electronics

MDPI

Article

Effective On-Chip Communication for Message Passing Programs on Multi-Core Processors

Joonmoo Huh [1] and Deokwoo Lee [2,*]

[1] Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695, USA; herjmoo@gmail.com
[2] Department of Computer Engineering, Keimyung University, Daegu 42601, Korea
* Correspondence: dwoolee@kmu.ac.kr; Tel.: +82-53-580-5268

Abstract: Shared memory is the most popular parallel programming model for multi-core processors, while message passing is generally used for large distributed machines. However, as the number of cores on a chip increases, the relative merits of shared memory versus message passing change, and we argue that message passing becomes a viable, high performing, and parallel programming model. To demonstrate this hypothesis, we compare a shared memory architecture with a new message passing architecture on a suite of applications tuned for each system independently. Perhaps surprisingly, the fundamental behaviors of the applications studied in this work, when optimized for both models, are very similar to each other, and both could execute efficiently on multicore architectures despite many implementations being different from each other. Furthermore, if hardware is tuned to support message passing by supporting bulk message transfer and the elimination of unnecessary coherence overheads, and if effective support is available for global operations, then some applications would perform much better on a message passing architecture. Leveraging our insights, we design a message passing architecture that supports both memory-to-memory and cache-to-cache messaging in hardware. With the new architecture, message passing is able to outperform its shared memory counterparts on many of the applications due to the unique advantages of the message passing hardware as compared to cache coherence. In the best case, message passing achieves up to a 34% increase in speed over its shared memory counterpart, and it achieves an average 10% increase in speed. In the worst case, message passing is slowed down in two applications—CG (conjugate gradient) and FT (Fourier transform)—because it could not perform well on the unique data sharing patterns as its counterpart of shared memory. Overall, our analysis demonstrates the importance of considering message passing as a high performing and hardware-supported programming model on future multicore architectures.

Keywords: message passing; scalable; on-chip communication; hardware design

Citation: Huh, J.; Lee, D. Effective On-Chip Communication for Message Passing Programs on Multi-Core Processors. *Electronics* **2021**, *10*, 2681. https://doi.org/10.3390/electronics10212681

Academic Editors: Juan M. Corchado, Stefanos Kollias and Javid Taheri

Received: 4 October 2021
Accepted: 1 November 2021
Published: 3 November 2021

1. Introduction

From programming perspectives, multicore is essentially synonymous with shared memory. Multicore programmers are free to stitch and sew programs together out of threads, locks, barriers, flags, and a wide variety of programming constructions with shared memory. The ubiquity of the assumption that shared memory equals multicore would lead one to conclude that message passing is not a reasonable choice for programming with these architectures. However, few studies have rigorously investigated this assumption since the dawn of modern multi-core processors. Message passing paradigms such as MPI (message passing interface) have been around for decades and provide great convenience for distributed systems because nodes are connected over networks, and furthermore, MPI abstracts communication to simpler messages. In the case of multi-core processors, however, the message passing abstraction has not solved a critical need, at least not yet, and even causes a disturbance. One reason for this is that message passing

frameworks such as MPI support a robust middleware library for communicating over a distributed network, but many of the features that make it compelling for distributed systems simply add an overhead on multi-core processors due to the additional copying of data. This large overhead can be easily seen and measured when an MPI library, such as OpenMPI or MPICH, runs on a single chip multiprocessor. However, when we eliminate these overheads, a different picture emerges. Rather than being a drag on performance, messaging provides a critical abstraction that can boost multicore performance: messages precisely describe how and when to move data around the chip to support the task at hand. This information can significantly improve the utilization of the cache hierarchy and on-chip interconnection. At the same time, as the number of cores on a chip scale increases, the internal architecture of the multi-core processors of a shared memory are more complicated, and hidden costs in the form of complex cache hierarchies and coherence protocols ultimately play a key role in system performance and power efficiency. After all, as the number of cores increases, coherence is responsible for managing data that could be spread over growing on-chip distances with unpredictable latencies. Since many of these behaviors are intentionally hidden from programmers' perspectives, optimizing such systems can be difficult. As an example of how this affects implementation, consider the optimization of synchronization primitives: in order to run faster, their code often becomes longer, not to mention the increased difficulty of determining its correctness [1].

Message passing may be a compelling alternative to some programs that require the fine-tuning of the control of the cache hierarchy. In this paper, we investigate the behavior of message passing on a conventional multicore architecture. In particular, we are interested in the requirements and/or advantages that may arise if message passing is designed as a first-class hardware-supported programming model. Unlike recent work that aims to design a multicore architecture to support MPI in a larger distributed system [2–4], we want to support message passing efficiently on a single chip. Our contributions can be stated as follows.

1. We design an infrastructure that allows us to investigate the algorithmic differences when an application is optimized for either shared memory or for message passing. NAS (network attached storage) parallel benchmarks are used since they have been designed for both MPI and OpenMP. Here, we eliminate the library and system-level overheads and focus only on the characteristics of the algorithms. Using this system, we have characterized the similarities and differences between applications optimized for both systems. We find that the patterns of data movement between the processes in a message passing architecture are similar to those for threads in shared memory architecture. In other words, there is no penalty for using explicit data movement in message passing architecture; this is not surprising when one considers that shared memory architectures tend to have implicit data movement that is carried out through the coherence protocol. We also see that the total volume of transactions on the interconnection is large with shared memory architecture since coherence messages are issued whenever the status of the shared block is updated. However, payload data movements are only issued to the interconnection when blocks need to be transferred to another core or when a block is read from/written to its private address space. Finally, we find that the performance of our shared memory architecture is more sensitive to the size of the last level of private memory hierarchy than the message passing architecture.
2. Our analysis suggests that while behaviors are similar, there are opportunities to improve message passing performance using methods that are not available on multi-core processors currently; in particular, support for the efficient bulk transfer of data between caches. As a result, we design messaging protocols that allow message passing programs to communicate efficiently either between memories or between high levels of the cache memory hierarchies. These protocols allow new capabilities in the interconnection, such as point-to-point messaging, which exploits

higher transfer rates. It also improves hit rates and increases IPC (instructions per cycle) in MPI programs.

3. We evaluate our multicore design with hardware support for message passing and show that it is competitive with a state-of-the-art shared memory multicore. In the experiments, in the best case, message passing achieves up to a 34% increase in speed over its shared memory counter-part, and it achieves a 10% increase in speed on average. The advantages of message passing come from both efficient bulk transfer and the elimination of the wasteful movement of data.
4. We also provide a characterization that offers insight into the scenarios under which message passing works better. In particular, message passing offers a greater advantage when the cache is a scarce resource. This is likely the result of the more efficient management of the cache hierarchy through explicit data movement. When private L2 caches are larger, in the 1 MB (mega bytes) to 4 MB range, message passing architecture is still better on average, but the differences of performance between the architectures decrease.

The rest of the paper is organized as follows. Section 2 covers some background and the motivation for our design; Section 3 describes the overall message passing architecture we propose; Section 4 provides the details of our design; Section 5 presents our evaluation; Section 6 covers important related work; and Section 7 concludes our article.

2. Motivation

2.1. Assumptions and Definitions

For the rest of this paper, when we refer to system architectures, we are referring to a single-node system in which all of the cores are on the same chip. This work excludes distributed systems from consideration that require a network protocol to communicate between threads/processes in the parallel computing applications. All of the communications between threads or processes remain on the chip and are brokered either through hardware or software mechanisms. We assume directory-based coherence for our shared memory chip. Focusing only on single node systems may seem narrow, but there is a growing need for high-performance embedded computer systems that support surveillance and security applications, often in support of national defense [5]. These systems could have orders of hundreds to thousands of cores in the near future. They are, in essence, supercomputer systems in an embedded system, and their performance should be high with low power consumption. The intermediate conclusion in this work is that shared memory is the best approach for programming these systems, but we argue that message passing is a competitive and important model in practice. Furthermore, MPI programs are usually a set of cooperating processes, while OpenMP programs are typically threads–processes that share address spaces [6–8]. For consistency, we refer to both MPI processes and OpenMP threads as threads of the program. This is imprecise, but it simplifies our language. Finally, for the sake of this work, message passing will refer to MPI programs. While our observations are likely more general than MPI, we need fine-tuned and optimized applications written for both shared memory and message passing, and MPI is the only suitable option. Hence, references to message passing implicitly refer to MPI-based implementations and architectures.

2.2. Shared Memory vs. Message Passing

Shared memory (SM). Shared memory provides programmers with the abstraction of a single address space through which all threads in a program can communicate. To communicate with another thread, only the location of a reserved memory is needed, and in the location, one thread can write and another one can read. Properly synchronizing those accesses is another important task, but this also can be solved using operations on shared memory. With this basic primitive of sharing a memory location, a wide variety of threading and synchronization primitives can be designed, thereby allowing programmers

to construct a wide variety of synchronization and communication mechanisms. Hence, shared memory is often touted for its intuition and flexibility.

Message passing (MP). In message passing, programmers craft their applications as separate threads with private address spaces that work on their own data sets. If necessary, they communicate data to other threads through messages. MP relies on programmers specifying messages explicitly in their programs. No communication happens between threads unless it is directed through such a message. Hence, communication and synchronization events are explicit in the code. The programmer may specify the kinds of communication, such as point-to-point or collective operations, or blocking or non-blocking communication. Table 1 gives a brief description of common MPI functions.

Why pick one over the other? Regardless of the programming model, the same basic reasoning steps are required to implement parallel code. The programmer must answer the following questions:

1. What can be done in parallel?
2. How are threads coordinated to achieve the goal?
3. How can parallelism be expressed to maximally exploit available compute resources?

