A Hierarchical Memory Management for a Load-Balancing Stream Processing Middleware on Tiled Architectures *

Nilesh Karavadara, Michael Zolda, Vu Thien Nga Nguyen, Raimund Kirner

University of Hertfordshire
n.karavadara, m.zolda, v.t.nguyen, r.kirner@herts.ac.uk

Abstract. Tiled many-core computer architectures are becoming increasingly popular, providing a viable solution to the complexity of resource management in parallel processors. One of the critical challenges in programming such tiled many-core architectures is the efficient use of available resources.
In this paper we present a hierarchical memory management approach for tiled many-core processors. This memory management approach is capable to provide shared memory across multiple OS instances running on different cores. This memory management approach made it possible for us to port LPEL, a dynamic load-balancing middleware for stream processing applications, to the Intel Single-Chip Cloud Computer (SCC), a research processor that shares many similarities with other tiled architectures. This is the first execution middleware running on the SCC that provides dynamic load balancing. An evaluation shows that our framework performs better than an MPI-based implementation.

1 Introduction

The demand for compute power is exceeding the supply [20]. Until recently, increasing the processor clock speed to meet the demand of performance had worked well, but heat and interference caused by increased clock speeds and shrinking size of transistors are starting to limit processor designs [5, 4].

The current strategy is to use several simple and power efficient cores [27] instead of a single complex and power-hungry core. Processors such as the Intel Xeon Phi [7], ARM (Cortex A7 & A15) [12], Tilera (Tile-Mx & Tile-Gx) [10, 25] and Kalray MPPA 256 [18] reflect a trend towards tiled architectures.

Parallel architectures with multi-core tends to be more energy-efficient than an equivalent single-core processor. However, programming multi-core processors is more complex and requires the refactoring of algorithms for concurrency. The identification and exposition of concurrency is not enough to exploit a high fraction of the potential computing power made available by a parallel platform. The scheduling of dynamic

* This work has been supported by the Material Transfer Agreement 2010-2013 for the Intel SCC Research Processor "Light-weight Parallel Execution Layer for Stream Processing Networks on Distributed Memory Architectures" and the FP7 ARTEMIS-JU research project "ConstRaint and Application driven Framework for Tailoring Embedded Real-time Systems" (CRAFTERS) under contract no 295371.

workload on these tiled platforms increases the difficulty in utilising available resources. Automatic load-balancing reduces the waste of resources and relieves the programmer.

Dataflow programming [17] is a particularly promising model for concurrent programming. Data flow languages expose concurrency directly through explicit modelling of data dependencies, in contrast to most traditional programming languages which are centered around control flow. *Coordination languages* [8] allow software engineers to build parallel applications from sequential building blocks. *Stream processing* [28] is a parallel execution model that is well-suited for architectures with multiple computational elements that are connected by a network. Put together, these mechanisms afford a powerful software development approach for multi-cores [14, 13, 24].

In this paper we introduce a hierarchical memory management approach for tiled many-core processors that provides shared memory across multiple OS instances. Like the Hoard memory manager [3] and scalloc [1] our memory management approach uses local memory pools per core. In contrast to Hoard and scalloc, ours also works with separate OS instances. Based on that memory manager we ported the *Light-weight Parallel Execution Layer* (LPEL) [24, 21] to the Intel SCC research processor, making it the first execution middleware running on the SCC with dynamic load balancing.

Sections 2.1 and 2.2 of this paper offer a brief review of the SCC tiled architecture. Sections 2.3 and 2.4 provide a short review of the stream programming paradigm and the stream execution model. Section 3 describes how our middleware maps streaming network on the SCC tiled architecture. In Section 4 we study the issue of caching and compare our middleware with an existing MPI [2] implementation. We discuss relevant related work in Section 5 and conclude with Section 6.

2 Preliminaries

2.1 The SCC Architecture

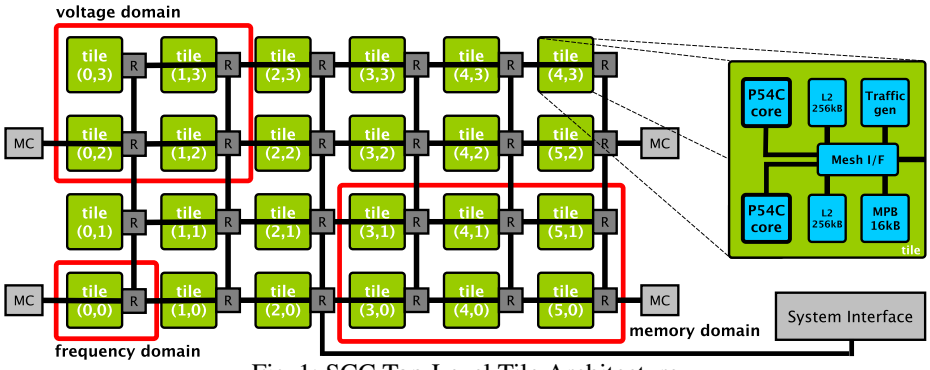

Fig. 1: SCC Top-Level Tile Architecture

The SCC [15, 26, 16] is an experimental tiled multi-core processor created by INTEL. The processor consists of 24 tiles in a 4x6 grid, connected by a high bandwidth, low latency, on-die 2D mesh network, resembling a cluster on a single chip, as shown in Fig. 1. Each tile hosts two modified P54C processor cores that support x86 compilers

and operating systems. Each core has 32 kB L1 and 256 kB L2 cache. Furthermore, each tile has a 16 kB block of SRAM called Message Passing Buffer (MPB), which is physically distributed, but logically shared. Each tile connects to the mesh network via a router.There is a Voltage Regulator Controller (VRC) to let programs dynamically manage the voltage and the frequency of cores, and four on-die Memory Controllers (MC), which support a total of 16 to 64 GiB off-die DRAM. Only one atomic test-and-set register and two atomic counter registers are available per core via the system interface, which is a limiting factor for efficient synchronisation.

