Accelerating Communication in Single-chip Shared Memory Many-core Processors

Farrukh Hijaz‡, Brian Kahne*, Peter Wilson*, Omer Khan‡
*Freescale Discovery Labs, Austin, TX, USA
†University of Connecticut, Storrs, CT, USA

Abstract—Shared Memory stands out as a sine qua non for parallel programming of many commercial multicore processors. For efficiency, shared memory is often implemented with hardware support for cache coherence and memory consistency among the cores. It optimizes patterns of communication that benefit common programming styles. As parallel programming is now mainstream, those common programming styles are challenged with emerging applications that communicate often and involve large data. Message passing has been mostly used as a parallel programming model for supercomputers and large clusters. However, this paradigm offers opportunities to explicitly and efficiently manage communication patterns. We propose to retain the shared memory model, however, introduce a set of lightweight in-hardware explicit message passing style instructions in the instruction set architecture (ISA). The explicit messaging protocol can be used as an “accelerator for communication” and improve bulk data transfers, eliminate the overheads of multi-party communication in shared memory protocols, and efficiently support synchronization primitives.

I. INTRODUCTION

Power dissipation has halted the drive to higher core frequencies. However, transistor density has continued to grow, and CPUs with many-cores have become the norm both in academia (e.g., Raw [8], Execution Migration Machine [2]) and industry (e.g., Tilera [1], Intel Phi [4]), with 1000 or more core processors predicted to arrive in a few years [2].

How will these massively parallel many-core chips be programmed? A shared memory abstraction stands out as a sine qua non for general-purpose programming: while architectures with restricted memory models (most notably GPUs) have enjoyed immense success in specific applications (such as rendering graphics), most programmers are familiar with a shared memory model, and commercial general-purpose multicore systems have supported this abstraction in hardware. The main question, then, is how to efficiently provide communication on the scale of hundreds or thousands of cores.

Explicit message passing protocol (MPI) is frequently used in the high-performance computing space [3], while shared-memory abstraction dominates the traditional small-computer and embedded space, as exemplified by the pthreads API. However, it can be notoriously difficult to get lock-based shared-memory communication to work correctly. Difficult to debug race conditions are the norm in such applications. Recently in the embedded domain, attention has been growing about the advantages of using message passing, e.g., the Multicore Communications API (MCAPI) is gaining traction. The advantage is one of dependability: if a lock is forgotten, an application may work correctly nine times out of ten, then fail unexpectedly. On the other hand, if a message is missing the program will always produce incorrect results. Traditionally, however, such message passing APIs have been built on top of primitives involving communication via coherent caches and coordination via special instructions, such as traditional RISC-based load-with-reservation and conditional stores [6].

The downside of this approach is that it can take many instructions to coordinate between cores. At a minimum, a spinlock must surround a critical section. However, this spinlock needs exponential back-off to avoid a livelock situation. This results in correct functionality, but at the expense of energy consumption and precluding other work from being completed. Thus, such locks often require the operating system to schedule additional work. When one core has released a lock, it signals the other core via an interrupt, which is a supervisor activity. The end result of all of this is that communication between two cores can potentially take hundreds of instructions to complete.

We propose to retain the shared memory programming paradigm, however introduce a set of lightweight in-hardware message passing style instructions to the instruction set architecture (ISA). Our key idea is to explore an “accelerator for communication” rather than always rely on traditional shared memory communication protocol to manage on-chip data. The idea to layover an explicit messaging protocol on top of traditional shared memory was explored in the 1990s as part of the MIT Alewife multiprocessor machine [5]. We are unique because we propose to study this hybrid paradigm in the context of single-chip many-core processors that offer new tradeoffs in managing data accesses over high bandwidth, low latency networks-on-chip (NOC).

The proposed explicit messaging protocol introduces a set of send and receive instructions in the ISA. These instructions are implemented in-hardware to enable efficient core-to-core communication. The advantage is that the cache coherence related indirections of shared memory protocols can now be replaced by private memory based explicit communication of data over the on-chip network, leading to a significant reduction in data access movement and a consequent lowering of energy usage. We outline the following distinct benefits from utilizing our proposed explicit messaging protocol:

- Explicit data access can substantially reduce the data movement traffic, which leads to reductions in energy consumption and latency.
- Many shared memory synchronization primitives are implemented in software using special ISA instructions. These one-to-all and all-to-all communication patterns can be improved significantly if the ISA supports explicit messaging protocol.
- In many applications such as streaming workloads, bulk data transfer is commonly used. Since shared memory coherence protocols are implemented at the cache line granularity and each transfer involves multi-party communication, bulk transfer of data accrue overheads quickly. On the other hand, explicit messaging protocol can efficiently transfer data in bulk.
- Explicit message passing works well with multi-threaded cores to hide latency and increase overall throughput.

We note that although explicit messaging protocol has many benefits as outlined earlier, it suffers from several challenges that must be considered when evaluating its utility. First, since the protocol is explicit, it works best with applications that are designed from the grounds up to use a message-passing style of communication. Second, the explicit messaging protocol requires per-core hardware support in the form of receive queues and logic for managing in-bound data that is placed into appropriate storage, whether that is the core’s local cache or a private tightly-coupled memory.

