Characterization and analysis of pipelined applications on the Intel SCC

Tommaso Cucinotta,Vivek Subramanian
Real-Time Systems(ReTiS) Lab
Scuola Superiore Sant'Anna
Pisa, Italy.
cucinotta@sssup.it, vivek@retis.sssup.it

***Abstract*—Many-core computing platforms can be used to parallelize computations by dividing the data to be processed into smaller chunks and processing them simultaneously on different cores. One possible approach in such parallelization is to set up a pipeline such that each smaller chunk of data passes in turn through all the processors involved. In this paper we examine some approaches to set up such a pipeline on the Intel SCC. We use a combination of the message passing and the shared memory capability of the SCC hardware through the interfaces provided by the RCCE library for our implementation. We build a model to analyze and compare the performance of such pipelines by measuring the total time for computation. This model is used to illustrate the effects of type of memory scheme used, ordering of cores in the pipeline and caching.**

***Index Terms*—pipeline, real-time streaming, message passing, shared memory, SCC**

I. Introduction

THE parallelization of a computing task is a well-studied problem. There are several approaches and methods to achieve parallelization depending on the nature of the computing task. One of them is the pipeline approach that may be applicable in situations where input data may be divided into smaller fragments, each of which must have a certain set of operations carried out in a specified order. This is particularly effective whenever the data to be processed is available progressively, for example, in multimedia streaming applications, where pipelining the application results in a higher sustainable throughput.

In several instances the current operation to be carried out on a certain fragment is not dependent on either the result of the previous or the subsequent fragment. Also, the computing task maybe divided to be carried out at different processing elements. As an example; an audio-processing application that processes an input audio file for streaming, or DES encoding of an input file. The operations carried out at each processing element could be different (e.g the audio-processing application, where a different filter is used each time) or identical (e.g the cryptographic application). Setting up a pipeline for processing such tasks serves to reduce the amount of computing that a single processing element needs to perform before switching to the next data chunk, thus increasing the possible throughput, or reducing the computing requirements on a single processing element.

There are several factors that affect the performance of a pipeline such as the latency in memory accesses and the overheads involved in moving data between processing elements. If the processing elements communicate over an interconnect network, then simultaneous use of the network, by more than one processing elements results in contention for bandwidth which affects the performance of the pipeline.

In this work, we introduce a set of variables and equations to describe the pipeline. We use experimental implementations to help validate these models of the pipeline. The aim of these experiments is to have a method to build models of the various building blocks of the pipeline and of the pipeline itself. These may then be used to gauge the expected performance of pipeline and this, in turn, be used to guide the process of actual implementation and deployment that leads to gains in throughput and processing times.

II. Related Work

The availability of the fast message passing buffers on the SCC allow for using these for inter-core communications. The RCCE library [2] provides a framework to implement message-passing on the SCC, however a number of authors addressed the problem of efficient inter-core communications on the SCC. For example, Rotta [8] presents design options for message passing protocols and discusses them. Villa et al.[11][11]study the efficiency and scalability of barrier synchronization in NoC-based many-core systems. The NoC-based architecture of the SCC that uses the mesh-network to access the off-chip RAM presents challenges introduced by this additional latency. Verstraaten et al. [4] presents methods to implement memory copy mechanisms aimed at increasing the throughput. Abts et al.[1]explores issues in placement of memory-controllers and the effect on latency. Petrot et al.[6]present a software-based solution for cache coherency and memory consistency in NoC-based multiprocessors. Prell and Rauber [7] address methods for achieving task parallelism on the Intel SCC using runtime task schedulers. Kierstscher et al. [10] present the effects of MPI applications having knowledge of the topology, while Tol et al. [3] discuss the mapping of a distributed implementation of the S-Net on the SCC. Bo et al.[9]discusses the optimization of data-parallel operations in the context of many-core platforms. Papagiannis and Nikoloppoulos [5] examines bottlenecks in scalability of the MapReduce algorithm and presents an implementation of the same for the SCC.