The SCC is a research processor, but many of its features are found in commercial processors, for example in Tilera TILEPro series: In Tilera's architecture, cores are also organised as 2D grid of tiles connected to a mesh via on-tile routers, and each core is capable to run an OS instance. Each tile has 16 kB L1 instruction cache, 8 kB L1 data cache, and a 64 kB combined L2 cache. In contrast to SCC's mesh network, the TILEPro has six independent networks to route traffic to different destination, i.e., tile, tile caches, external memory, and IO Controllers. The Special Purpose Registers (SPRs) are nearly identical to SCC's control register buffer (CRB). There are three memory modes: In default mode the hardware maintains cache coherence, but it does not do so in non-coherent mode, and in non-cacheable mode all the data blocks are not cached at any level. Dynamic Distributed Cached Shared Memory (DDC) on TILEPro serves same purpose as MPB on SCC [30].

2.2 The SCC Memory Architecture

The SCC offers three address spaces:

- A private off-chip address space in DRAM for each core. This memory is cache-coherent with an individual core's L1 and L2 caches.
- A shared off-chip address space in DRAM. This memory can optionally be configured as cached, but ensuring cache-coherence is the programmer's responsibility.
- The MBP, a physically distributed, logically shared address space in SRAM.

Tiles are organised in four *memory domains* of six tiles each. Each memory domain maps to a particular MC. Private memory is accessed through the assigned MC, shared memory can be accessed through any of the four MCs.

Each core has its own 256-entry Lookup Table (LUT) to translate 32-bit core addresses to a 46-bit system addresses. Each core can address 4 GiB of physical memory, even though the SCC supports up-to 64 GiB in total. The LUTs provide a mechanism to translate 32-bit physical core address to 34-bit physical system address. The upper 8 bits of a physical core address index a LUT entry, which contains 22 bits, of which the upper 12 are routing information. The lower 10 bits are prepended to the lower 24 bits of the core address, resulting in the 34-bit memory address. The LUTs are loaded with default values at boot time, but it is possible to change them dynamically.

The interaction of the memory with caches depends on its mapping. The part of the DRAM that is mapped as a shared region between cores can be configured to be cached or uncached. If memory is configured as cached, read/write accesses go through the L1 and L2 caches and manual flushing of the L2 cache is required to commit data to main memory. The SCC does not provide cache coherence, hence concurrent accesses to shared data may cause memory consistency issues in cached mode. If the memory is configured as uncached, read/write accesses go directly to main memory (DRAM).

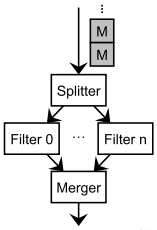

Fig. 2: Image Filter

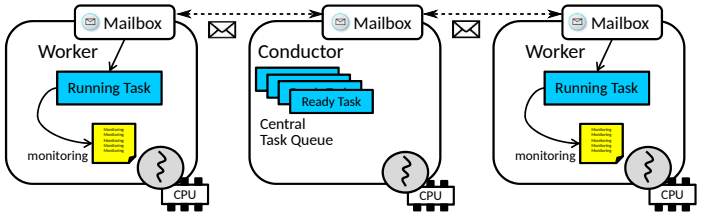

Fig. 3: LPEL Execution Layer

There is a tag for data in the MPB called Message Passing Buffer Type (MPBT) that identifies L1 cache lines. Tagged data bypasses the L2 cache and goes directly to the L1 cache in the case of reads. Write operations to tagged memory are stored in the Write Combine Buffer (WCB) until an entire cache line is filled or a write access to a different cache line happens. Intel has also extended the Instruction Set Architecture (ISA) with an instruction to invalidate all tagged cache lines in L1. Accessing invalidated L1 cache lines forces an update of the L1 cache lines with the data in the shared memory.

2.3 Stream Programming

In stream programming, a program is structured as a set of computation processes called nodes and a set of directed communication channels between them called streams. Stream programs can be viewed as a graphs whose vertices are nodes and whose edges are streams. Streaming data is presented as an infinite sequence of messages. Examples of stream programming can be found in [6, 14, 29].

Fig. 2 shows an example of an S-Net [14] program — an image filter. The *Splitter* node consumes an image and splits it into sub-images. The number of sub-images varies depending on the size of the original image. The sub-images are sent to different branches where *Filter* nodes perform the actual filtering operation. The processed sub-images are sent to the *Merger* node, which combines them into a complete image.

