Efficient memory copy operations on the 48-core Intel SCC processor

van Tol, M.W.; Bakker, R.; Verstraaten, M.; Grelck, C.; Jesshope, C.R.

Publication date
2011

Document Version
Author accepted manuscript

Published in
3rd Many-core Applications Research Community (MARC) Symposium

Link to publication

Citation for published version (APA):
van Tol, M. W., Bakker, R., Verstraaten, M., Grelck, C., & Jesshope, C. R. (2011). Efficient memory copy operations on the 48-core Intel SCC processor. In D. Göhringer, M. Hübner, & J. Becker (Eds.), *3rd Many-core Applications Research Community (MARC) Symposium* (pp. 13-18). KIT Scientific Publishing. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023937

Efficient Memory Copy Operations on the 48-core Intel SCC Processor

Michiel W. van Tol, Roy Bakker, Merijn Verstraaten, Clemens Grelck and Chris R. Jesshope
Informatics Institute, University of Amsterdam
Sciencepark 904, 1098 XH Amsterdam, The Netherlands

***Abstract*—The Single-chip Cloud Computer (SCC) is a 48-core experimental processor created by Intel Labs targeting the many-core research community. It has hardware support for sending short messages between cores, while large messages have to go through off-chip shared memory. However, memory copy operations on this chip are expensive and inefficient. In this paper we provide insight in the SCC's memory architecture and describe and evaluate a few memory copy methods. We propose a novel method, unique to the SCC, which we believe achieves the maximum possible throughput for a single core on this chip. In order to efficiently implement this approach we introduce dedicated cores that run a memory copy service which can be used asynchronously by other cores.**

I. Introduction

The Single-chip Cloud Computer (SCC) experimental processor [1] is a 48-core *concept vehicle* created by Intel Labs as a platform for many-core software research. It provides an on-chip message passing network, a non cache-coherent off-chip shared memory and dynamic frequency and voltage scaling. We are investigating possible implementations on this platform of SVP [2], a hierarchical concurrent execution model, and S-Net [3], an asynchronous stream processing coordination language. The dataflow-style execution properties of both models would provide us with a handle for adaptive power management.

As the SCC effectively is an on-chip distributed system, we can already run the two available distributed implementations [4], [5] of the models without any modification. As these are based on more coarse grained communication primitives such as TCP/IP sockets and MPI, we plan to rewrite them to optimally use the hardware messaging support on the SCC. However, the on-chip message passing buffers are only efficient for relatively small messages; up to 8KB using RCCE [6], or 128KB using the pipelined iRCCE [7] approach. This is sufficient for many message passing programs that only need to communicate small updates on every iteration, but in SVP and S-Net we potentially move a lot more data around between cores. Therefore we have investigated efficient ways to copy large pieces of data on the SCC.

In this paper we make an analysis of several approaches to implement efficient memory copy operations between cores on the SCC. We do this by first giving an overview of its relevant hardware features and performance properties (Section II), then we discuss existing communication libraries and several approaches using different memory access methods in Section III. We present a novel method to copy memory, unique to the SCC, in Section IV. We believe that this method achieves the highest possible throughput for a single core copying blocks of memory larger than 256KB. This requires dedicated *copy cores* which support copy offloading that we discuss in Section V. We conclude with comparing the discussed approaches and our experience with the SCC platform in Section VI.

II. SCC platform

The SCC is a single chip with 48 Intel IA-32 P54C cores connected by an on-chip mesh network which has a 256 GB/s bisection bandwidth [8]. Its features are intended to allow CPUs scaling up to hundreds and potentially thousands of cores. The mesh is organized as six voltage islands containing four tiles with two cores per tile, creating a 6x4 mesh with 24 tiles total. Each tile has its own mesh router to access the mesh for memory access and inter-core communication.

The basic communication paradigm for the SCC is message passing, and therefore each tile has a local 16KB Message Passing Buffer (MPB). The MPB is suitable for sending short messages between cores. A special library (RCCE) to easily use the MPBs to send/receive messages is available, as well as an MPI channel implementation. With two cores per tile, each core has an 8KB area in the local MPB only by convention, as all MPBs are memory mapped and can be addressed directly by each core.