III. Preliminaries

A. Modeling memory access

Consider the Intel SCC which uses a tile-based architecture with a mesh NoC that connects the tiles and the memory controllers. Each tile has two cores, their caches and a small local memory (the local memory buffer or the message passing buffer).Each core on the SCC is assigned space in the message passing buffer (MPB). The RCCE library uses this buffer to implement message passing between the cores.

Let $mpb(i,b)$ denote the time taken by a core i to write b bytes of data into its own MPB. Let $coord(i)$ be the coordinates of the tile that contains core i, such that $coord(i).x$ and $coord(i).y$ indicate, respectively, the x-coordinate and the y-coordinate. Let $dist(i,j)$ denote the routing distance between elements i and j. Note that, the elements may be either cores or memory-controllers. As the SCC uses dimension-ordered routing, we may write $dist(i,j)$as:

$$\begin{aligned} dist(i,j) &= |coord(i).x - coord(j).x| + \\ & \quad |coord(i).y - coord(j).y| \end{aligned} \tag{1}$$

If the data rate of the links of the NoC are denoted by μ, then the time taken to transfer b bytes from i to j can be expressed as t_t:

$$t_t(i,j,b) = \frac{dist(i,j) \cdot b}{\mu} \tag{2}$$

The above expression assumes that only a single transfer is happening over the set of links. We assume this simplistic way to model the memory access and further assume that it would be an upper bound on the time it takes to access memory in the worst scenario in this simple case. This might not always be the situation and there may be more than one core using the same links of the NoC. In this case, the effective data rate may be lower (see VII).

Define a function $mem(i)$ similar to $coord(i)$, but instead of indicating the coordinates of i, $mem(i)$, indicates the coordinates of the memory controller that has the private memory of i. Similarly define $shmem(i)$ to indicate the coordinates of the memory controller that has the shared memory that i is using.

B. Modeling message passing

The RCCE library provides synchronous blocking *send()* and *receive()* interfaces for transferring messages between cores. The *send()* method accepts the rank of the core that is the destination and the *receive()* method accepts the rank of the core that it expects to receive a message from. These calls have to be matched - for every send executed to j from i, j must execute a matching receive from i.

RCCE implements this mechanism such that the sending core writes the message from its private memory to the MPB, and signals the destination core. The destination core reads the message from the source's MPB (via the lookup table entries) and stores into its own private memory. Thus, a send and receive operation consists of one off-chip memory read by the source, one write by the source to its own MPB, one read of the source's MPB by the destination and one write to the off-chip memory by the destination. For a message of size b bytes, we may write this time taken as t_m:

$$\begin{aligned} t_m(i,j,b) &= t_t(i, mem(i), b) + \\ & \quad mpb(i,b) + \\ & \quad t_t(i,j,b) + \\ & \quad t_t(j, mem(j), b) \end{aligned} \tag{3}$$

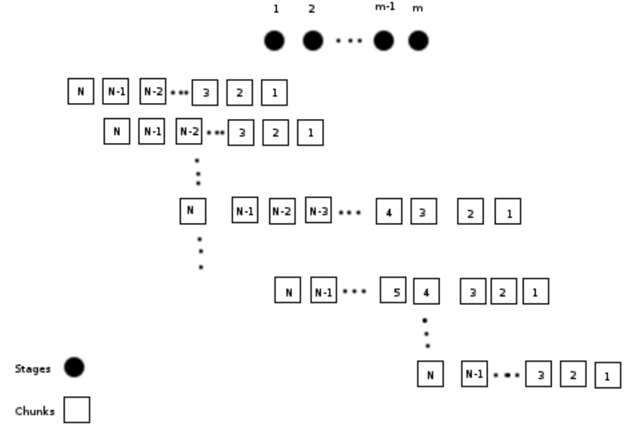

Figure 1. Representation of the pipeline

IV. Pipeline

The pipeline that we consider has several stages through which each chunk of data must be processed. For the purpose of this study, we have kept the operation performed at each stage to be identical. Also, a single core is mapped to exactly one stage in the pipeline. Each stage in the pipeline does the following:

- receive a single chunk of data from the previous stage
- perform the operation on that chunk
- send the chunk to the next stage in the pipeline

Since we use the RCCE library for message passing and synchronization, the send and receive steps are synchronous. Thus, all the cores are almost simultaneously doing one of the three steps described above. The first and the last stages of the pipeline are slightly different from the other intermediate stages - the first stage instead of receiving a chunk, reads a chunk from the input buffer and, the last stage instead of sending a chunk forward, writes to an output buffer. Figure 1 shows a representation of the pipeline.

