4c). The requesting core will write the data that arrives first to its private cache. As there is no acknowledgement of a proximity hit from the requester, every core that provided the data will mark its cache line as forwarded. Hence, for a single address, several cores can point to a single requester.

Figure 5 shows the actions taken if a load operation does not hit in any of the neighboring caches. As before, P_A sends out proximity requests to its neighboring caches (step ❶). As none of the caches contains a copy of the data that can be forwarded to P_A, they all respond with a PROXMISS message (step ❷). After P_A has collected all replies, it sends a GETS message to the directory responsible for this cache block (step ❸).

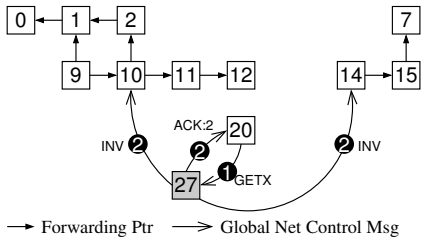
3.2 Invalidations

As figure 4 shows, any cache that forwards data to another core records this action in the forwarded vector for that line. This process can occur several times, forming an acyclic *forwarding graph*, as figure 6a illustrates. As a cache must hold a line to be the source of a forwarding pointer, it is impossible to form a cycle in the graph. Cores P10 and P14 originally have received their data from the directory (located for this particular address in core P27). Core P10 has forwarded the data to cores P2, P9 and P11, while core P14 has forwarded it to core P15. These cores in turn have forwarded the data to other cores, as indicated by the forwarded arrows. When core P1 has requested the data, both cores P2 and P9 return a copy. Therefore, both cores hold a record that they forwarded the data to core P1. As the directory has sent the data only to cores P10 and P14, it holds only a pointer to these cores. For this reason, on an invalidation, it is necessary to follow the forwarded links in order to reach and invalidate all copies of the data. The following paragraphs present examples of the two types of invalidations found in Proximity Coherence:

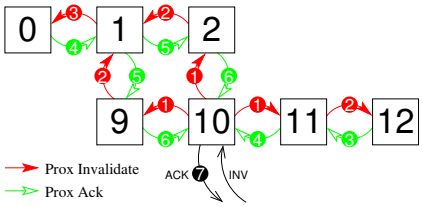
External Invalidations occur when a core, which is not part of the forwarding graph, needs to modify shared data. In figure 6b, core P20 requires exclusive access to a cache line. As in a normal MESI directory protocol, core P20 sends a GETX message to the directory (step ❶). The directory responds by sending invalidates to the two sharers it has knowledge of (cores P10 and P14) and in parallel notifies core P20 that it should wait for two acknowledgements (step ❷). The protocol now diverges from the standard MESI behavior. Before cores P10 and P14 can reply with an acknowledgement, they have to invalidate the cores to which they have forwarded the data. Figure 6c shows how core P10 invalidates these cores (core P14 acts in a similar way, but for simplicity we focus on core P10). Core P10 sends PROXINV messages to cores P2, P9 and P11 (step ❶). Since these cores also forwarded the data, they too must send PROXINV messages (step ❷). A special case is core P1, since it received data from both core P2 and P9. As such, P1 will possibly receive two PROXINV messages before it receives confirmation from core P0, to which it forwarded the data. In order to remember the cores to which PROXINV messages were sent, the function of the forwarding vector is changed; instead of keeping track of to whom the cache line has been forwarded, it keeps now track which cores send a PROXINV message. When the end of the forwarding chain is reached, the final core replies with a PROXACK message (see core P0 in step ❹). This in turn causes the previous core in the chain to generate a PROXACK message. Once all PROXACK messages have been collected by core P10, it



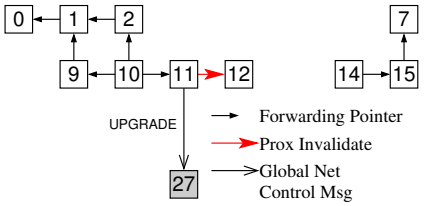
(a) Initial situation: the directory is aware of 2 sharers, P10 and P14. P10 is the root of the left half of the forwarding graph, while P14 is the root of the right half. Tile P20 and P11 are emphasized since they will start the process that will lead to the invalidation of all sharers.



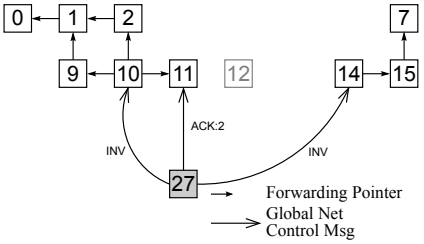
(b) P20 sends a GETX message to the directory in order to gain exclusive ownership of the cache line. The directory responds by sending INV messages to the known sharer and an ACK:2 message to P20. All these messages are sent over the global on-chip network.



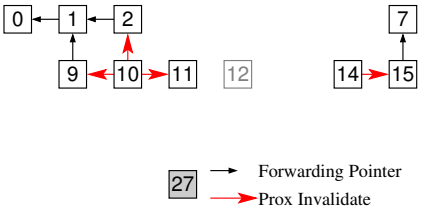
(c) Invalidation of the left part of the forwarding graph. The PROXINV and PROXACK messages are sent over the direct links between neighboring cores.



(d) P11, which is part of the sharer graph, sends an UPGRADE message to the directory in order to gain exclusive ownership of the cache line. Since it also has forwarded the data to its right neighboring core, it also sends a PROXINV message to this core – proactively invalidating it.



(e) By the time the directory processes the UPGRADE request, core P12 has already invalidated its copy of the data and is shown in gray. The directory responds by sending INV messages to the known sharer and ACK:2 message to P11.



(f) After core P10 and P14 have received the INV messages, they send PROXINV messages to the cores they have forwarded the data to. Since P11 is the originator of the request, it ignores the PROXINV by acknowledging it without invalidating its copy of the data.

Figure 6: Example of external (b – c) and internal (d – f) invalidations. For this cache line, the directory is located in core P27, indicated by solid gray shading.

sends an ACK message over the global on-chip network to the new exclusive owner of the cache line (step ❼). The remaining actions are identical to those in a standard MESI protocol.