The chip features extensive frequency and voltage control on a per tile and voltage island basis. For all measurements in this paper the cores were clocked at 533 MHz, and the mesh network and memory controllers at 800 MHz.

A. Caches

Each core has both a 16KB L1 data and an instruction cache which cache the complete address space, including the MPBs. Therefore an extra memory type for MPB data (MPBT) was added to the virtual memory system, together with an instruction to invalidate all cachelines in the L1 D-cache that are flagged with this type. As the P54C core originally only supports a single outstanding write request, a Write Combine Buffer (WCB) has been added to combine adjacent writes up to a whole cacheline which can then be written back at once. However, this is only used for MPBT flagged writes.

Each SCC core has a private unified 256KB L2 cache which does not feature a cache coherency protocol. Also, in contrast to the L1 cache, there is no native way to flush or invalidate the

L2 cache; the WBINVD/INVD instructions that can be used to flush or invalidate L1 do not affect the L2 cache. To solve the cacheability issues, users can turn off the L2 cache on a per-core basis. It is also possible to set the cacheability for each individual virtual memory page. This can be done with the standard PCD cache disable flag to disable both caches, or with the MPBT flag to bypass L2 and use the WCB. The L2 cache can be reset separately from the core using a special control register, which initializes all lines into invalid state. However, this operation halts the core and therefore can not be used to invalidate the cache during execution.

Both the L1 and the L2 are 4-way set associative with a cacheline size of 32 bytes, are *write back*, and do not allocate on write miss, i.e. are *write around*.

B. *Memory structure and look-up tables*

The individual P54C cores have a 32 bit core-physical address space, while the chip has four DDR3 memory controllers which each use 34 bits addressing. Combining the 34 bits with the memory controller address, the system can address 64GB in total. The translation of core-physical addresses to system-physical addresses is done through look-up tables (LUTs). Each core has a private LUT with 256 entries, where each entry covers 16MB of the 4GB core-physical address space, which we refer to as a *LUT page*.

The translation is done by indexing the LUT with the highest 8 bits of a core-physical address, and then extending it to form a 34 bit address and adding routing information for the mesh network. An entry can map to a special memory anywhere on the chip or to an addresses on any of the four memory controllers. The MPB and System Configuration Registers of each tile are examples of such special on-chip memories, but also the LUT itself. The LUTs are usually set up when booting a core, but can be changed dynamically when the system is running, having effect immediately. This allows sharing data between cores without having to copy it. A core can map system-physical memory used by any other core at the granularity of these 16MB LUT pages.

C. *Memory subsystem properties*

In the standard configuration, each memory controller runs at 800 MHz, corresponding with a theoretical peak transfer rate of 6.4 GB/s. Figure 1 shows the measurements of the maximum memory throughput for a single core. This benchmark reads or writes data in 4-byte operations from/to memory areas ranging in size from 8KB to 2MB, where each area is prefetched before every measurement. This shows a reading and writing bandwidth around 400 MB/s for the L1 cache for size 8-16KB, 285 MB/s for reading the L2 cache at size 32-256KB and 107 MB/s for reading external memory. Writes to the L2 cache perform at 116 MB/s, or 130 MB/s when using the write-combine buffer (WCB) by flagging the data as MPBT. Writing to external memory shows the real benefit of the WCB where MPBT flagged data is written at 125 MB/s against 22 MB/s for non-MPBT writes. MPBT tagged reads do not benefit from the L2 cache as it is bypassed, but therefore perform slightly better than non MPBT writes to memory.