The pipeline has a set of parameters associated with it:

- D is the total size of data (in bytes) to be processed by the pipeline.
- C is the size of each chunk (in bytes)
- $N = D/C$ is the number of chunks
- m is the number of stages in the pipeline
- Z is the size of a token
- $t_c(i,b)$ is the time take to compute b bytes at stage i - each compute step is a read from memory (private or shared), process and write to memory(private or shared).

- $T_{pipeline}(D, C, m)$denotes the time taken by a pipeline with m stages to process D bytes of input in chunks of C bytes.

We have implemented the pipeline using two of the memory types available on the Intel SCC. The following subsections describe each of these approaches.

A. Private memory

The private memory of a core is visible and accessible only to that core. In the implementation of the pipeline using private memory all the buffers that each stage uses are allocated in the private memory using the standard *malloc()*. The chunks are sent from one stage to the next using the *send()* and *receive()* methods of the RCCE library. Denote the time taken to process the nth chunk of size C bytes at stage i by $t_p(i, C)$, as:

$$\begin{aligned} t_p(i, C) &= t_m(i-1, i, C) + \\ &\quad t_c(i, C) + \\ &\quad t_m(i, i+1, C) \end{aligned} \tag{4}$$

For the pipeline to proceed, one chunk at a time must be processed and placed in the output buffer. For this to happen, $m-1$ messages have to be sent (or received) and since the messaging is synchronous we need to consider only either the time for sending or receiving - the time spent sending in at stage $i-1$ will be equal to the time spent in receiving at i. When the chunk from the last stage is placed in the output buffer, exactly one more new chunk may be admitted at the first stage. Thus, the maximum time spent at each step of the pipeline in processing is the time taken by the stage that has the maximum $t_c(i, C)$. That is, if the pipeline were to be stalled - some stage is in the processing step, while the stages before this stalled stage are waiting to send and the ones after the stalled stage are waiting to receive - then the pipeline would progress only when the stalled stage finishes processing and sends the chunk on to the next stage. In the time that this stage took to process the current chunk, all the other stages would have processed exactly only one chunk. Hence, we may write the time taken for pipeline to complete as:

$$\begin{aligned} T_{pipeline}(D, C, m) &= N \cdot \left[\sum_{i=1}^{m-1} t_m(i, i+1, C) \right] + \\ &\quad (N+m) \cdot max \left\{ t_c(i, C) \right\}_{i=1}^{m} \end{aligned} \tag{5}$$

$$\begin{aligned} \leq \quad & N \cdot (m-1) \cdot max \left\{ t_m(i, i+1, C) \right\}_{i=1}^{m-1} + \\ & (N+m) \cdot max \left\{ t_c(i, C) \right\}_{i=1}^{m} \end{aligned} \tag{6}$$

B. Shared memory

The SCC provides a shared memory area that can be accessed by all cores on the platform. We use this shared memory as one of the ways to implement the pipeline. In this case,the first core allocates the space in shared memory for the buffers and sends the offset from the start of the shared memory region to the other cores. One of the buffers that is created in the shared memory is a queue. Access to each slot in the queue is managed using tokens (denoted a Z) which the first stage initially generates. A stage may only access the slot to which it holds the token. The first stage reads data in chunks from the input buffer into the slots of the queue and the last stage reads out data from the queue into the output buffer. Unlike in the previous method where the entire chunk was transferred over the NoC, here only the token is transferred from stage to stage. Due to the synchronous nature of communication, the number of slots in the queue has an upper bound equal to $m-1$.