2.4 LPEL - A Stream Execution Layer with Efficient Scheduling

Our streaming middleware includes two layers: a *runtime system* (RTS) and an *execution layer*. At the RTS layer, each stream is represented as a FIFO message buffer and each node of the stream program is transformed into a task. A task is an iterating process that reads messages from its input streams, performs the associated node's computations, and writes output messages to its output streams. The role of the RTS is to enforce the semantic of stream programs, i.e., to ensure that each task reads from and writes to its appropriate streams. The execution layer below the RTS provides primitives for task and stream management and a scheduler that distributes tasks to cores.

The Light-weight Parallel Execution Layer (LPEL) [24] is an execution layer providing user-space threading and communication mechanisms for stream programs on shared memory platforms. It provides functions for creating, reading, writing and modifying streams and a task component to create a wrapper around each node before sending it to the scheduler.

LPEL offers two different schedulers: DS-LPEL uses a global mapper to allocate tasks to cores and a local scheduler for each core. The local scheduling policy is round-robin, whereas the mapper uses either a round-robin policy or a static mapping [24].

HRC-LPEL follows a centralised approach of automatic load balancing [21], where one *conductor* core is dedicated to manage the set of ready tasks; cf. Fig. 3.

3 LPEL on the SCC

To obtain good performance in terms of throughput and latency, the HRC-LPEL scheduler uses the notion of data demands on streams to derive the task priority that is used to decide which task will be executed on available core. We deployed HRC-LPEL on the SCC, as it has been shown to be more efficient than the DS-LPEL [21]. Furthermore, it better suits our future plan to extend the scheduler with power management features.

HRC-LPEL requires shared memory to efficiently move tasks and their associated states between the workers, however the SCC is by default a distributed memory platform: Each core runs separate OS instance.

Although the SCC offers a fast network between the cores and on-chip shared memory (MPB), the default configuration does not offer enough shared memory. The limited number of hardware locks also makes it difficult to deploy HRC-LPEL on the SCC, as we need at least one lock per stream. Software mechanisms like mutexes from the POSIX thread library are not safe to use on distributed memory platforms.

For all these reasons we re-configure the SCC as a shared memory platform.

3.1 Shared Memory Creation

We use the LUT entries described in Section 2.2 to configure the SCC such that it behaves as a shared memory platform. There are 256 LUT entries, of which 0–40 are used by OS that is running on the core. Entries 192–255 are mapped to the MPBs and configuration registers. This leaves LUT entries 41–191 unused.

To create shared memory we improve upon technique used in the RCCE [31] library. In RCCE, 4 LUT entries are mapped to same physical address-space range on all the cores. As each LUT entry points to a 16 MiB segment of physical memory, this mapping provides 64 MiB of memory that is shared between all cores. There are two problems with this approach: Firstly, 64 MiB are not enough to deploy HRC-LPEL and run some real-world application. The lack of shared memory can be addressed using the remaining LUT entries. As each entry points to a 16 MiB chunk of memory this provides us approx 2.5 GiB of shared memory. As mentioned before, 41 of the 256 entries point to physical memory needed by the individual OS instances. To obtain more shared memory we disable 4 of the 48 cores and use entries from those cores to populate unused entries of LUT.

The second problem is that with memory mapped as described in RCCE you get a globally visible shared memory, but the virtual address range is not the same for all the cores. In this case, pointers are not globally valid. Using offsets from the beginning of the address-space instead would introduce an additional overhead. We solve the problem by mapping the address-space to same virtual address range on all cores.

Calling standard malloc will allocate space in private rather than shared region of memory. We have written our own malloc and free functions that are based on K&R malloc and free [19] to address the issue.

3.2 Shared Memory Management

By using LUT entries to create shared memory, all cores get the same view on the memory. There are multiple ways we can allocate this shared memory to cores.

The first approach is to have a global allocator that allocates memory to each core as and when requested. As shared memory is global each core has to grab the lock, allocate memory and release lock. The lock is necessary as we do not want meta-data within the memory allocator to be corrupted due to simultaneous accesses by multiple cores. This creates unnecessary contention and adversely impacts the performance.

In the second approach, the global memory is divided into chunks of equal size and then each core can locally manage its chunk. The problem with this approach is that not all tasks need the same amount of memory. If we distribute equally-sized chunks of memory to all the cores, we waste resources.

To alleviate this problem we can fuse the first and second approach to create a hybrid allocator. We can have a global allocator that allocates chunks of memory as and when required by participating cores and then cores allocate memory locally from these chunks. When cores do not have enough memory to fulfil the next request for memory allocation, it will request another chunk from the global allocator.