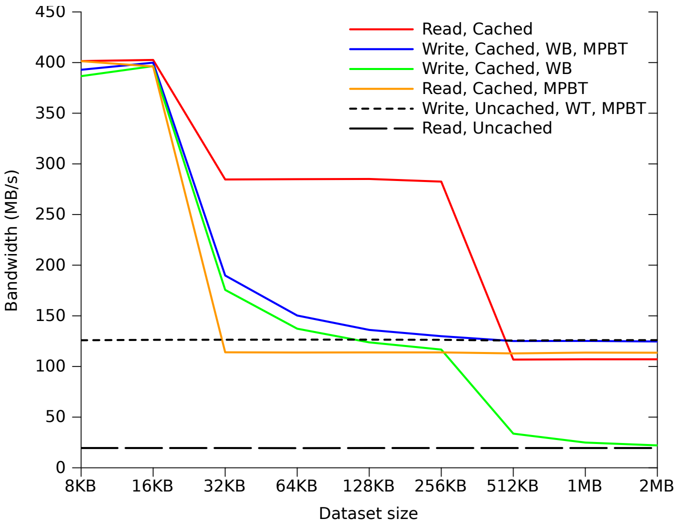

Figure 1. Memory throughput benchmark result of one core reading and writing prefetched memory areas of several sizes, showing the effect of different memory flags.

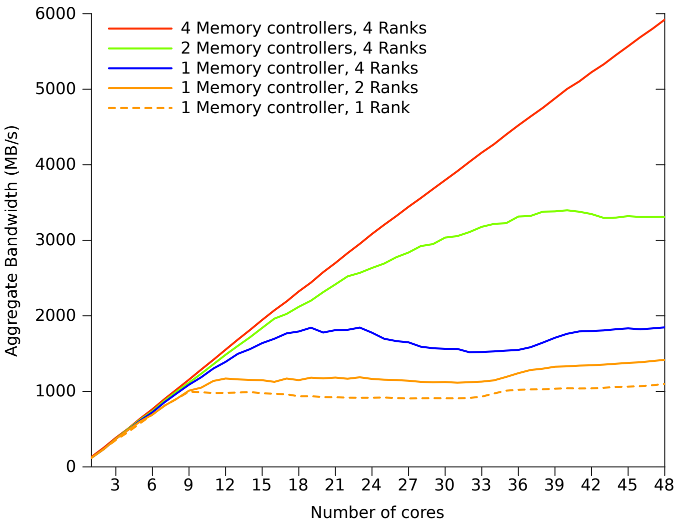

Figure 2. Memory throughput benchmark result showing the aggregate memory bandwidth of 1 to 48 cores performing memory reads divided over a different number of memory controllers and ranks.

The results in Figure 1 show us that a single core can get nowhere near saturating a memory controller. Figure 2 shows how many cores are required to saturate one or more memory controllers, or conversely, how many cores can access the same controller without a large impact on performance. Each memory controller has two banks that each consist of two ranks, and requests to different banks and ranks are interleaved [9]. However, the controller does not interleave the address space between the four ranks. The default LUT mappings map the main memory address space of a core to a contiguous physical address range on a single controller, mapping it to only a single rank. Rank and bank conflicts impact memory performance, so we measured what it takes to saturate a rank, a bank, the whole controller, two controllers, and all four controllers.

Figure 2 shows us that a single rank is saturated by 9 cores

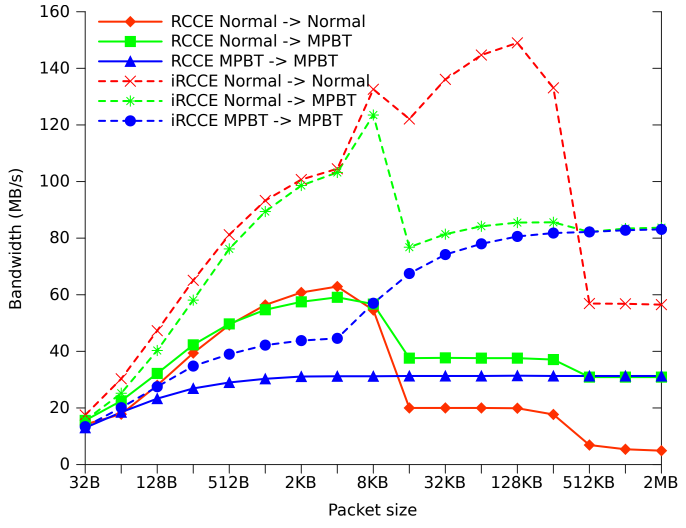

Figure 3. Pingpong benchmark results using the RCCE and iRCCE communication libraries for a range of packet sizes, showing the effect of different memory flags on send and receive buffers.

at an aggregated bandwidth of 1.0 GB/s. A whole controller saturates at 19 cores delivering 1.8 GB/s, two controllers at 40 cores with 3.4 GB/s and all the 48 cores together are unable to saturate the four memory controllers, scaling linearly to deliver a peak of 5.9 GB/s.

