

MASTER COMPUTER SCIENCE



UNIVERSITY OF AMSTERDAM

Exploring the Intel Single-Chip Cloud Computer and its possibilities for SVP

Roy Bakker
0583650
bakkerr@science.uva.nl

October 7, 2011

Supervisors:

prof. dr. Chris R. Jesshope
drs. Michiel W. van Tol

Exploring the Intel Single-Chip Cloud Computer and its possibilities for SVP

Master's Thesis

written by

Roy Bakker

under the supervision of

prof. dr. Chris R. Jesshope and **drs. Michiel W. van Tol**,
and submitted in partial fulfilment of the requirements for the degree of

M.Sc. in Grid Computing

at the *University of Amsterdam*

Date of public defence:

October 19, 2011

Members of the Thesis Committee:

prof. dr. Chris R. Jesshope

drs. Michiel W. van Tol

dr. Inge Bethke

dr. Alban Ponse



UNIVERSITY OF AMSTERDAM

Abstract

Current multi-core processor architectures face several scalability issues. The Single-Chip Cloud Computer (SCC) is an experimental 48-core processor created by Intel Labs for the Many-Core applications research community. The architecture combines several new design approaches that should allow for a better scalability of many-core systems. The SCC features relatively simple but fully functional general purpose cores on a scalable on-chip mesh network with large bandwidth. The University of Amsterdam was selected to receive one of the prototype boards for use in their research on many-core programming methods.

In this thesis we explore the properties, possibilities and pitfalls of this new architecture. Based on these results we propose and make modifications to an implementation of the SVP concurrency model developed at the University of Amsterdam. SVP is an abstract concurrent programming model that is applicable to multiple levels of granularity and is communication deadlock free under the assumption of sufficient resources.

Acknowledgements

First of all I would like to thank all people in the Computer Systems Architecture (CSA) group for taking me as a Master student as a member of their research group and inviting me to group meetings and presentations. We had discussions about almost everything during lunch, tea breaks, cake breaks and in the office. I'll remember the endless discussions we had during the early SCC meetings with Clemens, Michiel and Merijn. At a certain point we even decided the best way to get some joy out of the SCC was to throw it out of the window, or use Merijn's ever lasting solution for everything: *a hammer*.

I think my family and many of my friends can be happy now as they have heard the same story over and over again for the last few months when asking about my thesis: "*Yes, it is almost finished, I only need to..*". Now it is really finished.

Special thanks are for Chris Jesshope for his supervision and his support during the project, even though I was not always in time with the work on my master project due to interest in education and involvement in several other projects. I always enjoyed his inspiring lectures and the discussions during meetings.

Last, but certainly not least I would like to thank Michiel van Tol for pulling me into this project, and providing me a comfortable workplace in his office. During the project he was always willing to act as a living *man page* for all my questions about architecture and programming. His knowledge about various computer architectures, operating systems and the C language and seemed endless to me.

Many thanks to all who have supported me during the project.

Contents

1	Introduction	11
1.1	Outline	12
2	Exploring The Intel Single Chip Cloud Computer	13
2.1	Overview and Set-up	13
2.1.1	Problems with the start up	14
2.2	Community	15
2.3	Network and Memory Details	15
2.3.1	Caches and Write Combine Buffer	15
2.3.2	Memory, Lookup Tables and Routing	16
2.3.3	On-Chip Message Passing	18
2.3.4	Flushing the L2 cache	18
2.3.5	Flushing the L1 cache	21
2.4	Power Management	21
2.5	sccLinux and sccKit	22
2.6	BareMetal	23
3	SVP	25
3.1	SVP programs	26
3.2	Implementations of SVP	27
3.3	A distributed implementation	27
4	Communication protocols on the SCC	31
4.1	RCCE	31
4.2	iRCCE	31
4.2.1	Asynchronous Communication	32
4.3	Memory Copy	33
4.3.1	Memory-map Settings	33
4.3.2	Memory Latency	33
4.3.3	Total memory bandwidth performance	34
4.4	Direct MPB access and Interrupts	34
4.5	Cacheable Shared Memory	36
4.6	Running the SCC in baremetal	37
4.6.1	ETI Baremetal Framework	37
4.6.2	Microsoft Baremetal	37
4.6.3	Bootstrap code and a C program	37
4.7	Copy Core	38
4.7.1	LUT Copy	38

4.8	Performance of Copy Cores	39
5	The SVP run-time on the SCC	41
5.1	The easy way: TCP-IP	41
5.1.1	Performance	41
5.2	SCC specific implementations	41
5.2.1	Using (i)RCCE	42
5.2.2	Dedicated MPB protocol	42
5.2.3	Direct data transfer	43
5.2.4	Memory Remapping	44
6	Evaluation	47
6.1	Benchmark Programs	47
6.1.1	Ping Pong	47
6.1.2	Matrix Multiplication	47
6.2	Results	49
6.2.1	Ping Pong	49
6.2.2	Matrix multiplication using one decomposition step	51
6.2.3	Matrix multiplication using two decomposition steps	53
6.2.4	Alternative approaches	54
7	Related Work	55
8	Conclusion	59
9	Future Work	61
A	P5 Architecture overview	67
B	SCC performance meter	69

Introduction

We moved to multi-core processor architectures several years ago. There is nothing new about that. The problems traditional single-core architectures face are generally known. One of the main problems is the *Memory Wall*: the Clock frequency of a processor unit increases faster than the speed of memory. Caches can not completely bridge this gap. Many techniques within a processor are developed to hide memory latency. For example the instruction stream can be re-ordered on the chip or the processor runs multiple independent threads at the same time to keep the processing unit busy. An other important problem is power dissipation, as the dynamic power (P) dissipated increases faster than the clock frequency (f). The maximum clock frequency is closely related to the operating voltage (V), where higher frequencies require higher voltages. In formula this is $P = C \cdot V^2 \cdot f$, where C is a capacitance constant. This formula does not include leakage current. Given a parallelizable program, multiple cores at a lower frequency can do the same amount of work while dissipating less energy.

As Moore's law still applies, the number of transistors that can be placed on a single chip still grows exponentially. As a consequence, the cost and time required for the logical design of chips grows with about same factor. This problem can be solved by designing a simpler core with less effort. Due to the limited required silicon space of this simple core, more of these can be put on a single chip, resulting in many-core chips instead of multi-core chips.

Intel started research on such new architectures in their Tera-Scale project [31, 4]. This project focuses on exploring new, energy-efficient designs for future many-core chips, as well as approaches for interconnect and core-to-core communications. Tera-Scale refers to processors that achieve more than a TeraFlops (one trillion floating point operations per second). Back in 1996, the ASCI Red Supercomputer was the first ever to deliver TeraFlops performance. That machine took up more than 185 square meters of space. It was powered by nearly 10.000 Pentium Pro processors, and consumed over 500 KW of power [31]. Only eleven years later in 2007, Intel's first Tera-Scale research chip achieves this same performance on a multi-core chip that could rest on the tip of a finger, and consumes only 62 Watts. The cores on this architecture are very simple and consist of just two floating point units. The chip is therefore not a general purpose computing device. Its successor, developed in 2009 and made available for a research community in the summer of 2010 features 48 fully functional cores. The new chip is introduced as a *Single-Chip Cloud Computer* (SCC) [14, 5]. The University of Amsterdam was selected to receive one of the

limited number of SCC systems under Intel’s research program.

The main problem that many-core architectures face is that the traditional memory architecture is not scalable [4]. Cache consistency and shared memory protocols do not scale over potentially hundreds or thousands of cores.

As a result, traditional programming methods will not hold for these new architectures. At the University of Amsterdam, an abstract concurrent programming and machine model (SVP) [16] is developed to express both coarse-grained as well as fine-grained concurrency over a potential wide area of current and future architectures.

In this thesis we describe and explore the properties and possibilities of the new SCC architecture. We propose options to port an existing distributed implementation of SVP to the SCC, and show the details and results of these prototype implementations.

1.1 Outline

Chapter 2 will give an overview of the SCC architecture and the results of experiments we did to gain knowledge of the system. In Chapter 3 we will give a summary of the SVP concurrency model and the current implementation status. In Chapter 4 we dive into the SCC memory system. Various communication methods, as well as the memory performance are discussed and benchmarked. Chapter 5 provides the details of the research undertaken to make an SVP implementation specific for the SCC. The results follow in Chapter 6. We make an analysis of related work in Chapter 7, and our conclusions follow in Chapter 8.

Exploring The Intel Single Chip Cloud Computer

2.1 Overview and Set-up

The Single-Chip Cloud Computer (SCC) experimental processor [14] is a 48-core *concept vehicle* created by Intel Labs as a platform for many-core software research. The machine has 48 Pentium 1 (P54C) cores on a very high performance scalable 6×4 mesh network.

The original P54C core is superscalar and has two data paths (5 stage in-order pipelines) that allow it to complete two instructions per clock cycle. The main pipe can handle any instruction, while the other can only handle the most common simple instructions. There is one floating point unit available. It was the first Pentium processor to operate at 3.3 volts instead of 5 volts, reducing energy consumption. A schematic overview of the P5 core architecture on which the P54C is based is drawn in Appendix A.

The relatively old and simple P54C cores are cheap in silicon design (*off the shelf*) and small in silicon area compared to current state of the art cores. Therefore it was possible to put 48 cores and an interconnect network on the same chip, build in 45nm CMOS. Figure 2.1 shows a photo of the actual SCC hardware we received from Intel.

At the moment, the SCC is not a stand-alone computer¹. In order to get it running, a management PC (MCPC) needs to be used. We got a HP proliant ML110 G6 as MCPC. The SCC connects to the MCPC through external PCI express. The MCPC requires a special PCI express 2.0 extension card and a compatible cable. The board features four gigabit Ethernet ports and a SATA port.

The SCC board is controlled by the Board Management Controller (BMC) which is a very small embedded device. The BMC is used to flash the on-board FPGA from a bit stream located on a USB stick. It can be used to read the current board status of the SCC, including power usage. The BMC is used to initialize the FPGA and several communication channels. It initializes the memory controllers and performs some sanity checks. The FPGA acts as the chipset for the SCC system and controls the SATA port and Ethernet ports on the board. It also acts as a bridge between the internal packet format and the PCIe interface. Since the FPGA is reprogrammable it

¹This is true for the current setup. With the use of the SATA port and a SATA driver on the FPGA, it should in theory be possible to run the SCC without MCPC.

allows updates without changing the hardware after the release of the platform. For example, Intel added support for atomic counters in the FPGA and routed TCP/IP traffic between the SCC cores and the MCPC through an additional Ethernet cable instead of the PCIe cable to fix a multiplexing bug. SATA support is not operational yet. One of the difficulties is that the programmable area in the FPGA is almost completely utilized. When SATA support has to be added, other features must be removed in order to make space.

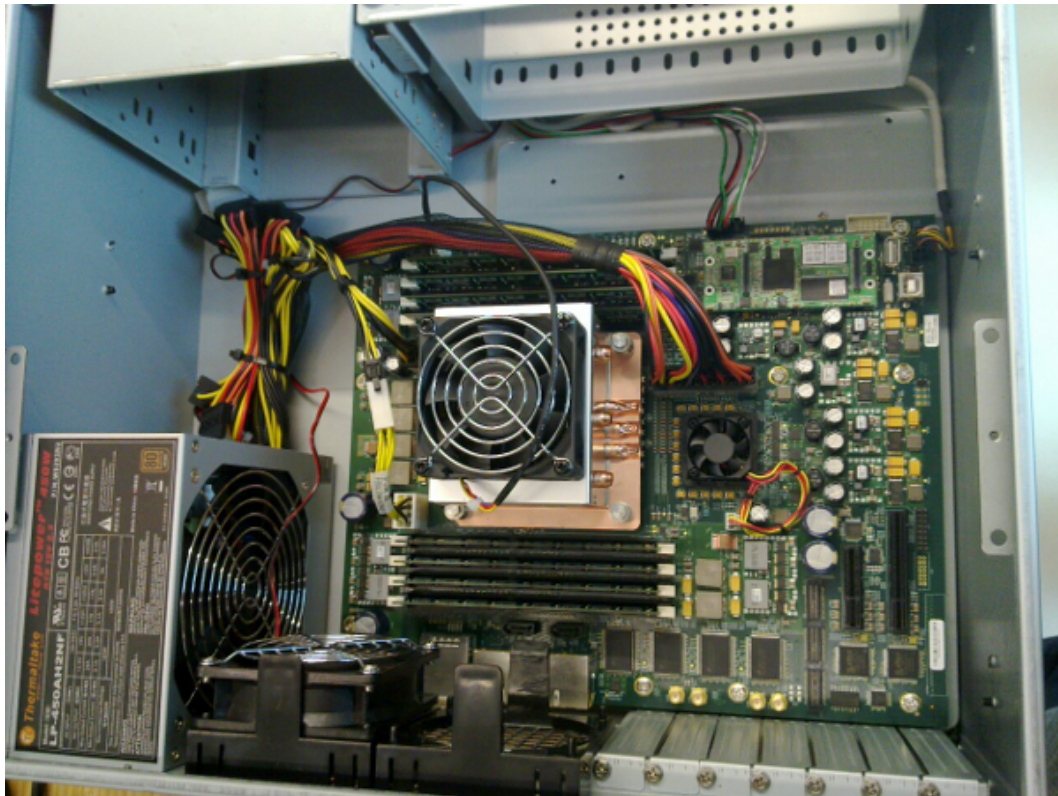


Figure 2.1: The SCC hardware consists of a board with an BMC, FPGA, DDR3 RAM and the actual SCC chip.

2.1.1 Problems with the start up

While setting up the system, the first problem we experienced was the use of an infrequently used PCI express (PCIe) extension card and cable. These had a shipping time of at least three weeks at the moment we ordered them. The SCC was already on our desk and we were not able to use it. When the equipment finally arrived, we experienced difficulties in getting the combination working. The PCIe card was not properly recognized by our MCPC, and therefore could not connect to the SCC. After some trial and error, we discovered that we had to put the PCIe x16 slot explicitly in PCIe v2.0 mode, and this could only be done after a BIOS upgrade of the MCPC.

2.2 Community

Apart from some early testing partners such as Microsoft and the Barrelfish team of ETH Zurich, the research on the SCC started in September of 2010. At the University of Amsterdam we were one of the first institutes that acquired an SCC board. At the time this project started, only a few details were available. As the SCC research is community driven with only a limited number of people from Intel involved, there is was not a lot of information and documentation available at the start of the project. The SCC community is part of the online Many-core Applications Research Community (MARC)[2]. To support the research community, Intel organizes MARC symposia all over the world. Since the beginning of the project, two of them were held in Europe. The first one was in November 2010, held in Braunschweig, Germany. This first symposium mainly focused on the status of SCC development and how to get started with the architecture. Both hardware and software specialists from Intel presented their contribution to the SCC and the possibilities and future features of the SCC. A few research teams invited by Intel presented their early results. The third symposium (second in Europe) was held in July 2011 in Ettlingen, Germany. This was the first opportunity for the community to share research on the SCC. In this symposium, 24 papers were presented in a talk or by a poster. Two of them [27, 30] were from our SCC team at the University of Amsterdam, of which one [27] won the best paper award.