$$\begin{aligned} t_p(i, C) &= t_m(i-1, i, Z) + \\ &\quad t_c(i, C) + \\ &\quad t_m(i, i+1, Z) \end{aligned} \tag{7}$$

As reasoned for the private memory case, in this implementation as well a similar reasoning can be applied. The differences are that since the queue is of a circular nature (due to a ring created by the passing of tokens among the stages), the number of messages at each step of the pipeline is m (the last stage sends the token to the first stage). Also, the stages move along the queue of chunks as opposed to the chunks moving from one stage to another.

$$\begin{aligned} T_{pipeline}(D, C, m) &= N \cdot \left[\sum_{i=1}^{m-1} t_m(i, i+1, Z) \right] + \\ &\quad N \cdot t_m(m, 1, Z) + \\ &\quad (N+m) \cdot max \left\{ t_c(i, C) \right\}_{i=1}^{m} \end{aligned} \tag{8}$$

$$\begin{aligned} \leq \quad & N \cdot m \cdot max \left\{ t_m(i, i+1, Z)_{i=1}^{m-1}, t_m(m, 1, Z) \right\} + \\ & (N+m) \cdot max \left\{ t_c(i, C) \right\}_{i=1}^{m} \end{aligned} \tag{9}$$

1) Uncached and cached shared memory: Shared memory on the Intel SCC can be made uncached - bypasses both the L1 and L2 cache, or cached - cached in both L1 and L2, by setting the relevant bits appropriately for the corresponding page table entries. RCCE provides a mechanism to achieve this at the time of compiling the library. The uncached and the cached shared memory is exposed, respectively, through the */dev/rckncm* and */dev/rckdcm* devices. Depending on how the library was compiled the shared memory is mapped to one of these devices. The RCCE interfaces to handle this memory does not change significantly.

Although, using the cached shared memory has the advantage of reducing the number of memory accesses during computation, it currently comes with the overhead of having to flush the entire cache on the core to ensure consistency of the shared memory. A core must flush caches before begining to read from shared memory and must flush after modifying data in shared memory. As a result, the number of flush operations is proportional to the number of chunks that the input data is split into times the number of cores participating.

C. Effects of ordering of cores

The trival method to order cores would be in order of their physical core ID. Though on the average, the distance between tiles is about as small as it can get, the distance is quite large at certain points (for instance, from core 11 to core 12 and from core 23 to core 24). Some advantage could be gained if the ordering of cores and mapping of pipeline stages were done in such a manner so as to keep the routing distances as small as possible. One possible method is to start at some corner (say tile $(0, 0)$) and then move in the direction of increasing x-coordinate values, then step up to the next y-coordinate, and move in the direction of decreasing x-coordinate, and follow this method till the last core. The gains from following such reordering is only significant if the time spent in message passing itself is comparable to the time spent in processing, and further if the differnce in latency in messaging cores with different routing distances itself is significantly appreciable. Nevertheless, some small gains are to be expected by reducing the routing distances between stages of the pipeline.

V. Experimental Results

All experiments were performed on the SCC with 32GB of off-chip memory. The SCC system was configured with the cores running at *533Mhz*, the mesh at *800Mhz* and the DDR at *800Mhz*. The RCCE library was compiled with the non-gory interfaces and without the power-management options enabled. The -DSHMADD flag was enabled to for increasing the size of the available shared memory. The LUT entries corresponding to the shared memory were re-arranged such that the first 15 entries pointed to shared memory on the memory bank connected to tile (0,0), the next 15 to the bank connected to (5,0), the next 15 to the bank connected to (0,2) and the last 15 to the bank connected to (5,3). The processing done on each core was a placeholder operation that simply incremented the value read in the input by 1 and wrote it back. The applications on the core were run at real-time priority by using SCHED_FIFO with a priority of 20.This section presents sample results from the experiments we have performed on the described setup.

Figure 2 plots the time taken to access (read and write) 4MB of shared memory from each of the cores against the distance (in terms of dimension-ordered routing) of the core from the memory controller. We see that the latency varies almost linearly with increasing distance with respect to each of the four controllers.