Table 1. Commonly used MPI functions.

Function	Description	Variants
send(msg,dest)	Calling thread sends msg to dest thread	Blocking/nonblocking
recv(msg,src)	Receive msg from src thread	Blocking/nonblocking
wait(msg)	Wait for non-blocking message to complete	Nonblocking
all-to-all(msg,size,group)	Send part of msg to all threads in group, recv msg from all threads in group	Blocking

In general, the programming model may alter these choices in many ways, but we expect that the overall characteristics of the two approaches should be similar. The reason is that architectural trends discourage frequent communication and synchronization. Hence, if these rules are disobeyed, performance will suffer. On the other hand, if the programming model exploits synergies with the underlying architecture, then performance may be considerably improved. This observation is what we explore in this paper: *what opportunities are there for hardware to support messaging synergistically, and how can they can be supported in multi-core processors?*

2.3. Key Architectural Differences

If we ignore the implementation differences commonly associated with message passing and shared memory, we can reduce the behaviors of these programs to essential operations. In particular, we assume that, with hardware support, sending a message involves passing a copy of data from one block of memory to another. In shared memory, on the other hand, no such copies are fundamentally necessary; however, copies do occur within the cache hierarchy of multicore systems to enable efficient use in the presence of data sharing. Considering this fundamental difference, we examine the key architectural differences of a shared memory cache hierarchy and a message passing one. We identify four important differences/opportunities, which both help and hinder both models depending on how they are used, and their impact on architecture.

2.3.1. Private Data Versus Shared Data

SM: All data are assumed to be shared. In SM, all data are potentially shared, so this incurs the overheads associated with coherence. For example, on a cold miss at the last level of a private cache, a directory lookup is required before accessing memory. This is also incurred unnecessarily when private data suffer frequent capacity or conflict misses.

MP: All data are assumed to be private. In MP, all data are assumed to be private. This has the advantage that all threads can benefit from a private memory hierarchy. A cache miss can directly proceed to memory. Of course, this comes at the expense of a shared cache which is effectively larger and has more temporal locality.

Upshot: MP hardware can eliminate some of the overheads of coherence.

2.3.2. Bulk Transfer Versus Fine-Grained Per-Block Transfer

SM: Fine-grained per block transfers. Cache coherence makes it possible to share the granularity of cache blocks. In a standard protocol, the request for a block involves sending a request to the directory, which determines where the block may be found on the chip and then carries out the necessary coherence actions to deliver the block to the requester. This means that small amounts of data, such as a single byte, can be modified and moved around to other threads that need it (of course, an entire block must be moved in conventional protocols). This can occur somewhat frequently and still be efficient. For data that are only occasionally modified but otherwise read-only, this is very effective. However, when communicating large amounts of data between threads, the overheads of the communications add up and delay the transfer.

MP: Bulk transfer. In message passing, communication always happens in bulk. This means that the communication of large amounts of data can be streamlined and supported as a single request. Hence, the messaging overhead is amortized for large amounts of data. However, the enumeration of all data to send is required, and it must be packed into a single message. This may lead to an extra cost of data movement in the private memory space to pack the data serially or result in sending unneeded data for the sake of the simplicity of the algorithm. Furthermore, many small messages may be inefficient, as setting up a bulk transfer comes at a higher cost: the message must be packed with communication protocol information, such as the destination, size, and group. Furthermore, in MPI, it is necessary for the receiver to set up the receive buffer before the send can begin.

Upshot: To be competitive with SM, MP hardware should optimize message passing so that large and small messages are efficient.

2.3.3. Collective Transfer versus No Global Coordination MP: Collective Communication

MP supports collective communication primitives, like all-to-all communication. These primitives describe how data in a message should be distributed and collected to other nodes in the system. Usually, the algorithm is minimally configurable, so programmers must adapt their communication needs to fit the pattern. These primitives are very valuable for some programs, and they supply a semantic meaning to operations. In terms of hardware, they can provide a context for global cooperation and coordination to move data around a chip. This allows more efficient use of the the interconnect because communication can be managed in a way to ensure that no part of the on-chip network is ever overloaded.

No Coordination In comparison, SM has no similar capability. Programmers must build up such mechanisms from scratch or use libraries. Fortunately, unlike the MP approach which would require that a communication template be followed exactly, SM can tailor communication to a specific application. As a result, it may be more efficient than following a prescribed MP template. The effort involved to design such an approach may be quite high.

Upshot: MP hardware should support collective operations to provide a big advantage for applications that need them.

2.3.4. Explicit Overlap Versus Implicit Overlap

MP: Explicit overlap. Point-to-point communication allows the programmer to specify whether computation should continue or wait for the communication to happen. This has the direct implication that the programmer can tune the overlap explicitly. Assuming that the hardware can actually perform the messaging without interrupting the core, this

will provide a performance improvement. During the window of communication, the contents of the message must remain fixed. This places a burden on the programmer to either copy the contents of the message into a new buffer for continued use or find other work to do that does not rely on the message. However, if a copy is needed, this usually occurs on the critical path before sending the message. Hence, overlapped communication requires adjustments to the code to compensate for the overlap.

SM: Implicit overlap. In SM, the programmer has less control when communication happens. From a positive perspective, coherence on different blocks can happen simultaneously without causing concern for the programmer. Hence, overlap occurs without additional effort much of the time. However, synchronization must be added to control when threads access data. When done properly, the overlap between communication and computation can occur at any time, leading to very good performance. Furthermore, newer chips have been released equipped with transactional memory, giving new tools for programmers to deal with synchronization bottlenecks.

Upshot: MP hardware should support efficient nonblocking communication.

2.4. Benchmarks

To illustrate some of the differences and similarities, we perform a brief characterization of applications in the NAS parallel benchmark suite [9,10]. The benchmark suites contain three versions of each benchmark: serial, OMP, and MPI. The serial programing model gives the baseline of each benchmark. OMP parallelizes the sequential program using the OpenMP library. Thus, only one address space is used for the entire program. On the other hand, MPI provides multiple threads with individual address spaces. We choose 8 benchmarks from the NAS suite that provide both parallel programming models, OMP and MPI, out of 11 total benchmarks. The three other benchmarks have only the OMP or MPI versions but not both, so we exclude those.

Characteristics

First, we categorize the memory overheads incurred by different programming models into problem space, algorithmic overheads, and parallelization overheads categories. Problem space refers to the size of the data structure, on the source code level, that contains the original input data set of the application—it remains unchanged across implementations. Next, we account for algorithmic overheads; this refers to temporal or duplicated storage needed to carry out the work of each algorithm. This category is dependent only on the kinds of parallel models and not on the number of parallel threads. Lastly, there is the extra parallelization overhead required to support scaling in parallel algorithms. This overhead, if present, tends to scale linearly with thread count.

Figures 1 and 2 show the three kinds of overheads in the benchmarks with a class 5 input set. Each bar measures the total bytes of declared variables to store or manage the input set at the source code level. In the graph, the 5 bars from left to right show the serial, OMP with a single thread, OMP with 16 threads, MPI with a single thread, and MPI with 16 threads configurations for each benchmark. Note that the SP benchmark is omitted because the characteristics of BT and SP benchmarks are very similar. Otherwise, each benchmark has different characteristics.

First, we point out the general trend that MPI tends to require more memory space than its OMP counterparts. This is because the MPI uses usually extra memory to hold copies of data. Some of this is purely algorithmic to support nonblocking communication, while some of it occurs to support messaging explicitly, and this can be seen in the general trend that memory usage scales with the number of threads. EP is an exception to this trend, with the MPI showing no significant difference to the corresponding OMP version. This is because the parallel algorithm in EP generates an independent problem space for each thread. It only sends messages to collect the final results from each thread, and the amount of results is very small. For the rest of the applications, the degree to which memory usage increases varies, and the reason for the difference varies with the algorithms.

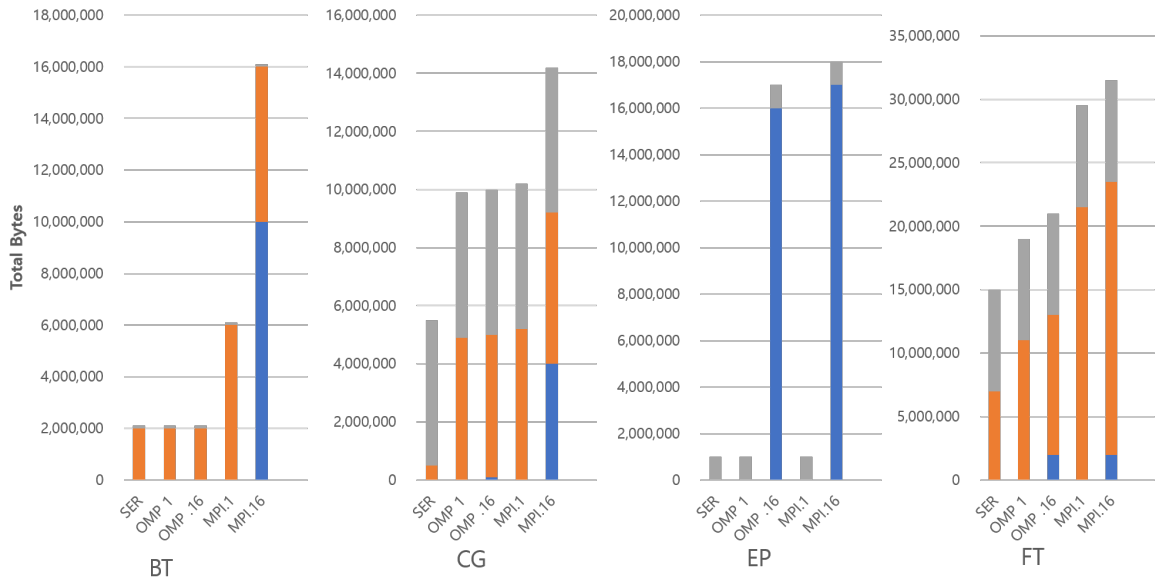

Figure 1. Categorizatio of parallel overheads in benchmarks (BT, CG, EP and FT). Note that the SP benchmark is omitted because the characteristics of BT and SP benchmarks are very similar.