2.3 Network and Memory Details

The SCC has a very fast and scalable on-chip 6×4 mesh network which has a 256GB/s bisection bandwidth [24]. Figure 2.2 shows a schematic overview of the chip. Each of the 24 routers manages the traffic for one tile consisting of two cores. A router has five links of 16 bytes each. The crossing latency is 4 mesh cycles per hop. The two cores on each tile have private L1 and L2 caches, and share an additional 16KB memory area which can be used as a *Message Passing Buffer* (MPB). Four tiles in the grid ((0,0), (0,5), (2,0) and (2,5)) are directly attached to a memory controller (MC). Each MC consists of two 4GB DDR3 banks, providing a total of 32GB of memory for the system we have. The 4GB banks could be replaced with 8GB ones, if there would be need for additional memory.

2.3.1 Caches and Write Combine Buffer

Each core has both a 16KB L1 data cache and instruction cache which caches the complete address space, including the MPBs which are accessible from all cores. An additional memory type for MPB data (MPBT) was added to the virtual memory system, together with an instruction to invalidate all cache lines in the L1 data 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 cache line which can then be written to memory at once. The WCB is only used for writes to memory which has the MPBT flag on. A pitfall of the WCB is that data is only written to memory when the entire cache line is filled, or when an other cache line is written to by a write instruction. At that point the new cache line will become active in the WCB. When the WCB is enabled, and the presence of the written values in memory should be asserted (for

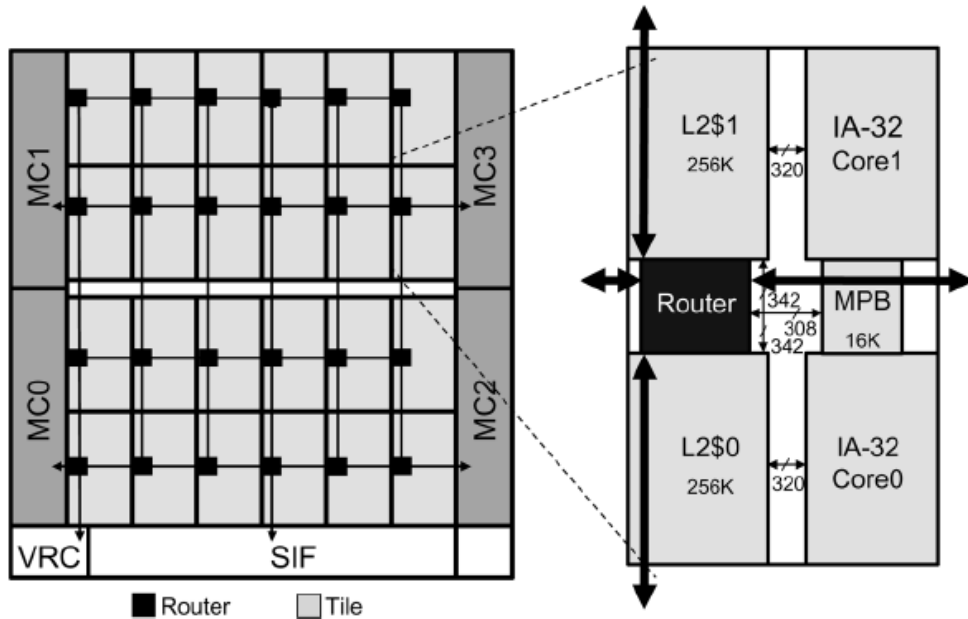


Figure 2.2: The network layout of the SCC. Image taken from [14].

example when an other core needs to read that data), a *dummy* write to a separate cache line is required to *flush* the WCB.

Each SCC core has a private unified 256KB L2 cache which does not feature a cache coherency protocol among other cores. 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 the L1 cache do not affect the L2 cache. To solve cacheability issues, users can turn off the L2 cache completely 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 cache disable flag (PCD) in the page table and disables both caches. Memory flagged as MPBT always bypasses the L2 cache. The L2 cache can be reset from the core using a special control register, which initializes all lines into invalid state. However, this operation halts the core and can therefore not be used to invalidate the cache during execution.

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

2.3.2 Memory, Lookup Tables and Routing

The individual P54C cores have a 32 bit core-physical address space, while the chip has four DDR3 memory controllers which each use 34 bit addressing. Combining the 34 bits with the memory controller address, the system can address 64GB in total, which is also the maximum supported. The translation of core-physical addresses to system-physical addresses is done through lookup tables (LUTs). Each core has its own LUT with 256 entries, where each entry covers 16MB of the 4GB core-physical addressable space, which we refer to as a *LUT page*. A schematic overview of the address translation is drawn in Figure 2.3.

The translation is done by indexing the LUT with the highest 8 bits of a core-

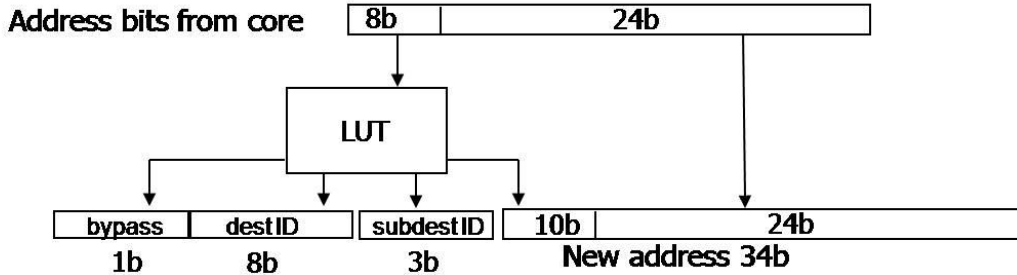


Figure 2.3: Lookup tables on the SCC. Image taken from [15].

physical address, and then extending it to form a 34 bit address and adding routing information for the mesh network. The routing information consist of an 8 bit tile ID (4 bit y, 4 bit x), and a sub-destination ID which indicates which router direction it should take at the destination tile. Sub-destination IDs can be the mesh exits on a router (North, East, South, West), or one of the on-tile destinations (Core0, Core1, CRB, MPB). At last, there is a bypass bit that can be used to address the on-tile message passing buffer. When set, the packet bypasses the tile router and stays within the tile clock domain. This would allow for fast MPB access. Due to a bug in the arbitration system between the two cores on a tile, the bypass bit should not be used. When using the bypass bit, it can corrupt data or hang the core as it stalls on a memory operation that will never complete.

A LUT entry can map to a special memory anywhere on the chip or to an area 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 set up when booting a core, but can be changed dynamically when the system is running, having effect immediately. This allows the sharing of 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. Note that in the default configuration, a core can change the LUT of any other core. This results in a highly configurable, but potential unsafe setup. After remapping memory and reading data from the new target, a core might still read stale data from the L1 or L2 cache since these are indexed with core-physical addresses and reside before the LUT. The only option is to flush both the L1 and L2 cache, which is in case of the L2 a very expensive operation. The lookup tables for the two cores in a tile are stored in the Core Register Bank (CRB), which are by default mapped on each core through the LUT itself. This requires 24 entries in the LUT to map all CRBs. An additional entry is provided for simplicity that always maps to the CRB that is local to that core.

The theoretical minimal memory latency is defined as:

$$40 \cdot C + (2 \cdot 4 \cdot n) \cdot N + 46 \cdot M$$

where C , N and M denote the times per clock cycle ($\frac{1}{frequency}$) for the core, mesh network and memory controller respectively, and n denotes the number of hops in the mesh network from the core to the memory controller. There are 4 mesh cycles required to forward a message to the next router, and this needs to be

multiplied by two because the request needs to travel back and forth between source and destination. Translation through the LUT is included in the 40 core cycles.

2.3.3 On-Chip Message Passing

On each tile, a Message Passing Buffer of 16KB (8KB per core by convention) is available that is both readable and writable from every other core. Combined with the fast network this allows for fast on-chip core-to-core communication. The MPB of each tile is mapped through a LUT entry, taking 24 times 16MB of addressable space for only 24 times 16KB of memory. An additional LUT entry is set on each core for simplicity of accessing the own MPB. When one addresses beyond the available 16 KB of physical space in the MPB, the most significant bits are discarded resulting in a *wrap around*.

We measured the throughput of the MPB, and the results are in Table 2.1. From core 0 we measured the latency and throughput of the MPBs in all tiles on the chip. The first column is the tile number, followed by the number of hops in the mesh network to the target MPB. We measured the latency by reading one byte from each cache line over 16MB. Before and after the measurement, we read the hardware timestamp counter (TSC) which gives the number of core cycles that it took to perform the complete operation. We divide this number by the number of accesses to get the latency per access. Before the measurement we flush the cache, and because we only access one byte per cache line we know we will never hit in the cache. This gives us the actual time that it takes to transfer minimal data from the target MPB to the core. We see an increase in latency of about six core cycles per hop in the mesh network.

The second measurement accesses each byte in the 16MB space at the granularity of a 4-byte integer. The first access transfers the complete cache line to our local L1 cache, so consecutive accesses within a cache line will hit in the L1 cache. From these numbers we can calculate the maximum throughput of the MPB in MB/sec. We performed the measurements for both reading and writing, but the results are not significantly different. Both operations operate at the granularity of a cache line. While reading, cache lines are captured by the L1 cache (L2 is bypassed with the use of the MPBT flag), and for writing cache lines are captured by the WCB. By enabling the bypass bit we could get the values for reading the local MPB down to an average of about 5-6 cycles per 4-byte read or write. However, setting the bypass bit results in unexpected behavior.

2.3.4 Flushing the L2 cache

There is no native way (flush instruction) to flush the L2 cache. We can only *flush* the L2 cache by reading 256KB of data that is certainly not yet in the cache. We can not reliably flush the cache from user-space, since the pseudo least recently used (pLRU) replacement policy has issues with context switches. A context switch that occurs during the flush routine may update the state of the pLRU, resulting in a different replacement scheme when the flush routine continues. It can not be guaranteed that each way of a set will be flushed. Therefore Intel proposed a kernel module which acts as a device driver. A write of 0 bytes to that device will trigger the flush routine. In kernel mode, interrupts can be temporarily disabled to make sure that no context switch occurs that may affect the flushing process. Flushing the L2 cache this way is a very expensive operation. Originally we measured it to be

Tile	Hops	1 byte read latency in cycles	Average cycle cost of a 4 byte read	Reading throughput in MB/sec	1 byte write latency in cycles	Average cycle cost of a 4 byte write	Writing throughput in MB/sec
0	0	60.94	9.81	207.20	60.45	9.83	206.92
1	1	66.50	10.19	199.50	63.44	10.57	192.29
2	2	72.49	10.94	185.80	69.45	11.50	176.79
3	3	72.52	11.32	179.64	72.52	11.34	179.27
4	4	80.55	12.07	168.51	78.47	12.08	168.35
5	5	84.58	12.82	158.61	84.52	12.83	158.49
6	1	66.50	10.19	199.53	63.73	10.60	191.78
7	2	72.49	10.94	185.81	69.45	11.33	179.52
8	3	72.53	11.35	179.09	72.50	11.34	179.33
9	4	78.79	12.07	168.48	78.48	12.08	168.32
10	5	84.55	12.82	158.65	84.52	12.83	158.47
11	6	90.53	13.56	149.89	90.48	13.58	149.73
12	2	72.49	10.94	185.82	69.44	11.33	179.53
13	3	72.53	11.32	179.64	72.50	11.34	179.34
14	4	78.55	12.07	168.50	78.48	12.08	168.30
15	5	84.55	12.99	156.50	84.51	12.83	158.48
16	6	90.52	13.57	149.83	90.50	13.58	149.72
17	7	96.53	14.32	142.01	96.49	14.33	141.90
18	3	72.55	11.32	179.64	72.50	11.33	179.38
19	4	78.53	12.07	168.49	78.48	12.08	168.35
20	5	84.55	12.82	158.59	84.51	12.83	158.48
21	6	90.52	13.57	149.86	90.48	13.58	149.72
22	7	96.51	14.32	142.01	96.49	14.33	141.89
23	8	102.53	15.07	134.93	102.49	15.08	134.85

Table 2.1: Measurements of raw access to all of the on-chip Message Passing Buffers (MPB) performed from core 0.

more than 1 million core cycles. This has been reduced to 900K cycles, by optimizing the flushing routine. Table 2.2 (Normal) shows the number of core cycles that it takes to flush the entire cache, and the average number of core cycles per cache line. There is a difference between flushing the cache that has all lines in allocated state, but unmodified (clean), and all lines in modified state (dirty), and need to be written to memory (write-back). When the cache is flushed again directly after a previous flush, the flush data is still present in the cache resulting in a faster flush.

Even after the use of a kernel module, there are still rare situations in which users experience stale data to be read from the flushed cache. In a later stage, a fix was proposed by our team that uses two separate blocks of flush memory that are alternating used to flush the cache from the kernel module. The reason that the first solution does not work properly is that when a cache flush is issued, there might still be data from the previous flush in the cache. This triggers a cache hit in the current flush that puts the pLRU replacement policy in a different state. Stale data may now still be in the cache and result in a cache hit later on. After the proposed fix, we did not see this issue anymore.

With the use of faster memory, we can reduce the costs for L2 flushing even more. Faster memory is available in the MPB. However, this is only 16KB of memory while the L2 cache is 256KB. It is possible to use the properties of the LUT in combination with the way the MPB is addressed. When the MPB size is exceeded during the addressing of the MPB, the access will *wrap-around* to the beginning of the MPB (i.e. only the lower bits are used to address the MPB, and the higher bits are discarded). From the L2 perspective however, these are independent addresses, and therefore cached separately. Within a 16MB LUT entry, we have plenty of space to flush the cache without having to touch the original MPB addresses. Note that this does not affect the data that is actually in the MPB since it will only be read. The memory in the LUT entry should be mapped as L2 cacheable, since by default, the MPB is not cached in L2 (property of MPBT type). The results from this new flush routine are included in Table 2.2 (MPB). We see that using MPB memory is much faster than using normal memory. For the MPB flush routine that switches between two regions (MPB2) we see no difference between flushing a clean cache and flushing a previously flushed cache because all data will be replaced just as it would happen during a clean flush.

Routine	Type	Min	Avg	Max
Normal	Clean	921022 (112.4 c/l)	938975 (114.6 c/l)	976709 (119.2 c/l)
Normal	Flush	206940 (25.3 c/l)	215265 (26.3 c/l)	251784 (30.7 c/l)
Normal	Dirty	1463223 (178.6 c/l)	1504708 (183.7 c/l)	1528075 (186.5 c/l)
MPB	Clean	578488 (70.6 c/l)	582352 (71.1 c/l)	608504 (74.3 c/l)
MPB	Flush	196338 (24.0 c/l)	198999 (24.3 c/l)	231444 (28.3 c/l)
MPB	Dirty	888915 (108.5 c/l)	1182089 (144.3 c/l)	1343956 (164.1 c/l)
MPB2	Clean	570773 (69.7 c/l)	573571 (70.0 c/l)	633198 (77.3 c/l)
MPB2	Flush	569133 (69.5 c/l)	571359 (69.7 c/l)	600177 (73.3 c/l)
MPB2	Dirty	1258989 (153.7 c/l)	1274740 (155.6 c/l)	1313061 (160.3 c/l)

Table 2.2: L2 flush measurements on SCC core 0, min/max/avg in total core cycles and average core cycles per cache line (c/l) over 256 measurements.

2.3.5 Flushing the L1 cache

Flushing the L1 cache is not as much an issue as flushing the L2 cache. When we flush the L2, the L1 is automatically flushed due to its smaller size. However, sometimes we only want to flush the L1, because we can set the page properties to bypass the L2 cache. This is for example the case in the default MPB mapping where memory is mapped as MPBT. Intel added an additional instruction to the SCC cores (CL1INVMB) to invalidate all lines in L1 that are tagged MPBT. This instruction will only invalidate, so data that had not been written back will be lost. To avoid this, writing to the MPB may never result in a hit in the L1 cache. Before writing to the MPB, the CL1INVMB instruction should be issued, resulting in a *write around* to the MPB.