Figure 3 and Figure 4 are for the private memory based implementation. We expect the time to process the input to be linearly dependent on the size from (6). The bumps in Figure 2 are at $C = 16KB$ are possibly due to interference from the L1 cache - since we would expect the previous chunk's data (and marked 'dirty') to be residing in the cache when the current chunk is being processed, thus accesses to the current chunk's data would cause evictions in the cache resulting in a write-back of the evicted data into memory.

Figure 5 and Figure 6 are using an uncached shared memory based model. As expected, there is a linear increase of total processing times but is independent of chunk sizes, since every

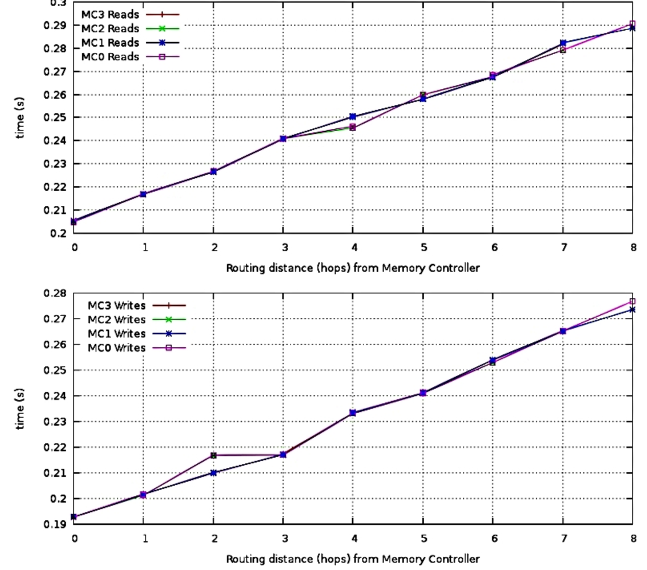

Figure 2. Read and write access times to shared memory for 4MB

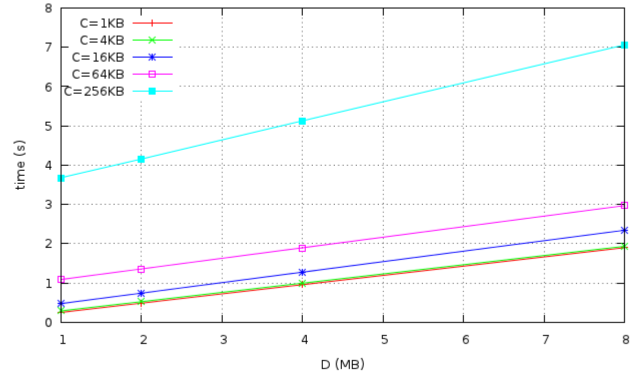

Figure 3. D vs $T_{pipeline}(D, C, m)$ with $m = 48$ for a private-memory implementation and a trivial ordering of core by ascending physical ID

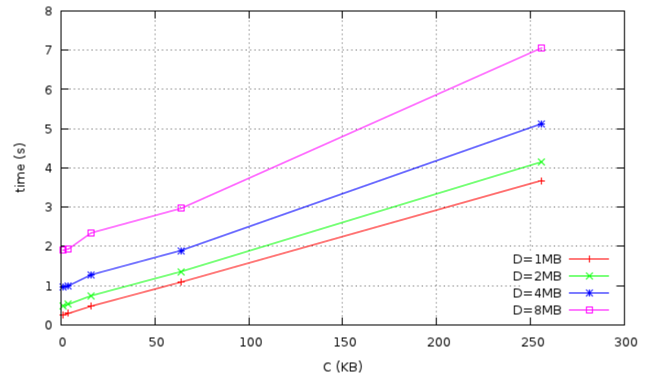

Figure 4. C vs $T_{pipeline}(D, C, m)$ with $m = 48$ for a private-memory implementation and a trivial ordering of core by ascending physical ID