Finally, it is possible that the use of message passing acts to increase the overall working-set size of a program. If all data is passed as messages, it would require multiple instances of the data in memory. However, the hybrid nature of our approach
is a key advantage in this situation. When large blocks of data need to be transmitted from one core to the next, this can be accomplished via the sending of a “pointer” to the actual data. Thus, a message is sent from a register in one core to the register in another core, with very little overhead. The data remains in the cache and is transferred using the traditional coherence protocol as requested by the consuming core. Thus, compared to a traditional multicore, there is essentially no increase in working set size and thus no increase in cache pressure, and the system has saved potentially hundreds of instructions which would have been required by obtaining and releasing locks.

II. PROPOSED ARCHITECTURE

Figure 1 shows a logical view of a tile within a representative many-core processor (similar to [1]). The three colored modules are introduced in the tile to support the proposed explicit messaging protocol on top of directory-based shared memory. The compute pipeline is extended to support the new send and receive instructions and their associated control logic. The send instruction is implemented as blocking or non-blocking. In case of blocking, an acknowledgment is required from the receiving core and can be thought of as a form of remote procedure call, while non-blocking variant sends the message on the network and retries the send instruction in the sender core. The data for the send instruction is either picked up from the register file or the level-1 data cache.

Similarly, the receive instruction requires hardware support at the receiving core for incoming messages. This is represented by the “MP buffers” in Figure 1 and takes the form of a small number of hardware-based receive queues and logic for implementing an arbitrary number of queues located in private memory. When a new message is received, it is buffered in a queue until handled via a receive instruction. The receive instruction then either loads the message into registers or stores the data into the level-1 cache. The send and receive instructions require additional networks, one for the requests and another for the replies. This ensures network protocol deadlock freedom in traditional mesh-based NOCs.

We note that an application must be re-compiled (and potentially re-written) using a compiler that supports the new hybrid parallel programming paradigm. We have implemented our protocol using an in-house industry-class simulator and associated toolchain for the LLVM-based compiler. The core models are simple RISC compute pipelines with hardware multithreading support. The cores are connected using an on-chip point-to-point NOC. For fast data access, a two-level cache hierarchy is implemented. The compiler itself does not inherently understand message passing. Instead, we simply wrap the send and receive instructions within assembly blocks, using the gcc extended asm block syntax to instruct the compiler as to what registers are inputs or outputs. This allows the compiler to allocate registers properly and schedule the code.

III. MOTIVATING EXAMPLE APPLICATION

To motivate the proposed architecture, we use a IPv4 packet forwarding kernel and quantify the communication latency. This kernel spawns many packet processing threads that generate packets with unique sequence numbers. The flow core communicates the packet and its sequence number to an ordering core (called the egress core in our kernel). The egress core receives the packets from the flow cores, possibly out of order, and use their sequence numbers to re-order them. This kernel requires communication between flow cores and the ordering egress core.

We implemented an efficient shared memory parallel program for this kernel. A shared “completion buffer” is maintained to order packet processing at the egress core. The flow cores generate their packet sequence number and set a valid bit in the completion buffer. The egress core either waits for the next packet in the sequence number to arrive, or reset the corresponding valid bit for the received packet. Shared memory communication is managed between the flow cores and the egress core through the corresponding valid bit. If valid bit is set, the flow core waits, else if the valid bit is reset, the egress core waits. This communication is efficiently managed using the underlying in-hardware shared memory coherence protocol. Our analysis shows that in case of a single flow core, the cycles per packet latency between the flow and egress cores is 76 cycles/packet. The majority of this latency is spent in moving the cache line associated with the valid bit between the flow core and the egress core. Note that we align the valid bits to remove the possibility of false sharing of cache lines. Since the communication pattern requires read-write sharing of the valid bit, the ping-ponging of valid bits incur coherence overheads. As more flow cores are added, the concurrency of packet processing is exploited and much of the latency due to valid bit communication is hidden. We observe a steady decrease in latency per packet as more flow cores are added. This behavior saturated at 256 cores, where a much improved 20 cycles per packet latency was observed.

We also implemented this kernel using our explicit messaging protocol. The flow cores generate the sequence number and use a send instruction to communicate the sequence number to the egress core. The egress core implements a dedicated buffer to receive packets from each flow core. A receive instruction at the egress core receives the packet sequence number and processes it in global order by advancing from one incoming port to the next and waiting for packet arrival when necessary. Our analysis of the explicit send/receive protocol shows that even at small core counts (in this case 4 cores), our communication protocol accelerated packet processing throughput and achieved a latency of 9.8 cycles per packet. This is a 2× speedup relative to the best case shared memory implementation of this kernel.

IV. CONCLUSION

In this paper we have introduced the idea of an “accelerator for communication” in future single-chip many-core processors. We plan to extend the applicability of our proposed communication protocol for a wide range of parallel applications, and quantify its performance and energy advantages.

REFERENCES