The instruction WBINVD performs a *write back* on the complete L1 cache and invalidates thereafter, and INVD instruction invalidates all lines in the L1 without write back. When the cache is in write back mode while issuing INVD, modifications to the data will be lost. While CL1INVMB can be issued from user-space, WBINVD and INVD are *privileged instructions*, and can only be issued from kernel-space. The performance measurements of these instructions are listed in Table 2.3. Per instruction there are three measurements on different initial states of the cache. *Clean* indicates allocated but not modified, *flush* is the state where the cache is already in flushed state, and *modified* indicates that all lines in the cache has been written to before the flush, making them *dirty*. These measurements include the time required for the system call.

Instruction	State	Min	Avg	Max
WBinvd	Clean Flush	9157	12399	17479
WBinvd	Flush Flush	6332	6386	7092
WBinvd	Dirty Flush	16180	20902	47896
invd	Clean Flush	1903	6121	33361
invd	Flush Flush	322	340	1035
invd	Dirty Flush	2852	7749	9390

Table 2.3: L1 flush measurements on SCC core 0, min/max/avg in core cycles over 256 measurements

2.4 Power Management

The SCC supports voltage and frequency scaling. The voltage can be set in six voltage islands with four tiles each. The frequency can be set on a per-tile (2 cores) basis. Frequency and voltage scaling can be performed (just as LUT mapping) by each core for each voltage or frequency domain. Voltage values are stored in a special Voltage Regulator Controller (VRC).

The frequency and voltage of the mesh network is also configurable. The frequency of the routers can either be 800MHz or 1600MHz. When 1600MHz is chosen, the frequency of the Memory Controllers can either be 800MHz or 1066MHz. When 800MHz is chosen for the routers, the frequency of the Memory Controllers is always set to 800MHz.

The frequency of the tiles can be set in 15 non-linear steps from 100MHz up to 800MHz. The voltage can be adjusted from 0 to 1.3V in steps of 6.25mV. The

maximum frequency is dependent on the voltage level. The power consumption of the chip is rated at 25W by 0.7V and 125MHz up to 125W by 1.14V and 1GHz. Note here that these are results from experiments performed by Intel. In normal operation it is not possible to clock the cores at 1 GHz.

All experiments presented in this thesis are performed at the default configuration, with the core clocked to 533MHz, and mesh network and memory controllers clocked to 800MHz.

2.5 sccLinux and sccKit

The 48 SCC cores are all independent from each other and run their own operating system. The SCC does not feature a BIOS. This is one of the reasons that we need a MCPC to control the hardware.

sccLinux is a modified version of the Linux 2.6.16 kernel which is capable of booting without a BIOS. All values that are normally obtained from the BIOS are hard coded in the kernel. Other modifications are based on timers and interrupts. Bits in page table entries may have an alternative meaning on the SCC. For example the page size extension (PSE) bit is used to indicate MPBT memory, while it was intended to indicate that the page size is 4MB instead of 4KB. There are additional devices created in `/dev` to control the memory, which are provided by the rckmem driver. The rckpc driver provides support for virtual network interfaces for TCP/IP communication between cores, and between the cores and the MCPC.

At the time of writing this thesis, a newer kernel (2.6.37) with a more advanced build system (BuildRoot) has been made available to the MARC community [25]. It features the same functionality as the the previous version of sccLinux, but is more modular and configurable.

When the reset pin of a core is released, the core boots in 16bit real mode starting at the fixed memory address `0xFFFFFFFF0`. This means that valid bootstrap code needs to be at that address when the reset pin is released.

The sccKit is a set of software tools provided by Intel to control the SCC hardware from the MCPC. It contains command line tools and the `sccGui`. Many of the command line tools have a corresponding button in the GUI. A graphical performance meter is provided to visualize the current load of all 48 cores, both individually and accumulated. We can also see the power usage of the system. A screen shot of the SCC performance meter is provided in Appendix B.

`sccBmc` is used to control the on-chip BMC. From here we can switch the SCC chip on and off, and read some status values from the board. A special initialization command is used to bring the chip into usable state. At the beginning of the initialization process we can choose at which initial speed we want to clock the cores, mesh network and memory controllers.

`sccBoot` is used to boot a range of SCC cores. In the boot process, the reset pins of the cores are triggered first, and the specified image is load to the appropriate memory location(s). After completion, the reset pins are released again. The `-s` flag will not boot the cores, but list the cores that can be successfully reached using `ping`.

`sccDump` is a tool used to read the contents of the SCC DDR3 memory, the control registers (CRB), or the message passing buffers (MPB). Since it is not easy to debug on the SCC cores, this tool can be used to inspect the memory state after a failure of the program or even a crash of the OS.

With `sccDisplay` it is possible to run graphical applications on the SCC. A special driver on the core writes to a frame buffer in main memory, which can be read and displayed with the display tool. For a single core at a time, it is even possible to forward mouse and keyboard input to the SCC core or hear the sound output on the MCPC.

2.6 BareMetal

In addition to running `sccLinux` on the SCC, it is also possible to run an other OS on the cores. The Barrelfish team ported their operating system to the SCC [22]. Also running without operating system is an option. Usually this is referred to as *baremetal*. There are two baremetal frameworks available from ETI and Microsoft. One of the community members made a framework in which users can run simple C programs directly on the SCC hardware. Some experiments and a more detailed description of the frameworks are described in Section 4.6.

The Self-adaptive Virtual Processor (SVP) [16] is a concurrency model developed at the University of Amsterdam. The goal is to have a method to express concurrency without having to manage it. SVP expresses groups of concurrent activities (tasks). *create* is used to delegate a group of tasks (as a *family* of threads) to a *place*. A place is defined as a resource where a task can execute. This can for example be a core, a group of cores or a complete machine, depending on the available hardware and implementation. After a *create*, the execution of the initiator continues at the point directly after the *create*. Each *create* is eventually followed by a *sync*. Upon a *sync*, the initiator blocks until the created task completes. The initiator can also interrupt a created task with a *kill*. The model works recursively, such that each created thread within a family can create new families. Within a family, each thread is uniquely identified by an index. SVP is a model that can be applied to multiple levels of granularity, from high level task parallelism to low level instruction parallelism [28]. Threads can consist of only a few instructions, up to an entire program. Within a thread, every action is sequential.

Communication between threads within a family is possible using channels. There are two types of write-once channels available, *shared* and *global*. Both channels have blocking reads. The global channel is a communication shot from the parent thread to all threads in the family. The shared channel allows threads within a family to communicate in a restricted way. The first thread in the family reads the value from the parent thread, the second thread from the first, the third from the second, and so on. The last thread in the family is connected to the parent again. A schematic example of these channels is given in Figure 3.1. Shared channels identify all data dependencies between threads.

A special property that a place can have is that it can be *exclusive*. Only one family can execute on an exclusive place at a time. This provides a mechanism for mutual exclusion.

The SVP model assumes a global shared memory. *create* and *sync* define where the memory needs to be consistent following a *release consistency* [10] model. It should be guaranteed that:

- A child family sees the same memory state as the parent saw at the point of *create*.
- The parent sees the changes to memory made by the child family only when the *sync* has completed.

- Subsequent families created on the same exclusive place see the changes to memory made earlier by other families on that place.

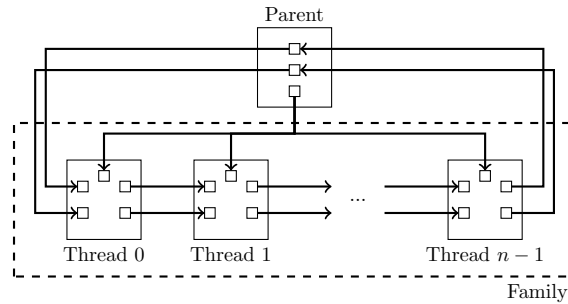


Figure 3.1: Unidirectional synchronization channels between threads. One global channel and two shared channels are shown in this figure.

3.1 SVP programs

An SVP program is written in a programming language that is an extension to the ISO C99 language to support the SVP concurrency model. The constructs and keywords added to the language are:

- *family* is a keyword to declare a family identifier variable.
- *thread* is used as a function identifier to specify that this function can be run as a thread (i.e. it can be *created*).
- *shared* is used as an identifier for thread function arguments that need the property of a shared variable. All other arguments are by default global.
- *index* is used as a keyword for a variable that should contain the thread index within a family after creation.
- *create* starts a family of threads according to its arguments. Create is placed in front of a normal function call to the thread function. create takes as arguments a family identifier, place identifier and a set of start, step and limit values. These indicate the thread indices to be generated from start to limit with a step size of step. At last, a block parameter can be given to indicate how many threads may run simultaneously.
- *sync* synchronizes (blocks) the parent on completion of a family. The only argument it takes is a family identifier.

A very simple SVP program is listed in Figure 3.1. In this example, the previous two Fibonacci values are passed to a thread as two shared variables, followed by a global array in which the threads write their result values. Note that this global array only shows the usage of arrays; the values are not actually needed since the shared variables are communicated from the last family thread back to the parent thread.

A programmer can explicitly program in this language, but it can also be used as a compiler target. Compilers exist to compile from plain C as well as from S-NET (a declarative coordination language for describing streaming networks of asynchronous components) [13] and Single Assignment C (a functional array programming language) [12] to an SVP enabled language.

3.2 Implementations of SVP

There are two main implementations of SVP. The first is a fine-grained implementation which is a cycle-accurate simulation of a large amount of microthreaded processors that implement the SVP model directly in hardware [19]. The architecture is called *the microgrid*.

The other implementation is more focused on coarse-grain concurrency [28]. The threads that are created in SVP are mapped to POSIX threads (pthreads). The operating system manages the scheduling of these threads. As the creation of a pthread, and switching between threads is quite costly, this implementation focuses on more coarse grained concurrency. For this implementation, the SVP program is transformed to C++ code by translating the SVP constructs to C++ constructs available in the SVP pthread library. This source code can be compiled with a standard C++ compiler and linked with the SVP pthread library. See Figure 3.2 for a diagram of this tool chain.

3.3 A distributed implementation

To run coarse grained SVP programs on a distributed system, an extension to the pthread implementation has been made [29]. For this implementation, places have been defined as either being *local* or *remote*. Remote places can be defined with an address; for example an IP address and port number for socket communication. As SVP assumes a shared memory system, this has to be emulated through communications. An additional data description function must be added to the program source code to indicate the data that needs to be shipped to remote nodes for a given thread function. This is necessary because when we use pointers as thread parameters, we do not know what the actual size and meaning of a variable is. There are constructs to make the difference between values required for input, output, or both (inout), such that data will only be transported when necessary. For the Fibonacci example, the data description function is listed in Figure 3.3. In case a thread parameter is a pointer, the write-once rule only applies to the pointer, and not to the data. It is up to the user to maintain consistency of this data. To support heterogeneous systems with a possible different data representation, the data stream is encoded using the External Data Representation Standard (XDR) [9] to a global standard format. On the other side of the channel, this stream must be decoded. XDR also gathers all data into a single buffer, which can be transported to the remote place at once.

```

1  #include <stdio.h>
   #include <assert.h>

   thread fibonacci_compute(shared int prev,
6     shared int prev2, int *fibonacci)
   {
       /* Our thread index. */
       index i;

11     /* Compute. */
       fibonacci[i] = prev + prev2;

       /*
16      * Pass the previous and current Fibonacci
       * number to the next thread
       */
       prev2 = prev;
       prev = fibonacci[i];
   }
21

   thread main(void)
   {
26     family fid;

       int n = 45;

       int *fibonacci;
       int prev, prev2;
31

       /* Allocate Array for Fibonacci Numbers from 0 to N */
       fibonacci = (int *) malloc(sizeof(int)*(n+1));
       assert(fibonacci);

36     /* Set two starting Fibonacci Numbers */
       fibonacci[0] = 0;
       fibonacci[1] = 1;

       /* Set shares to initial values */
41     prev2 = fibonacci[0];
       prev = fibonacci[1];

       /* Create n-1 threads of fibonacci_compute, index 2 to n+1 */
46     create(fid ;; 2;n+1;;) fibonacci_compute(prev, prev2, fibonacci);

       /* Wait for all threads to complete */
       sync(fid);

       free(fibonacci);
51 }

```

Listing 3.1: SVP code for Fibonacci.

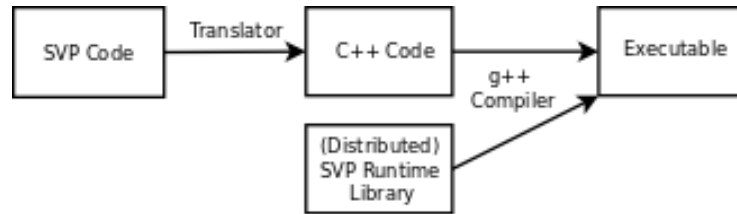


Figure 3.2: The translation and tool-chain.

```

DISTRIBUTABLE_THREAD(fibonacci_compute)(DataBuffer* db, int*& prev,
                                        int*& prev2, int& size, int*& fibonacci)
{
4   SINPUT(prev);
   SINPUT(prev2);
   GINPUT(size);
   ARRAY_SIZE(fibonacci, size)
9   for (int i = 0; i < size; i++)
   {
       GINOUT(fibonacci[i]);
   }
}
ASSOCIATE_THREAD(fibonacci_compute);

```

Figure 3.3: Data description function for the fibonacci example.

Communication protocols on the SCC

4.1 RCCE

RCCE [26] is Intel's approach to message passing. RCCE is an MPI-like library that provides communication through the on-chip message passing buffer (MPB). The downside of RCCE communication is that it needs to know sender, receiver and the size of the message beforehand, and all calls are blocking. RCCE performance is shown in Figure 4.1 for various memory mappings of the data locations to transfer. In the original mapping (normal \rightarrow normal), it has its peak performance at 60MB/s only for 4KB messages. Due to some overhead in the MPB (a small reserved area for flags and meta data), an 8KB message does not fit completely into a core's available MPB space. For larger messages up to 256KB (size of the L2 cache) it drops to around 20MB/s and for even larger messages the performance collapses to 5MB/s, because the input data is not present in the L2 cache. The graph also shows the influence of use of the WCB on the performance. The WCB is only enabled when the MPBT flag is set, and as a result of the MPBT tag, the L2 cache is bypassed.

An extension to RCCE, *RCCE_comm* [3] from the University of Texas is available which implements missing communication functions like scatter and gather. Having these functions, one can almost directly map MPI programs to RCCE programs.

4.2 iRCCE

While RCCE does not support multiple outstanding communication requests and has poor performance, RTWH Aachen developed an extension to RCCE (*iRCCE*) [8] which implement some optimizations to the standard RCCE functions such that they have a better performance. They also added non-blocking communication. A user can issue multiple communication requests and can at any time thereafter check if they have completed, both blocking and non-blocking. Multiple requests can be put together in a *waitlist*. The waitlist can be checked for any or all of the requests to be completed, either blocking or non-blocking.

The performance difference of RCCE versus iRCCE is visualized in Figure 4.1. The main reason for performance difference between RCCE and iRCCE is that iRCCE uses pipelining. It splits the available MPB space for a core (8KB) into two parts such that the read and write operations do not happen in lockstep. While one core is reading from one part of the MPB, the other can write the other part of the MPB at the same time, and switch regions when both are finished with their part.