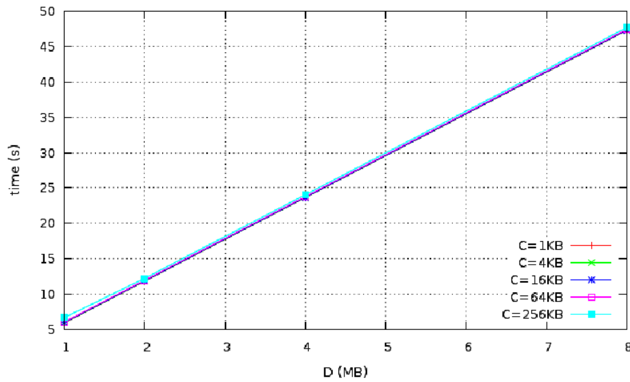

Figure 5. D vs $T_{pipeline}(D, C, m)$ with $m = 48$ for a uncached shared memory implementation and a trivial ordering of core by ascending physical ID

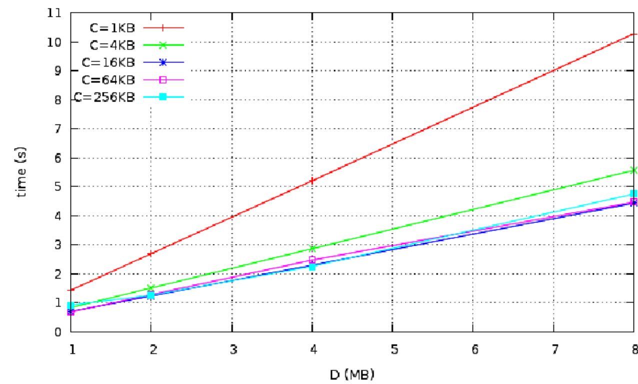

Figure 7. D vs $T_{pipeline}(D, C, m)$ with $m = 48$ for a cached shared memory implementation and a trivial ordering of core by ascending physical ID

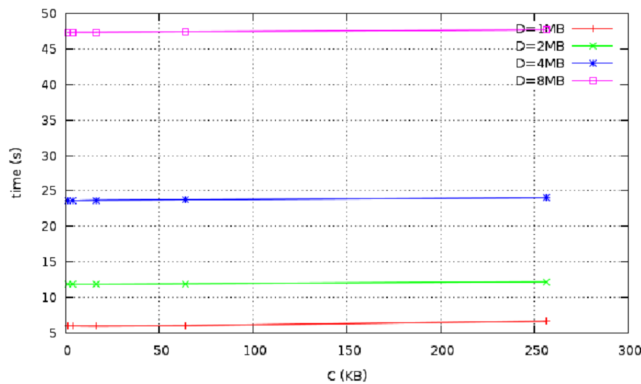

Figure 6. C vs $T_{pipeline}(D, C, m)$ with $m = 48$ for a uncached shared memory implementation and a trivial ordering of core by ascending physical ID

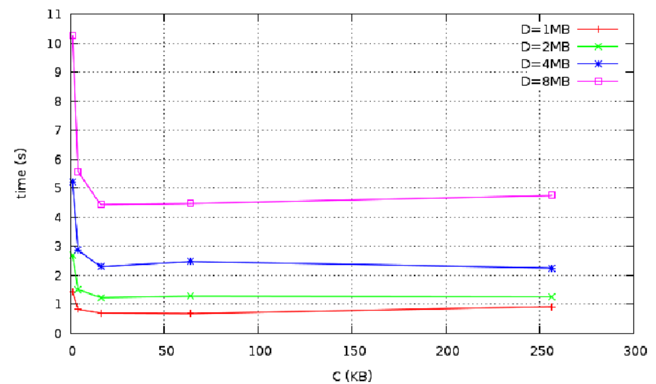

Figure 8. C vs $T_{pipeline}(D, C, m)$ with $m = 48$ for a cached shared memory implementation and a trivial ordering of core by ascending physical ID

access is from the off-chip memory. Also, in Figure 6 we see that the total time for processing doubles with doubling of input data size.