III. Memory Copy Operations

A. RCCE and iRCCE

The initial intuitive approach to copy data from one core to another is to use the on chip message passing buffers. This can be done by using the supplied RCCE [10], [6] framework. However, as we can see in [6], as well as in Figure 3 where this is shown as *RCCE normal->normal*, it has its peak performance at 60 MB/s only for 4KB messages. For larger messages up to 256KB it drops to around 20 MB/s and for even larger messages the performance collapses to 5 MB/s.

iRCCE [7] was developed by RWTH Aachen to improve RCCE performance. It uses pipelining when sending messages larger than the MPB so that the read/write operations do not happen in lockstep, and prefetches the target addresses into the L2 cache so that data is not suffering from *write around* for every 4 bytes. A specialized memcpy function is used to copy data to and from the MPBs. This improves throughput a factor of two compared to RCCE, and with messages larger than 4KB the pipelining gives an even greater advantage. At 128KB messages the peak performance is reached, around 145 MB/s. However, as soon as messages are larger than the L2 cache, the performance drops to 60 MB/s. Figure 3 shows these results in the plot as *iRCCE normal->normal*.

Section II-C showed us that using MPBT flagged memory accesses can improve throughput, therefore we ran two more experiments with RCCE and iRCCE. In the first experiment we flag the receive buffer as MPBT type memory, and in the second both the send and the receive buffers. Figure 3 shows that this improves the performance for iRCCE messages larger than 256KB to a throughput of 84.9 MB/s, and improves the throughput of standard RCCE from 5 to 31 MB/s. As MPBT flagged operations bypass the L2 cache, it shows worse performance in our measurements than the normal RCCE/iRCCE for messages between 4KB and 256KB. However, this is a red herring; the pingpong test sends the same message back and forth many times, effectively measuring the throughput when the message data is already present in the cache. If the data is not in the cache, the MPBT modification would outperform normal iRCCE, even for small messages, similarly as it does for messages larger than 256KB.

Another issue with these message passing approaches is that it keeps two cores busy to copy data while they can not perform any computation. This is less of an issue in message passing based SPMD programming paradigms, such as MPI, where usually all processes alternate between a computation and communication step, but it is inefficient for our dataflow style approaches. In our case it is more beneficial for the sending core to copy the data into a shared memory location and then asynchronously signal the receiving core that the data is ready to be used. In the meantime, the receiving core can potentially do other useful work without being tied up in the copy process. Furthermore, when we are sending large messages, data comes from memory, goes through the MPB and is then written back to memory again by another core. Therefore, we expect that we can be more efficient with a direct memory to memory copy.

B. Optimizing memcpy

Naive memory copy operations with the standard *memcpy* function performs very poorly on the SCC at only 17.4 MB/s. This is because it does not take the cache hierarchy into account which has a *write around* policy on a write miss. When copying data to a new location, it is not likely to be in the cache, which means that every 4 byte chunk is individually written to memory instead of whole cachelines of 32 bytes at a time. Also, the read data is unnecessarily cached in both L1 and L2 cache, while only the spatial locality of a single cacheline will help for the copy throughput, as the SCC does not use prefetching.