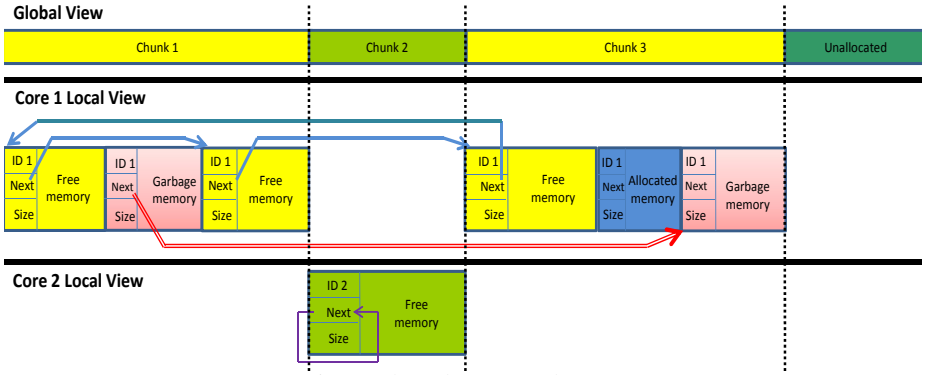

Fig. 4: Shared memory layout

Figure 4 depicts the view of shared memory from perspective of different cores. When a core makes a request to the global allocator it receives a chunk of memory. The chunks that are managed by a core may not be contiguous. Thus the core keeps its free storage as a circular list of free blocks, e.g., core 1 manages chunk 1 and 3. If we consider core 1's view at the chunks then both chunks are divided into small multiple blocks. The local allocator manages two lists; first to keep track of free storage, known as *free list*. Second list is *garbage list* to keep track of garbage storage that needs to be added back to free list. Each block contains a header indicating its size, a pointer to the next block, and an owner id, followed by the actual memory.

Algorithm 1 allocates n bytes of memory from local shared memory. The local allocator scc_malloc_local uses a "first fit" algorithm [19] that is not thread-safe. It is therefore protected by a lock (lines 1–3). A return value of NULL means there is no block available from where core can allocate required memory (line 4). The core then requests a new chunk of memory from the global allocator (line 5). In order to ease the contention on global allocator the core always requests m bytes where $m > n$. To make this newly allocated memory chunk available to the requesting core, we call scc_free_local on it so that it gets added to the free list (line 7).

The function scc_malloc_local is called again to allocate memory. If the return value is still NULL that means there is not enough memory available and an error is returned (lines 8–13). All the calls to scc_malloc_local and scc_free_local are protected by a mutex from the POSIX thread library to make them thread-safe.

Algorithm 2 is a simple allocator that keeps track of the size of the global shared memory and the starting point of this memory as meta-data. When a request for memory allocation is made by a core, the global allocator performs three steps: First it checks, if there is enough memory to allocate. If so, it continues to the next step—otherwise it returns an error. Next it uses a lock implemented by using a test-and-set register to avoid any corruption of meta-data. Finally it allocates the required memory block and adjusts the size and starting point of the global shared memory before releasing the lock. This hierarchical malloc means we will also need a hierarchical free.

Algorithm 1 *scc_malloc_local* to allocate n bytes from local shared memory

```
1:  mutex_lock()
2:  memptr ← scc_malloc_local(n)      ▷ K&R malloc
3:  mutex_unlock()

4:  if memprt = NULL then
5:      chunk ← scc_malloc_global(m)
6:      mutex_lock()
7:      scc_free_local(chunk)
8:      memptr ← scc_malloc_local(n)
9:      mutex_unlock()

10: if memprt = NULL then
11:     Not enough memory available, return Error
12: else
13:     Return memptr
```

Algorithm 2 *scc_malloc_global* to allocate m bytes from global shared memory

```
Require: m ≤ available_global_memory
1:  tas_lock()
2:  chunk ← m bytes from global memory
3:  tas_unlock()
4:  Return chunk
```

Algorithm 3 *scc_free_local* to free memory pointed by p

```
1:  if not (shmStart < p < shmEnd) then
2:      standard_free(p)   ▷ p points to private memory
3:      return
4:  if owner id = core id then
5:      mutex_lock()
6:      scc_free_local(p)                ▷ K&R free
7:      mutex_unlock()
8:  else
9:      acr_lock()
10:     add p to garbage list of core with owner id
11:     acr_unlock()
```

Algorithm 4 *scc_free_garbage* to free memory from garbage list

```
1:  acr_lock()
2:  glfirst ← gl          ▷ copy the garbage list gl
3:  gl ← NULL             ▷ empty the garbage list
4:  acr_unlock()

5:  mutex_lock()
6:  while glfirst ≠ NULL do
7:      glnext ← block after glfirst
8:      scc_free_local(glfirst)          ▷ K&R free
9:      glfirst ← glnext
10: mutex_unlock()
```

Algorithms 3 and 4 free the shared memory that was allocated with our own allocator function scc_malloc_local. We know the range of the global shared memory region

and can check if the memory that is being freed is within this shared region or not (line 1). If the memory pointed by p was allocated in private region using standard malloc, then we need to free it using standard free (line 2). If it was allocated in shared region then owner id from memory block and core id are compared (line 4). If the owner id and core id are the same, then we call function scc_free_local, which is the standard free function corresponding to scc_malloc_local [19]. As mentioned earlier calls to scc_free_local are protected by a lock to ensure thread-safe operation (lines 5–7).

Each core maintains a garbage list of blocks to be freed. Access to this garbage list is protected by locks. In case of id mismatch, the core will add the block to the garbage list of the core that allocated the block (lines 9–11).

Algorithm 4 frees the memory blocks that were added to its garbage list by some other cores. This algorithm is executed periodically during scheduling cycle.