Figure 7 and Figure 8 refer to the cached shared memory based model. Here the nature of the plot is due to the additional time taken in explicity flushing caches before reading from shared memory to prevent reading stale data from the cache and after writing to shared memory to ensure changes are flushed from the cache to the memory. The flushing causes the entire cache to be flushed. For every chunk that a core processes, two flush operations are needed, hence, doubling the chunk size (hence, halving the number of chunks) halves the number of cache flushes needed.

Figure 9 and Figure 10 compare the performance of the three approches. The current limitation in the cache flushing causes the cached shared memory implementation to be somewhat worse than the private memory implementation for smaller chunk sizes, due to a large number of flushing operations. But, for larger chunk size it performs better.

We compared the times for two orderings of the cores which are:

1) A *trival* ordering in order of physical ID of cores - 0,1,2,3,4,....,45,46,47.
2) A reduced inter-core distance ordering denoted by *min-routing* - 0,1,2,...,9,10,11,23,22,21,...,14,13,12,24,25,26,...,33,34, 35,47,46,45,...,38,37,36

Table I lists the comparison of total processing time for the three memory models based on the ordering of cores and with $C = 256KB$. Though there are gains, the effect is most noticeable in the case of the cached shared memory implementation.

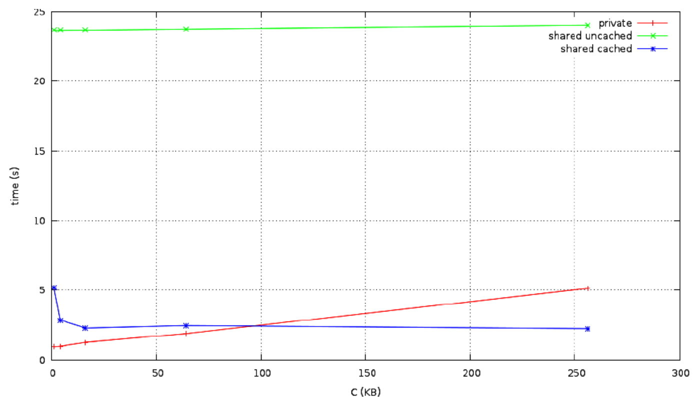

Figure 9. C vs $T_{pipeline}(D, C, m)$ with $m = 48$ and $D = 4MB$ - comparison

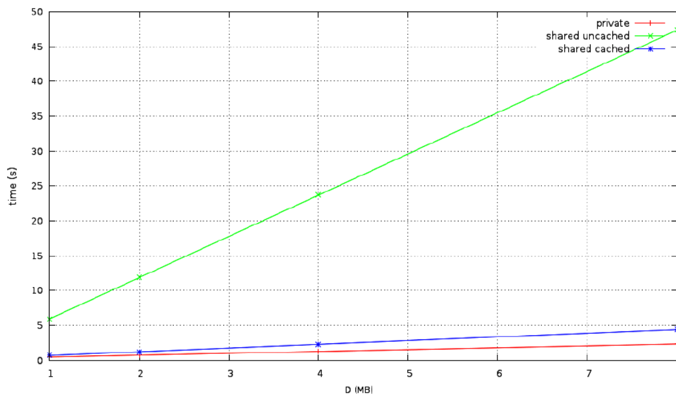

Figure 10. D vs $T_{pipeline}(D, C, m)$ with $m = 48$and $C = 16KB$ - comparison

$D(MB)$	$T_{pipeline}(D, 256KB, 48)$(sec)	
	Trivial ordering	Min-routing ordering
	Private memory	
1	3.678	3.670
2	4.155	4.151
4	5.128	5.117
8	7.060	7.054
	Shared uncached memory	
1	6.635	6.553
2	12.200	12.118
4	24.031	23.948
8	47.693	47.611
	Shared cached memory	
1	0.920	0.868
2	1.256	1.143
4	2.252	2.042
8	4.745	4.313

Table I
COMPARISON OF TOTAL PROCESSING TIMES AGAINST ORDERING AND MEMORY-MODEL

VI. CONCLUSIONS

From our experiments we concluded that in deploying pipelined applications factors of memory access latencies need to be accounted for improving the performance of computation. Further, it will help in improving performance by running data-intensive stages on cores that are closer to memory.