To improve memory write performance, the SCC has the WCB but this is only enabled for MPBT flagged data. As this can be set per virtual memory page it can be used to force normal memory writes to use the WCB. By applying this to the target buffer we achieved a dramatic improvement in throughput up to 69.4 MB/s. Besides the largest advantage that comes from having a single 32 byte burst instead of 8 individual 4 byte writes on the memory bus, this also removes the delay of accessing (and then writing around as it is a miss) the L2 cache. The latter inspired us to a further optimization; by flagging the input buffer as MPBT as well, the data is loaded directly into the L1 cache. This still gives us the locality advantage for subsequent reads, but removes the delay of allocating in L2, then moving and allocating in L1. Using this approach we measured a throughput of up to 70.9 MB/s.

We also investigated the optimized memcpy implementation that was developed for iRCCE. It moves data through two registers taking advantage of the dual-issue pipeline in the

P54C core, in contrast to the standard memcpy which uses the special repetition prefix and 4 byte copy instruction. This apparently has its advantages when copying data into the MPB, but not when copying from memory to memory. In the best case, again achieved by flagging the source and target memory as MPBT, it reaches a throughput of 49.7 MB/s.

C. L2 Cache flush

An important issue when communicating through shared memory is the L2 cache. As the SCC does not have support for flushing or invalidating the L2 cache, care has to be taken when the receiver wants to use the data copied by the sender. There might be address conflicts in the L1 and L2 cache causing the receiver to read stale data. This is not a problem for the L1 as it can be flushed with an instruction, but the L2 can only be flushed by making sure that the contents of the entire cache are replaced, i.e. reading in 256KB of *clean* data. This is not an issue on the sending side, assuming we use the MPBT based memcpy method; MPBT flagged data does not go to the L2 cache, and even the L1 cache can be turned off for the target addresses at as good as no performance hit as shown earlier in Section II-C. Of course the sender needs to make sure that the data it wants to send is not in dirty state in its own L2 cache, so it would require a flush before starting the copy operation, unless it can be absolutely certain that it can not be in dirty state in its cache. Note that the L1 cache can be used as write through, but the L2 cache can only operate as write back.

We have worked on optimizing the L2 cache flush routine by using on-tile MPB mapped addresses to flush the cache. As the MPB is mapped into a single LUT-page, it only occupies a small portion of the 16MB address space. Memory accesses beyond the MPB within the LUT-page wrap around back into the MPB as the logic probably ignores the higher address bits. However, from the perspective of the L2 cache, these are still unique addresses, so they can be used to flush the cache. By using a 256KB region beyond the MPB to avoid interference with normal MPB accesses, we improved the flush operation from over 1 million cycles to around 580K cycles. Of course this is still a heavy impact on performance, as this takes slightly more then 1 ms.

IV. LUT Based Copy

The disadvantage of both the message passing and memcpy approaches discussed in the previous section is that all data needs to go through the core, being read and written in 4 byte quantities. The L1 cache and the WCB alleviates the problem slightly so that beyond that point only 32-byte cachelines are transferred, but still 8 read and 8 write operations have to be performed individually by the pipeline of the core to transfer the contents of each line.

The SCC has a unique feature with the programmable LUTs which do a second layer of address translation outside the boundary of each core. As this happens transparently to the core, this property can be exploited to efficiently duplicate large blocks of memory. If two cores need to share data and it is guaranteed to be used read-only, the receiving core can simply map the memory anywhere (on a 16MB granularity) into its address space by adding the correct LUT entries and the sender only needs to make sure that the off-chip memory is up to date with a cache flush.

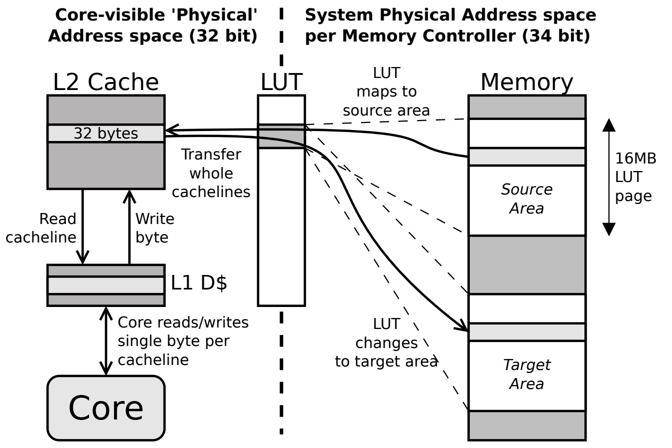
Core-visible 'Physical' Address space (32 bit)
System Physical Address space per Memory Controller (34 bit)
L2 Cache
LUT
Memory
LUT maps to source area
32 bytes
Transfer whole cachelines
16MB LUT page
Source Area
Read cacheline
Write byte
L1 D$
Core reads/writes single byte per cacheline
LUT changes to target area
Target Area
Core