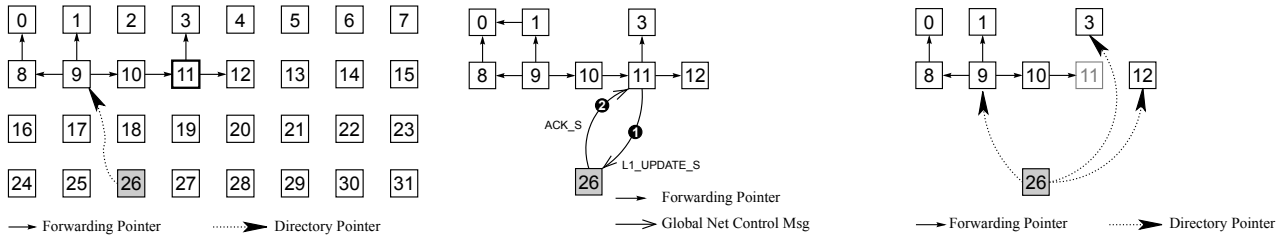
Internal Invalidations occur when a core, which is part of the sharer graph, needs to modify shared data, such as core P11 in figure 6d. Core P11 sends a UPGRADE message to the directory to request exclusive access to the cache line. Since P11 also has forwarded the cache line to other cores, it sends PROXINV messages to these cores. For simplicity, we show a situation in which the PROXINV messages are acknowledged before the GETX message is processed by the directory. However, this is not a requirement of the protocol; the events are allowed occur in any order. The directory responds in the standard manner sending out two INV messages and one ACK message that tells core P11 how many sharers there were (see figure 6e). Once core P10 and P14 have received the invalidate messages, they send out PROXINV messages to cores P2, P9, P11 and P15. As such, P11 will receive an invalidate message, even though it originated the request. To prevent P11 from invalidating itself, the PROXINV messages must contain a field identifying the original requester. Therefore, if a core receives a PROXINV message for which it is the originator, it can ignore the message and reply with a PROXACK.

3.3 L1 Cache Replacements

At any time during the life of a forwarding graph, a participating cache can evict its data. If the protocol were to behave as a standard MESI protocol and perform a silent eviction, the graph would be irreparably broken. To prevent this, we modify the mechanics of an L1 replacement:

- If the cache has not forwarded the data to any other core, it behaves as in the standard MESI protocol and simply replaces the cache line, without informing the directory. If it later receives a PROXINV message for the replaced address from any neighboring cores or an INV message from the directory, it acknowledges the message.
- If the core has forwarded the data, then it must inform the directory of the other sharers before it can replace the cache line. This action is similar to an L1 cache trying to replace a cache line that contains dirty data: before the cache line can be replaced, it has to be written back to the L2 and the directory has to be informed. We use the same simple mechanism. This mechanism also deals with cases when, during an L1 replacement, another L1 tries to gain exclusive access to the data and wins the arbitration at the directory.

Figure 7 illustrates such a scenario. The starting situation is shown in figure 7a. The directory in core P26 is only aware that core P9 has a copy of the data, while core P11 wants to perform a replacement, having forwarded the data to cores P3 and P12. P11 sends an UPDATE_S (Update Sharer) message to the directory (step ❶). Upon receiving this message, the directory adds the sharers contained to its sharer vector and sends an ACK_S message back to core P11 (step ❷). To prevent protocol races against external invalidations, P11 must retain the sharer information for the



(a) Initial situation: the directory is aware of 1 sharer, P9. P9 is the root of the forwarding graph. Core P11 has forwarded the cache line to core P3 and P12. It now wants to replace the cache line.

(b) In order not to break the forwarding graph, it sends an L1_UPDATE_S message to the directory. The directory adds the sharer contained in this message to its sharer vector and acknowledge the receipt with an ACK_S message.

(c) Final situation: core P11 has invalidated its copy of the cache line, shown in gray. The directory is now aware that core P3 and P12 have a copy of the data and holds a direct pointer to them.

Figure 7: Example of an L1 replacement in case of forwarded data. For this cache line, the directory is located in core P26, indicated by solid gray shading.

cache line until the ACK_S message is received. Figure 7c shows the situation after the replacement: the directory is now aware that cores P3, P9 and P12 have a copy of that cache line. Core P10 maintains a forwarded pointer set towards core P11, but this has minimal impact; P11 simply receives a PROXINV message for the replaced address.

3.4 Forwarding from Modified and Exclusive

In addition to supporting *Load on S* forwarding described so far, Proximity Coherence also allows data to be forwarded from a line that is held with exclusive permissions. Forwarding is supported from the **M**odified and **E**xclusive states of the baseline MESI protocol.

When a proximity-request is received for a cache line held in the **M** or **E** states, the data is returned as a proximity-hit and the cache line is moved immediately to a new **F**orwarded state. This **F** state indicates to the forwarding cache that the line's permissions have been downgraded, without the directory's knowledge, to read-only access. A processor holding a line in the **F** state is responsible for any copies it forwarded on. This means that, should the core receive an invalidate, it must invalidate all copies of the data forwarded to adjacent processors.

Supporting forwarding in this way is important, as when a line is first loaded into the system it arrives with exclusive permissions in the requesting private cache. Hence, without the addition of *Load on M* forwarding, the first proximity-request is guaranteed to miss, creating unnecessary traffic to the directory.

In a situation similar to the one described in section 3.3, the forwarding graph is broken into two parts. As such, the replacing cache sends an L1_UPDATE_S message to the directory. For this reason, maintaining the read-sharers vector in the directory state machine is necessary even when the line is believed to be held with exclusive access in a private L1 cache. No extra storage is required to support this extension. In the special case that the replacing cache is the root of the forwarding graph, a message is returned to the directory containing both the forwarding vector and, if the line is dirty, the data. Again, the directory state machine is augmented to allow for such messages to be processed.

Before a processor can write to a cache line that is held in the **F** state, the processor must reacquire exclusive access. This is achieved by invalidating the forwarded read-access copies of the data using proximity invalidates, and in parallel, sending an UPGRADE request to the directory. When all forwarded copies are invalidated and confirmation is received from the directory, the cache line returns to **M**odified and the write completes.

