Early experience with the Barrelfish OS and the Single-Chip Cloud Computer

Simon Peter, Adrian Schüpbach, Dominik Menzi and Timothy Roscoe
Systems Group, Department of Computer Science, ETH Zurich

***Abstract*—Traditional OS architectures based on a single, shared-memory kernel face significant challenges from hardware trends, in particular the increasing cost of system-wide cache-coherence as core counts increase, and the emergence of heterogeneous architectures – both on a single die, and also between CPUs, co-processors like GPUs, and programmable peripherals within a platform.**

The *multikernel* is an alternative OS model that employs message passing instead of data sharing and enables architecture-agnostic inter-core communication, including across non-coherent shared memory and PCIe peripheral buses. This allows a single OS instance to manage the complete collection of heterogeneous, non-cache-coherent processors as a single, unified platform.

We report on our experience running the Barrelfish research multikernel OS on the Intel Single-Chip Cloud Computer (SCC). We describe the minimal changes required to bring the OS up on the SCC, and present early performance results from an SCC system running standalone, and also a single Barrelfish instance running across a heterogeneous machine consisting of an SCC and its host PC.

I. Introduction

The architecture of computer systems is radically changing: core counts are increasing, systems are becoming more heterogeneous, and the memory system is becoming less uniform. As part of this change, it is likely that system-wide cache-coherent shared memory will no longer exist. This is happening not only as specialized co-processors, like GPUs, are more closely integrated with the rest of the system, but also as core counts increase we expect to see cache coherence no longer maintained between general purpose cores.

Shared-memory operating systems do not deal with this complexity and among the several alternative OS models, one interesting design is to eschew data sharing between cores and to rely on message passing instead. This enforces disciplined sharing and enables architecture-agnostic communication across a number of very different interconnects. In fact, experimental non-cache-coherent architectures, such as the Intel Single-Chip Cloud Computer (SCC) [1], already facilitate message passing with special hardware support.

In this paper, we report on our efforts to port Barrelfish to the SCC. Barrelfish is an open-source research OS developed by ETH Zurich and Microsoft Research and is structured as a *multikernel* [2]: a distributed system of cores which communicate exclusively via messages.

The multikernel is a natural fit for the SCC that can fully leverage the hardware message passing facilities, while requiring only minimal changes to the Barrelfish implementation for a port from x86 multiprocessors to the SCC. Furthermore, the SCC is a good example of the anticipated future system types, as it is both a non-cache coherent multicore chip, as well as a host system peripheral.

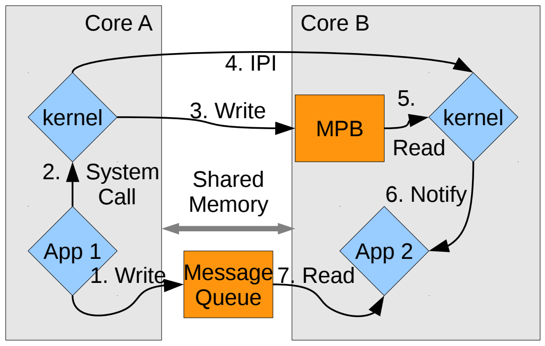

Fig. 1. Sending of a message between SCC cores

We describe the modifications to Barrelfish's message-passing implementation, the single most important subsystem needing adaptation. We give initial performance results on the SCC and across a heterogeneous machine consisting of an SCC peripheral and its host PC.

II. Message Passing Design

Message passing in Barrelfish is implemented by a *message passing stub* and lower-level *interconnect* and *notification drivers*. The message passing stub is responsible for (un-)marshaling message arguments into a message queue and provides the API to applications. Messages are subsequently sent (received) by the interconnect driver, using the notification driver to inform the receiver of pending messages. Messages can be batched to reduce the number of notifications required.

Message passing is performance-critical in Barrelfish and thus heavily tied to the hardware architecture. In this section, we describe the interconnect and notification driver design between cores on the SCC, as well as between host PC and the SCC. We mention changes to the message passing stub where appropriate.