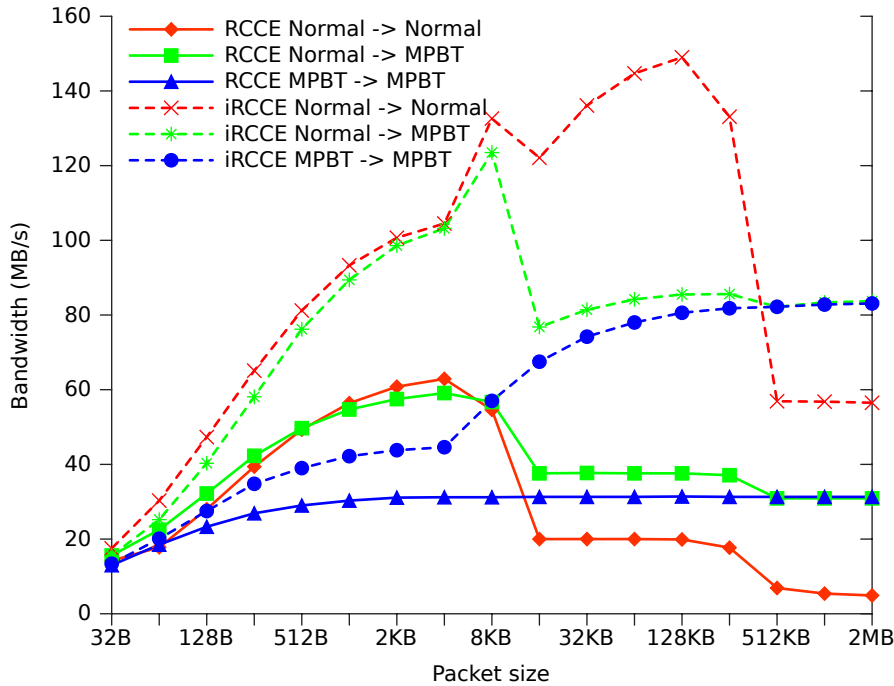


Figure 4.1: (i)RCCE performance

iRCCE prefetches the target addresses into the L2 cache such that it is not suffering from a *write around* for every write operation. 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 at around 145MB/s. As soon as messages are larger than the L2 cache, the performance drops to 60MB/s. This is a significant speedup compared to RCCE.

4.2.1 Asynchronous Communication

With iRCCE we still have to provide the identifier of the other core and the size of the message in a request. This would not allow for asynchronous communication of an arbitrary size in the same way for example TCP/IP provides.

We have developed a solution for this using iRCCE. On start-up we issue a non-blocking receive request with a small size for every other active core. We collect these in a waitlist and from time to time check if any has completed. When a request completes, we issue another request of the same size and add it to the waitlist again. The waitlist is now ready to accept a new incoming message. Based on the information in the message we can issue an other directed receive request to the sending core, now with the actual size of the message. Note that this approach has substantial overhead and latency, since we usually do not want to check the waitlist too often. There is also the overhead of sending an additional message with meta information. An other option would be to always send messages of the same length and split larger messages into multiple and padding the last message to fit the message size. While implementing this, we experienced some bugs in the iRCCE library which were solved after a discussion on the MARC community forum.

4.3 Memory Copy

Communication (data transfer) using RCCE or iRCCE is not very efficient. Besides that the throughput is far from optimal, it takes two cores to copy the data from one place to an other. This wastes CPU time that may have been used better, and consumes more energy. The message passing approach does not seem to be very effective when all cores are able to directly access the complete memory area using dynamic mapping. For large messages, we might be better off with normal memory copy operations. We measured the performance of memory (copy) operations under different circumstances.

4.3.1 Memory-map Settings

On a per page basis, we can map memory in different ways [27]. There are three flags that can be set to influence memory performance. The MPBT flag, which is obviously intended for the MPB, can be used on normal memory as well. The MPBT flag turns the Write Combine Buffer (WCB) on, and therefore gives a performance gain when writing memory. As it bypasses the L2 cache, the MPBT flag is less suitable for reading. The other flags we can set are PCD (cache disable) and PWT (cache write through or write back). PCD turns off caching completely and therefore performs badly, but there are no cache coherency problems with this setup. In case of the SCC the PWT bit only affects the L1 cache. This is a result of the original P54C core that only supports an external (off-chip) L2 cache.

For each of the possible flag combinations, a memory device was created to map memory using that combination. Memory operations were issued and measured from and to each of these devices. Figure 4.2 shows the results of these experiments. We can clearly see the influence of the L1 cache (16KB) and the L2 cache (256KB) versus uncached memory, the absence of the L2 using MPBT flagged memory for reading, and the function of the WCB in MPBT flagged memory for write operations.

After some bug reports on the MARC forum, we found that combining the uncacheable flag and MPBT flag results in bad data from memory. The conclusion is that with the MPBT flag on, the memory management unit (MMU) always wants to get a complete cache line from memory (and put it into the L1 cache). When we use uncacheable memory the data does not end up in the cache. The cache line ends up on the 64 bit wide bus of the core, and therefore the core always only sees the first 64 bits of a cache line, no matter what address in that line has been requested. We should therefore not combine the uncacheable and MPBT flag.

4.3.2 Memory Latency

Because the cores are located on a mesh network while there are four memory controllers located at the edges of the chip, the cores do not see the same memory latency and throughput. Figure 4.3 shows the average latency measured by reading one single byte per cache line over 16MB. It shows the results measured at each of the 24 tiles, for each of the 4 memory controllers. The graph may look a bit chaotic at a first glance, but the individual lines are clearly visible. The average is more or less the same for the first 18 tiles, but grows for the upper six tiles. This indicates that some cores have on average a better memory performance than others. The difference in average is the result of the asymmetric placement of the memory controllers on the chip. The upper tiles have on average a larger path in the mesh to

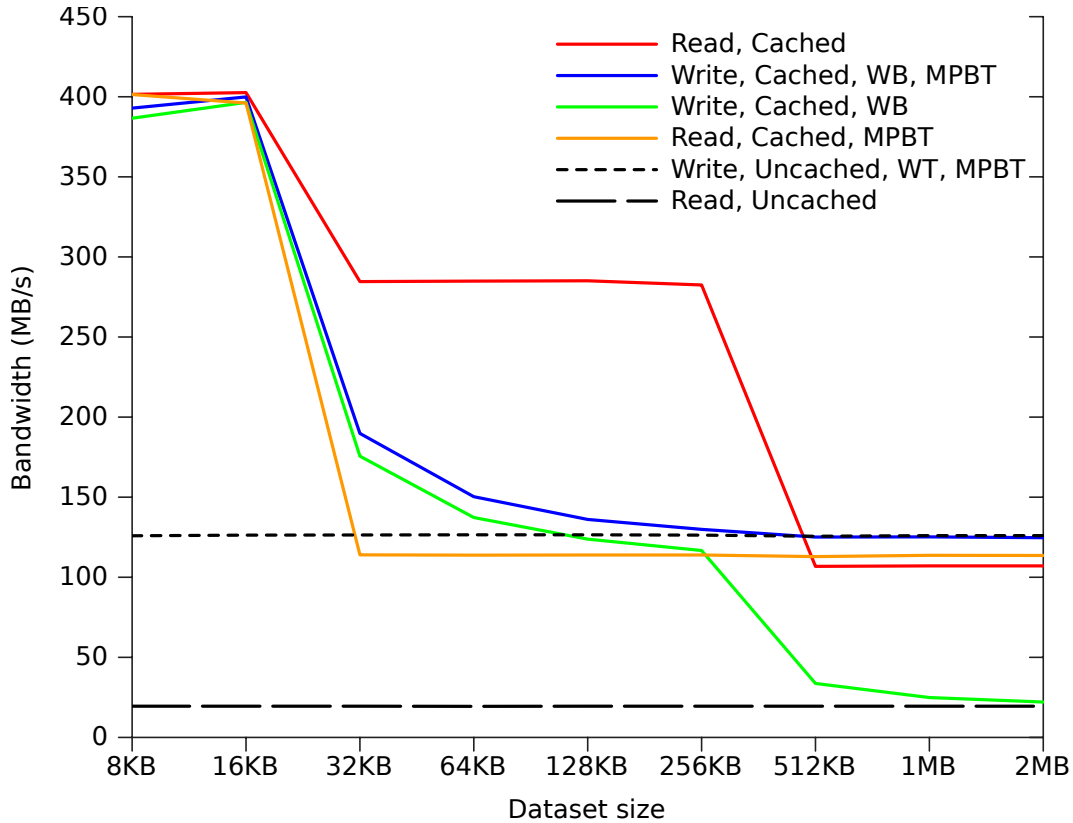


Figure 4.2: Memory throughput benchmark result of one core reading and writing pre-fetched memory areas of several sizes, showing the effect of different memory flags.

the memory controllers. This might be of interest in the mapping of tasks to cores when we know the memory needs for tasks in advance. Measurements for memory throughput shows the same patterns and are therefore not included.

4.3.3 Total memory bandwidth performance

We measured the total performance of the four memory controllers and what happens when we access them from various cores at once. Each memory controller has two banks (DIMMs) of 4GB. Each bank consists of two ranks. Figure 4.4 shows the results of our measurements. We see that the memory performance (accumulated bandwidth) scales linear with the number of cores, as long as the load is equally spread over the four memory controllers and their ranks. As soon as we divide the load over lesser memory controllers or ranks, we see a decrease in accumulated bandwidth. The current memory system seems good enough for this amount of cores, but does not tend to scale within the current setup. With the current number of cores, the memory controllers are at their maximum throughput capacity. Adding more cores to a mesh would also require more or faster memory controllers.

4.4 Direct MPB access and Interrupts

There is no need to use the MPB driven by RCCE or iRCCE, as they are just libraries provided as a way to use the MPB. We can choose to use the MPB in any other

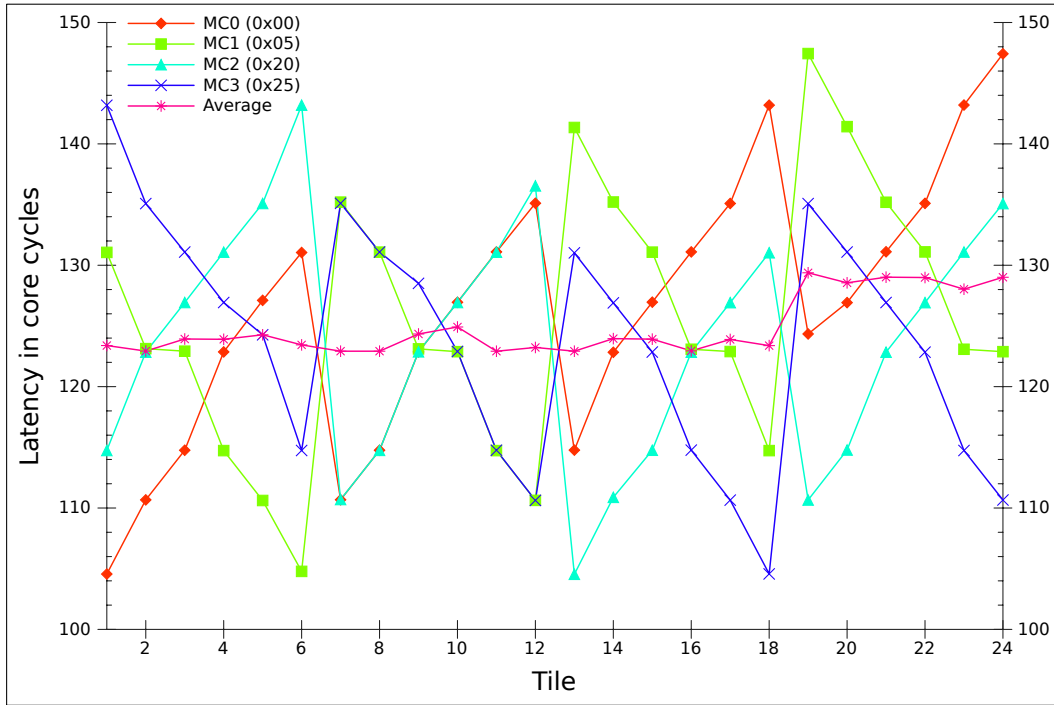


Figure 4.3: The latency in core cycles for a memory access, measured from each tile to the different memory controllers, and the average.

way.

We implemented our own messaging protocol that uses the MPB, based on the idea from the Barrelfish implementation [22] from ETH Zurich. Short messages through the MPB initiate larger commands on a core. We used this in our *Copy Core* implementation (for details see Section 4.7). We also used direct MPB access in one of our distributed SVP implementations for the SCC.

We need to make sure that there are no conflicts in the access of the MPB. We need a way to arbitrate the access to the MPB of a core. The SCC cores feature a lock in the Core Register Bank (CRB) that can be used by all cores. Reading from the address sets the lock if possible and returns 1 when the lock is set, and 0 when the lock could not be acquired (i.e. is already locked). A write to the register address resets the lock.

A circular buffer is implemented in the MPB. We use the first cache line as a dummy one, to fool the Write Combine Buffer (WCB) (see Section 2.3.1). The second line is used to contain meta information such as the head and tail entry of the circular buffer.

To avoid polling on a message in the MPB, we can use interrupts. A small framework for sending interrupts was developed at the University of Amsterdam for future use in the distributed S-NET implementation for the SCC [30]. A core can send an interrupt to each other core, which is handled by a kernel driver. The kernel driver can forward the interrupt to a user-space process by sending it a POSIX signal. The application can catch this signal and take appropriate action. In some cases interrupts are preferred above polling because it avoids unnecessary computation, locking and power consumption by the receiving core. In the meantime, a core can do other work or wait in idle state and consume less power. With voltage and frequency scaling on the SCC, we can potentially benefit from this even more by

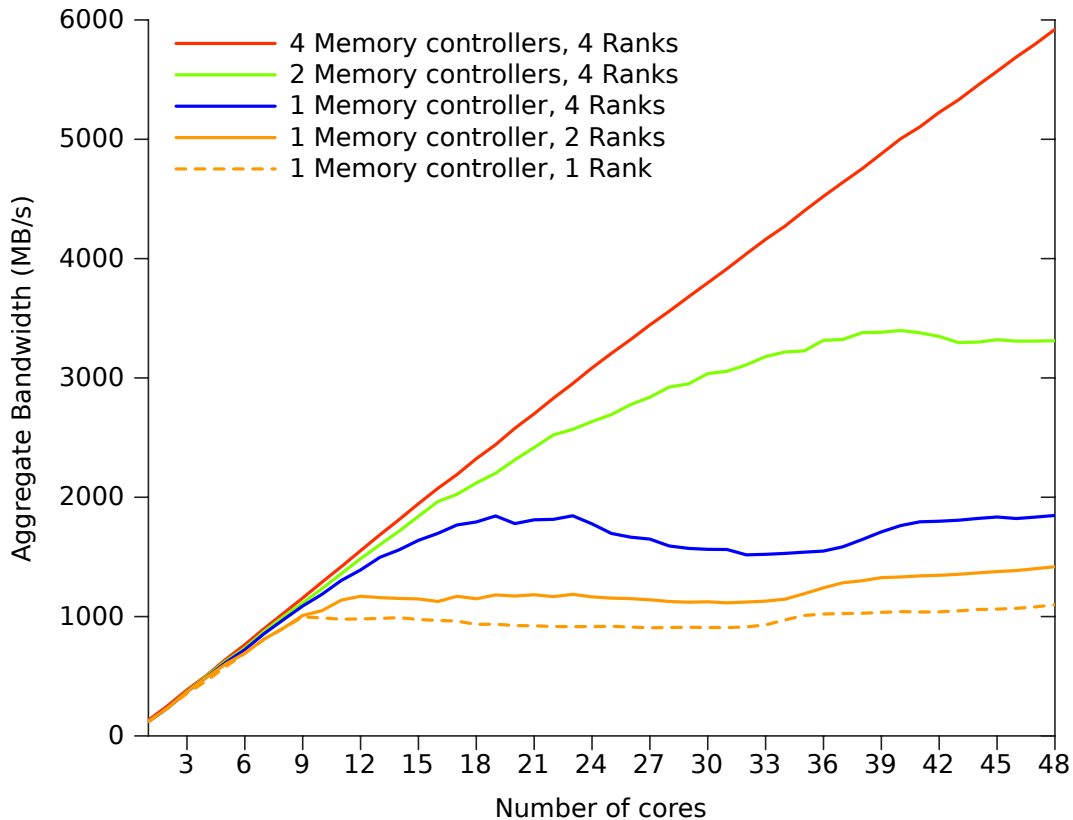


Figure 4.4: Memory throughput benchmark result showing the aggregate memory bandwidth of 1 to 48 cores performing memory reads divided over a different number of memory controllers and ranks.

turning an idle or waiting core to the lowest frequency.