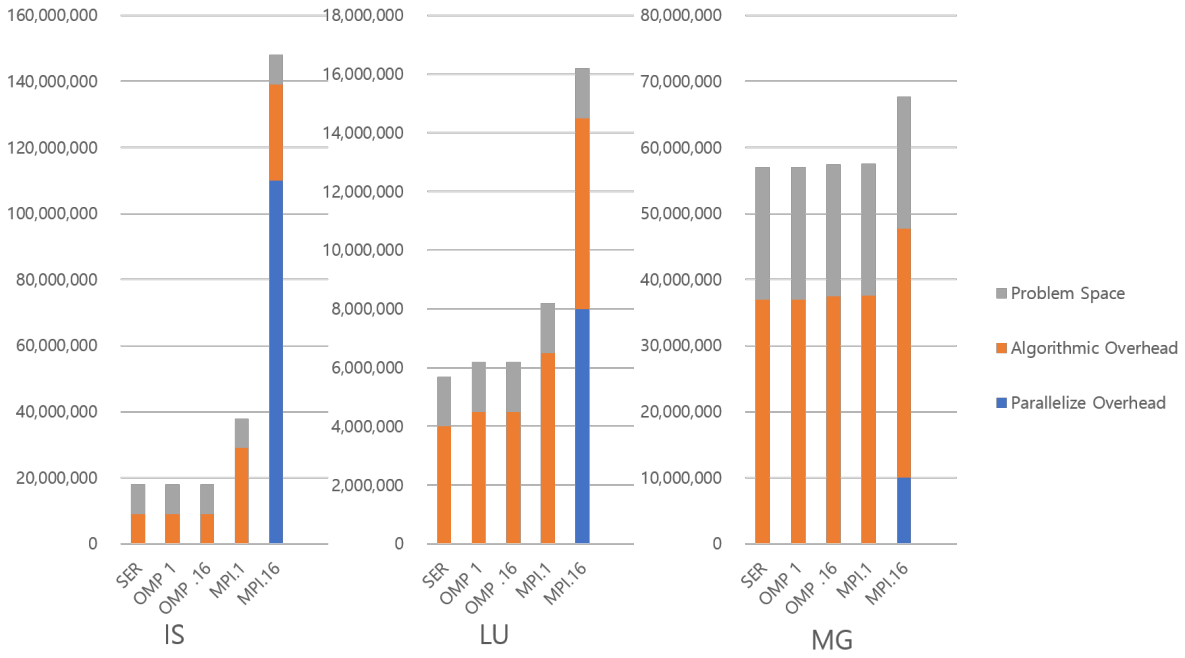

Figure 2. Categorization of parallel overheads in benchmarks (IS, LU and MG). Note that the SP benchmark is omitted because the characteristics of BT and SP benchmarks are very similar.

Figures 3 and 4 shows the total memory references in the benchmarks with the class W input set. All memory accesses from 16 threads are accumulated in OMP and MPI with 16 threads, respectively. Interestingly, even though the MP programs tend to use more memory address space, they do not show a proportionally larger number of memory accesses. This suggests that their accesses are spread over a larger amount of memory. This makes sense as each thread has its own private address space. CG has a unique characteristic that most of the memory references are reads under the SER and

OMP parallel version. Thus, the additional reads and writes on MPI are possibly used to transfer shared data between the threads. This shows the fundamental downside of MPI programming. FT has also the interesting characteristic that the large amount of additional algorithmic memory space leads to a large number of memory writes. Other benchmarks have a relatively small portion of memory writes, even though they have a large amount of algorithmic overheads, such as BT and LU. The reason is that FT performs the matrix transposition of input sets and uses the out-of-place transposition algorithm. Furthermore, in the MPI case, since the matrices are distributed to private memory address spaces, additional data movement is required prior to using all-to-all communication primitive. Thus, there are many writes that come from the data movements due to the transpositions. In most other benchmarks, the different numbers of memory accesses between SM and MP are independent of the size of memory usage. This suggests that if those accesses can be supported efficiently by the architecture, then MP programs could run as fast as—if not faster than—shared memory.

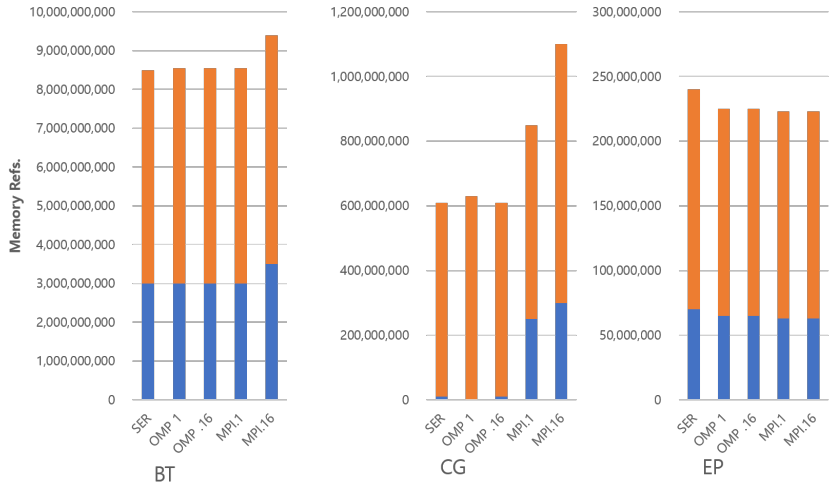

Figure 3. Number of memory references in benchmarks (BT, CG and EP).

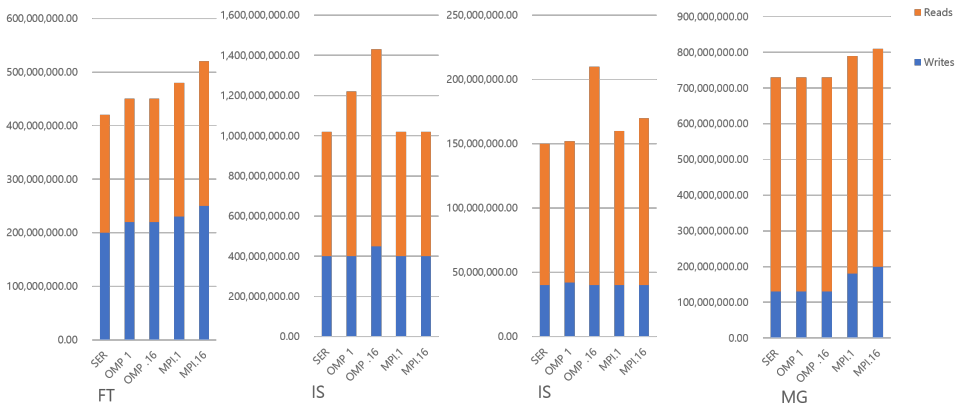

Figure 4. Number of memory references in benchmarks (FT, IS, IS and MG).

3. Message Passing Architecture

3.1. Overview

We propose a new multicore architecture with explicit support for message passing at the microarchitectural level. Our architecture needs two key modifications: (1) a software interface that supports MPI operations, (2) hardware that performs message delivery, and (3) a low-level MP Runtime Layer.

3.2. MP Runtime Layer

As part of the architecture, new coordination needs to happen between hardware and software to guarantee correct and efficient execution. So, a new MP Runtime Layer (MPRL) is needed. First, when an MP program boots up, each thread is mapped to a core. When threads communicate, the hardware must know which thread is mapped to which core. However, MP programs specify destinations in terms of thread IDs. One job of this layer is to set up a map table for thread IDs to core IDs at the beginning of the program. We assume no multiprogramming in this work, but we can envision approaches to virtualize this table. Using the map table, hardware can seamlessly convert thread IDs to core IDs. This map table can also be used to record communication groups for collective communication operations. The MPRL also helps ensure that thread IDs are arranged in a reasonable way among core IDs. For example, MPI provides templates that say how to arrange threads, and we can do the same. For example, if two threads communicate frequently, we want them to be near each other. Furthermore, we may want to ensure threads are given a unique address space that is conveniently located near the memory controller closest to the thread; this would require coordination with the OS.

3.3. The Hardware–Software Interface

We extend the hardware–software interface to support point-to-point messages and collective communication. Point-to-point communication is the most fundamental form of communication and will be directly supported in hardware. Furthermore, because point-to-point communication can be used as a primitive to construct more elaborate communication graphs, we do not require additional interface support. However, some efficiency can be gained by coordinating global communication. So, we also support all-to-all communication directly at the interface level. We do not add instructions because the number of operands needed is too large. Usually, a send or receive requires an argument that specifies a buffer to hold the message, a sender or receiver ID, and a buffer size. So, we support the operation through special memory mapped registers that can begin a message and check for its completion. If blocking communication is used, the thread will stall upon writing to the register until the communication is complete.

3.4. Messaging Architecture

3.4.1. Point-to-Point Communication

At the level of the microarchitecture, we support a point-to-point send and receive directly in hardware. When an MP thread sends a message, it specifies a buffer, a size, and a destination thread. The receiving thread also specifies a buffer, a size, and source threads. The hardware is responsible for moving data from the source buffer in one address space into the destination buffer of the other address space. To make this data movement work, hardware should manage several steps: (1) it should determine that the two threads are ready to communicate, (2) it should know the buffers that will be copied from/to, and (3) it should notify the threads of completion. We have explored two architectures to support this: memory-to-memory messaging and cache-to-cache messaging.