3.5 Hardware Costs

Implementing Proximity Coherence incurs only a small hardware overhead. In contrast to similar works [10, 11], no additional complexity is required in either the processor or network routers. First, we need additional wires for the point-to-point links that are used for the proximity requests. These wires are, on average, the length of one tile and do not require deep buffering. Flow control is provided by a simple not-ready wire applying backpressure. Moreover, there are a large number of such wires available in modern fabrication processes [1, 9], particularly in wiring channels between tiles. Second, each cache line needs additional bits to store to where the cache line has been forwarded. In this particular implementation of the scheme, data can be forwarded to any of the four neighboring cores requiring an additional 4 bits per cache line. This increases the stored information in each L1 cache by less than 1%, assuming 64 byte cache lines with 51 bit tags. Finally, as discussed in section 2.3, it is not necessary to increase the number of cache read ports. All new structures employed in Proximity Coherence are distributed and will scale well to larger core counts without incuring additional hardware overheads.

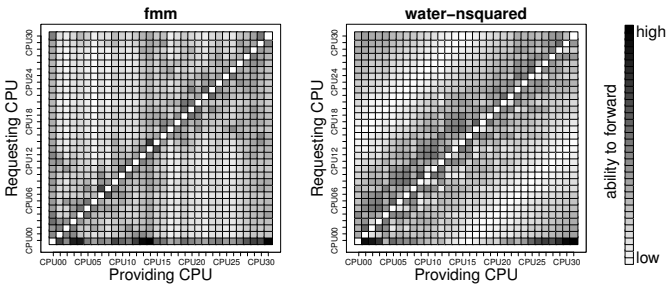
4. EVALUATION SETUP

To evaluate the performance benefits of Proximity Coherence, we implement a cycle accurate version of the protocol. The systems and methodology used are described below.

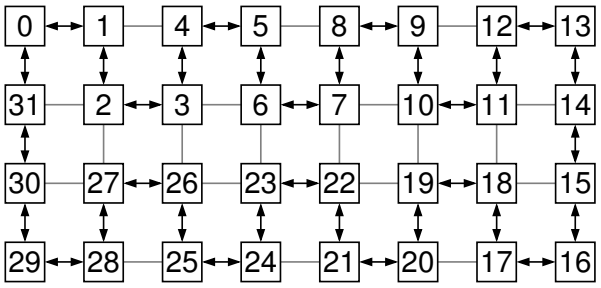
4.1 Simulation Parameters

For our full-system simulation we used Virtutech Simics [15] and the Wisconsin GEMS tool set [16]. These tools provide full OS support and a customizable memory model. Using the GEMS SLICC language, we define the extended state machine with all transient states and the necessary storage additions to hold forwarding vectors for each cache line. We have thoroughly stress-tested the protocol using the supplied SLICC protocol tester, to check for race conditions and consistency violations. We have augmented the existing GEMS network model with fast point-to-point links between neighboring tiles, as described in section 3.5.

Table 1 lists the parameters of the simulated system. These parameters are in line with recently proposed industrial architectures, such as Intel's Larrabee processor [20]. In order to account for variability in simulating a multi-threaded workload on a full-system



(a) Preferred neighbour analysis for *fmm* and *water-nsq*.



(b) H-tree thread placement.

Figure 8: Thread mapping considerations: (a) shows the best neighbor lists for *fmm* and *water-nsquared*. A darker color indicates that this core is more likely to be able to forward data to the requesting core. We notice a dark region around the diagonal, which resulted in the approximate thread placement strategy shown in (b).

Processors	32 Sparc V9 cores, 3 GHz, single-issue, in-order, non-memory IPC = 1
OS	Solaris 9
L1 cache	32 kB per core, split I/D, 4 way associative, 2 cycles latency, 64 byte lines
L2 cache	8 MB, 32 banks interleaved, 8 way associative, 16 cycles latency, 64 byte lines
Memory	1GB, 4 banks, 250 cycles latency
Directory	L1 tag replication, 32 banks interleaved, MESI protocol
Network	8x4 mesh topology, 2-cycle routers, 1-cycle link latency, 36 bytes wide
Prox-Links	1-cycle link latency, 36 bytes wide, single-depth buffers

Table 1: Parameters used in our full-system simulation to evaluate Proximity Coherence.

simulator, we randomize the memory access latency slightly for each data point as described by Alameldeen and Wood [3] and run each benchmark many times to produce results with sufficient confidence. Error bars showing standard deviation are included where applicable.

4.2 Benchmark Selection

In figure 3, we show a selection of Splash-2 [24] and Parsec [5] benchmarks that exhibit behaviors that warrant further investigation. For full-system simulation, we eliminate benchmarks that exhibit less than 10% proximity hit rate for the chosen snoop width of 4, excluding *swaptions*. However, *ocean*, a benchmark with a low proximity hit rate of 13%, exemplifies the behavior of Proximity Coherence with less favorable programs.

To capture all temporal phase behavior we run the entire parallel phase of each benchmark. For Splash-2, we use the recommended input size and for Parsec, the *simmedium* input set.

We found that simulation times for Parsec benchmarks are orders of magnitude greater than for Splash-2, preventing the inclusion of results for *bodytrack* and *fluidanimate*.

4.3 Thread Mapping

The results presented in section 2.2 provide us with lists of preferred neighbors for each core. These lists define an ideal thread placement for each benchmark. We find these ideal mappings impossible to achieve when mapping to a 2-D mesh topology. Computing the optimal 2-D mesh mappings would place additional strain on the compiler or runtime environment; instead, we choose a suitable approximation.

Figure 8a shows the preferred neighbor lists for two of our chosen benchmarks. We observe that any core i is likely to find cores $i-1$ and $i+1$ high in its preferred neighbor list. This leads us to the use of the H-tree thread mapping shown in figure 8b. Our trace-driven analysis shows that if a random mapping were to be used, an average of 35% of proximity hits would be lost, confirming the use of this simple, fixed scheme.