Figure 4. Memory copy operation using L2 cache and LUT remapping

Memory duplication of large blocks using the programmable LUTs works as follows, and is illustrated in Figure 4; the core reads in the first block of 256KB into the L2 cache. It only needs to read a single byte per cacheline to fill the entire cache. While doing this, it makes sure that every cacheline is put in modified state by writing each byte it reads back again to the L2 cache. The content of the data is unchanged, but the L2 is unable to tell the difference. After the cache is filled with modified lines, the core switches the LUT entry for that core-physical address range to map to the destination range in the system-physical address space. The core-physical addresses that are present in the L2 cache now map onto other system-physical addresses. Data is then pushed out of the cache, for example with a cache flush, and is therefore copied to the target location.

When a multiple of 256KB blocks needs to be transferred, the copy procedure can be pipelined; Using a second address range in the core-physical address space (i.e. another LUT entry) the next block of 256KB is read in while it pushes out the previous block to the target address range instead of the cache flush. This needs to be a second core-physical range as the first range is still used to push data out, and can therefore not yet be reused. For the block after that, the first entry can be used again, and so on, alternatingly using the two LUT entries until all data has been copied.

We believe that this method achieves the highest possible throughput for memory copy operations on a single core, which we measured at 73 MB/s for large blocks. Similarly to memcpy using MPBT flagged data only whole cachelines are transferred from and to the memory controllers, most effectively using the powerful on-chip network [8]. The advantage it has over the memcpy approach is that only a single byte per cacheline needs to pass through the core itself, resulting in less read and write operations to copy a cacheline. Still a whole cacheline is transferred between L2 and L1, but by using L1 in write-through mode, only a single byte is written back from L1 to L2. Unfortunately the L1 can not be bypassed completely.

The proposed approach has a few downsides. First of all the target address needs to be aligned at the same offset within a 16MB LUT page within the system-physical address range, though it does not have to be on the same memory controller. This restriction can be compensated somewhat by the use of virtual memory mappings, but as these map using a 4KB page granularity, you still have to use the same offset within such a 4KB page. A second issue is that both the source and target areas need to be contiguous blocks in physical memory, or at least at a granularity of the L2 cache which is 256KB. Otherwise, conflicts would happen and as a side effect data outside the source area would be copied as well as. A third disadvantage is that care has to be taken that the L2 cache is not influenced during a copy operation. This can be done by using the MPBT flags to bypass L2 for all other memory used by the program to avoid interference, however this impacts performance. To keep this manageable, we can use dedicated cores to which we delegate these memory copy operations.

V. Dedicated Copy Cores

We introduce dedicated *copy cores* that run a memory copy service. These cores can be asynchronously messaged by writing meta-data into their MPB and sending them an interrupt. This uses a mailbox protocol similar to what the Barrelfish developers proposed in their report [11]. A message tells a copy core to initiate a memory copy operation of a given size between two addresses, and to send a notification on completion, but not necessarily back to the requesting core. As this requires minimal functionality, it can be implemented as a small efficient kernel running directly on the *bare metal* hardware, likely even fitting completely in the 16KB I-cache. This makes it very suitable for our LUT based copy approach that we just described, as it will have no cache interactions. Furthermore, it has the advantage that we can completely control the virtual memory types used, and therefore guarantee that for any memory interaction required to run the code, the L2 cache is bypassed and therefore remains untouched. A second advantage is that such a kernel does not suffer from the penalty of around 2K cycles to switch between kernel and user mode on receiving interrupts, making message delivery much cheaper taking around 600 cycles [11]. These cores are not limited to using the LUT based copy approach, but can implement any memcpy method discussed previously.