4.5 Cacheable Shared Memory

The originally configured shared memory was only 16MB per memory controller, and could only be accessed with caching disabled due to coherency. This results in a horrible performance. To gain more out of shared memory, Intel came up with a special memory sharing idea. It is called *privately owned public shared memory* (POPSHM). The concept is that each core offers a part of its private memory and shares this (which can easily be done using LUT mapping) with some or all other cores. Caching is again an issue here, because the cores all have a private non-coherent L1 and L2 cache. The memory used in POPSHM can only be accessed through the library such that the library can take care of flushing the caches when necessary. This makes it quite cumbersome to use, as the shared memory still needs to be copied to private memory to do a computation. Since POPSHM relies on L2 cache flushing, it could only be used reliably from the moment that the cache flushing issues had been solved.

In addition to POPSHM, a Chinese Intel team developed Software Managed Coherency [32] for the SCC. They have published this, and it should be working, however the implementation has not yet been released due to licensing issues. The implementation is portable over multiple architectures. It supports systems with an OS per core such as the SCC as well as single OS systems. For the multi-

OS systems a virtualization layer has been implemented. The system acts as a virtual machine that intercepts system calls and handles the consistency on a page granularity following the release consistency model [10]. After an *acquire*, it captures all writes and synchronizes with a *Golden Copy* in the main thread at the point of *release*.

4.6 Running the SCC in baremetal

We experimented with three baremetal frameworks in an early stage of the development, but the conclusion after initial testing was that none of them would easily be usable in scope of this project. The frameworks are briefly discussed in this section.

4.6.1 ETI Baremetal Framework

E.T. International Inc. (ETI) made a baremetal framework for the SCC¹ which is a development toolchain for C programs with full support for libc, gdb and MPI, as well as a communication library that uses the MPB. Programs are loaded onto the SCC with a special launcher application from the MCPC. Output is forwarded to the MCPC which prefixes each line with the core identifier.

Although the framework runs for fairly simple programs, we were not able to use the beta version. The SCC kept crashing when we tried to run more sophisticated code. The code of the framework was not open for the community, being a *black box* with very restricted licensing. A new version was only recently released, and some community users have used this with success.

4.6.2 Microsoft Baremetal

Microsoft developed a framework in which we are able to run any user code directly on the SCC. This framework is a bit complicated, and the setup is as follows. The SCC stays connected to a Linux MCPC, which now runs a `sccTCPSTerver`, providing an interface to the SCC from any other machine over TCP/IP.

Microsoft uses a Windows machine with Visual Studio 2010 to communicate with the SCC through the MCPC. From Visual Studio with the SCC add-on, a user can select a special SCC project which generates a framework where you can write your SCC program in. The add-on adds a special SCC menu to Visual Studio with shortcuts to build and run the SCC project.

We used Windows Server 2003 within a VirtualBox running on the MCPC. After some experiments, we concluded that this framework was not suitable for an implementation of SVP. The environment was unstable in our set-up. We encountered random crashes of the system while loading and running application on the SCC. Most of our test programs did not run in the Visual Studio environment, and were not easy to port for preliminary testing purpose.

4.6.3 Bootstrap code and a C program

One of the MARC research groups put together the minimal code that is required to get a core ready to execute code at a given location in memory. They made a buffer in memory to which calls to `printf` at the cores write to. After the bootstrap,

¹<http://www.etinternational.com/index.php/projects/>

the framework calls an arbitrary C function that takes no arguments and no return value. A very minimal C library is available.

A program on the MCPC reads out the print buffers and writes the content to the terminal together with the core id, so we can distinguish between the different cores output. This is a very promising approach, because we can experiment with the architecture without any overhead of an operating system.

The problem we experienced was that the framework does not setup paging on the core. A page table is required to use for example the MPB and WCB due to configuration flags in the page table entry.

We decided not to put more effort into baremetal at the moment, because even when we could have setup paging, we are far from running an SVP run-time on the SCC. This would mean to implement threading ourselves.

4.7 Copy Core

After exploring the SCC and measuring its memory performance as described in the section above, we decided that introducing *Copy Cores* [27] was one of the best things to do due to the bad memory performance of the system. The idea is that we use dedicated cores that provide a service to copy memory areas. For this implementation we developed a lightweight protocol through the MPB as described in Section 4.4. We use the MPB to send small messages between cores. In this case a message contains the source and target address in real-physical memory, the size and optionally a core identifier. The core identifier is used to signal an other core than the sending one when the task completes. We must assume that memory that needs to be copied is contiguous in real-physical memory.

The problem we experience is that sccLinux uses virtual memory. Even though we can access all real physical memory locations from all cores, we do not know directly where the virtual memory address is located in physical memory. The memory device driver of sccLinux offers us a routine to obtain the core physical address from a virtual address. When we have the core physical address we can retrieve the real physical address from the LUT. As sccLinux uses a 4KB paging system, contiguous virtual memory does not necessarily need to be contiguous in core-physical memory. A small program has been written that allocates an amount of memory and finds the core physical address and the real physical address. Results show that 4KB pages of memory can be widely spread over the core physical address range, and therefore even so for the real physical address space. In our prototype implementation we did not deal with this, since the implementation was only made for performance measurements. Besides the virtual memory issue, we need to be careful when we exceed a LUT page boundary. We need to make sure that the next LUT entry maps to adjacent memory.

4.7.1 LUT Copy

The architecture of the SCC allows us to use a more eccentric method to copy memory. This relies on the fact that we have lookup tables behind a cores L2 cache. We can dynamically change the mapping of the LUT without the memory system of a core noticing this. Data can be copied from one physical memory location to an other using the following mechanism: At the start a LUT entry is pointed to the source physical address. At most 256KB (size of the L2 cache) is read into the L2

cache. We only need to access one byte to transfer a complete cache line. The same LUT entry is now pointed to the target physical address, and the data in the L2 cache is touched such that it will be in modified state. When the cache is flushed, the data is pushed out of the cache to new location in memory. Figure 4.5 shows a schematic overview of the LUT copy procedure. To copy larger chunks of data, a form of pipelining can be used when we alternately use one from a set of two LUT entries. When data is in the modified state in the L2 cache, we start reading from the source area through the second LUT entry. This will push out the modified data to the first LUT entry, and read fresh data into the cache. An additional flush is now not necessary anymore. LUT copy has a few downsides, to make use of lutcopy we must ensure that there are no context switches or what so ever during the transfer as it would influence the data in the cache. One way to achieve this is running in baremetal in a single process environment. To use this under sccLinux, we could implement LUT copy in a kernel module and disable interrupts during the copy process. Switching LUT entries causes a little overhead, but we think this can be compensated by the fact that lesser data (only a single byte per cache line, instead of each byte) needs to be loaded into the core.

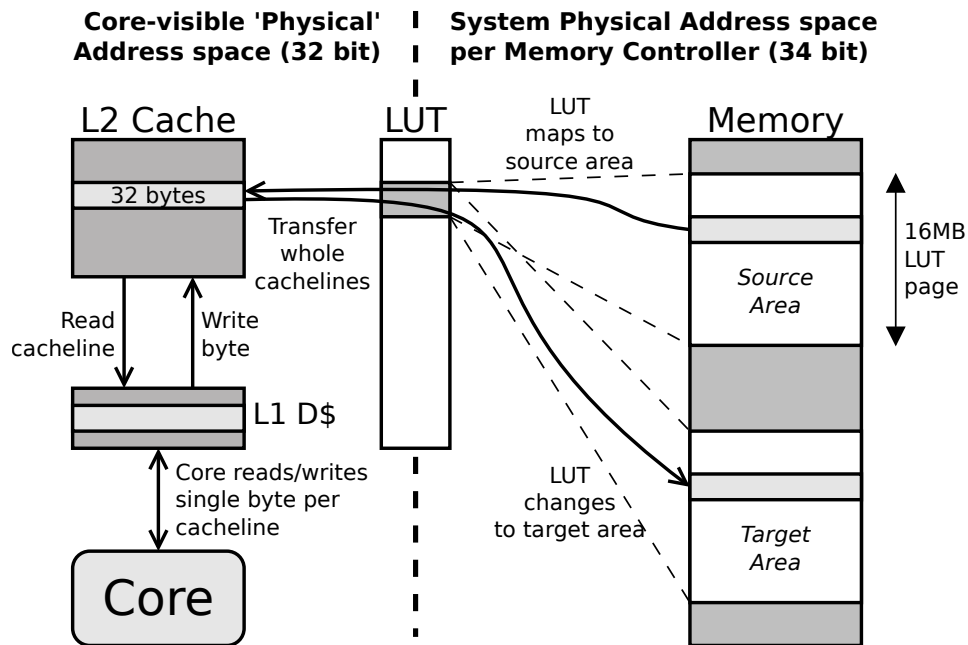


Figure 4.5: The approach to copy memory through the L2 cache.

4.8 Performance of Copy Cores

As a first evaluation of Copy Cores for the 3rd MARC symposium paper [27], we implemented the functionality to copy real physical address ranges. We evaluated the performance of both the normal `memcpy` function and LUTcopy. The result is visualized in Figure 4.6. In this graph we see that using the LUT copy method is in most cases not more efficient than a the regular `memcpy` operation. Only in the case that we need to copy 16 MB or more and we are using 4 copy cores for this, we get a slightly better performance. We see that dividing a copy operation over multiple copy cores leads to an almost linear speedup. Programs that need to copy

large amounts of data can benefit from copy cores due to the parallel execution.

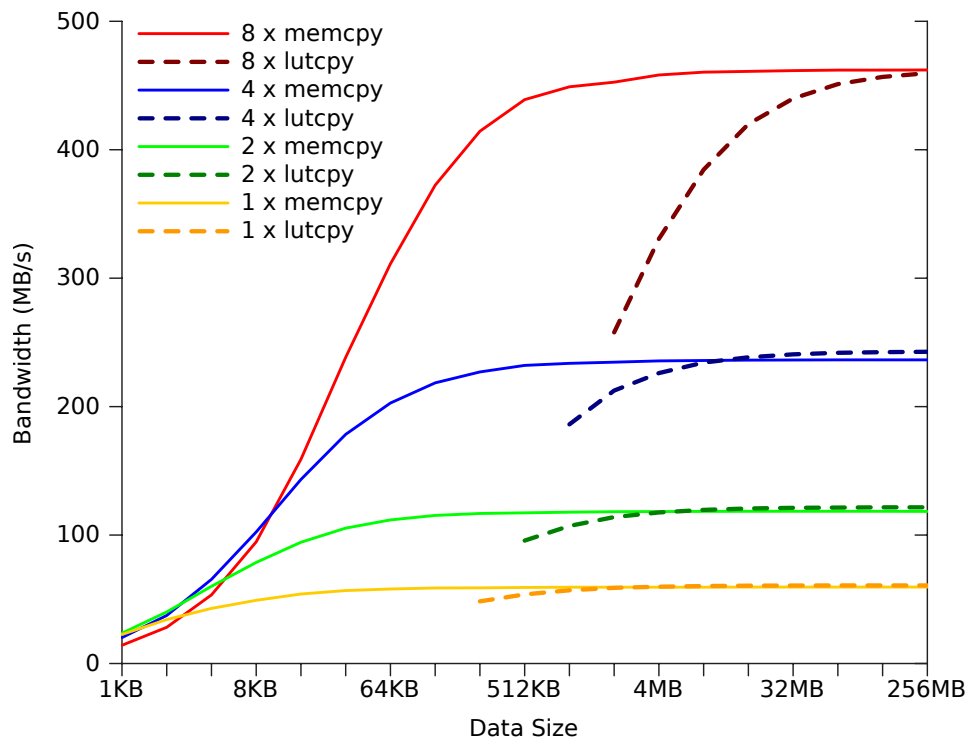


Figure 4.6: Throughput for the Copy Core.

The SVP run-time on the SCC

5.1 The easy way: TCP-IP

At the start of this project, there was already a distributed implementation of SVP available that uses TCP/IP for communication between places [29] that do not need to be on the same physical machine. Since the SCC cores can communicate with each other through TCP/IP, this implementation runs on the SCC with very little effort. However, as described in Section 3.3, the implementation causes overhead that is necessary to support heterogeneous platforms which may have a different data representation. All information that needs to be shared is captured in a buffer which is encoded with XDR and then sent to the other *place*, where it needs to be decoded and put in the correct data structures. On the SCC we can avoid this, since it is a homogeneous system with the potential to use shared memory.

5.1.1 Performance

Problems we encountered with the implementation were mostly due to bugs in the SCC. When we generated too much traffic between the SCC and the MCPC, the system crashed. Intel solved this issue by enabling one of the Ethernet ports to split control messages from data. An additional Ethernet card was installed in the MCPC and connected with an Ethernet port on the SCC. With some initial tests of a distributed matrix multiplication algorithm we experienced that the core that distributes the work to the other cores cannot keep the other cores busy. Many of the workers are waiting for a new job, while the initiator keeps busy. The main issue is the communication, which is sequential in the SVP implementation and slow on the SCC. A remote create call will only return when data communication is finished. Consecutive remote creates with no dependent sync in between can theoretically be issued in parallel.

5.2 SCC specific implementations

After we investigated the properties of the SCC system, we came up with a few possibilities to speed up the communication. As all communication functions are separated in a module which implements an abstraction of communication functions it is not very difficult to implement additional communication channels. It is still based on the idea of sockets in the way that there is a connect phase that sets up a

virtual channel on a physical device. This virtual channel can be used to send and receive data independent from other virtual channels.

5.2.1 Using (i)RCCE

The first approach was to make use of the existing libraries RCCE and iRCCE to replace the communication layer of the existing implementation. We could easily set this up and replace all send and receive calls with the appropriate iRCCE functions. For the connection establishment we use an iRCCE waitlist with a receive request for each possible sending core as described in Section 4.2.1.

We experienced some problems with this implementation. The (i)RCCE implementation seems to only match requests on core identifier. As a result, we cannot have multiple outstanding receive requests for a single sending core. The first message that fits the size will complete. Therefore we cannot issue another receive request in the connection establishment function while there is already a connection active. The messages in the connection establishment function are rather small, and therefore match any other sized message. We see that messages sent through a previously established connection now initiate a new connection, and fail thereafter. iRCCE does not support virtual channels, and is not usable in the SVP implementation in its current form.