5. EXPERIMENTAL RESULTS

In this section, we evaluate Proximity Coherence in detail. We measure high proximity hit rates for our selection of benchmarks, in line with the predicted values. As a direct consequence, our scheme provides considerable improvements in memory access latency, which in turn improves overall program execution time. We confirm that in delivering these benefits, Proximity Coherence does not impose unrealistic demands on network resources. In fact, the system reduces the energy requirements of the cache hierarchy, creating a faster and more efficient coherency protocol.

We evaluate three versions of Proximity Coherence, one implementing only *Load on S* sharing (referred to as *Prox*) and the second also providing support for *Load on E/M* sharing (referred to as *ProxF*). The third version, used to evaluate the impact of the point-to-point links, is a modified implementation of the *ProxF* protocol, where neighboring caches are snooped via the global on-chip network (referred to as *ProxF-N*).

5.1 Impact on Memory Latency

Figure 9 shows the effects of Proximity Coherence on L1 load and store miss latencies. We observe that *Prox* achieves load latency reduction of up to 32% and 14% on average. *ProxF* provides further improvements, lowering load miss latency by an additional 2.3% on average, resulting in a maximum reduction of 33% in the case of *fmm*. Improvements are obtained by avoiding unnecessary indirections to the directory, as discussed in section 3. *ProxF-N* also benefits from physical locality of shared data, but due to latencies introduced by unnecessary router traversals, we see diminished gains.

When using Proximity Coherence, store miss latencies can be marginally increased. The worst degradation in latency occurs in *cholesky* – an increase of 2.4% – due to the serialization of invalidations in forwarding graphs. A standard directory protocol is able to send invalidations to every sharer in parallel. In Proximity Coherence, however, some sharers can only be reached through the traversal of the forwarding graph, causing the observed increase in latency.

However, on average the *ProxF* scheme improves store miss latency by 1.4%, due to *ProxF* more efficiently supporting the reacquirement of write permissions. This is particularly important

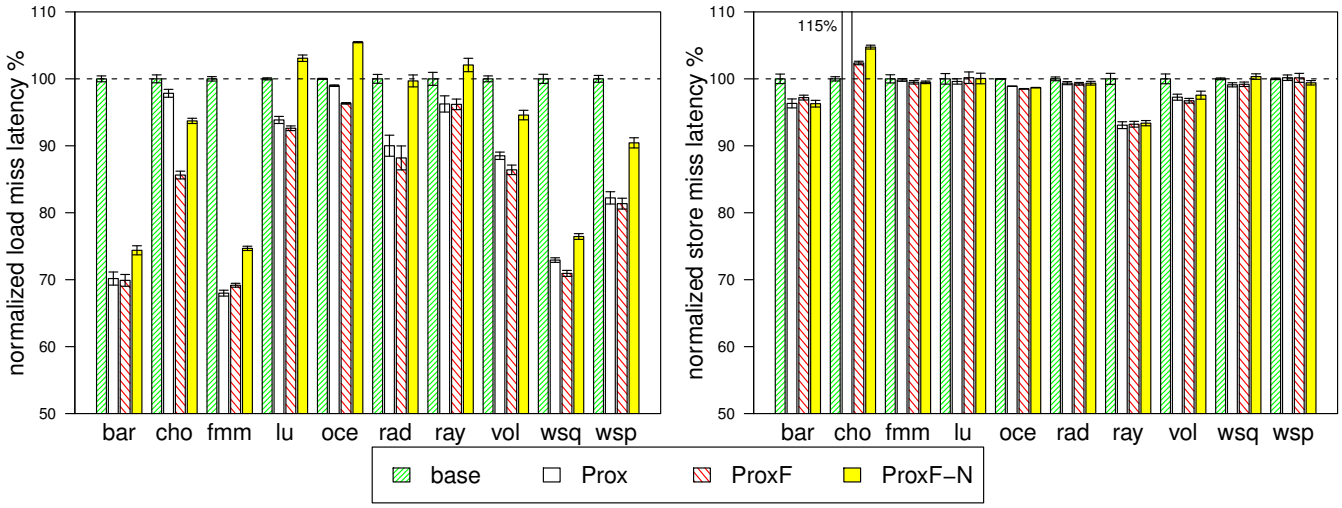


Figure 9: Cache miss latency reduction in % compared to a system using the MESI baseline protocol.

in producer-consumer relationships, a common data sharing pattern. For example, should a cache line be held in state F, the core can normally re-obtain write permission with a 2-hop transaction, as described in section 3.4. In *Prox* however, where no F state is implemented, a 3-hop transaction is required.

5.2 Invalidation Chain Length

In addition to the fixed overhead of checking adjacent caches, Proximity Coherence serializes invalidations within the forwarding graphs of shared data. If a forwarding graph is deep, an invalidation request will take many cycles to propagate to the end of each branch, causing slow state transitions. In order for Proximity Coherence to provide good performance the depth of any forwarding graphs frequently invalidated must be low. Figure 10 shows the depth of invalidations encountered when using the *ProxF* scheme.

The graphs invalidated most frequently are of depth one, showing that data was forwarded only once before being invalidated. Over 98% of proximity invalidations are of depth less than or equal to 2. This minimizes the serialization penalty and ensures good invalidation performance for data shared through proximity hits.

5.3 Proximity Hit Rate

Figure 11 shows the measured proximity hit rates for both *Prox* and *ProxF*. For *ProxF*, we distinguish between *Load on S* and *Load on M* hits.

Our implementations of Proximity Coherence achieve hit rates of up to 54%, enabling the latency improvements already described. Our results show that in almost all cases, the measured proximity hit rate is close to our predicted value presented in section 2.2. This is especially interesting, as the expected hit rates have been generated using an ideal thread placement, while the measured results use only an approximate placement, as described in section 4. Additional variation is introduced through operating system interference. *Radix* is especially affected, as it is a particularly short running benchmark: a significantly higher proximity hit rate is observed in the full-system simulation results than the predicted value.