A. SCC Inter-core Message Passing

The SCC interconnect driver reliably transports cache-line-sized messages (32 bytes) through a message queue in non-coherent shared memory. Shared memory is accessed entirely from user-space, shown by steps 1 and 7 in Figure 1, using the SCC write-combine buffer for performance. The polling approach to detect incoming messages, used by light-weight

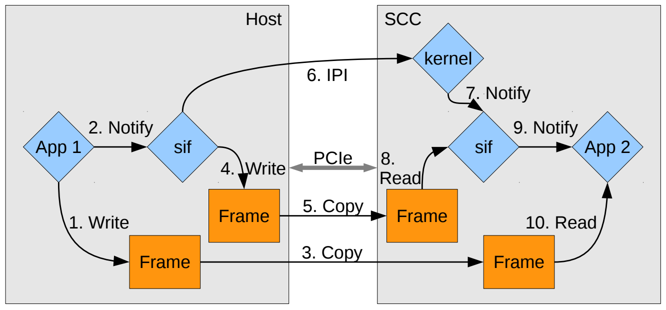

Fig. 2. Sending of a message from host to SCC

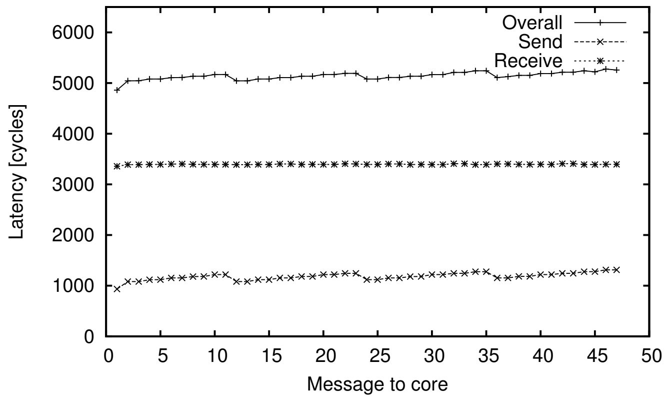

Fig. 3. Average notification latency from core 0 (*Overall*). *Send* and *Receive* show time spent in sending and receiving, respectively.

message passing runtimes, such as RCCE [3], is inappropriate when using shared memory to deliver message payloads, since each poll of a message-passing channel requires a cache invalidate followed by a load from DDR3 memory.

Consequently, the notification driver uses inter-core notifications, implemented within per-core kernels, to signal message arrival. Notifications are sent by a system call (2) via a ring-buffer on the receiver's on-tile message passing buffer (MPB) and reference shared-memory channels with pending messages (3). An inter-core interrupt (IPI) is used to inform the peer kernel of the notification (4), which it forwards to the target application (6).

At first sight, it may seem odd to use main memory (rather than the on-tile MPB) for passing message payloads, and to require a trap to the kernel to send a message notification. This design is motivated by the need to support many message channels in Barrelfish and more than one application running on a core. The SCC's message-passing functionality does not appear to have been designed with this use-case in mind. We discuss this issue further in Section IV.

B. Host-SCC Message Passing

The SCC is connected to a host PC as a PCI express (PCIe) device and provides access to memory and internal registers via a system interface (SIF). The host PC can write via the SIF directly to SCC memory using programmed I/O or the built-in direct memory access (DMA) engine.

The interconnect-notification driver combination used between host PC and SCC, called SIFMP, employs two proxy drivers. One on the host, and one on the SCC. New SIFMP connections are registered with the local proxy driver. When the interconnect driver is sending a message by writing to the local queue (1), the notification driver notifies the proxy driver (2), which copies the payload to an identical queue on the other side of the PCIe bus (3). The proxy driver then forwards the notification to the receiver of the message via a private message queue (4, 5), sending an IPI to inform the receiving driver of the notification via its local kernel on the SCC (6, 7). The peer proxy reads the notification from its private queue (8) and forwards it to the target application (9), which receives the message by reading the local copy via its interconnect driver (10). This implementation, shown in Figure 2, uses two message queues (one on each side) and two proxy driver connections (one for each driver) for each SIFMP connection.

III. Evaluation

We evaluate message passing efficiency by running a series of messaging benchmarks. All benchmarks execute on a Rocky Lake board, configured to 533MHz core clock speed, 800MHz memory mesh speed and 266MHz SIF clock speed. The host PC is a Sun XFire X2270, clocked to 2.3GHz.