Approach 1: Memory-to-memory communication. We can achieve this using a simple protocol through the on-chip interconnection. First, the sender flushes its message contents to memory. Once the receiver is ready, direct transfer between the memories is initiated. Details of the protocol are shown in the next section. Memory-to-memory transfer makes sense for large messages. However, if the contents of the message are in the cache, flushing the data to memory is a waste. Similarly, if the receiver requests the contents of the message in a short period, receiving the data to memory is a waste. This leads to our other approach.

Approach 2: Cache-to-cache communication. The cache-to-cache approach uses the same basic protocol. However, instead of flushing data to memory, we first transfer data in the cache directly to the receiver. Any data that are not in the cache are then transferred

from memory. The received data are written normally to the cache in the receive side. The dirty blocks on the cache will be naturally written back to the memory.

3.4.2. Collective Communication

We support collective communication as a sequence of point-to-point messages. The architecture schedules the point-to-point operations serially in a way that balances on-chip network usage while still providing high throughput.

4. Design and Implementation

4.1. Overview

Figure 5 shows two different chip designs: a conventional SM design and an MP design. Both have the same number of cores, L1 caches, and L2 caches on a chip. All L1 and L2 caches are private caches. An on-chip interconnection is placed between the L2 caches and main memory. The left diagram shows the hardware design of shared memory while the right diagram shows the hardware design of message passing. To support shared memory, a directory-based cache coherence protocol is selected since it is scalable. We extend the MP design with our message-passing architecture. In particular, in place of a directory, we add the MP controller. This controller coordinates message passing among all threads. In the following sections, we describe how it supports three main operations: memory-to-memory messaging, cache-to-cache messaging, and collective communication. We begin with a discussion of the basic protocol messages for message passing.

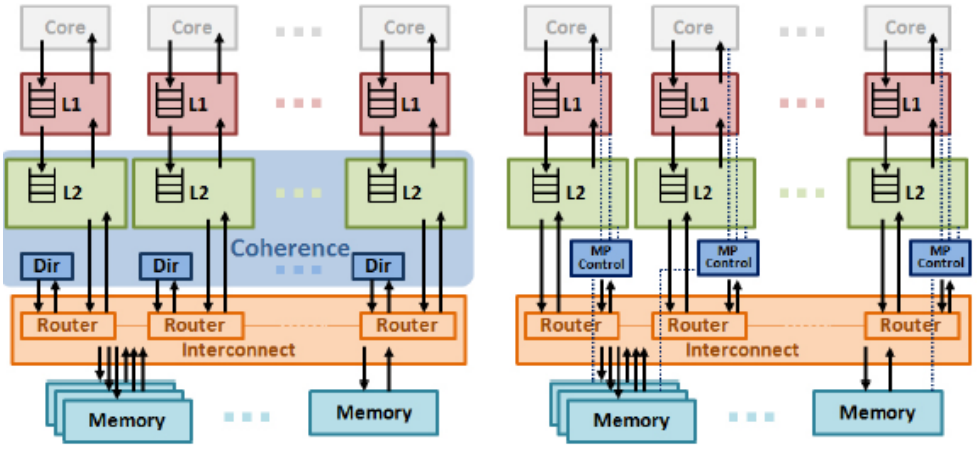

Figure 5. Hardware Design.

4.2. Basic Protocol for Message Passing

The new hardware protocol needs seven kinds of requests: `Invalidate`, `Flush`, `Writeback`, `ReadyToSynchronize`, `Transfer`, `Send` and `Receive`.

1. `Invalidate`: Invalidate the selected block in the message regardless of whether each blocks is dirty or not. This supports the receipt of messages, since the old contents of the buffer do not need to be preserved.
2. `Flush`: Invalidate the selected blocks. The blocks that are dirty will be written back to the next level of the memory hierarchy. This supports message sending.
3. `Writeback`: The replaced blocks due to the write misses will be written back to the next level of the memory hierarchy. This is the same as the approach in a conventional cache hierarchy, but it supports a writeback of multiple blocks.
4. `ReadyToSynchronize`: Synchronize the send and receive sides. The receive side issues synchronize message to the send side. The send side should wait until the synchronize message arrives.
5. `Transfer`: Transfer the selected blocks from the source buffer to the destination buffer. The request is similar to a burst copy in Direct Memory Access (DMA).

6. `Send:` (Cache-to-cache only) Read and send out the selected blocks in the designated message buffer. The missed blocks will be read from the next level of memory hierarchy.
7. `Receive:` (Cache-to-cache only) Receive the blocks and update the cache. The replaced blocks due to the write misses will be written back to the next level of memory hierarchy.

Finally, the hardware communication protocol is designed with support of the new requests above.

4.3. Memory-to-Memory Communication

4.3.1. Blocking

Memory-to-memory messaging transfers data from one memory bank to another. Figure 6 shows the set of sub-requests to support the blocking memory-to-memory transfer between two threads.

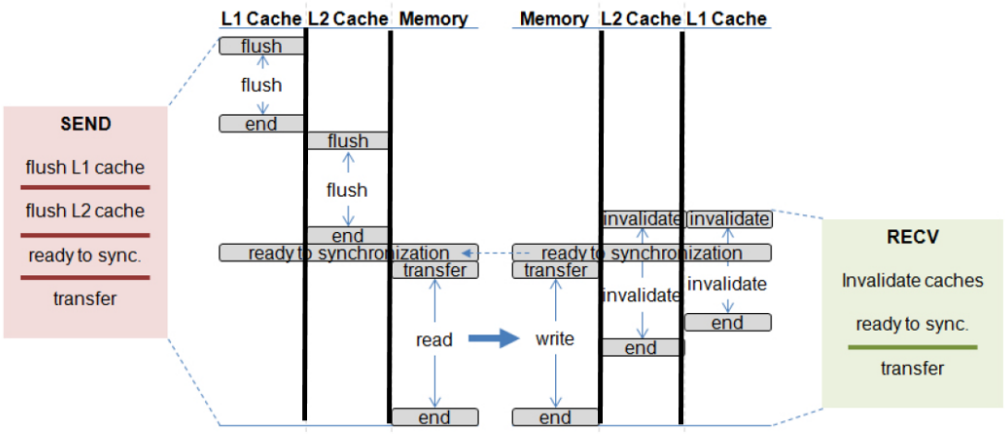

Figure 6. Blocking memory-to-memory transfer.

The red block on the left side presents the sequence of sub-requests that is required to be issued to each component on the send side. Requests that are separated by a bold line in the block should be issued sequentially. In other words, a request below a bold line cannot be issued until all requests above the line are completed. In the same way, the green block on the right side contains the sequence of requests that is required to issue to each component on receive side. The timing of activated components by the message passing request is drawn on the center of the figure. Vertical lines show the flow of time. On the send side, all the data to transfer have to be sequentially flushed from the highest level cache to the memory. Once the flush requests are completed, the actual transfer is initiated on the memories if the receive side also reaches the ready-for-synchronization phase. On the other hand, a message passing request on the *receive* side designates the address space that will be overwritten by receiving data. All of the blocks in the designated message buffer that are in the cache are invalidated, meaning that the most up-to-date version of the data will correctly be fetched from memory when requested for the first time. The invalidation requests can be simultaneously issued when receiving the message because the process does not write back data. The blocking transfer is performed only when all the sub-requests are completed.

4.3.2. Non-Blocking

We support non-blocking message passing by allowing access to the cache while the data transfer occurs. A set of sub-requests and its timing on non-blocking memory-to-memory transfer is shown in Figure 7.

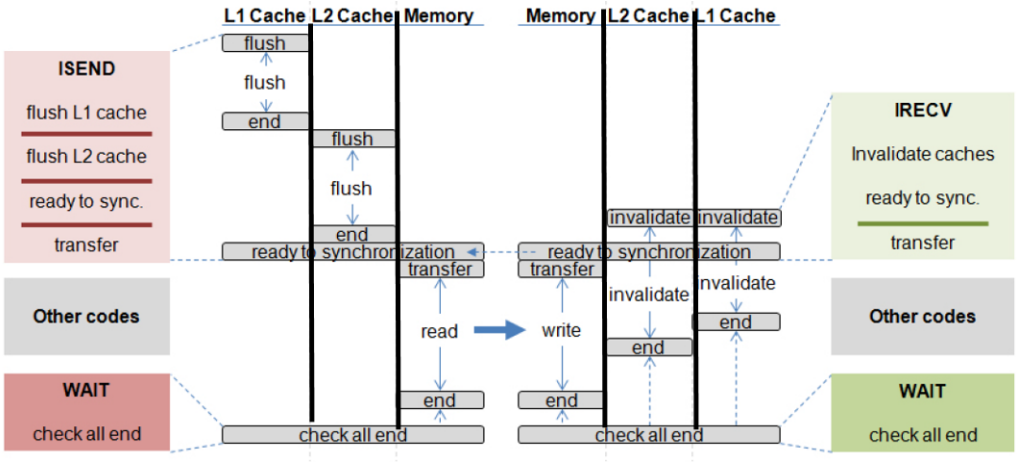

Figure 7. Non-blocking memory-to-memory transfer.

The sequence of protocol events is exactly the same as that in blocking communication, but the caches can be freely used once the actual transfer begins in the memory. This allows normal memory requests after non-blocking is sent to access the caches. At this point, all operations are restored, and requests to the messaging buffers can be serviced. Finally, the programmer should specify a wait point—a point in the code past which execution should not continue until the request finishes. If the non-blocking request is not completed, the program will be blocked at that point. The program may be stalled at this wait point, but that is not a concern of the hardware.