When examining the *ProxF* results, we notice that for each benchmark, *Load on M* forwarding provides only a small proportion of proximity hits, on average 2.6% and, excluding *cholesky*, just 1.4%. However, this small improvement means that more sharers

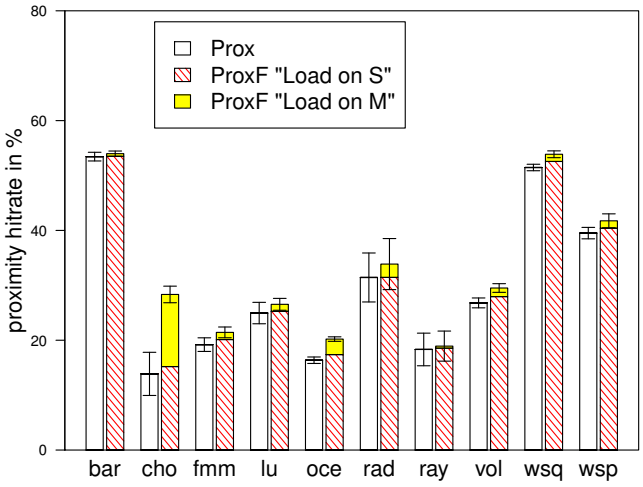


Figure 11: Measured proximity hit rates for *Prox* and *ProxF*.

are available in the system sooner and these sharers can offer data via *Load on S* forwarding, as reflected by the increased *Load on S* events for *ProxF*. This behavior improves average proximity hit rate by an additional 3.3%. These two effects combined deliver higher than expected latency benefits, as shown in figure 9. *ProxF* increases latency reduction by up to 7%, justifying the additional complexity.

5.4 Execution Time Improvements

Figure 12 shows the overall execution time improvements Proximity Coherence provides. The *ProxF* scheme delivers benefits of up to 13% with only *ocean* suffering a slight slow down. *ocean* was included as an example of a program with a low proximity hit rate, leading to a marginal execution time increase of 1%. Importantly however, network traffic and energy consumption are still reduced. *ProxF-N* cannot match these improvements and for six benchmarks delivers worse runtime results than the baseline system.

Although Proximity Coherence is an effective optimization, its impact on execution time is limited by the high L1 cache hit rates

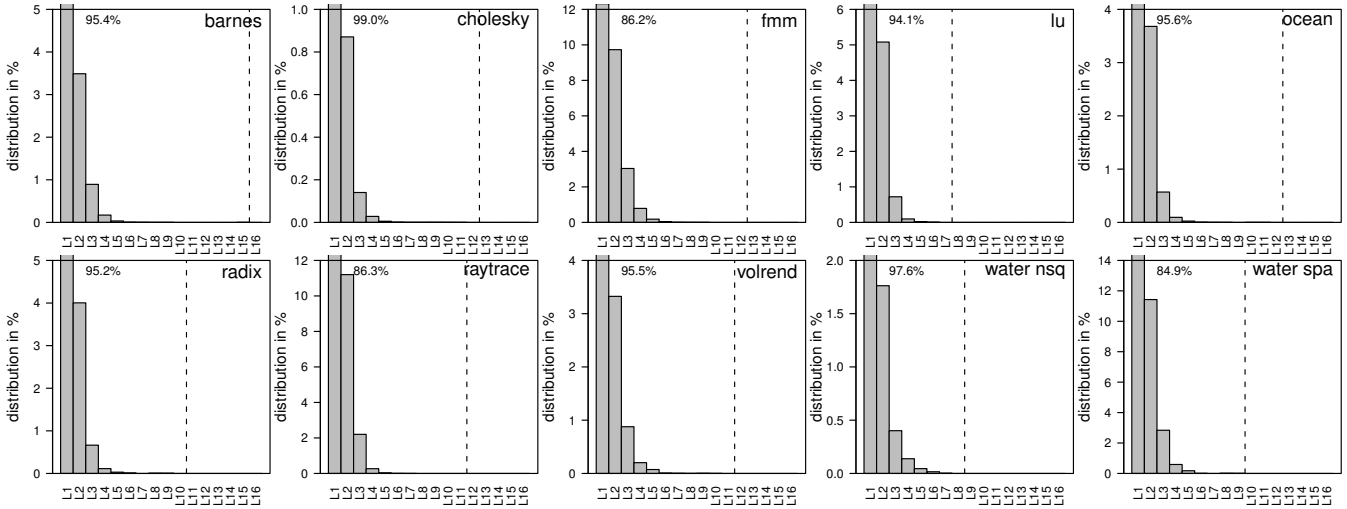


Figure 10: Distribution of the depth of the sharer graph at the time of an invalidation request, when using the *ProxF* scheme. The graph has in most cases only a depth of 1, resulting in negligible overhead. The vertical dashed line indicates the maximum depth observed in that program.

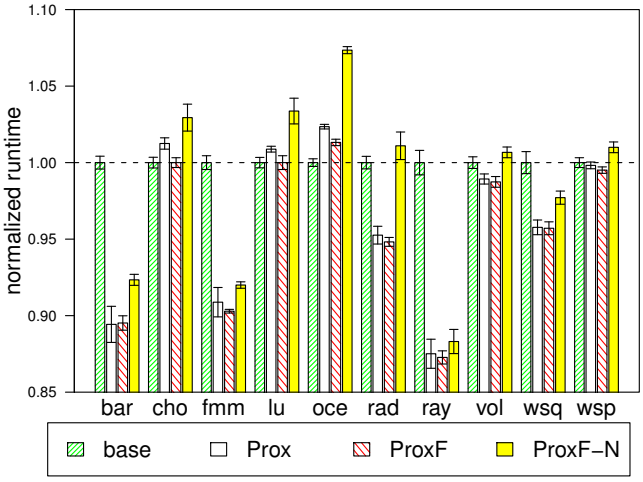


Figure 12: Runtime reduction compared to a system using the MESI baseline protocol.

observed in our chosen benchmarks. The data forwarding mechanisms of the protocol are only exercised during L1 cache misses.

5.5 Impact on Network Traffic

As Proximity Coherence optimizes the communication in many-core systems, analyzing its impact on on-chip network traffic is important. In this study, we distinguish between two types of traffic: proximity messages that are carried on the new dedicated links described in section 3.5 and standard messages that use the global on-chip interconnect. We make this distinction as the two networks have significantly different characteristics.