Here we use two different locks. The first lock is to protect the garbage list from being corrupted due to the concurrent access by other cores (lines 1,4). We use atomic counter registers of the SCC to implement this lock. The second lock is local to the core to ensure thread-safe operation of scc_malloc_local and scc_free_local by using a pthread mutex (lines 5,10). Once the garbage list is copied and the original list is emptied, we can release the lock so that other cores can start adding memory block to be freed (lines 2,3). Then we loop through copied list and add blocks to free list (lines 6–9) by calling function scc_free_local (line 8).

3.3 Conductor/Worker Initialisation

When deploying the HRC-LPEL scheduler on the SCC, it makes sense to create exactly as many workers as there are cores, as the cores of SCC are single-threaded. As there is no shared memory at the beginning, we can not just create conductor/workers on a single core and then distribute them amongst participating cores. For this purpose, when the execution of a program starts, a configuration file is used to decide which core will be the conductor based on the physical core id.

As mentioned in Section 3.1, to create a truly globally shared memory, all cores have to map part of program's address-space to same virtual address range. At the beginning there is no shared memory, apart from the MPB. Meta-data, including a flag necessary to establish communication between cores is located in a pre-defined location in MPB.

If a core is a conductor, it starts by initialising the shared memory, tasks, streams and the static parts of streaming network. If core is a worker, it will busy-wait on a flag located in MPB. Once the conductor has mapped the LUT entries and created the shared memory, it places the relevant LUT configuration in the MPB and sets the flag. Once the flag is set the worker cores configure their LUTs to map shared memory to same virtual address range as conductor.

HRC-LPEL uses mailboxes to facilitate communication between conductor and workers. Each mailbox is protected by a lock to ensure that no messages are lost. Once the mailboxes are setup, the workers request tasks to execute from the conductor and the conductor will fulfil these requests based on demand and task priority. When there are no more messages to be processed, the conductor sends a termination message to all the workers via mailbox.

3.4 Synchronization primitives

HRC-LPEL requires a number of means to synchronise at different points.

– when initialising, the conductor/workers need a shared flag
– the meta-data of the global shared memory may be accessed by conductor/workers concurrently
– meta-data of the local shared memory needs to be protected against concurrent access by multiple threads
– the mailbox is an example of producer/consumer, where messages are added/removed from queue. This queue needs to be protected against concurrent access to ensure messages are not lost
– streams are used to transfer data/messages between tasks. Streams are implemented as FIFO buffers, and these buffers need protection to ensure integrity of data (during reading/writing to the stream).

We already use all the hardware registers provided by SCC for synchronisation. The MPB is used to store the shared flag. We use the test-and-set registers to implement locks that protect the meta-data of the global shared memory. We use the atomic counter registers to implement locks to protect the garbage list and the mailboxes.

We still need synchronisation primitives to protect the streams and for allocating core local shared memory. For this purpose we use POSIX (pthread) mutexes. The SCC runs an OS instance on each core, so we create mutexes with the process shared attribute set. When different worker threads try to access the same mutex, it will be seen as it was accessed by different processes.

4 Experiments

We evaluate the efficiency of HRC-LPEL with dynamic load balancing on the SCC and compare it to DS-LPEL with manual load balancing. In the latter each core has its local round-robin scheduler, and the cores communicate via MPI. We also evaluate the scalability of HRC-LPEL for varying numbers of cores.

4.1 Experimental Setup

In our experiments we used a default sccKit 1.4.2 configuration, with the cores running SCCLinux at 533MHz, and memory and mesh running at 800MHz. We used the SCCLinux driver for memory mapping.

We used four benchmarks implemented using the S-Net coordination language [14]:

– DES: Encrypts data using DES. This benchmark performs computationally intensive operations on relatively small chunks of data of 2 kB.
– FFT: Calculates a fast fourier transform. This benchmark performs computationally less intensive operation on relatively large chunks of data of 64 kB.
– HIST: Calculates histograms of images. This benchmark performs computationally intensive operations on relatively large chunks of data of 127 kB.
– FILT: Applies a series of filters on images. This benchmark performs computationally intensive operations on relatively large chunks of data of 127 kB.

Each benchmark contains a pipeline performing the application's main function. To increase the level of concurrency, S-Net provides parallel replication to create multiple instances of the pipeline.

We used 4 out of the 48 cores as donors for shared memory, and 4 further cores to model an external source/producer and sink/consumer for stream programs. Since we need at least one conductor and one worker, our baseline is 2 cores.

The SCC does not provide cache coherency and offers no direct control over cache flushing, so we have to ensure consistency when using the cache. We use two variants of HRC-LPEL: In DLB only the task stack—consisting of non-shared data—is cached, whereas in NDLB we do not use caching. For DS-LPEL with manual load balancing MPI is used and memory is not shared, so we can make full use of caching. In this approach, which we denote MLB, each benchmark is mapped to achieve the best load balance, i.e., each instance of the pipeline is mapped on a separate core.

The first core is special: Besides processing messages, it is also responsible for receiving input messages from the environment, distributing messages to the other cores and collecting them, and sending them out to the environment. The MPI communications occur only between the first and all other cores. To ensure the message order, MPI must be used in blocking mode.