4.4. Cache-to-Cache Communication

Cache-to-cache communication also supports blocking and non-blocking transfers. Figures 8 and 9 show the set of sub-requests and its timing for blocking cache-to-cache transfer and non-blocking cache-to-cache transfer, respectively.

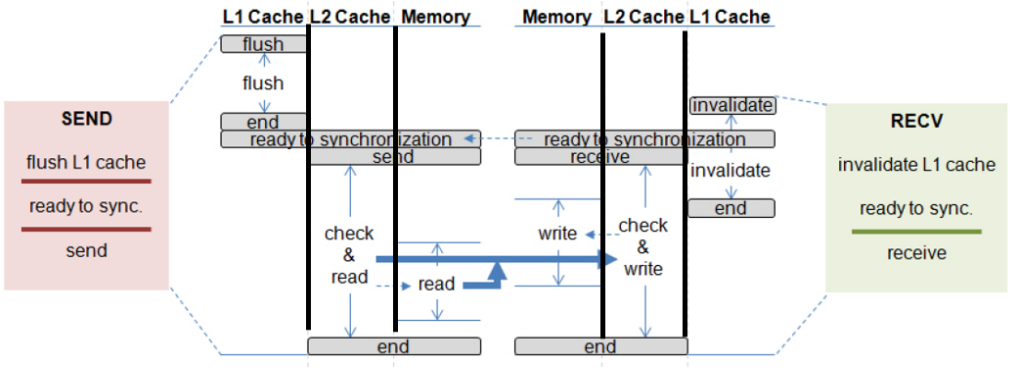

Figure 8. Blocking cache-to-cache transfer.

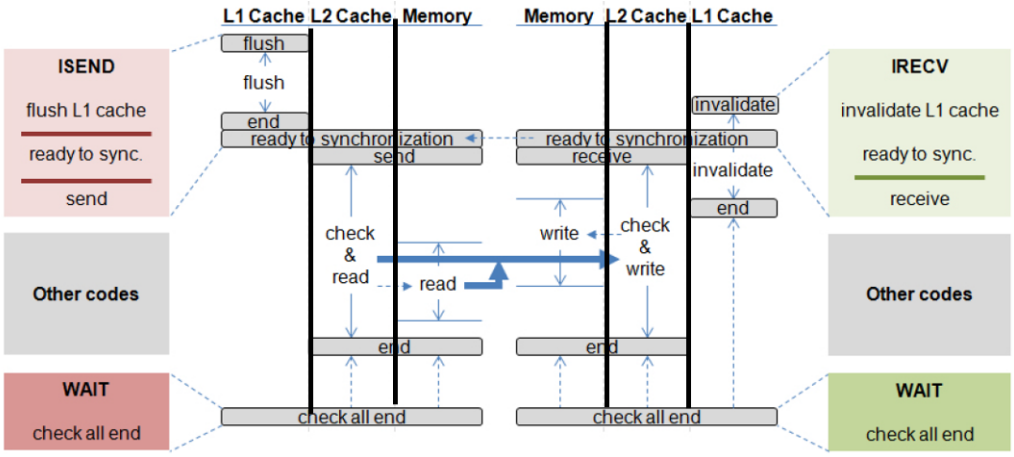

Figure 9. Non-blocking cache-to-cache transfer.

This process is similar to memory-to-memory transfer, except the actual transfer occurs in both the L2 cache and the memory. On the *send* side, all the transferred data are flushed from the L1 cache into the L2 cache. Once the flush request is completed, the send request is initiated on the L2 cache if the receive side is ready. Hopefully, all the cache blocks are in the L2 and are copied from one L2 to another. However, it is possible that some of the data might no longer be cached and only available in the memory. For these blocks, we initiate additional memory requests. The detailed steps of the send request on cache-to-cache transfer are shown in Figure 10.

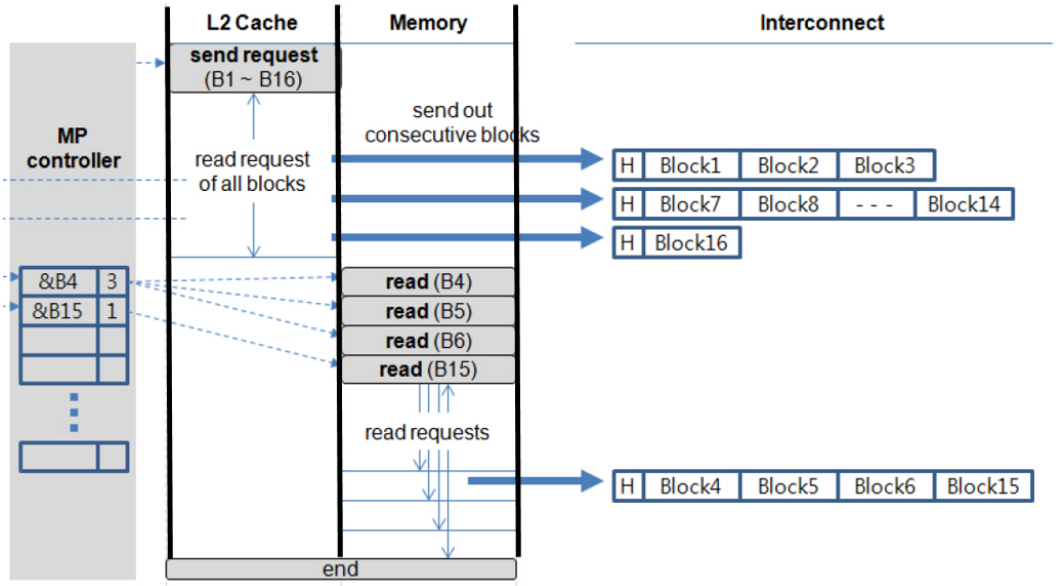

Figure 10. Sequence of cache-to-cache transfer.

Once the ready-for-synchronization phase is passed, the MP controller issues the send request to the L2 cache. The request is similar to a series of read requests, but the consecutive hit blocks are sent out to the interconnection as a message. On the other hand, missed requests are recorded by the MP controller. The recorded requests are issued to memory after the request on the cache is completed, and a message is formed that identifies a series of memory read requests to be transferred out. In the example shown in Figure 10, 16 blocks are to be sent. When the send request is processed in the L2 cache, the blocks are read sequentially starting from the first block. Blocks 1, 2, and 3 all hit; they are packaged up as a message and sent to the receiver. However, Blocks 4, 5, and 6 are completely missed; these are recorded in a buffer in the MP controller. Next, Blocks 7–14 are sent from the

cache. Then, Block 15 misses and is added to the table. Once all the blocks found in the cache are transferred, the missing blocks are read and transferred from the memory. Note, that because we may send blocks out of order, we must add a header to each message specifying its offset in the message so that the receiver can place it in the proper location. In the worst case, the number of missed blocks that do not occur consecutively could be so large that they overflow the table. In this case, we can simply flush the rest of the message and fall back to the memory-to-memory protocol. Interestingly, with a modest-sized buffer, this never occurred in our experiments. So, it can be treated as a rare case that does not need to be highly optimized.

4.5. Collective Communication

Collective communication can be categorized into two groups: one-to-many communication and many-to-many communication. All collective communications are performed as blocking communication. All cores involved have to reach each synchronization point to start initiating the actual transfer. Figure 11 shows the timing of communication in broadcast communication as an example of one-to-many communication.

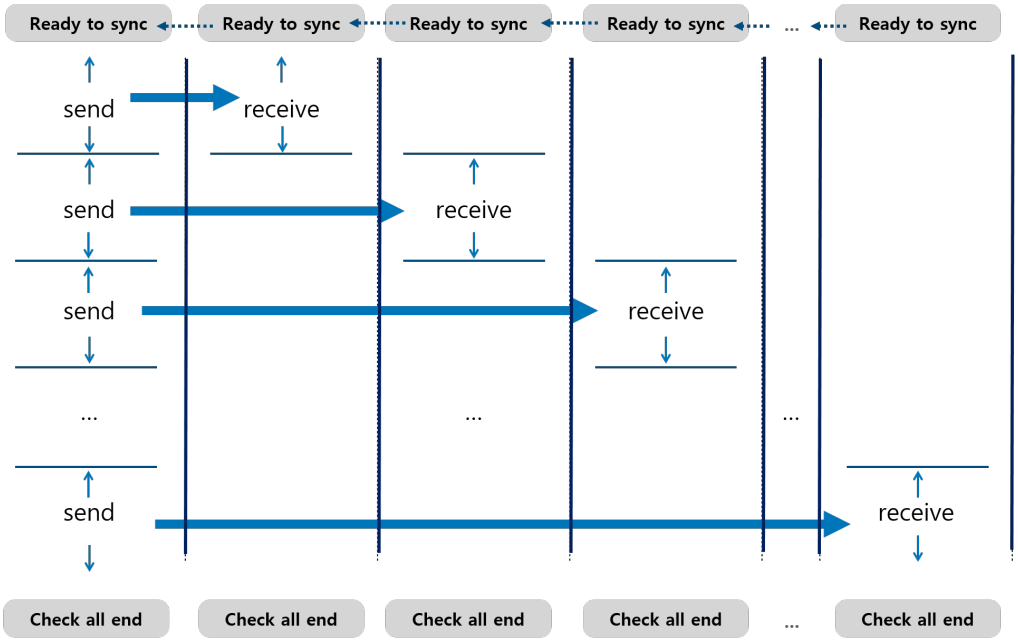

Figure 11. One-to-many collective transfer.