Figure 13 shows the aggregate number of bytes transferred a single hop by on-chip network. Since all proximity messages travel on only one point-to-point link to reach their destination, they have a fixed hop count of 1. However, global network-on-chip messages may have to travel through several routers to reach their destination.

Over all benchmarks, Proximity Coherence achieves a reduction in global network-on-chip data transferred of between 8% and 42%. In *Prox* and *ProxF*, cache misses that would have been serviced using the global network are satisfied using the proximity network.

As discussed, *ProxF* provides several benefits over the simpler *Prox*. However, our network analysis shows that these improvements create no increase in proximity link traffic. This is as expected, since "Load on M" forwarding effectively turns control traffic (negative reply to a proximity request) into data traffic (positive reply). The number of requests sent and replies received remains constant.

The *ProxF-N* scheme succeeds in reducing the amount of data traffic, however, as control messages to neighboring cores still need to traverse two routers, the total control traffic increases to the point that it negates the savings made by reduced data traffic. For all benchmarks, *ProxF-N* generates more traffic than the baseline system, highlighting the importance of the new proximity links when implementing Proximity Coherence.

5.6 Impact on Energy

To confirm that Proximity Coherence is feasible to implement, we estimate the energy consumed in the two networks and the energy required for snooping the four neighboring caches. For this study, we make three assumptions. First, we assume that network energy consumed is proportional to the amount of data transferred. Work by Banerjee et al. [4] shows that, with effective clock-gating, this is the case. In Proximity Coherence, data messages are approximately nine times larger than control messages. As such, we assume that they consume nine times more energy. Second, we assume that when transferring a message, the energy consumed in a router is four times that which is consumed in the link. This assumption is based upon work presented by Kundu [14]. As the proximity network is composed of simple point-to-point links with no routers, we assume that the energy required to send a single proximity message is equal to the amount consumed by a global network link. Finally, as discussed in section 2.4, we assume that the energy required for a single L1 cache lookup is equivalent to the amount consumed by a router processing one message. For simplicity, we do not consider the energy saved by not performing an L2 lookup after a proximity hit.

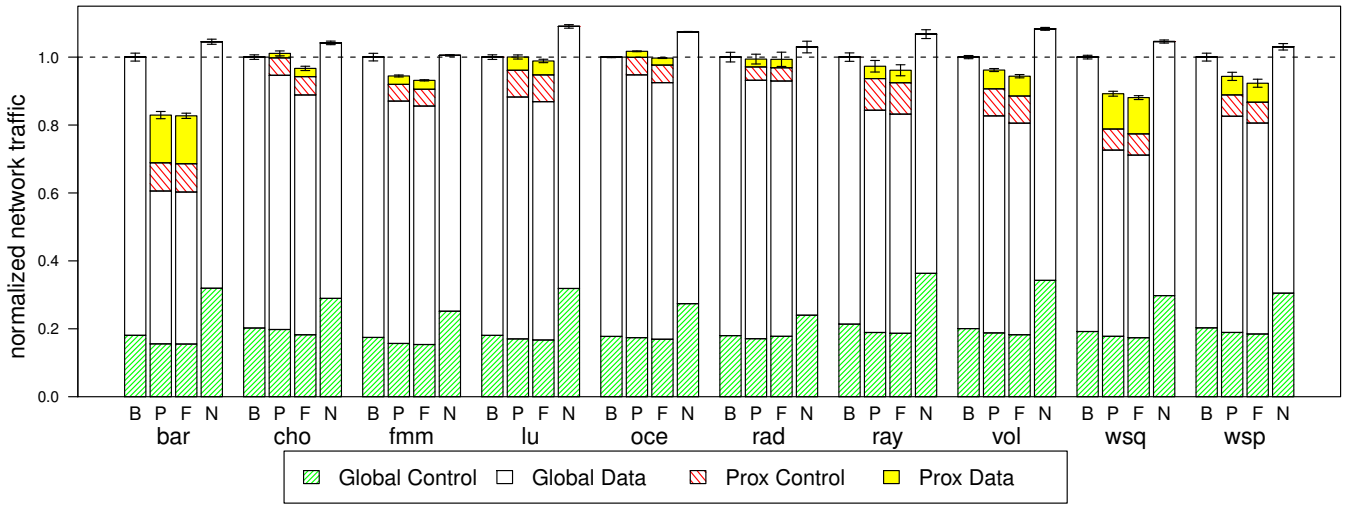


Figure 13: Normalized network traffic compared to a system using the MESI baseline protocol. "B" refers to the baseline system, "P" refers to *Prox*, "F" refers to *ProxF*, and "N" referes to *ProxF-N*.

Figure 14 shows the total network energy consumption under the discussed assumptions. We also show the energy overhead associated with snooping caches. When using the baseline MESI protocol, we see that only 19% of energy is spent on control messages, despite their greater contribution to overall network traffic. Using either of the Proximity Coherence implementations that employ proximity links results in a reduction of between 5% and 30% in total network energy. Importantly, the reduced consumption in the global network is not nearly matched by the energy spent in the proximity links. Moreover, we see that the total network energy saved more than offsets the additional expense of lookups in neighboring caches. We find that as *ProxF-N* only uses the global on-chip interconnect, its energy requirements are up to 55% higher than *ProxF* (24% on average), further motivating the inclusion of proximity links in architectures implementing Proximity Coherence. A more detailed analysis is left to future work.

6. RELATED WORK

To the best of our knowledge, we are the first to suggest the use of dedicated wires to snoop neighboring caches in a many-core processor. However, prior work exists that attempts to exploit proximity in a chip multiprocessor or considers the special properties of chip multiprocessors as opposed to multi-node systems.

Cheng et al. [8] optimize the energy demand of the on-chip interconnect by providing different networks for different coherence message types. Unlike our scheme, they do not explore the new opportunities of a many-core design and focus solely on optimizing the on-chip network for an existing cache coherence protocol.