A. Notification via MPB

We use a ping-pong notification experiment to evaluate the cost of OS-level notification delivery between two peer cores. Notifications are performance critical to notify a user-space program on another core of message payload arrival. The experiment covers the overheads of the system call required to send the notification from user-space, the actual send via the MPB and corresponding IPI, and forwarding the notification to user-space on the receiver.

Figure 3 shows the average latency over 100,000 iterations of this benchmark between core 0 and each other core, as well as a break-down into send and receive cost. As expected, differences in messaging cost due to topology are only noticeable on the sender, where the cost to write to remote memory occurs. The relatively large cost of receiving the message is due to the direct cost of the trap incurred by the IPI, which we approximated to be 600 cycles, and additional much larger indirect cost of cache misses associated with the trap.

B. Host-SCC Messaging

We determined the one-way latency of SIFMP for a cache-line size message from host to SCC to be on the order of 5 million host cycles. As expected from a communication channel that crosses the PCIe bus, SIFMP is several orders of magnitude slower than messaging on the host (approximately 1000 cycles). To gain more insight into the latency difference, we assess the performance of the PCIe proxy driver implementation, by evaluating read access latency of varying size from SCC memory to the host PC, using DMA.

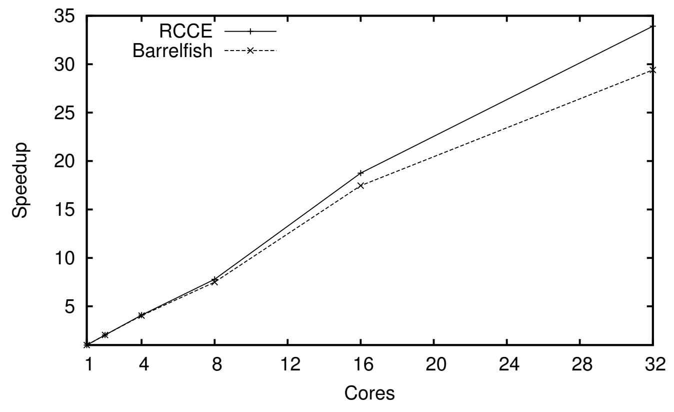

Fig. 4. RCCE LU benchmark speedup comparison

The results show a baseline read overhead of about 500,000 host clock cycles, which is 10% of the messaging overhead. Thus, PCIe bus latency explains only a fraction of the measured messaging overhead and we have yet to determine the cause of these overheads.

C. Application-level benchmarks

The standard software environment available on the SCC uses RCCE [3], a library for light-weight communication, highly optimized for this platform, requiring exclusive access to the MPB. We implemented a substrate supporting the RCCE message-passing interface using Barrelfish stubs and interconnect drivers for messaging, which multiplexes MPB access to applications. We evaluate the LU benchmark shipped with RCCE to compare the performance achieved on Barrelfish to that of RCCE.

From the result, shown in Figure 4, we can see that, at the application level, Barrelfish shows only slightly lower performance and scalability than direct MPB access via RCCE. This overhead is due to multiplexing and the early version of our port, which we seek to improve.

IV. Discussion

The port of the IA-32 version of Barrelfish to the SCC required 2235 lines of SCC-specific C code and 130 lines of assembly, only 17% of which execute in privileged mode. By and large, the bring-up for SCC was straightforward and took about 2 person-months. The prototype is fully functional and shows adequate initial application performance.

We experienced no significant problems with Barrelfish due to the lack of coherent caches on the SCC. This was not a big surprise to us, but it was a confirmation of our expectations, and a validation of the OS design. We regard the lack of coherent caches as a useful feature from a research perspective.

In this section, we cover the most influential architectural issues to our design and discuss possible improvements in either software or hardware.

A. Cache issues

The caches do not allocate a cache line on a write miss, treating it as an uncached write to memory. Furthermore, a core is allowed only one outstanding write transaction; when such a write miss occurs, any subsequent memory or L2 cache access causes the core to stall until the write to memory completes (typically around 100 cycles). Combined with the lack of a store buffer, this policy causes severe performance degradation for word-sized writes to data not already present in the cache. When storing to a fresh stack frame, or saving registers in a context switch path, each individual memory write instruction will stall the processor.