The one-to-many communication consists of multiple blocking point-to-point communications, and each blocking point-to-point communication is performed in the same manner as described in the previous subsections. Because there is no overlap between point-to-point communications, there is no additional requirement for interconnection bandwidth compared with point-to-point communication. Many-to-many collective communication is also designed in the same way. Figure 12 shows the timing of communication in all-to-all communication. The pattern of many-to-many communication can be seen as a combination of one-to-many communications.

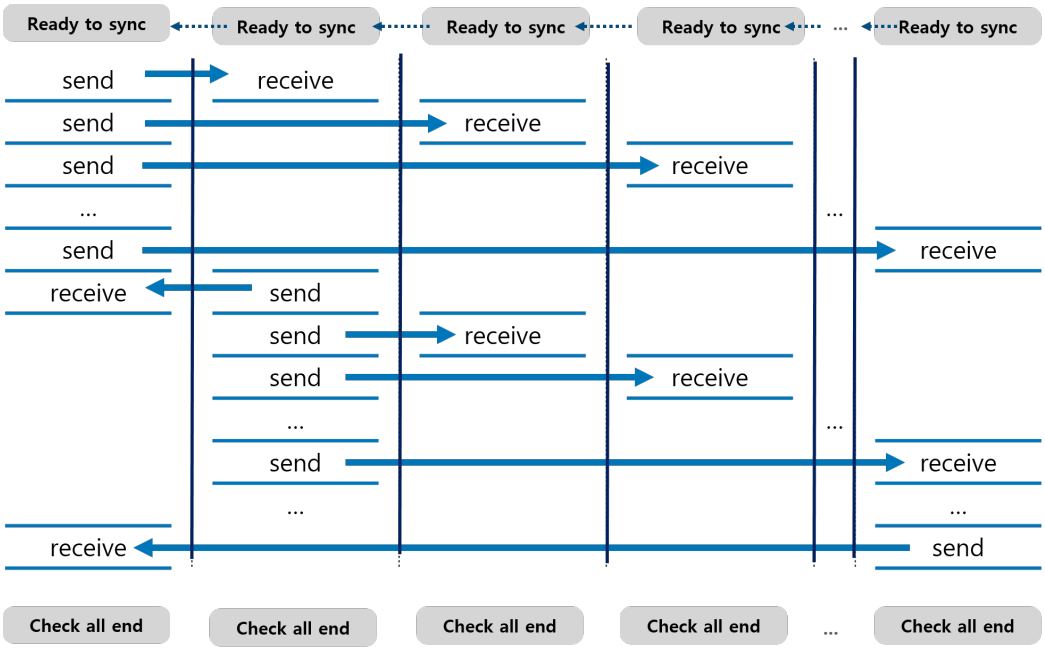

Figure 12. Many-to-many collective transfer.

Consequently, our protocol supports all the basic communication primitives present in MPI standards by carefully designing both blocking and non-blocking point-to-point communication.

5. Evaluation

5.1. Trace Generation

To evaluate the benefits of our new message passing architecture, we collected traces of the NAS parallel benchmarks running on 16 thread systems for OpenMP and MPI versions of the benchmarks with class W input sets. We implemented our own Pintool [11,12] to record the traces; we designed the tool to eliminate MPI library calls from the trace, but it records all MPI calls and their arguments. This allows us to replay the messages using our new architectural design without simulating the work of the MPI library, which is not needed. For the OpenMP versions of the programs, we eliminate threading overheads, such as thread creation and join, so that we can simulate identical costs for those events in both systems. Synchronization events through standard means, such as barriers and locks, are recorded explicitly in the trace so that we can replay their ordering. If there are homemade flags in the benchmarks, those may not be recorded, but after the inspection of the source code, we are not aware of any such flags.

5.2. System Configuration

Table 2 shows the architectural parameters that are common to both our message passing and shared memory architectures.

The interconnect and memory are equivalent for both systems. The latencies and bandwidths of components are carefully chosen to reflect a high performance processor architecture today [13]. Table 3 shows the differences in cache hierarchy design.

The Shared Memory design includes a directory-based MESI protocol, and the L1 cache uses a write-through policy to maintain inclusive properties with a write buffer. The message passing design uses write-back caches, and messaging is performed using our new architecture. All MPI benchmarks run on the message passing design, while all OpenMP (OMP) benchmarks run on shared memory. The configurations for various results are shown in Table 4.

Table 2. Common configuration of components for each architecture.

Component	Configurations
Number of cores	16
L1 cache	32 kB size, 2-way association, 64 B block, 2 cycles latency, write-buffer.
L2 cache	512 kB size, 8-way association, 64 B block, 20 cycles latency, write-buffer, fully pipelined (2 cycles).
Interconnect	32 B/cycles bandwidth each port of routers, 2 cycles latency, input-buffer, no contention model, fully pipelined (2 cycles).
Memory	Unlimited size, 64 B block, 102.4 B/cycles bandwidth (off-chip) 200 cycles latency, fully pipelined (10 cycles).

Table 3. Different configuration of caches for each architecture.

Component	Shared Memory	Message Passing
L1 cache	Write-through	Write-back
L2 cache	MESI protocol	Write-back
Coherence/ communication	MESI directory (unlimited size), 2 cycles latency	Message passing controller, 2 cycles latency

Table 4. Simulated configurations.

Name	Description
OMP	16-core OpenMP on SM.
MtoM	16-core MPI on MP with memory-to-memory communication.
CtoC	16-core MPI on MP with cache-to-cache communication.

5.3. Overall Performance

Figure 13 shows the overall performance measured by execution time across three configurations.

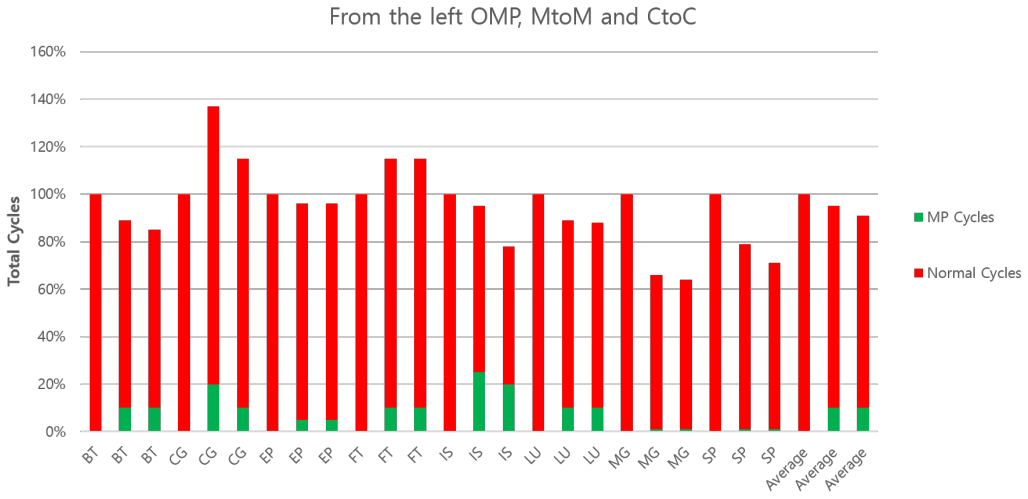

Figure 13. Performance comparison of all benchmarks on each architecture with different configurations.

On average, CtoC and MtoM are 10% and 5% faster than OMP. In the best case, CtoC requires 66% of the cycle time of OMP for MG. CG and FT, on the other hand, are 17% and 15% slower than OMP, respectively. CtoC outperforms MtoM across the board, demonstrating the need for a cache-to-cache protocol. Among the benchmarks studied, BT, EP, IS, LU, MG, and SP all benefit from the message passing architecture. In the CG case, memory references of CG consist of 97% reads and 3% writes in OMP. This provides a huge benefit to SM by sharing data naturally without any unnecessary copying of data. However, MP might have additional memory space to transfer sharing data explicitly, whether the data are read only or not, and this leads usually to more copies. In FT case, the transpose operation of an entire data set leads to additional data movement/alignment, even for each private memory space, as we mention in an earlier section.

The figure also shows the fraction of MP cycles and normal cycles. MP cycles are the cycles when the MP controller is active; otherwise, the cycles are normal cycles. Since there are 16 MP controllers, the final MP cycles are averaged. Hence, the OMP bar has no MP cycles. We can understand one property directly from this breakdown and some others indirectly. The direct measure shows that the total cycles used to communicate messages with the cache-to-cache approach run only 1% faster than the memory-to-memory approach with a normalized scale. However, the CtoC configuration runs even faster due to the increased cache hits at the sender and receiver. Since neither thread is forced to reload message content from the memory, they make quicker progress after communication.

5.4. Total Bytes of Messages on Interconnection of All Benchmarks on Each Architecture with Different Configurations

Figure 14 compares the total number of bytes transferred over the on-chip network for the three 16-core configurations; each bar is partitioned into four categories, and the height of the bar is normalized to OMP.

For SM, bytes of control messages include all transactions between caches and directories excluding payload data, such as a read request or invalidation. In MP, bytes of the control message are the headers and synchronization messages. Bytes of requests to memory are separately collected. Next, the payload data transfer from the other cache-by-cache coherence protocol is classified as data movement between caches in SM. This is compatible to the data transfer by the cache-to-cache protocol in MP. Finally, data movement to/from memory is payload data that come from or head to the memory in SM. This also includes the data transfer by the memory-to-memory protocol in MP.