Brown et al. [6] describe an augmentation to the coherence mechanism that takes into account the proximity of available sharers when the directory serves an L1 cache miss and cannot provide a copy from its L2 cache bank. Unlike our work, Brown's scheme does not avoid the extra hop to the directory and cannot utilize an inexpensive point-to-point network that provides a copy from a neighboring sharer. Furthermore, the proposed changes are orthogonal to our scheme and combining both schemes may prove beneficial.

Eisley et al. [10] propose a coherence mechanism that is directly embedded into the interconnection network routers. The mechanism works by constructing tree structures in the network routers that redirect requests to the directory towards a nearby sharer, if the request happens to traverse a node that is part of the tree. However, depending on the routing, it is completely possible that the request will miss an adjacent sharer and will proceed across the network. Our scheme will always probe neighboring tiles and is guaranteed to find adjacent copies. Furthermore, this scheme increases the processing time of the router, dealing with both routing and coherence protocol tasks. Additionally this work does not present execution time statistics, which prevents any direct comparison of performance.

Enright Jerger et al. [11] propose a protocol that uses a tree structure to maintain coherence across several sharers. The root of the tree acts as an ordering point for requests. While their scheme uses a coarse-grained coherence mechanism, we maintain coherence at cache line granularity. In addition, their scheme also results in an increase of global network traffic by a factor two to three over a standard directory protocol, drastically reducing the efficiency of the proposed scheme. Proximity Coherence delivers improved performance and reduces energy consumption.

Hossain et al. [12] present a scheme where an L1 cache also sends a request to a neighboring cache instead of sending a request to the directory. However, since they use the global on-chip network for such requests, rather than our novel dedicated links, their definition of neighboring is a more relaxed "close-by" instead of adjacent. Furthermore, while the data is provided by this "close-by" cache, the directory functions are not delegated to this cache. Instead, the directory is immediately informed and the provided data can only be used once the directory has acknowledged the forwarding. The main performance gain in their system comes from control messages having a lower latency than data messages. In our system, we assume a global network that delivers data and control messages with the same latency. Additionally, our work models state of the art router latencies [4]. As detailed in Hossain's work, using such a low latency network reduces the benefits gained through their scheme. Finally, we delegate coherence responsibility to the L1 that forwarded the data, such that the data is usable immediately; an acknowledgement from the directory is not needed.

Cache coherence protocols have been proposed that uses linked-lists to track sharers in a multi-processor system [13, 17]. Although sharers are also tracked using pointers in Proximity Coherence, our

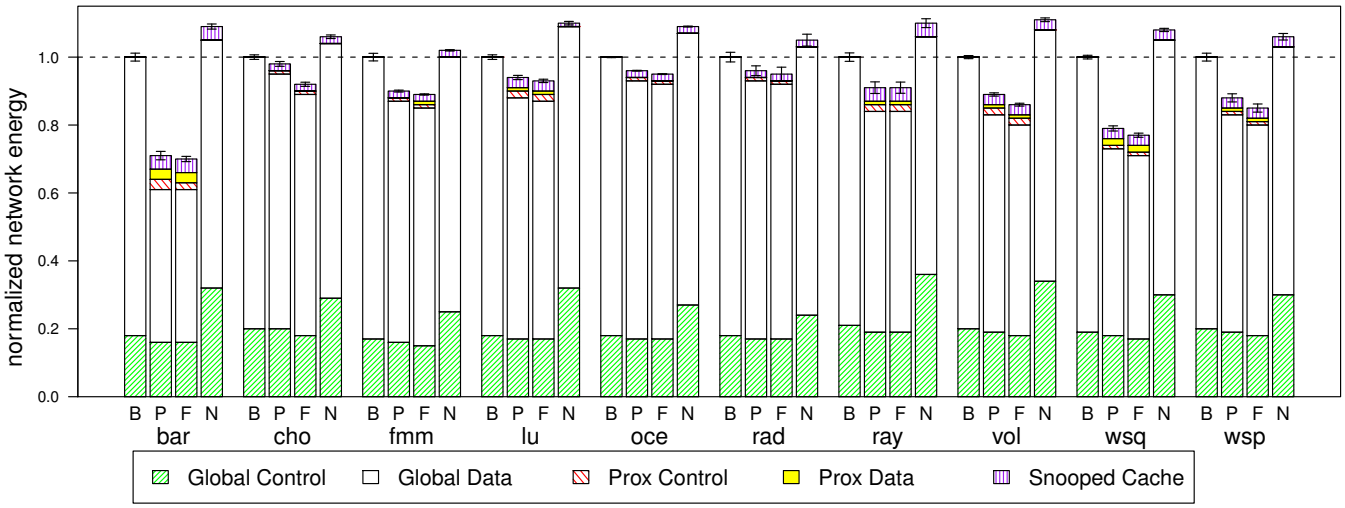


Figure 14: Normalized estimated network energy consumption compared to a system using the MESI baseline protocol. "B" refers to the baseline system, "P" refers to *Prox*, "F" refers to *ProxF*, and "N" referes to *ProxF-N*. In addition, we show the energy required to perform a cache lookup in the case of a servicing a proximity request.

scheme differs significantly: it tracks sharers in an acyclic graph and takes physical locality information into account. Further differences are found due to the proximity-link network introduced by our work.

Cheng et al. [7] propose a scheme that delegates directory responsibilities to other nodes in the system. The goal is to transform 3-hop transitions into 2-hop transitions. However, their design is optimized for a multi-node system and unlike our system, the delegations only happen after a stable producer-consumer relationship has been detected. Our scheme uses an optimistic mechanism and establishes delegation immediately.

Ros et al. [19] propose a cache coherence protocol for tiled CMPs. Similar to the work by Cheng et al., this scheme aims to avoid long latency 3-hop transitions by delegating the directory responsible to the owner node. While the protocol considers the limited storage requirements in a CMP system, it does not take advantage of the opportunities offered by the low latency on-chip interconnect. Implementing this scheme in in a multi-node system may obtain similar improvements.