5.2.2 Dedicated MPB protocol

Due to the problems with iRCCE we decided to implement our own MPB protocol. This protocol uses the complete core MPB for a single communication shot. A core's MPB space (8KB, the MPB is split equally between the two cores on a tile) will be used by other cores to put messages for that core (*push communication*). The receiving core can get the messages out of its own MPB. At the moment it uses polling to check the MPB for messages, but interrupts could also be an option. Polling has more overhead, while interrupts are more difficult to handle as they can occur at any time, also when a message is not expected. Only a single core may access MPB at a time. The core lock register is used to arbitrate the usage of the MPB for that core. This locking protocol causes overhead, and results in a lot of busy waiting when several cores want to send to the same core. Messages that do not fit the MPB will be split in multiple communication shots.

To allow for virtual channels, each message that will be put into a cores MPB has two tags. First is the core identifier of the sending core, the second one is a channel identifier for that core. Each core has a server thread running that checks the MPB for messages with a channel ID larger than the latest known channel identifier. If it discovers such a message, it initiates a new thread that handles this new connection. That new thread is now responsible for all incoming messages with the same core and thread identifiers.

One problem that we experienced with the existing implementation is that we use virtual channels over a physical channel that can only be used once. All virtual channels run in their own thread at the remote side where each incoming connection will set up a virtual channel that is handled in its own thread. At the local (creating) side, one thread can issue multiple requests to the same core resulting in multiple virtual channels in the same thread. In this implementation this can lead to deadlock. For example, in the following execution sequence given in pseudo code:

```
create(family1, place1);
```

```
create(family2, place1);
sync(family2);
sync(family1);
```

both families are created at the same place. Now, the sync instruction is blocking, so the sync for family1 is not issued before the sync of family2 completes. At the remote place, the execution of family1 may complete before the execution of family2. In this case the sync message for family1 is put in the MPB of the issuing core, and blocks the physical channel until it is taken out. At the issuing core, the message will not be taken out of the channel, because that thread is waiting for a message on the virtual channel for family2. The channel remains blocked, resulting in deadlock.

We solved this issue by introducing an additional message from the issuing core to the remote place. It now explicitly requests the synchronization of a family. The remote place will not send the synchronization message before the sync request is received. Of course, this causes additional overhead.

Given the small size of the MPB compared to the potentially large amount of data that we need to transfer, and given that the overhead that it takes to transfer data in chunks through the MPB, the MPB might not be the most efficient method to use.

The problem with virtual channels blocking the physical channel might also be solved by splitting the MPB in chunks, where each chunk represents a channel. In this way, a single communication shot does not block the complete channel, but only part of it. In the best case we want to have unlimited virtual channels. However, this makes the problem of the MPB being small only more relevant. An other option is to temporarily buffer unexpected messages. When a thread finds an unexpected message in the MPB, it can take the message out of the MPB, and store it in off-chip memory. The channel is now free from blocking, but throughput is even worse than using off-chip memory directly.

5.2.3 Direct data transfer

Before sending data to a remote place, the current implementation encodes and gathers all data in a buffer using XDR. The XDR library provides a data representation that is not machine or architecture specific since the SVP protocol can run on a heterogeneous set of systems. The SCC is a homogeneous system and as long as we do not want to mix the SCC cores with other systems, the XDR encoding just causes overhead. We adapted the socket implementation in such a way that it does not encode the data with XDR, and an additional copy to a send buffer is not required anymore. Each data element that is part of the data description function will now be sent separately through the socket. For scalar values this causes a large communication overhead, since they are now all pushed separately through the channel. For (large) arrays this means a great reduction in the overhead of encoding and copying. We adapted the data description function a bit to support sending arrays in a single shot. To send arrays, we no longer need to explicitly touch each single element of the array, but just provide a pointer to the first element, and the number of elements in that array.

For the fibonacci example, the new data description function is listed in Figure 5.1. The original function is listed in Figure 3.3

```

2  DISTRIBUTABLE_THREAD(fibonacci_compute)(DataBuffer* db, void *conn,
    int*& prev, int*& prev2, int& size, int*& fibonacci)
    {
        SINPUT(prev);
        SINPUT(prev2);
        GINPUT(size);
7   ARRAY_SIZE(fibonacci, size);
        GINOUT_ARRAY(fibonacci, size);
    }
    ASSOCIATE_THREAD(fibonacci_compute);

```

Figure 5.1: New data description function for the fibonacci example, for direct communication.

5.2.4 Memory Remapping

The SCC allows us to remap memory from one core to another. This means that we can avoid the communication of large data chunks through a channel, and share our memory directly with another core.

This seems easier than it actually is due to problems with virtual addressing. Linux uses a virtual memory system which means that the addresses seen in a program (in process) are not the same as the physical addresses. It is not known where in physical memory the data actually is. A function in the special SCC Linux memory kernel driver can provide the address in physical memory given a virtual address. The sccLinux memory system uses 4KB pages. Chunks of contiguous virtual memory that span more than one page, do not necessarily map to contiguous core-physical memory. Once we know the core physical address, we do not need to worry that it might change as it could on a regular system which uses a swap mechanism. The SCC cores do not have swap space. Once the core-physical memory address is known, the real-physical address can easily be obtained from the LUT.

Results from a test program that has been written to gain insight of address ranges shows as expected a non contiguous mapping between virtual and physical memory. We have to realize that we can not use memory remapping without changing the way memory is allocated.

To make effective use of memory remapping, we need to maintain our own memory region in our implementation. We use an sccLinux image that has only 320MB of private memory configured, which leaves about the same amount of memory for the application to manage as there is 656MB of private memory reserved for each core. All memory regions that may be transported through memory mapping, must be allocated through our own allocation function. The memory is allocated using a very simple first-fit algorithm.

When using memory remapping, we can either use cached memory or non-cached memory. In order to use cached memory, we must flush the L2 cache at both sides. Due to the write back caching method, we must flush the L2 cache at the sending side to make sure all data will be in physical memory. As we already pointed out, flushing the L2 cache is a very expensive operation. There is an option to avoid the L2 flush at the receiving side. As we map the memory with the MPBT tag on, the L2 cache is bypassed, but multiple accesses within the same cache line will hit in the L1 cache. We only have to issue the cheap CL1INVMB instruction that

invalidates all data in the L1 cache with the MPBT tag. The use of non-cached memory is also expensive, since every access needs to go to main memory then.

In this implementation, the sending core will send the core physical address through the socket to the receiving core. The receiving core checks the LUT of the sending core to obtain the real physical address. It will dynamically map this address on free LUT pages and start a the memory copy operation to the receive buffer.

As the initial socket communication, memory mapping procedure and cache flushing will cause overhead, the implementation should only use memory remapping as the message is large enough to compensate for this overhead.

Evaluation

6.1 Benchmark Programs

6.1.1 Ping Pong

The first benchmark that we use is a program to measure the latency and throughput on the network channel. This application is a simple Ping-Pong which sends chunks of data with incremental size to the remote place. The only thing that the remote place does is sending that data back immediately. The data will be transported as an array of integers. The sizes we measured range from 1 up to 8M integers (32 bit), resulting in a data transfer of 4 bytes up to 32MB.

6.1.2 Matrix Multiplication

A benchmark that fits the distributed implementation of SVP with the potential to copy lots of data is matrix multiplication. We implemented a decomposition algorithm that splits a matrix in sub matrices and performs the calculations on sub matrices only. Figure 6.1 shows the decomposition algorithm. We can apply the decomposition recursively as long as the square matrix size is still dividable by two. Each step splits the calculation in eight parts that can execute concurrently, followed by four additions that can also execute concurrently. Note that the addition can be performed on the individual sub matrices, or on the combined larger matrices. In this example we perform the addition on the sub matrices, since this exposes more concurrency without changing the representation again. This implementation works on square matrices and operates on double precision floating point values.

$$A \times B \rightarrow \begin{vmatrix} a1 & a2 \\ a3 & a4 \end{vmatrix} \times \begin{vmatrix} b1 & b2 \\ b3 & b4 \end{vmatrix} = \begin{vmatrix} a1 \times b1 & a1 \times b2 \\ a3 \times b1 & a3 \times b2 \end{vmatrix} + \begin{vmatrix} a2 \times b3 & a2 \times b4 \\ a4 \times b3 & a4 \times b4 \end{vmatrix} = \begin{vmatrix} c1 & c2 \\ c3 & c4 \end{vmatrix} \rightarrow C$$

Figure 6.1: Matrix decomposition: Matrices A and B (both $N \times N$ are split into four $\frac{N}{2} \times \frac{N}{2}$ matrices each. Eight matrix multiplications and four matrix additions are performed on the sub matrices. From the resulting four sub matrices (c1 to c4), the result matrix C can be constructed.

In order to make the decomposition more time and space efficient, the matrix

representation in memory is a column of pointers that all index a row in the matrix. All matrix rows together form one contiguous block of memory. This representation is visualized in Figure 6.2. The normal lines indicate a pointer to an element in memory, while the dashed lines refer to the same element in the corresponding matrix. Pa, Pb, Pc and Pd are pointers to arrays with pointers to sub matrix rows. This allows us to do the decomposition by creating a new array of pointers and assign the pointers to elements in the original matrix rows. There is no need to copy data to change the matrix representation.

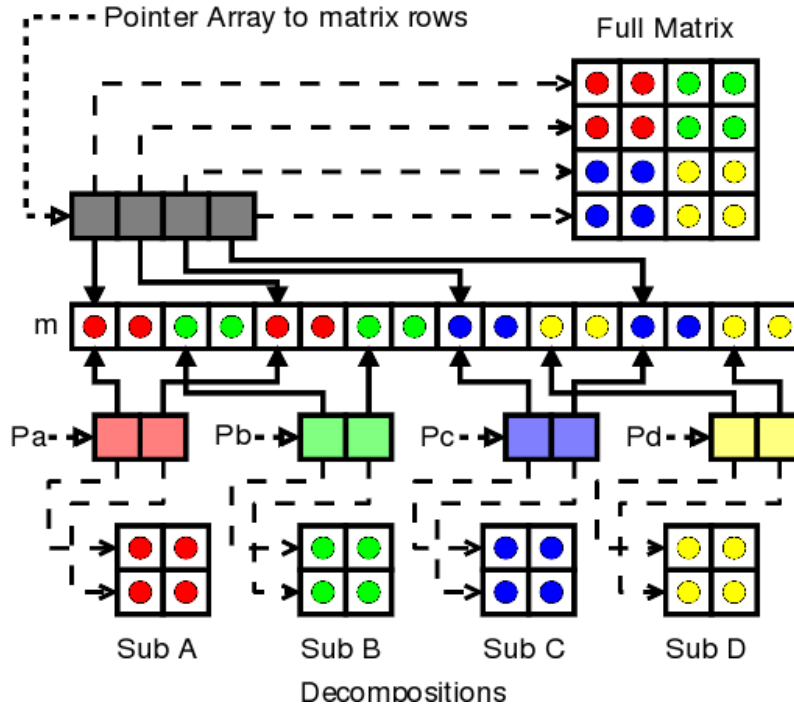


Figure 6.2: Representation of the original and decomposed matrices in memory.

We also have a version of the matrix multiplication benchmark in which there is only one master node that decomposes the matrices and sends the sub matrices to worker nodes. Experiments have shown that only a single master node can not keep the other cores busy when we use two decomposition steps creating $8 \times 8 = 64$ concurrent multiplications on only 47 (or 48, when the master is included) nodes. The overhead for the decomposition and communication is too much for the master node. In the TCP/IP implementation, we could *cheat* a little, by letting the much faster MCPC act as master node. For the SCC-only implementations we need an other solution. We let one master node do a single recursion step, and then delegate the work to eight worker-master nodes which in turn create the 64 tasks for the actual worker nodes. In this way, we divide the communication overhead over multiple nodes.

6.2 Results

6.2.1 Ping Pong

The first results obtained are from the Ping Pong benchmark. We benchmarked the program against the original implementation, and four different implementations that have been made using the techniques described in Section 5.2:

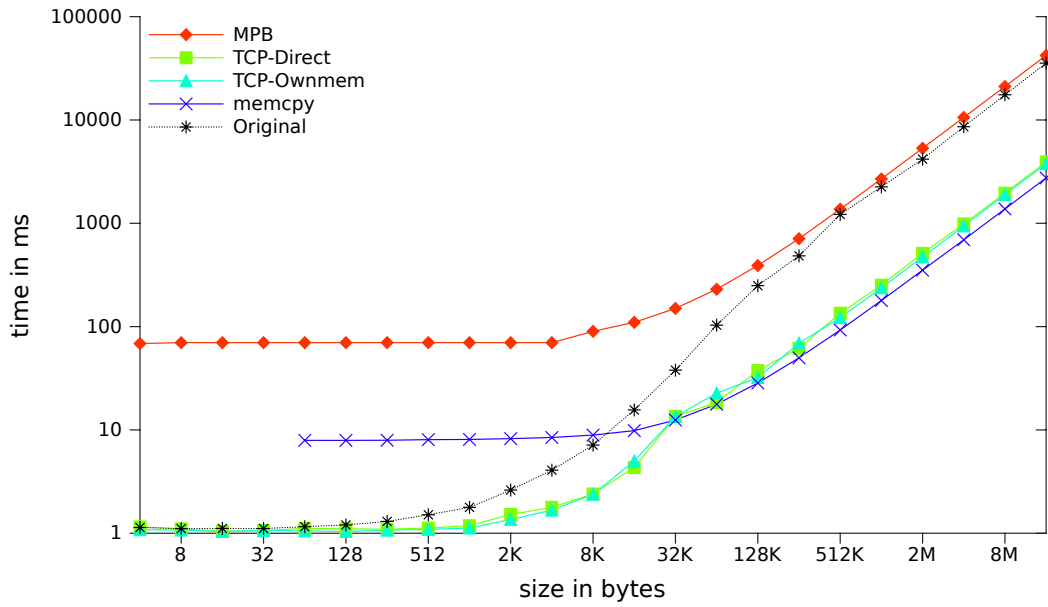
1. The original implementation.
2. An implementation which communicates data directly without copy operation to a buffer and XDR encoding.
3. An implementation as 2, but uses our own memory allocator.
4. An implementation as 2, but uses the MPB as communication channel.
5. An implementation as 2, but uses a memory copy operation from remapped memory.

In Figure 6.3, the results of the Ping Pong benchmark are visualized in two graphs. The first graph (Figure 6.3a) shows the actual running time of the thread creation, followed by a synchronization directly thereafter. All results are the minimum over 10 measurements, as the TCP/IP implementation on the SCC suffers from timeouts regularly, generating some outliers that would large impact on the average. The second graph (Figure 6.3b) shows the speedup of the implementations 2-4 with respect to the original implementation. For the remapping implementation results are shown for messages of 64 bytes and larger. As the communication of the addresses has to go through TCP/IP, the threshold for using remapping is set to 64 bytes.