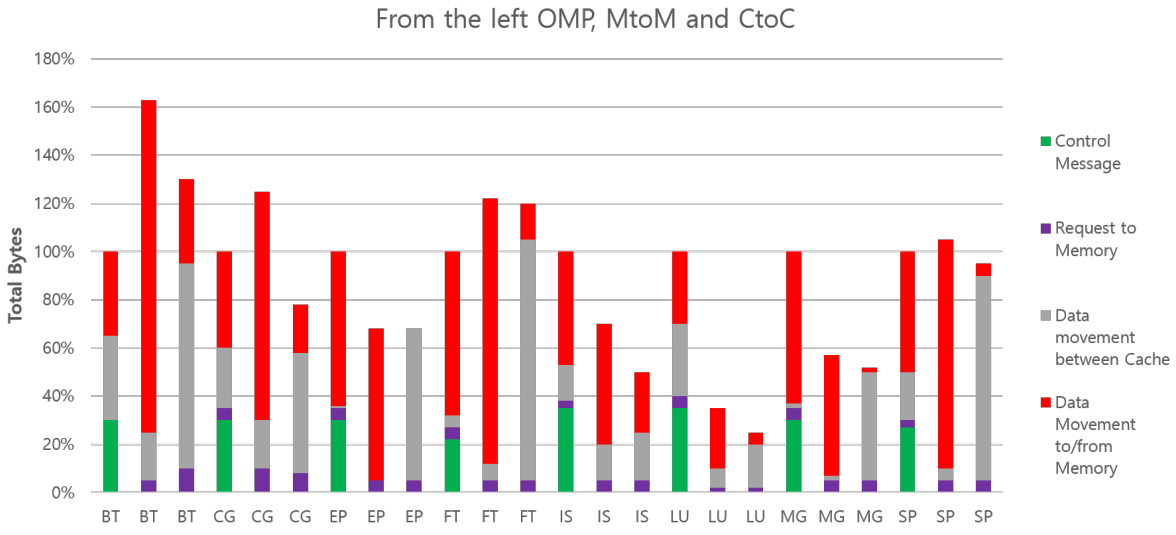

Figure 14. Performance comparison of all benchmarks on each architecture with different configurations.

In comparison, MtoM moves more bytes on average than CtoC, and this is expected since it always moves data to memory first before copying between memory banks. However, in the case of EP and FT, MtoM is on par with CtoC. This is because the size of the problem data set of these benchmarks is relatively large compared to the size of private caches. The transferred data are finally written back to memory, even though they are copied directly from cache to cache. However, in the case of BT and CG, MtoM performs much worse than CtoC because the working data set is small in the benchmarks. In CtoC, once the working data are read initially from the memory, the data do not need to be written back to memory even in the message passing phase.

In MtoM, the transferred data always go to memory. Finally, it is interesting to note that the total amount of bytes of data moved by OMP is between those amounts required by MtoM and CtoC in CG and SP cases. For these benchmarks, SM performs better than copying between memory banks and worse than a streamlined cache copy. This is likely caused by shared data which move more times than necessary between threads, due to the fine-grained nature of block movement used in cache coherence. In these particular workloads, CtoC is able to take advantage of explicit communication to reduce sharing, and our architecture successfully capitalizes on that advantage. We also measured the total number of transactions, as shown in Figure 15.

All bars are normalized to OMP. This only counts the number of unique messages and does not weigh them in any way. Not surprisingly, OMP has the largest count due to the large number of coherence control messages. On the other hand, we cannot see the control messages of MP in the figures since they are very small compared to others. MtoM has more messages than CtoC because it creates additional messages when writing back data to memory. CtoC is the most efficient since most of its messages are successful at moving data directly between the caches. Note, each separate block of data that is communicated is factored into the CtoC measure. CtoC communicates the smallest total number of transactions for all benchmarks across the board. For most benchmarks, it also has the smallest number of data bytes moved. The main reason for this is that CtoC has a minimal impact on a sender's working set, and it can usually place messaging data directly into the receiver's L2 cache. This means that subsequent accesses need not go to memory. This improves the locality of both the sending and receiving thread.

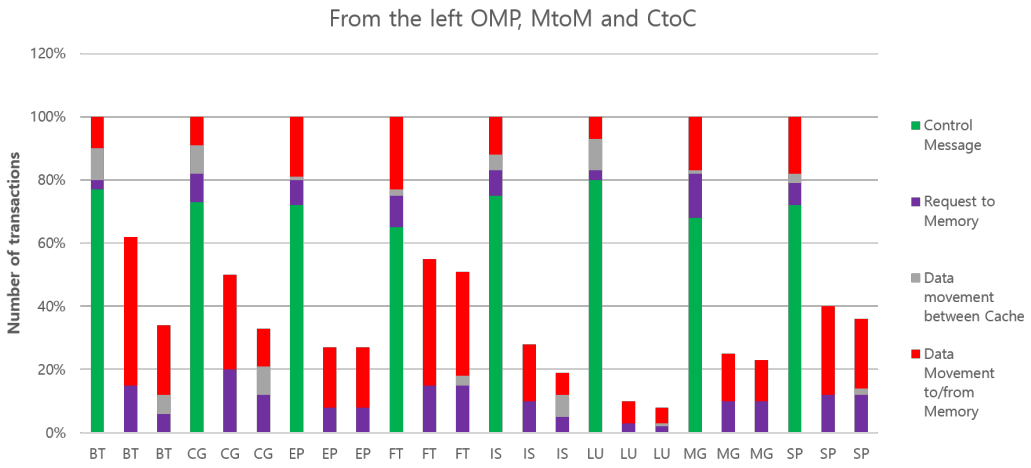

Figure 15. Total number of transactions on interconnection of all benchmarks for each architecture with different configurations.

5.5. Sensitivity Study

We also performed the same simulation under different capabilities of L2 caches, with the last level of private memory hierarchy. Figure 16 shows the overall performance of all benchmarks while the size of the L2 caches increases from 128 KB to 4 MB.

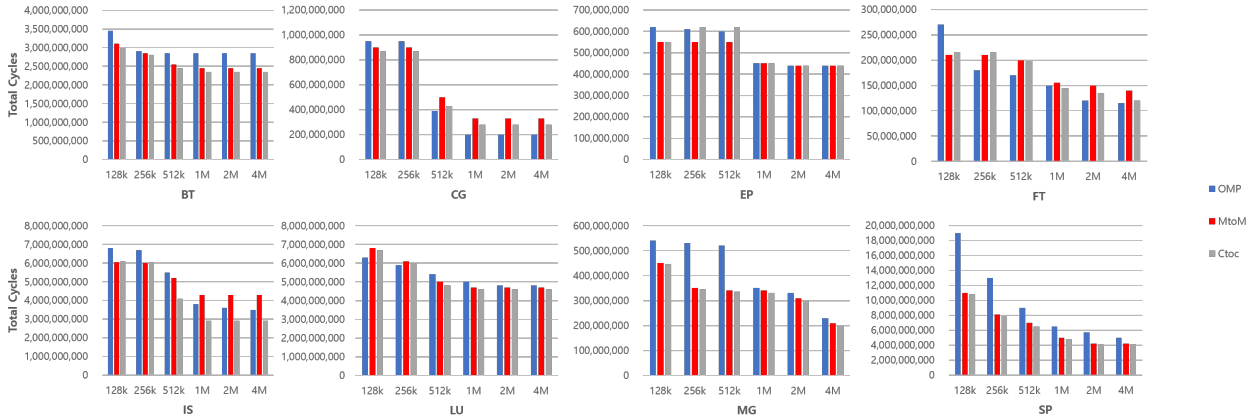

Figure 16. Performance comparison of all benchmarks on each architecture with various L2 sizes.

Note that latencies of L2 caches are fixed to see only the impact of the difference of capacities. In general, message passing offers advantages when L2 caches are small, as shown on the left groups of each benchmark. On average, MtoM and CtoC are 14% and 15% faster than OMP with 128 KB of L2 caches. This is because message passing is able to manage the cache more effectively in hierarchies of memories. This always concerns only the private data through the private memory hierarchy unless explicit data movement is requested. The performance gap between the architectures decreases as the size of L2 caches is increased. At the largest configuration, on average, MtoM is 11% slower than OMP, but CtoC is still 3% faster than OMP. In CG and FT cases, OMP performs even better than CtoC once the L2 cache is large enough for the working set to fully fit to the cache. BT and LU contradict the general trends because the smallest configuration, 128 KB of L2 caches, is able to cover most of the relatively small data sets of the benchmarks. Thus, we cannot see the large coherence overheads of OMP for those benchmarks. We argue here that the differences in the algorithms for each benchmark have influences on the performance of the architecture, and the size of the private cache in the last level is more sensitive to SM than MP. The size of a private cache in the last level is even more important when we consider much larger data sets that are used as inputs. Since the last levels of private caches become smaller, there is a higher possibility of degrading the performance of SM.

Figure 17 shows the percentages of missed blocks out of the total blocks that are requested to transfer out while the size of L2 caches increases from 128 KB to 4 MB.

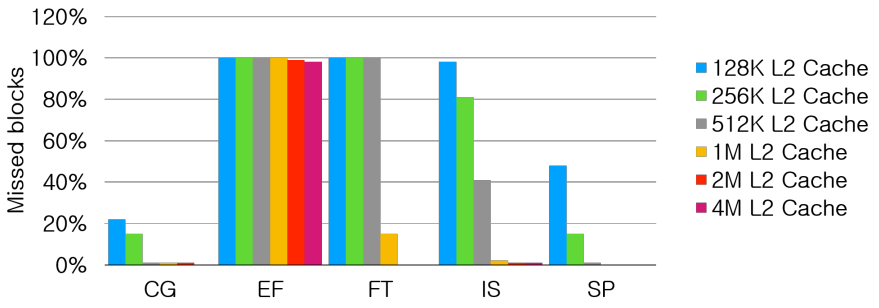

Figure 17. Percentage of missed blocks with cache-to-cache transfer with various L2 sizes.