7. CONCLUSION

In this work, we present Proximity Coherence, a novel protocol that exploits the physical locality of shared data to provide an efficient cache coherence in many-core architectures. Our design delivers a 14% reduction in L1 load miss latency, while reducing global on-chip network traffic by 19%. For our selection of benchmarks, we achieve execution time improvements of up 13%. Proximity Coherence effectively trades off network traffic and latency against additional L1 cache accesses, while simultaneously reducing energy consumed by the memory hierarchy.

Benefits emerge through the use of new dedicated links between neighboring cores. Using these links, data is optimistically requested from adjacent cores. Coherence is then maintained through delegation of responsibility, from the directory to caches that have forwarded data. An implementation without these links is not feasible, as using the global on-chip interconnect increases the energy required by the network by 24% and reduces the obtainable latency improvements. Additionally, it is impossible to run the baseline MESI protocol transactions over the simple proximity links – such messages require more comprehensive routing, flow control, and buffering. Furthermore, the resources required to form proximity links are so minimal, that reassigning their use to further increase the global-network bandwidth is not possible – increasing bandwidth requires larger crossbars and associated datapaths, not just more wires.

Looking forward, Proximity Coherence presents many opportunities for additional research. First, reducing the number of unsuccessful cache snoops by using dynamic prediction may be possible. We are also interested in implementing an OS-based scheme to disable Proximity Coherence in situations it is either not required or has detrimental effects on performance. Such a scheme would require simply changing a single state transition, disabling snooping. Second, we are interested in the challenge of implementing Proximity Coherence on a strictly non-inclusive cache hierarchy that maximizes on-chip storage utilization. Third, we believe that a processor architecture employing chip-stacking would allow for a greater number of proximity-links to be added, further improving the chances of delivering a proximity hit. Finally, we believe that restructuring the benchmark algorithms can increase the physical locality of shared data, improving the proximity hit rate. In such a scheme, Proximity Coherence would provide efficient support for message-passing style communication between physically local cores, while still supporting a fallback of a fully coherent shared-memory. Optimizing communication then becomes an optional performance layer, offering an interesting new platform for software and hardware engineers alike.

8. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and the Computer Architecture Group at Cambridge University for their constructive comments and suggestions. In particular, we would like to thank Arnab Banerjee and Robert Mullins for their insightful discussions, Christophe Dubach and Timothy Jones for providing valuable feedback, and Kate Aufses for her assistance editing the final paper. This research was supported in part by the Engineering and Physical Sciences Research Council (EPSRC). This work has made use of the resources provided by the Edinburgh Compute and

Data Facility (http://www.ecdf.ed.ac.uk/). This facility is partially supported by the eDIKT initiative (http://www.edikt.org.uk).

9. REFERENCES

[1] International Technology Roadmap for Semiconductors. `http://public.itrs.net/`.

[2] A. Agarwal, L. Bao, J. Brown, B. Edwards, M. Mattina, C.-C. Miao, C. Ramey, and D. Wentzlaff. Tile Processor: Embedded Multicore for Networking and Multimedia. In *Hot Chips 19*, Aug. 2007.

[3] A. R. Alameldeen and D. A. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proceedings of HPCA 9*, pages 7–18, Feb. 2003.

[4] A. Banerjee, P. T. Wolkotte, R. D. Mullins, S. W. Moore, and G. J. Smit. An Energy and Performance Exploration of Network-on-Chip Architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(3):319–329, Mar. 2009.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of PACT 17*, pages 72–81, Oct. 2008.

[6] J. A. Brown, R. Kumar, and D. Tullsen. Proximity-Aware Directory-based Coherence for Multi-Core Processor Architectures. In *Proceedings of SPAA 19*, pages 126–134, June 2007.

[7] L. Cheng, J. B. Carter, and D. Dai. An Adaptive Cache Coherence Protocol Optimized for Producer-Consumer Sharing. In *Proceedings of HPCA 13*, pages 328–339, Feb. 2007.

[8] L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J. B. Carter. Interconnect-Aware Coherence Protocols for Chip Multiprocessors. In *Proceedings of ISCA 33*, pages 339–351, June 2006.

[9] W. J. Dally and B. Towles. Route Packets, Not Wires: On-Chip Inteconnection Networks. In *Proceedings of the 38th annual Design Automation Conference (DAC)*, pages 684–689, June 2001.

[10] N. Eisley, L.-S. Peh, and L. Shang. In-Network Cache Coherence. In *Proceedings of MICRO 39*, pages 321–332, Dec. 2006.

[11] N. D. Enright Jerger, L.-S. Peh, and M. H. Lipasti. Virtual Tree Coherence: Leveraging Regions and In-network Multicast Trees for Scalable Cache Coherence. In *Proceedings of MICRO 41*, pages 35–46, Nov. 2008.

[12] H. Hossain, S. Dwarkadas, and M. C. Huang. Improving Support for Locality and Fine-Grain Sharing in Chip Multiprocessors. In *Proceedings of PACT 17*, pages 155–165, Oct. 2008.

[13] D. V. James, A. T. Laundrie, S. Gjessing, and G. Sohi. Distributed-directory scheme: Scalable Coherent Interface. *IEEE Computer*, 23(6):74–77, 1990.

[14] P. Kundu. On-Die Interconnects for next generation CMPs. Presentation at Workshop on On- and Off-Chip Interconnection Networks for Multicore Systems (`http://www.ece.ucdavis.edu/~ocin06/program.html`), Dec. 2006.

[15] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[16] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, Nov. 2005.

[17] A. Nowatzyk, G. Aybay, M. C. Browne, E. J. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Processing*, volume 1, pages 1–10, Aug. 1995.

[18] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of ISCA 11*, pages 348–354, June 1984.

[19] A. Ros, M. E. Acacio, and J. M. García. DiCo-CMP: Efficient Cache Coherency in Tiled CMP Architectures. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, Apr. 2008.

[20] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Trans. Graph.*, 27(3):1–15, Aug. 2008.

[21] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20, HP Labs, 2008.

[22] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, Jan. 2008.

[23] S. Vangali, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyerl, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskotel, and N. Borkarl. An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*, pages 98–99, 589, Feb. 2007.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of ISCA 22*, pages 24–36, June 1995.