For example, in Barrelfish kernel code, we observed that simple function calls in hot paths of the system regularly have an order of magnitude greater overhead (in cycles) on SCC compared to newer x86 processors. Our code is not optimized for this behavior and shows major inefficiencies, in particular on function call boundaries immediately after kernel crossings. Our tentative explanation is that caller-saved registers will be pushed onto the stack upon a function call and then restored upon return from the function. Both cases miss in the cache, but the call incurs particularly high overhead, as each individual write goes to memory, and the cache lines are only allocated when reading them on return. We have also observed substantially increased costs for exception and trap handling, which save register state to memory not commonly present in the cache.

It is possible that an OS workload is a particularly bad case for this cache design. More work is required to both confirm this as the cause, and explore possible solutions. Ideally this could be fixed in hardware, through changes to the cache architecture, the addition of a store buffer, or simply allowing the write-combining buffer to be used for non-message-buffer memory, which would mitigate the problem by allowing full cache-line writes to memory. A possible software fix would involve reading from each cache line before it was written, to ensure its presence in the cache; in the case of context save code this could be done explicitly, but for stack access would probably require compiler modifications.

B. Message-passing memory

The ability to bypass the L2 cache for areas of address space designated as messaging buffers (MB), combined with an efficient L1 invalidation of all such lines, is one of the most interesting features of the SCC.

As with other message-passing features of the SCC, this functionality may have been designed with a single-application system in mind. When using MB memory for the operating system, as in Barrelfish, we typically have a number of communication channels in use at any one time. For this reason, although the CL1INVMB instruction is extremely cheap to execute, its effects may be somewhat heavyweight, since it invalidates all messaging data, some of which we may wish to have remain in the L1 cache.

In our message-passing implementations, we generally know precisely which addresses we wish to invalidate. Consequently, we would find more fine-grained cache control very useful. An instruction which would invalidate a region around a given address would be ideal for us.

Better still would be to extend such functionality to the L2. Receiving data in an MPB generally involves an L1 miss (ideally to the on-tile MPB, but see below why this is problematic), followed by a miss to main memory caused by copying the data somewhere where it can be cached in L2, followed by a second L2 miss when the data needs to be subsequently read, due to the non-allocation policy of the cache on a write miss. The final penalty can be mitigated somewhat by performing a read of the destination location (and so populating the L2) before writing the received data.

This coarse-grain cache access can have far-reaching implications for applications. For example, in its current form the cache architecture seems to prohibit any efficient zero-copy I/O implementation, since if the message passing buffers are used, any cached data will be invalidated any time further I/O occurs. Our position overall is that explicit cache management is good, and Barrelfish (and, we believe, other OS code) would benefit from future SCC implementations providing much more fine-grained control over it.

C. *The on-tile message passing buffer*

We experienced two significant challenges when using the on-tile message passing buffers on SCC. These challenges are related, but different.

1) Size: The small size of 8192 bytes constrains the size of message queues. If messages cannot be lost (a typical design assumption for message-passing applications, including Barrelfish), this results in blocking and tighter coupling between communicating processes.

Barrelfish uses message-passing throughout for communication, and consequently requires a large number of independent message channels to share the MPB. This leads to the question of how to allocate space in the MPB to message channels. Obviously, the space is too small to reserve parts for message payload. For example, reserving 8 cache lines for each message queue would allow only 32 message channels per core, which is impractical.

It is conceivable to multiplex all Barrelfish channels onto a single channel per pair of cores, requiring only 47 channels in the MPB and privileged code to demultiplex incoming messages. This allows 170 bytes of buffer per core pair, still small, but possibly useful for small messages. The downside to this approach is a kernel crossing and increased contention.

Finally, message payload could be written to the MPB per application, but this increases contention even further. In both cases, direct payload access by applications is prohibited, as the MPB has to be freed up as quickly as possible to reduce contention. This requires copying the message in and out of the MPB, which is a costly operation.

2) Multiplexing: An operating system must mediate access to the MPBs to ensure safe sharing of the buffers between applications. Unfortunately, the very high MPB access performance is completely dominated by the cost of kernel crossing to validate the access. As an additional cost, since multiple cores can be expected to be accessing each tile's MPB, write access to each core's memory must be done under a lock.

Like most resources, the MPBs can be multiplexed in both in space and time.