BT, LU, and MG are excluded from the figure because the percentage of missed blocks is lower than 2% even for 128K L2 caches. For those benchmarks, the performance gap between CtoC and MtoM is almost tje same for all configurations in Figure 16. However, FT and IS show a dramatic change in the number of missed blocks at 512K and 1M L2 caches. Thus, the performance gap between CtoC and MtoM becomes larger at this point for the two benchmarks.

6. Related Work

Although debates comparing message passing to shared memory have been conducted for many years [14–17], these have largely been ignored in the context of multicore architectures. A prevailing perspective is that cache coherence is here to stay [18] and will scale effectively [19,20]. This is reasonable given the vast prevalence of programs written for such architectures. However, it is important to consider alternative programming paradigms and how multi-core processors might support them. In the present work, we simulate a multicore architecture devoted to message passing. We show that it outperforms shared memory and uses the interconnection more effectively; for some workloads, message passing may provide a better scaling trend. SCC [21] is the one of experimental many-core chips that have distributed memory systems on a single chip. There have been research works on the libraries that provide message passing functionality on the SCC [8,22–24]. A hardware design that supports the message passing protocol is also proposed in [25]. However, these studies are focused on the features that boost the performance of SCC based on the MPI standard, such as the bandwidth of interconnection or computation throughput. We propose a viable message passing architecture and compare it in a fair way to the most popular parallel programming model for a multi-core processor (shared memory).

7. Conclusions

In this paper, we have studied the differences in the behaviors of message passing programs and shared memory programs. Interestingly, message passing programs are not fundamentally more expensive or costly because they use messages. Furthermore, if messaging support is added to multicore architectures, the advantages of message passing can be exploited to create an efficient and high-performing alternative to shared memory programming. For many of the applications we studied, when evaluated on our new message passing architecture, the MPI programs outperformed the shared memory programs. So far, we have simulated our design as a message passing-only architecture, and we have shown that it offers many advantages. However, in practice, we do not expect the industry to drop cache coherence or shared memory designs, because they are important for a wide variety of workloads, such as CG, and many programs would be difficult to translate into a message passing style. As a way forward, we think it would be compelling to imagine future chips that offer heterogenous programming models. When appropriate, applications could choose to operate in either shared memory or message passing modes in order to exploit the architecture most effectively. Our design represents a first analysis to understand the opportunity and merits of such a design. In future work, we hope to better understand the synergies gained by combining the best features of both models to finely tune and optimize system behavior.

Author Contributions: Funding acquisition, D.L.; Investigation, J.H.; Project administration, J.H.; Software, J.H.; Supervision, D.L.; Validation, J.H.; Writing—original draft, J.H.; Writing—review and editing, D.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT, and Future Planning (2019R1G1A1100172).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Giacomoni, J.; Moseley, T.; Vachharajani, M. FastForward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel, Programming,Salt Lake City, UT, USA, 20–23 February 2008; pp. 43–52.
2. Friedley, A.; Bronevetsky, G.; Hoefler, T.; Lumsdaine, A. Hybrid MPI: Efficient message passing for multi-core systems. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 17–21 November 2013; pp. 1–11.

3. Dupont de Dinechin, B.; Graillat, A. Feed-Forward Routing for the Wormhole Switching Network-on-Chip of the Kalray MPPA2 Processor. In Proceedings of the 10th International Workshop on Network on Chip Architectures, Boston, MA, USA, 14 October 2017; pp. 1–6.
4. Ma, W.; Ao, Y.; Yang, C.; Williams, S. Solving a trillion unknowns per second with HPGMG on Sunway TaihuLight. *Clust. Comput. Vol.* **2020**, *23*, 493–507. [CrossRef]
5. Xu, Z.; Mauldin, T.; Yao, Z.; Pei, S.; Wei, T.; Yang, Q. A Bus Authentication and Anti-Probing Architecture Extending Hardware Trusted Computing Base Off CPU Chips and Beyond. In Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain, 30 May–3 June 2020; pp. 749–761.
6. Yu, Z.; Xiao, R.; You, K.; Quan, H.; Ou, P.; Yu, Z.; He, M.; Zhang, J.; Ying, Y.; Yang, H.; et al. A 16-Core Processor With Shared-Memory and Message-Passing Communications. *IEEE Trans. Circuits Syst. Regul. Pap.* **2013**, *61*, 1081–1094. [CrossRef]
7. Baker, J.; Gold, B.; Bucciero, M.; Bennett, S.; Mahajan, R.; Ramachandran, P.; Shah, J. SCMP: A Single-Chip Message-Passing Parallel Computer. *J. Supercomput.* **2004**, *30*, 133–149. [CrossRef]
8. Wijngaart, R.; Mattson, T.; Haas, W. Light-weight communications on Intel's single-chip cloud computer processor. *ACM Sigops Oper. Syst. Rev.* **2011**, *45*, 73–83. [CrossRef]
9. Bailey, D.H.; Barszcz, E.; Barton, J.T.; Browning, D.S.; Carter, R.L.; Dagum, L.; Fatoohi, R.A.; Frederickson, P.O.; Lasinski, T.A.; Schreiber, R.S.; et al. The NAS parallel benchmarks. *Int. J. High Perform. Comput. Appl.* **1991**, *5*, 63–73.
10. Loff, J.; Griebler, D.; Mencagli, G.; Araujo, G.; Torquati, M.; Danelutto, M.; Fernandes, L.G. The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Gener. Comput. Syst.* **2021**, in press. [CrossRef]
11. Luk, C.-K.; Cohn, R.; Muth, R.; Patil, H.; Klauser, A.; Lowney, G.; Wallace, S.; Reddi, V.J.; Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, IL, USA, 12–15 June 2005, pp. 190–200.
12. Villa, O.; Stephensen, M.; Nellans, D.; Keckler, S. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In Proceedings of the MICRO '52: The 52nd Annual IEEE/ACM International Symposium on Microarchitecture, Columbus, OH, USA, 12–16 October 2019; pp. 372–383.
13. Pavlovic, M.; Etsion, Y.; Ramirez, A. Can many cores support the memory requirements of scientific applications? In Proceedings of the 2010 International Conference on Computer Architecture, ser. ISCA'10, Phoenix, AZ, USA, 22–26 June 2012; pp. 65–76.
14. Chandra, S.; Larus, J.R.; Rogers, A. Where is time spent in message-passing and shared-memory programs? In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA, 5–7 October 1994; pp. 61–73.
15. Tasoulas, Z.-G.; Anagnostopoulos, I.; Papadopoulos, L.; Soudris, D. A Message-Passing Microcoded Synchronization for Distributed Shared Memory Architectures. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2018**, *38*, 975–979. [CrossRef]
16. Chodnekar, S.; Srinivasan, V.; Vaidya, A.; Sivasubramaniam, A.; Das, C. Towards a communication characterization methodology for parallel applications. In Proceedings of the Third International Symposium on High-Performance Computer Architecture, San Antonio, TX, USA, 1–5 February 1997; pp. 310–319.
17. Kubiatowicz, J.; Agarwal, A. Anatomy of a message in the alewife multiprocessor. In Proceedings of the 7th International Conference on Supercomputing, ser. ICS '93, Tokyo, Japan, 19–23 July 1993; pp. 195–206.
18. Martin, M.M.K.; Hill, M.D.; Sorin, D.J. Why on-chip cache coherence is here to stay. *Commun. ACM* **2012**, *55*, 78–89. [CrossRef]
19. Daya, B.K.; Chen, C.H.O.; Subramanian, S.; Kwon, W.C.; Park, S.; Krishna, T.; Holt, J.; Chrakasan, A.P.; Peh, L.S. SCORPIO: A 36-core research chip demonstrating snoopy coherence on a scalable mesh NoC with in-network ordering. In Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), Minneapolis, MN, USA, 14–18 June 2014; pp. 25–36.
20. Sanchez, D.; Kozyrakis, C. SCD: A scalable coherence directory with flexible sharer set encoding. In Proceedings of the IEEE 18th International Symposium on High Performance Computer Architecture (HPCA), New Orleans, LA, USA, 25–29 February 2012; pp. 1–12.
21. Held, J. Single-chip cloud computer. *Euro-Par-Workshop* **2010**, *6586*, 85.
22. Mattson, T.G.; Riepen, M.; Lehnig, T.; Brett, P.; Haas, W.; Kennedy, P.; Howard, J.; Vangal, S.; Borkar, N.; Ruhl, G. The 48-core SCC processor: The programmer's view. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, New Orleans, LA, USA, 13–19 November 2010; pp. 1–11.
23. Urea, I.A.C.; Riepen, M.; Konow, M. RCKMPI: Lightweight MPI implementation for Intels single-chip cloud computer (SCC). *Lect. Notes Comput. Sci.* **2011**, *6960*, 208–217.
24. Urea, I.A.C.; Riepen, M.; Konow, M.; Gerndt, M. Invasive MPI on Intel's single-chip cloud computer. In Proceedings of the Architecture of Computing Systems ARCS 2012, ser. Lecture Notes in Computer Science, Munich, Germany, 28 February–2 March 2012; Volume 7179, pp. 74–85.
25. Howard, J.; Dighe, S.; Hoskote, Y.; Vangal, Y.; Finan, D.; Ruhl, G.; Jenkins, D.; Wilson, H.; Borkar, N.; Schrom, G.; et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In Proceedings of the Solid-State Circuits Conference Digest of Technical Papers (ISSCC), San Francisco, CA, USA, 7–11 February 2010; pp. 108–109.