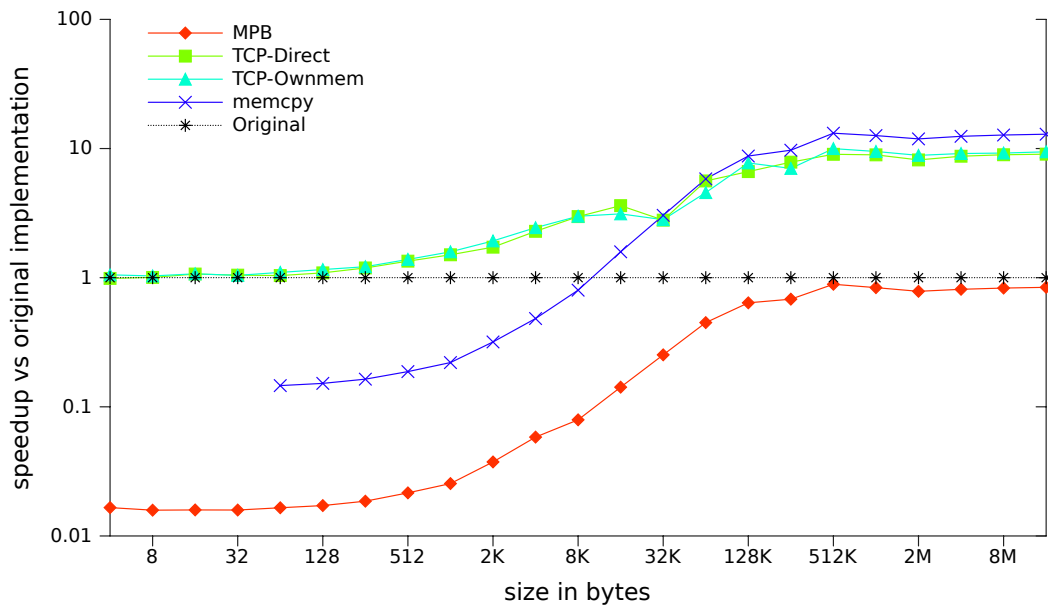
We see that using the MPB approach does not result in any speedup at all. The overhead in latency due to locking the core and polling is about seven times higher as the values obtained from a TCP/IP implementation, and the throughput is worse due to several of these communication shots that are required for large messages.

The other implementations show a significant speedup. The direct communication approach reduces the execution time by almost an order of magnitude compared to the original implementation. The speedup peaks at a factor of 9.5 for messages larger than 512KB. We obtain the same results for the direct approach that uses our own memory manager as for the implementation that uses memory allocated by the OS.

The memory mapping approach is only efficient for large messages due to the L2 flush that is required, and some overhead on mapping the memory. From a message size of 8KB it is faster than the original implementation, but to be more efficient than direct communication, messages need to be at least 32KB. Remapping memory has a maximum speedup factor of 13.



(a) Running time for the different implementations.



(b) Speedup versus the original implementation.

Figure 6.3: Results for the Ping Pong benchmark on the SCC.

6.2.2 Matrix multiplication using one decomposition step

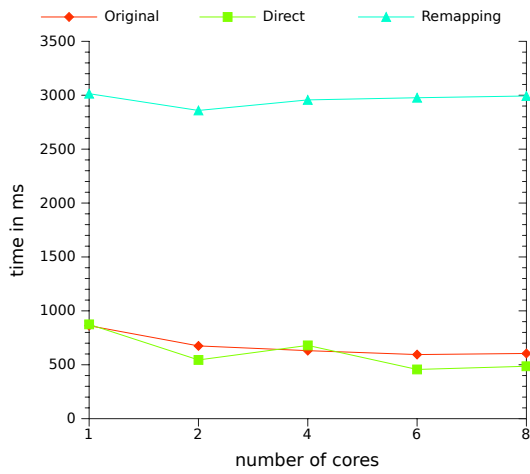
We ran the matrix multiplication example as described in Section 6.1.2. In this benchmark we only use one decomposition step, resulting in eight remote creates. We ran the benchmark for square matrix sizes of 128, 256, 512 and 1024 elements, resulting in sub matrices of half that square size. We use one master node that initializes the matrices, performs the decomposition and distributes the work over 1, 2, 4, 6, and 8 remote places. The results are visualized in Figure 6.4. All results are the average over three runs.

In this section, we benchmarked three implementations:

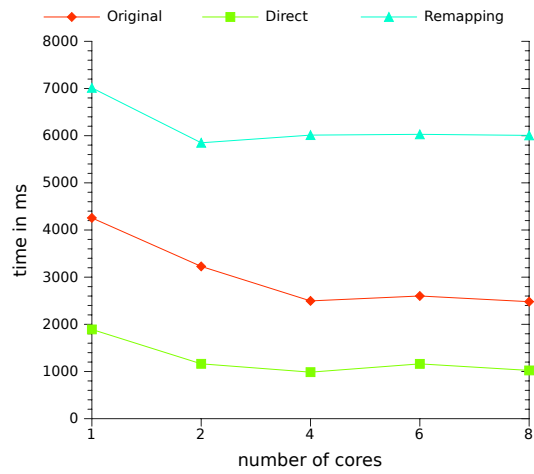
1. The original implementation, referred to as *original*.
2. An implementation which communicates data directly without copy operation to a buffer and XDR encoding, referred to as *direct*.
3. An implementation as in 2, but uses a memory copy operation from remapped memory, referred to as *remapping*.

For a small matrix size of 128×128 elements (Figure 6.4a), we see the impact of the large overhead of L2 flushing in the remapping implementation. The original and direct implementation perform about the same. In this case, the messages are rather small, since the matrix multiplication program uses pointers to the different rows in the matrix. All messages consist of 64 double precision floating point elements of 8 bytes each, resulting in a message size of 512 bytes. Using the direct implementation, each of these messages will be send separately, resulting in communication overhead. In this case, the communication overhead is about the same as the encoding and copy overhead in the original implementation. From the horizontal lines we can conclude that spreading the load over multiple cores does not result in a speedup. The computation is too small compared to the communication overhead.

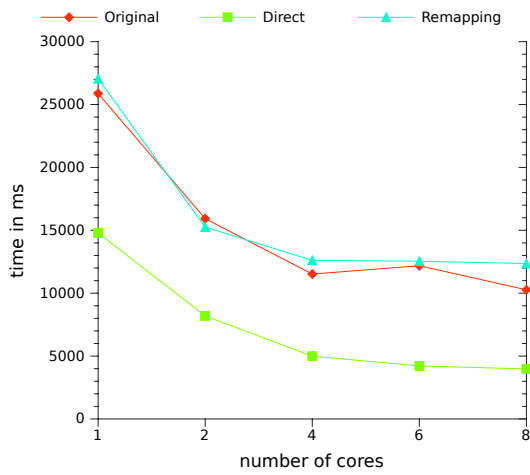
In the second graph (Figure 6.4b, matrix size 256), we still not see a lot of speedup although there is a difference between 1, 2 and 4 worker nodes. The data size per message increased to 1KB, which is enough to make a clear difference between the direct implementation and original implementation, where the direct implementation performs better. The third graph (Figure 6.4c, matrix size 512, message size 2KB) shows the point at which the remapping implementation and the original implementation perform about the same. We also begin to see speedup as number of cores increases. The last graph (Figure 6.4d, matrix size 1024, message size 4KB) finally shows that the mapping implementation performs better than the original implementation. The direct implementation still performs the best. We see speedup for all cores added to the worker pool.



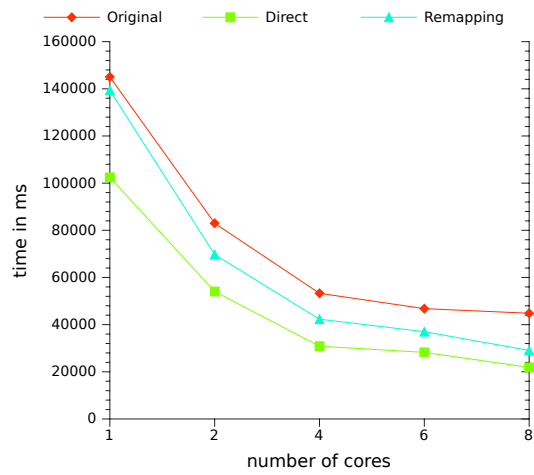
(a) Matrix size 128 × 128.



(b) Matrix size 256 × 256.



(c) Matrix size 512 × 512.



(d) Matrix size 1024 × 1024.

Figure 6.4: Matrix multiplication benchmark with 1 decomposition step.

6.2.3 Matrix multiplication using two decomposition steps

The matrix multiplication benchmark exposes eight times the concurrency as an additional decomposition step is performed. However, an additional recursion step leads to more communication overhead due to messages of a smaller size. We have chosen to benchmark matrices of 1024×1024 , and 2048×2048 elements. We were unable to run the 2048×2048 benchmark on the original implementation as there is not enough memory available at the master node to store all matrices and communication buffers. The application fails at a late stage of receiving result data from worker nodes while allocating memory to buffer the received message before decoding. The implementations that do not use communication buffers, but direct transport have enough memory to run this benchmark.

In Figure 6.5, we see no significant speedup for the original implementation when using additional cores, while the direct implementation has some speedup up to 20 cores. The explanation for this effect is that the master node is fully occupied with the distribution of tasks while most of the workers are waiting for new task. Note that for remapping the 2048 matrix takes twice as long as the 1024, but with the direct implementation the time required for 2048 is four times the time required for 1024. We can conclude that the remapping implementation has better scalable communication. This can be explained by the fact that the remapping approach is implemented such that the sender only needs to flush the L2 cache, and send the address to copy data from to the receiver. For the sending side, communication has finished at this stage. The receiver performs the memory copy operation. This reduces the communication time required for the master, so it can distribute tasks faster.

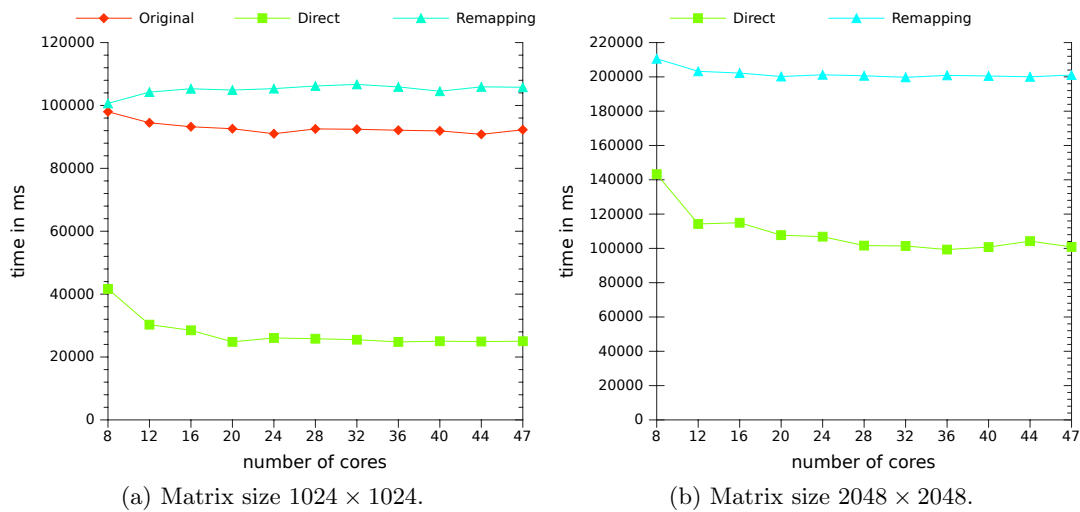


Figure 6.5: Matrix multiplication benchmark with 2 decomposition steps.

6.2.4 Alternative approaches

Recursive decomposition over multiple nodes

A solution that decreases the load of a single master node, is to split the recursion steps over multiple nodes. The master node performs one decomposition step and distributes the tasks over one up to eight worker-master nodes. The worker-master nodes perform an additional recursion step and distribute the work over four or five worker nodes each. We first need to distribute the data for the first recursion step from one node to several others, which have to distribute the data further. This approach introduces a lot of additional communication, as the decomposition on a single node can be done without additional copy operations due to the pointer to matrix row implementation of the decomposition.

The benchmark results of the original and direct implementation visualized in Figure 6.6 show a reduction in time for the original implementation with a matrix size of 1024 compared to two decomposition steps on a single node. The direct implementation has about the same runtime for both matrix sizes. Due to the extra communication we create, we can not achieve a speedup with approach.

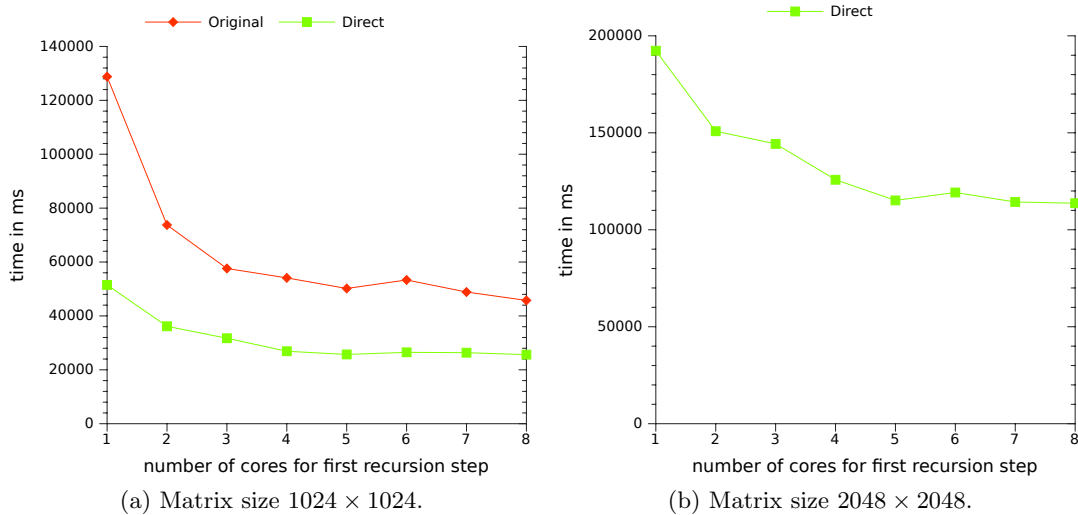


Figure 6.6: Matrix multiplication benchmark with distributed decomposition.

Related Work

Architectures comparable to the SCC

The SCC is currently one of a kind. No other processor features that many fully functional cores on a single chip. There are however other architectural approaches.

For example the Cell Broadband Engine [17]. The Cell BE processor is based on the Power architecture and features only a single fully functional core, the *Power Processing Element* (PPE), and multiple SIMD cores, referred to as *Synergistic Processing Elements* (SPE). The Cell bridges the gap between conventional desktop processors and more specialized high-performance processors, such as graphics-processors. The Cell processor is for example used in Sony's PlayStation 3. The PPE has control over the SPE's and can start, stop, interrupt, and schedule processes on the SPE's. In the SCC we can only control some parts of other cores directly. Full remote management has to go through a software layer. The PPE and SPE's are linked by an internal high *Element Interconnect Bus* (EIB). The communication mechanism was first designed as crossbar switch, but later moved to the EIB because that was cheaper in silicon area. The EIB is a ring consisting of four 16B-wide unidirectional channels. The SPE's work on their own memory (Local Store). Data needs to be transported between the local stores and memory. Transfers between local stores and memory are coherent in the system. The local stores are also addressable by the PPE.

The original Niagara [18], used in the UltraSPARC T1 was the first multi-threaded multi-core processor. The processor has up to eight cores with four threads per core. It is a shared memory system, and caches are coherent. The use of multiple threads per core allows for latency hiding of long latency operations as threads are temporarily on hold while the operation is in progress. In the meantime other threads can execute. In later generations the amount of cores and threads per core were increased to 8 cores with 8 threads each for the T2 and 16 cores with 32 threads per core in the T3. For the T4 that has just been released, the amount of cores and threads drops again to 8 cores and 8 cores per thread while increasing the sequential performance per thread.