The ordering of the cores also shows effect on the total computation time. Ordering of the stages that such that stages that have a bulk of inter-stage communication requirements on cores that are close will lead to better computation times. In this work, in our experimental pipeline the communication pattern between stages is near-identical and quite predictable.

The simplistic assumptions we have used for memory and communication give us some estimation as to the performance of the pipeline, these need to be generalized further to improve the capabilities (See VII).

Though using messaging to transfer data between stages of the pipeline performs better than the shared memory (cached) approach, a finer method of just flushing only modified locations from the cache to the memory will greatly improve performance.

VII. FUTURE WORK

In the future we plan to investigate methods to dynamically map stages of a pipeline to cores based on constraints such as the maximum acceptable end-to-end delay or latency of the application. We will also measure how using the on-die TCP/IP communication affects the performance. Further we will explore the effect of the degree of parallelization of stages in the pipeline and methods to incorporate this factor into how stages are mapped onto cores.

In this work we assume a very simple model of the memory access, we plan to consider cases with concurrent accesses by different cores and the load this imposes on the NoC to be able to be more accurate in estimating performance. Also, the communication pattern between stages currently is near-identical and quite predictable. This is usually not the case in a general deployment and we will explore ways to model such general cases. This will enable us to create a more realistic model which will help further in the task of dynamic mapping of stages to cores.

VIII. ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Community's Seventh Framework Programme under grant agreements n.248465, in the context of the S(o)OS Projects.

REFERENCES

[1] Dennis Abts, Natalie D. Enright Jerger, John Kim, Dan Gibson, and Mikko H. Lipasti. Achieving predictable performance through better memory controller placement in many-core cmps. In *Proceedings of the 36th annual international symposium on Computer architecture*. ACM, 2009.

[2] Tim Mattson and Rob van der Wijngaart. *RCCE: a Small Library for Many-Core Communication*. Intel Corporation.

[3] Michiel Van Tol Roy Bakker Merijn Verstraaten, Clemens Grelck and Chris Jesshope. Mapping Distributed S-Net on the 48-core Intel SCC processor. In *3rd Many-core Applications Research Community (MARC) Symposium*, 2011.

[4] Merijn Verstraaten Clemens Grelck Michiel W. Van Tol, Roy Bakker and Chris R. Jesshope. Efficient Memory Copy Operations on the 48-core Intel SCC Processor. In *3rd Many-core Applications Research Community (MARC) Symposium*, 2011.

[5] Anastasios Papagiannis and Dimitrios S. Nikolopoulos. Scalable Runtime Support for Data-Intensive Applications on the Single-Chip Cloud Computer. In *3rd Many-core Applications Research Community (MARC) Symposium*, 2011.

[6] F. Petrot, A. Greiner, and P. Gomez. On cache coherency and memory consistency issues in noc based shared memory multiprocessor soc architectures. In *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, 2006.

[7] Andreas Prell and Thomas Rauber. Task Parallelism on the SCC. In *3rd Many-core Applications Research Community (MARC) Symposium*, 2011.

[8] Randolf Rotta. On Efficient Message Passing on the Intel SCC. In *3rd Many-core Applications Research Community (MARC) Symposium*, 2011.

[9] Byoungro So, Anwar M. Ghuloum, and Youfeng Wu. Optimizing data parallel operations on many-core platforms. Intel Corporation.

[10] Simon Kiertscher Steffen Christgau and Bettina Schnor. The Benefit of Topology-Awareness of MPI Applications on the Intel SCC. In *3rd Many-core Applications Research Community (MARC) Symposium*, 2011.

[11] Oreste Villa, Gianluca Palermo, and Cristina Silvano. Efficiency and scalability of barrier synchronization on noc based many-core architectures. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 2008.