In an SPMD setting, these copy cores are probably not of much use. In most cases, it would be better for the performance to have the core participate in the computation than to have it copy memory for other cores. However, this is not the case in our dataflow style runtimes. Work might not be divided as evenly as with SPMD, and in order to make more progress at a critical point, having the aid of another core to copy memory can be very beneficial. For example, if core A requires a copy of a range of memory in order to start a new computation on core B, it can ask copy core C to copy in the background while A continues working, and C can notify B directly when it is done. Furthermore, if a large amount of data needs to be transferred, A could split the range and employ more than a single copy core to copy the data, exploiting data parallelism.

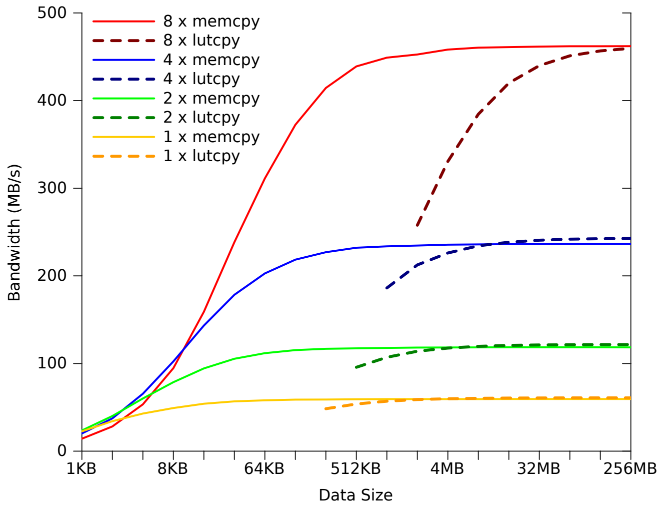

Figure 5. Memory copy throughput benchmark using 1 to 8 *copy cores* to copy data using either MPBT flagged *memcpy* or the LUT based *lutcpy* copy method.

This then delivers more throughput, as a single core can get nowhere near saturating the bandwidth of a mesh network link or a memory controller. We measured that it takes 9 cores to saturate a single bank, single rank on a memory controller.

A. Benchmark

Figure 5 presents some preliminary results of a copy core implementation. This implementation is currently still running under SCC Linux and uses polling instead of interrupts to receive messages, but it gives us an initial idea of the results that can be achieved with the techniques that we have described. Our implementation supports both the LUT based copy operation, *lutcpy*, and a *memcpy* operation that uses MPBT flagged source and destination areas which delivers the best memory throughput as discussed in Section II-C. It should be noted that the LUT based copy method will not reliably copy all the data in this environment, but the measurements will still show the correct performance characteristics. Furthermore, *lutcpy* can only be used to copy data larger than 256KB per copy core.

The results in Figure 5 show that *lutcpy* is only marginally faster than *memcpy* on very large copy operations. *memcpy* outperforms *lutcpy* because it uses uncached MPBT flagged memory for its target address range, which means it does not have to perform an expensive L2 cache flush at the end of the operation. For larger copy operations this becomes a less dominant factor for *lutcpy*, and then it slightly outperforms *memcpy* in the way originally expected.

The most important result of our copy core benchmark is that the technique scales very well. Even for very small piece of data, such as 4KB, the copy operation benefits from being split across two copy cores. However, this would still be slower than performing the operation locally due to the messaging latency. The advantage starts at 8KB, where two copy cores together deliver 78.7 MB/s, that is including messaging overhead, whereas the requesting core itself would only be able to copy at 70.9 MB/s, see Section III-B.

VI. Conclusion

In this paper we have surveyed several options for efficiently copying memory on the Intel SCC. Using the RCCE or iRCCE message passing implementations has the advantage that it does not require cache flushes, but has the disadvantage that two cores are occupied in the copy process. It originally has a low throughput for regions that are larger than the MPB and/or L2 caches. We then showed how the standard *memcpy* operation can be improved a four-fold by enabling the use of the WCB with MPBT flags on virtual memory pages, and that this also improves message passing performance.