Space-multiplexing the MPBs requires a high-performance protection mechanism to divide the buffer between applications or other resource principals. For main memory, this is performed by each core's MMU. The P54C chip used in the SCC provides hardware protection of memory segments of sizes smaller than a page and thus could be used to space-multiplex the MPB when small messages are sufficient. However, the size of the MPB prohibits space multiplexing of larger message queues.

Time-multiplexing the MPBs requires copying each application's state into or out of the MPBs on a context switch, or performing this lazily. This is potentially 8KB of message data, a substantial context-switch overhead when caches do not allocate on a write miss.

Unlike memory which is under the exclusive use of an application, memory used for communication between applications on different cores is shared between a pair of principals. Time-multiplexing on-tile MPB memory in software is possible via co-scheduling [4] of communicating principals on different cores. However, this constrains the system-wide schedule and requires considerable communication overhead in itself [5].

In addition, our experience with Barrelfish so far suggests that some kind of inter-core notification mechanism is an important complement to polled message-passing support. The fact that we can access interrupt pins on cores remotely on the SCC is very nice in this regard, but even better would be some kind of fast user-space control transfer. One option is to introduce address space identifiers (these should be orthogonal to virtualization in any case), and cause a lightweight same-address-space jump if and only if that address space is running.

D. *System Interface*

A number of different approaches are possible for communication across a PCIe bus, such as using only a single driver in the host PC which controls all memory operations on the SCC. While this approach is both simpler and more resource efficient than our current implementation, the lack of notification mechanism from the SCC to the host PC makes it untractable. The host PC has to poll every possible message channel in SCC memory, incurring a huge performance overhead. The double proxy approach reduces this cost by requiring only one channel to be polled.

To be able to better leverage a single image OS across both host and peripheral and facilitate message passing across the PCIe bus, we are missing an efficient notification mechansim from SCC to host. Given PCIe bus latencies, a simple PCI device interrupt would be sufficient.

V. Related Work

Helios [6] introduces the concept of satellite and coordinator kernels to enable heterogeneous computing. Satellite kernels are light-weight run-times that execute on the peripheral and provide a limited set of services. System calls pertaining to functionality implemented in the coordinator kernel are relayed to the host PC. This enables applications and operating system components to run unmodified on the peripheral. Barrelfish has no concept of satellite kernels and follows a fully distributed model.

The standard operating system model for the SCC treats the chip as a cluster of independent machines and runs a complete Linux operating system on each core that communicate via the TCP/IP network protocol [3]. Using TCP/IP to communicate incurs high overheads for unnecessary functionality, such as message fragmentation, replay and sequencing. Furthermore, executing a complete OS on each core has high memory overhead, while complicating global resource management, which has to be carried out by coarse-grained user-level cluster resource management software.

VI. Conclusion

We have demonstrated that it is possible to run a single image OS across a heterogeneous, non-cache-coherent machine consisting of an SCC and its host PC with reasonable performance (the remaining issues with our PCIe interconnect driver notwithstanding).

Our experience so far has provided insight into messaging performance on the SCC when the on-tile message buffers have to be multiplexed by an operating system. Barrelfish is a work in progress, and we believe that we can improve on the performance numbers presented here. Nevertheless, we feel our SCC experience provides useful insights for future designs of more OS-friendly message-passing hardware.

In addition to performance, asymmetry and latency will continue to be issues for future multicore architectures. We are looking at using improved scheduling to address this challenge, using the SCC port of Barrelfish as a research vehicle. For example, threads that communicate or synchronize frequently should not be placed on cores separated by a high latency link, and the cost of transferring a program and its working set to a different core should not outweigh the expected speed-up.

References

[1] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *International Solid-State Circuits Conference*, Feb. 2010, pp. 108–109.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems," in *Proceedings of the 22nd ACM Symposium on Operating System Principles*, Oct. 2009.

[3] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight communications on intel's single-chip cloud computer processor," *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 73–83, February 2011.

[4] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 306–318, 1992.

[5] J. Ousterhout, "Scheduling techniques for concurrent systems," in *IEEE Distributed Computer Systems*, 1982.

[6] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, "Helios: heterogeneous multiprocessing with satellite kernels," in *Proceedings of the 22nd ACM Symposium on Operating System Principles*, 2009, pp. 221–234.