4.2 Experimental Results

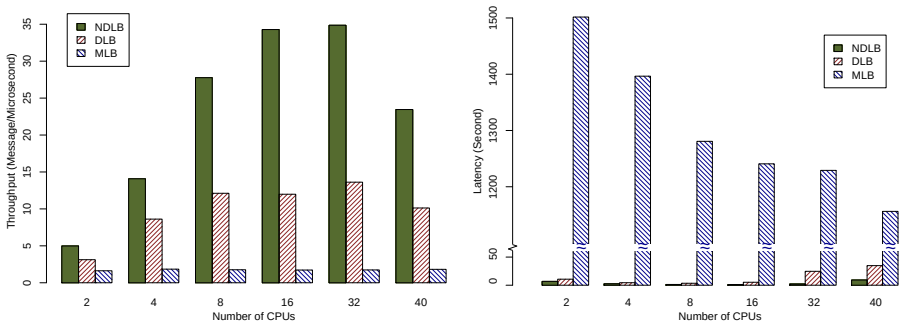

Fig. 5: Performance of FFT on NDLB, DLB and MLB

Fig. 5 shows the maximum throughput and minimum latency of the FFT benchmark. NDLB outperforms DLB by a factor of at least 1.5 for both, throughput and latency, even though caching is disable in NDLB. Since the SCC is configured as a shared memory platform, the caches need to be flushed to ensure data integrity among cores. This causes a significant overhead that caching cannot compensate.

MLB has the lowest throughput, because the communication performance of MPI is inferior to direct memory access. The maximum communication bandwidth between 2 cores is around 2.78 MiB/s for MPI. Transferring 64 kB between 2 cores takes more than 22 ms via MPI but only 15 ms via direct memory access. With 2 cores the throughput for MLB is smaller than for DLB and NDLB, and for more cores the MPI bandwidth is shared. MLB requires one core to communicate with all other cores, sending input messages and receiving output messages. Due to similar load on the cores, this communication is likely to coincide. MPI introduces a (de)serialising and (un)packing overhead and operates in blocking mode and this forces each core to wait while sending messages via the MPI interface. As a result the MLB throughput for MLB can be 7 times smaller than for DLB and 20 times smaller than for NDLB, as shown in Fig. 5.

MLB has a higher latency than NDLB and DLB. Besides the beforementioned reasons, the HRC-LPEL scheduler affords control over the consumption rate of input mes-

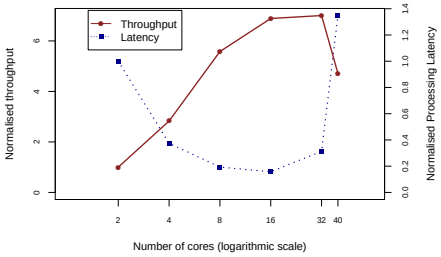

Fig. 6: Scalability of FFT on NDLB

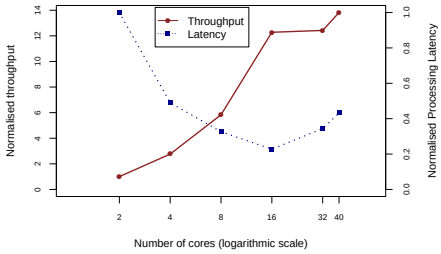

Fig. 7: Scalability of DES on NDLB

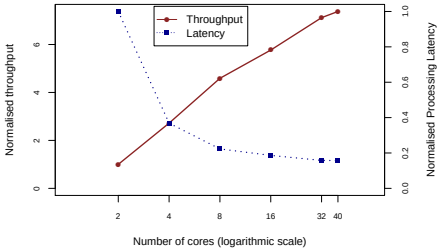

Fig. 8: Scalability of Histogram on NDLB

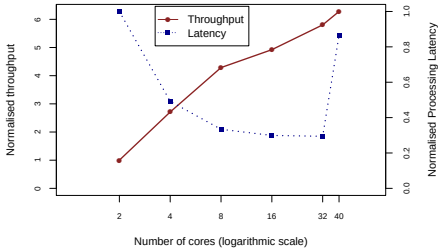

Fig. 9: Scalability of Filter on NDLB

sages to optimise latency [21]. MLB lacks this feature and allows the program to consume input messages even when it is overloaded and unable to process them. The latency for MLB can be 370 and 900 times higher than for DLB and NDLB, respectively.

Fig. 6 shows how NDLB scales for the FFT benchmark. From 2 to 16 cores the throughput scales roughly linearly, but more cores imply more memory accesses. Memory is managed by 4 memory controllers and extensive access can cause contention. Therefore throughput does not scale well between 32 and 40 cores. Although FFT operates on a sizeable amount of data (64 kB), the computation time is relatively small. On average each task takes 65 ms to process a message, so each core must access a large amount of data frequently.

In contrast, DES requires extensive computation on a small amount of data. Each input message is 2 kB and each task takes 194 ms on average to process a message. For this reason, the throughput of DES scales better, as shown in Fig. 7.

The latency depends on the immanent concurrency level of the stream program. Increasing the number of cores takes advantage of the concurrency within the stream program and helps to reduce the latency. However, more cores also imply higher communication costs, as tasks are spread among cores. Figs. 6 and 7 show that the latency decreases when we increase the number of core up to 16. For 32 and 40 cores the communication overhead surpasses the benefit of concurrency. The latency of DES and FFT therefore does not scale well for 32 or 40 cores.