We proposed a novel approach to copy memory, unique to the SCC platform, by switching LUT entries to copy data with the L2 cache using less interaction with the core. It performed 3% faster than the fastest memcpy method in our initial results, and we argued that this achieves the highest possible throughput for a single P54C core. The disadvantage of this approach is that it is cumbersome to use, with restricted alignments and sizes for the data that is copied.

Complementary to the LUT based copy method, but orthogonally to the improved memcpy approach, we proposed the introduction of *copy cores* to be able to asynchronously offload copy operations, similar to DMA engines. We then showed the benchmark results of a preliminary copy core implementation, comparing MPBT flagged memcpy and LUT based copy. Offloading to copy cores scales very well, but the LUT based copy performance was not as good as we expected. This is because a copy core requires an L2 cache flush at the end of a LUT based copy operation, which is not required for MPBT flagged memcpy.

The largest bottleneck for reading/writing memory, and therefore also for inter-core message communication, as this involves reading/writing an MPB, is the fact that a P54C core can only have a single outstanding memory operation, and stalls until it completes. For read operations this is partially alleviated by the L1 cache as a few consecutive reads will hit in the same lines. For write operations the WCB can be used, however, as it is only enabled for MPBT flagged data this is not easy. A solution to this could be to add one or more programmable DMA engines capable of having multiple outstanding memory requests to the platform. They would be more simple than a P54C core, but could achieve a much higher memory throughput. Now we require the combination of multiple copy cores to achieve a higher throughput when copying a large amount of data, possibly wasting precious computing cycles. And to generally have a more optimal memory and communication performance in a runtime system, it needs to fully manage both virtual and physical memory, applying MPBT flags where necessary to speed up operations. This is what we are currently planning to do in the future for our SVP and S-Net runtimes on the SCC.

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. V. D. Wijngaart, and T. Mattson, “A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS,” pp. 108–109, February 2010.

[2] C. R. Jesshope, “A model for the design and programming of multi-cores,” *Advances in Parallel Computing*, vol. High Performance Computing and Grids in Action, no. 16, pp. 37–55, 2008.

[3] C. Grelck, S.-B. Scholz, and A. Shafarenko, “A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components,” *Parallel Processing Letters*, vol. 18, no. 2, pp. 221–237, 2008.

[4] M. W. van Tol and J. Koivisto, “Extending and implementing the self-adaptive virtual processor for distributed memory architectures,” *CoRR*, vol. abs/1104.3876, April 2011.

[5] C. Grelck, J. Julku, and F. Penczek, “Distributed S-Net: High-level message passing without the hassle,” in *1st ACM SIGPLAN Workshop on Advances in Message Passing (AMP’10), Toronto, Canada, 2010* (G. Bronevetsky, C. Ding, S.-B. Scholz, and M. Strout, eds.), ACM Press, New York City, New York, USA, 2010.

[6] 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.

[7] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, “Evaluation and improvements of programming models for the Intel SCC many-core processor,” in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS2011) – to appear, Workshop on New Algorithms and Programming Models for the Manycore Era (APMM)*, (Istanbul, Turkey), July 2011.

[8] P. Salihundam, S. Jain, T. Jacob, S. Kumar, V. Erraguntla, Y. Hoskote, S. Vangal, G. Ruhl, and N. Borkar, “A 2 Tb/s 6×4 mesh network for a Single-chip Cloud Computer with DVFS in 45 nm CMOS,” *Solid-State Circuits, IEEE Journal of*, vol. 46, pp. 757–766, April 2011.

[9] Intel, “SCC extended architecture specification,” November 2010. Revision 1.1.

[10] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, “The 48-core SCC processor: the programmer’s view,” in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’10, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.

[11] S. Peter, T. Roscoe, and A. Baumann, “Barrelfish on the Intel Single-chip Cloud Computer,” Tech. Rep. Barrelfish Technical Note 005, ETH Zurich, September 2010. http://www.barrelfish.org.