Fig. 8 and Fig. 9 shows throughput and latency for HIST and FILT benchmark respectively. In contrast to DES and FFT here we can see roughly linear scaling in throughput from 2 cores all the way to 40 cores. This was expected, as HIST and FILT are computationally more intensive than DES and FFT. For HIST the latency continues to decrease up to 40 cores. In contrast, a decrease in latency can be observed for FILT

for up to 32 cores, after which it rises sharply. One reason can be the higher number of message queuing at merge point in the stream network, which can be a bottleneck.

Table 1 shows the minimal and maximal execution time for each task in the benchmarks. Some of these tasks have multiple instances occurring in the separate parallel pipelines created by S-Net. We can see that all benchmarks show a considerable variation in execution times of tasks. This can be attributed to high work-load imbalance which depends highly on input messages. These numbers underline the need for a load balancing scheduler like the one we have presented.

The table shows that the <collect> task of the FILT benchmark has nearly 4000% variation on 40 cores (for 32 core run this variation is 495.30%). The <collect> task merges messages from multiple streams and forwards them to the subsequent component. Such a high variation indicates that at some point multiple messages were waiting to be merged, resulting in the sharp increase in latency seen in Fig. 9.

Benchmark	Task	Min	Max	Diff (%)
FFT	initP	1.1232s	1.9954s	77.65s
	stepP	10.3226s	15.2775s	48.00s
HIST	<collect>	0.9477s	1.5814s	66.86s
	<split>	0.8987s	4.2756s	375.74s
	split	3.9660s	5.0427s	27.15s
	calHist	22.3738s	28.7144s	28.34s
FILT	<collect>	0.6123s	27.0231s	4313.34s
	<filter>	0.1787s	0.4919s	175.19s
	<parallel>	0.4500s	1.5947s	254.38s
	<split>	0.3979s	11.5076s	2792.24s
	filt	134.2071s	470.1736s	250.33s
	split	1.0512s	6.3557s	504.62s

Table 1: Minimal and maximal task execution time on 40 cores

5 Related work

Verstraaten's SCC port of S-Net [32], where the core allocation is determined via static user annotations at the S-Net level, is closely related to our approach. In his approach the programmer must manually specify the allocation at compile time, which can be difficult and precludes system-wide load balancing under dynamic demand.

In our approach cores are allocated dynamically at the LPEL level beneath the S-Net runtime system. The approach involves keeping track of the system-wide workload and resource availability, enabling efficient task scheduling to maximise throughput and to reduce latency by dynamic load balancing. Also our approach can easily be extended towards dynamic power management.

The two approaches also differ in overhead. The distributed version of S-Net runs several extra tasks per core, such as the input manager, the output manager and the worker. This incurs extra overhead due to OS-level context switches. Our approach has only worker tasks, which considerably reduces the context switching overhead.

In [1] authors present a memory allocator called *scalloc* that is fast, multicore-scalable and provides low-fragmentation. The allocator is made-up of two parts; a frontend to manage memory in spans and a backend to manage empty spans. spans are same concept as superblocks in Hoard [3]. The spans are organised in 29 different size classes ranging from 16 bytes to 1MB. Any request for memory over 1MB is allocated directly from OS using mmap. Span-pool is a global concurrent data structure that holds spans in different pools using arrays and stack. Each span is used to fulfil memory request in terms of blocks, when all the blocks in span are freed, i.e. span has no allocated block it is returned to span-pool. With regard to memory allocation and deallocation we have similar approach, for example notion of ownership of memory block and separate lists to hold memory blocks that needs to be freed. For example, add block to local free list when allocation was done by same core, or add to remote free list otherwise, in case of *scalloc* it will be threads not cores. The main difference in our approach is that our allocator works across different instance of OS and uses less complex data structure and, can handle allocation bigger than 1MB in size.

In Intel's [9] *Privately Owned Public Shared Memory* (POP-SHM) approach, each core offers some private memory to share data with other cores. For computations, however, the data must be copied to private memory. In contrast, our middleware hides the details of memory management, enabling programming at a high level of abstraction.

Software Managed Cache-coherence (SMC) [33] provides coherent, shared, virtual memory, but it is the responsibility of the programmer to ensure that data is placed in the shared region and that operations to shared data are guarded by release/acquire calls. SMC is a library that provides coherent, shared memory, where as our middleware provides a high-level abstraction that simplifies programming.

MESH [23] is a framework for memory-efficient sharing. It uses remote method invocation to pass access to shared object between cores. The MESH framework uses POP-SHM for shared memory. It provides a higher level of abstraction than POP-SHM, but in contrast to our approach it does not provide a scheduler that is geared toward maximising throughput and reducing latency in streaming applications and relieving the programmer from worrying about load balancing.

Prell et al. [22] have presented an implementation of Go's [11] concurrency constructs on the SCC. Their approach uses Intel's RCCE [31] as communication library and employs work-stealing. The work shows that the implementation failed to scale due to limitation such as the number of simultaneously used channels and the size and number of data items exchanged over channels. In contrast, our middleware provides automatic load balancing and avoids these limitation. Furthermore, our middleware can easily be extended to exploit SCC-specific power management functionality.

6 Summary and Conclusion

We have presented a hierarchical memory management approach for tiled many-core processors. This memory management approach is capable to provide shared memory across multiple OS instances running on different cores. Based on that memory manager we were able to port the *Light-weight Parallel Execution Layer* (LPEL) to the Intel SCC research processor, making it the first execution middleware with dynamic load balancing to run on the SCC. We have studied the abstraction of communication, local cache deployment, and the resource-efficient use of the cores on the SCC research processor, which serves as an example of a tiled many-core architectures.

Our results show that our middleware is superior to an MPI-based implementation in throughput and latency. LPEL relies on multi-threading to offer high-performance lightweight tasks switching, which requires MPI to use TCP-based *sock* channels.

We also found that exploiting local caches is basically limited to non-shared data objects, resulting in inferior performance compared to the non-cached version. Using a cache-coherent tiled architecture may yield better performance. Further work is required to study the influence of cache locality and core interconnect topology for cache-coherent architectures.

References

1. M. Aigner, C. M. Kirsch, M. Lippautz, and A. Sokolova. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. *arXiv preprint arXiv:1503.09006*, 2015.
2. Argonne National Laboratory. Mpich. `http://www.mcs.anl.gov/research/projects/mpi/` accessed 12-Dec-2013.
3. E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000.
4. S. Borkar. Thousand core chips: a technology perspective. In *DAC '07: Proc. 44th annual Design Automation Conference*, pages 746–749, New York, NY, USA, 2007. ACM.
5. S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54:67–77, May 2011.
6. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3), Aug. 2004.
7. G. Chrysos. Intel® xeon phi coprocessor-the architecture. *Intel Whitepaper*, 2014.
8. P. Ciancarini and T. Kielmann. Coordination models and languages for parallel programming. In *Proc. Int. Conf. on Parallel Computing (PARCO99)*, pages 3–17. Imperial College Press, 1999.
9. I. Corporation. The scckit 1.4.0 users guide. Technical report, Intel Corporation, March 2011. Revision 1.0.
10. T. Corporation. AMD FX Processor Product Brief, 2015. `http://www.tilera.com/files/drim__EZchip_LinleyDataCenterConference_Feb2015_7671.pdf` accessed 10-Mar-2015.
11. Golang.org. The go programming language. `http://golang.org` accessed 19-Apr-2014.
12. P. Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 2011.
13. C. Grelck, S. Scholz, and A. Shafarenko. Asynchronous Stream Processing with S-Net. *Int. Journal of Parallel Programming*, 38(1):38–67, 2010.
14. 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*, 18(2), 2008.
15. 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. *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, feb. 2010.
16. Intel Corporation. SCC External Architecture Specification (EAS). Technical report, Intel Labs, November 2010. Revision 1.1.

17. W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, Mar. 2004.
18. KALRAY Corporation. KALRAY MPPA MANYCORE Flyer, 2012. `http://www.kalrayinc.com/IMG/pdf/FLYER_MPPA_MANYCORE.pdf` accessed 10-01-2015.
19. B. W. Kernighan, D. M. Ritchie, and P. Ejeklint. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, 1988.
20. Microsoft. The Manycore Shift White Paper. Technical report, Microsoft Corporation, 2007.
21. V. Nguyen and R. Kirner. Demand-based scheduling priorities for performance optimisation of stream programs on parallel platforms. In J. Koodziej, B. Martino, D. Talia, and K. Xiong, editors, *Algorithms and Architectures for Parallel Processing*, volume 8285 of *Lecture Notes in Computer Science*, pages 357–369. Springer International Publishing, 2013.
22. A. Prell, T. Rauber, et al. Go's concurrency constructs on the scc. In *Proc. 6th Many-core Applications Research Community (MARC) Symposium*, pages 2–6, 2012.
23. T. Prescher, R. Rotta, and J. Nolte. Flexible sharing and replication mechanisms for hybrid memory architectures. In *Proc. 4th Many-core Applications Research Community (MARC) Symposium. Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam*, volume 55, pages 67–72, 2012.
24. D. Prokesch. A light-weight parallel execution layer for shared-memory stream processing. Master's thesis, Technische Universität Wien, Vienna, Austria, Feb. 2010.
25. C. Ramey. Tile-gx100 manycore processor: Acceleration interfaces and architecture. *Tilera Corporation*, 2011.
26. 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*, 46(4):757–766, April 2011.
27. B. Schauer. Multicore Processors–A Necessity. *ProQuest Discovery Guides*, pages 1–14, 2008.
28. R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.
29. W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proc. 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag.
30. Tilera Corporation. Tile Processor Architecture Overview for the TILEPro Series, 2013. `http://www.tilera.com/scm/docs/UG120-Architecture-Overview-TILEPro.pdf` accessed 10-01-2015.
31. 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.*, 45:73–83, February 2011.
32. M. Verstraaten. High-level Programming of the Single-chip Cloude Computer with S-Net. Master's thesis, University of Amsterdam, 2012.
33. X. Zhou, H. Chen, S. Luo, Y. Gao, S. Yan, W. Liu, B. Lewis, and B. Saha. A case for software managed coherence in manycore processors. In *Poster on 2nd USENIX Workshop on Hot Topics in Parallelism HotPar10*, 2010.