In the SCC, all cores are independent systems that run their own operating system, while for the other systems a single OS controls the complete system. Virtualization techniques make it possible to divide the single OS system into multiple virtual systems, or make a multi OS system appear as a single system.

Research on the SCC

The SCC related work in this section is for a large part based on the work presented at the 3rd MARC symposium in Ettlingen, Germany. This was the first community event where SCC researchers could present and share their work. At the first MARC symposia there were only a few invited speakers. A few papers with work related to our work will be discussed here. Most of the work described in these papers was carried out in the same period as the work presented in this thesis.

The authors of [21] explored the possibilities for data-intensive workloads on the SCC. They use Google's MapReduce programming model for dividing data intensive workloads across multi-core systems. They use RCCE for the communication. They report linear speedup, however, this is only measured relative to execution on four cores. They are not clear on the exact benchmarks and data sizes, and it is not clear how the initial data is distributed over the cores, and if they have taken that into account in the measurements. They make a vague comparison with a Cell processor, where they state that the SCC performs better. In the future work they state that they want to make use of the shared memory possibilities of the SCC.

In [23], the author presents a view on how to implement message passing protocols in general, and the SCC in specific. The focus here is low latency for very small messages (< 200 bytes). The author measured the latency in cycles for the mesh network. These figures match our results, taken into account the difference in core and mesh speed.

A second project on the SCC is currently carried out at the University of Amsterdam. This project is exploring the possibilities of a more advanced implementation of the S-Net run-time on the SCC [30]. S-Net is a declarative coordination language for describing streaming networks of asynchronous components. Just as SVP, it needs asynchronous communication and therefore faces similar problems. In S-Net there are many small messages, while in coarse-grained SVP messages can be large, making the use of the MPB in S-Net more feasible.

The Barrelfish team developed an SCC specific version of their operating system [22]. They use a technique in which they notify cores through Inter Processor Interrupts (IPI), put meta-data of at most one cache line (32 bytes) in the MPB and communicate the actual (potentially large) message data through off-chip memory. The Barrelfish operating system fits the SCC architecture closely. The SCC report states several issues we also discovered during our research. For example due to the small size of the MPB, it can not efficiently be space multiplexed nor time multiplexed. They also wish for more fine-grained control over the L1 cache and control over the L2 cache. Also not allocating on a write miss in the caches is seen as shortcoming of the architecture. Fortunately the use of the WCB can make up a little for this.

A detailed performance analysis of the SCC is published in the IEEE International Conference on Cluster Computing (Sept. 26-30, 2011) at the time of finishing this thesis [11]. The analysis with respect to memory performance reflect our own measurements and conclusions. They describe a mechanism for communication through shared memory via fixed shared memory regions. The mechanism works in three steps where (1) the memory is copied into a shared region, (2) the receiver must be notified, and (3) memory is copied from the shared region to private memory from the receiver. We have shown in our implementation that we can do on-demand mapping which requires only one memory copy stage. In addition, the power management tools are benchmarked, which are of interest for our future work.

Related Work on SVP and DSVP

Most of the relevant related work to the coarse-grained (distributed) SVP implementation and distributed memory systems was recently documented in [29].

Relevant for the work presented in this thesis is X10 [7]. X10 is an object oriented language based on Java for high performance and high productivity Non-Uniform Cluster Computing (NUCC). A prototype implementation for the SCC has been made [6]. X10 has a lot of similarities to SVP, it has the notion of places, uses *async* for the create of an asynchronous task, and *finish* for a sync. Just as in SVP, the goals are to provide a deadlock free, scalable and flexible programming model. The SCC prototype implementation uses MPI and RCCE communication. The X10 implementation uses the same concepts as our original DSVP implementation: the X10 communication API sends serialized object graphs. The findings about RCCE are the same as ours: RCCE send/rcv are blocking, and RCCE receive calls require knowing the sender. The X10 implementation is a bit simpler than ours, they only allow a single worker thread per place, making RCCE suitable for communication.

Habanero-C [1] is a project based on X10 but based on the C language. The Habanero goals are lightweight dynamic task creation and termination using the X10 constructs, collective and point-to-point synchronization, mutual exclusion and isolation, and locality control using hierarchical place trees. Also for this project a port to the SCC has been made [20]. They mainly discuss work stealing and work sharing algorithms and scalability of those, and implemented queues for this in the MPB, similar to what we have done in our copy core implementation.

Conclusion

We have presented an SVP implementation that reduces communication overhead on a homogeneous system as the SCC by almost an order of magnitude. The main reason for this speedup is avoiding the unnecessary data copy and encoding operations. We have shown that making use of the SCC property to remap memory from one core to another gives a significant communication speedup for large messages. As a consequence of the reduction of communication overhead, there are more independent smaller communications, making the memory remapping approach less effective. The memory remapping approach is not efficient for small messages due to the overhead of remapping and flushing the L2 cache.

We need more control over the L2 cache to make efficient use of shared memory. A flush instruction to flush the complete cache or flush only data that has been tagged as shared memory would provide a faster flush, reducing the current overhead of flushing the cache.

The SCC has poor performance compared to a normal desktop machine or server, but we should keep in mind that the SCC is a research platform and Pentium 1 cores were used in its design. In that sense we can not compare the SCC performance to any other state of the art architecture. Our main issue with the SCC core is that it can have only one outstanding memory operation, resulting in a poor memory performance. A core that supports out of order execution would perform better on this.

The added value of the MPB seems good in theory, but our results show in practice that they are not as usable as one might think. The best latency for the local MPB is 60 core cycles, while the worst is 102 core cycles. Latency for accessing main memory is only slightly worse, adding 46 memory controller cycles. We should note here, that a working bypass would have reduced the latency for the local MPB significantly, so this issue has already been taken into account and occurs only due to a bug in the prototype hardware. The size of the MPB is rather small, so it does not allow for high flexibility as a communication channel without ending up copying data to off-chip memory anyway. The MPB can efficiently be used to share small datastructures between cores as we used in the copy core implementation. The SVP MPB implementation suffers from large overhead due to locking and polling, and the absence of a suitable virtual channel approach over the MPB.

Regarding to the proposed copy cores which in our case waste the silicon of a fully featured x86 core, special smaller DMA engines could be designed to perform this task. When these are put closely to the memory controllers, we can benefit optimally.

The fully featured cores should only have to deal with the real computational work, while the DMA engines, together with a very fast local memory per core, would allow for background data communication, either between multiple cores or between a core and off-chip memory. The local memory could for example be a larger MPB with a working bypass.

The mesh architecture seems scalable, but the current layout where all memory controllers are located at the edges of the chip, introduces a latency difference between the cores. With thousands of cores on such a chip, the resource allocator needs to take the location of a core into account for the placement of tasks. Tasks that require more memory accesses needs to be placed at cores that are close by a memory controller. This would make the homogeneous architecture appear heterogeneous. The placement of tasks is also import when tasks are communicating directly (core to core) as the latency and throughput is dependent on the distance in the mesh.

However, even with these few drawbacks in mind, the SCC is still a promising architecture, allowing researchers and programmers to develop new programming models and prepare for future architectures. Hopefully, Intel will take the advices provided by the MARC community for future prototype architectures. As cache coherency and shared memory do not scale between many-cores, it might be an option to support coherency and shared memory between a number of cores. As voltage island have been introduced that share the same voltage at any time, coherency islands would also be an option, effectively creating a *many- multi-core* architecture.

Future Work

Even though we have shown a significant speedup in the implementation presented in this thesis, there are still several possibilities for an even more efficient implementation of SVP.

Data distribution can be more efficient when it is handled in parallel. In the current implementation, a remote create returns when the distribution of the data is finished. In theory, multiple consecutive remote creates can be issued in parallel, as long as there is no dependent sync in between. The data distribution for a single thread may be performed in parallel by using copy cores. With some restrictions to the use of data (for example read only or exclusive access for the remote place), we can choose to only remap and use data from one core to an other without copying it. This can be captured by the data description functions in the DSVP implementation.

The overhead of locking and polling in the MPB implementation could be reduced by using an alternative protocol that requires less locking and uses interrupts instead of polling. However, as used under sccLinux interrupts still cause a lot of overhead as described in [30]. There are two context switches required between kernel and userspace, and the user application has to stall to handle the signal. More experiments and results from the S-NET implementation are required to investigate this.

As the current placement of tasks is managed statically in the SVP program, this does not allow for high flexibility. A resource allocator should dynamically map tasks to places (a core, or a set of cores in case of the SCC) depending on certain values as load and availability.

The use of baremetal was out of the scope for this project due to complications we encountered during initial tests and the limited time frame of this project. The current effort of the community to work on better baremetal frameworks and a better insight of the architecture could be a trigger to reconsider an SVP implementation in bare-metal (as an *SVP kernel*) using a light-weight threading mechanism. This would decrease the overhead that is now introduced by the operating system and allow for a more fine-grained granularity of threads. This would probably require smaller communications, which can be handled through the MPB. For larger communications a message through the MPB can initiate a memory copy operation, or pass a pointer to an off-chip memory location depending on the use and consistency requirements of the data. We can use copy cores to transfer data in parallel for messages larger than 4KB, as we have shown in the copy core performance evaluation. The use of baremetal would also allow for more flexibility in the (virtual) memory

system as all memory can be managed explicitly, including the different mapping approaches for fine-tuning performance. It would also make the use of copy cores would be more easy.

The SCC voltage and frequency scaling provides tools that make energy aware computing possible. In a future implementation, these possibilities should be integrated in the proposed SVP kernel and resource allocator, such that the program will execute at the minimal energy consumption, given a certain performance value.

Bibliography

- [1] Habanero-C. <https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C>.
- [2] Many-core Applications Research Community (MARC). <http://communities.intel.com/community/marc>.
- [3] Rcce.comm. http://communities.intel.com/servlet/JiveServlet/downloadBody/5629-102-1-8718/RCCE_comm.pdf.
- [4] M. Azimi, N. Cherukuri, D. Jayasimha, A. Kumar, P. Kundu, S. Park, I. Schoinas, and A. Vaidya. Integration challenges and tradeoffs for tera-scale architectures. *Intel Technology Journal*, (11(3)):173–184, August 2007.
- [5] M. Baron. The Single-chip Cloud Computer. *Microprocessor Report*, April 2010.
- [6] K. Chapman, A. Hussein, and A. Hosking. X10 on the Single-chip Cloud Computer (presentation slides). <http://dist.codehaus.org/x10/documentation/papers/X10Workshop2011/X10ontheSCC.pdf>.
- [7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioğlu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40:519–538, October 2005.
- [8] C. Clauss, S. Lankes, P. Reble, and T. Bemberl. 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.
- [9] M. Eisler. XDR: External Data Representation Standard. RFC 4506 (Standard), May 2006.
- [10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 18:15–26, May 1990.
- [11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the International Conference on Cluster Computing, ICC ’11*. IEEE, 2011.

- [12] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.
- [13] C. Grelck, Shafarenko, A. (eds):, F. Penczek, C. Grelck, H. Cai, J. Julku, P. Hölzenspies, Scholz, S.B., and A. Shafarenko. S-Net Language Report 2.0. Technical Report 499, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom, 2010.
- [14] J. Howard, S. Dighe, S. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, and R. Van Der Wijngaart. A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling. *Solid-State Circuits, IEEE Journal of*, 46(1):173 –183, jan. 2011.
- [15] Intel Corp. SCC extended architecture specification, November 2010. Revision 1.1.
- [16] C. R. Jesshope. A model for the design and programming of multi-cores. *Advances in Parallel Computing, High Performance Computing and Grids in Action(16)*:37–55, 2008.
- [17] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49:589–604, July 2005.
- [18] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. *Micro, IEEE*, 25(2):21 – 29, march-april 2005.
- [19] M. Lankamp, T. Bernard, M. Hicks, C. Jesshope, and L. Zhang. Evaluation of a hardware implementation of the SVP concurrency model. 2010.
- [20] D. Majeti. Lightweight dynamic task creation and scheduling on the Intel Single Chip Cloud (scc) processor. In *Proceedings of the Fourth Workshop on Programming Language Approaches to Concurrency and Communication-Entric Software <http://places11.di.fc.ul.pt>*, pages 31–34, 2011.
- [21] A. Papagiannis and D. S. Nikolopoulos. Scalable runtime support for data-intensive applications on the Single-chip Cloud Computer. In *3rd Many-core Applications Research Community (MARC) Symposium. (KIT Scientific Reports 7598) <http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023937>*, pages 25–30. KIT Scientific Publishing, 2011.
- [22] S. Peter, T. Roscoe, and A. Baumann. Barrelfish on the Intel Single-chip Cloud Computer. Technical Report Barrelfish Technical Note 005, ETH Zurich, September 2010. <http://www.barrelfish.org>.
- [23] R. Rotta. On efficient message passing on the intel scc. In *3rd Many-core Applications Research Community (MARC) Symposium. (KIT Scientific Reports 7598) <http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023937>*, pages 53–57. KIT Scientific Publishing, 2011.
- [24] 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.

- [25] J.-A. Sobania, P. Troger, and A. Polze. Linux operating system support for the SCC platform - an analysis. In *3rd Many-core Applications Research Community (MARC) Symposium. (KIT Scientific Reports 7598)* <http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023937>, pages 37–44, 2011.
- [26] 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.
- [27] M. W. van Tol, R. Bakker, M. Verstraaten, C. Grelck, and C. R. Jesshope. Efficient memory copy operations on the 48-core Intel SCC processor. In *3rd Many-core Applications Research Community (MARC) Symposium. (KIT Scientific Reports 7598)* <http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023937>, pages 13–18. KIT Scientific Publishing, 2011. 1st Place Best Paper Award.
- [28] M. W. van Tol, C. R. Jesshope, M. Lankamp, and S. Polstra. An implementation of the sane virtual processor using posix threads. *J. Syst. Archit.*, 55(3):162–169, 2009.
- [29] M. W. van Tol and J. Koivisto. Extending and implementing the self-adaptive virtual processor for distributed memory architectures. *CoRR*, abs/1104.3876, April 2011.
- [30] M. Verstraaten, C. Grelck, M. W. van Tol, R. Bakker, and C. R. Jesshope. On mapping distributed S-NET to the 48-core intel SCC processor. In *3rd Many-core Applications Research Community (MARC) Symposium. (KIT Scientific Reports 7598)* <http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023937>, pages 41–46. KIT Scientific Publishing, 2011.
- [31] S. Yan, X. Zhou, Y. Gao, H. Chen, S. Luo, P. Zhang, N. Cherukuri, R. Ronen, and B. Saha. Terascale chip multiprocessor memory hierarchy and programming model. In *HiPC*, pages 150–159, 2009.
- [32] X. Zhou, H. Chen, S. Luo, Y. Gao, S. Yan, W. Liu, B. Lewis, and B. Saha. A case for software managed coherence in many-core processors. 2010.

P5 Architecture overview

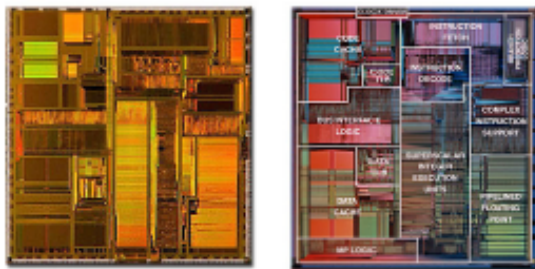
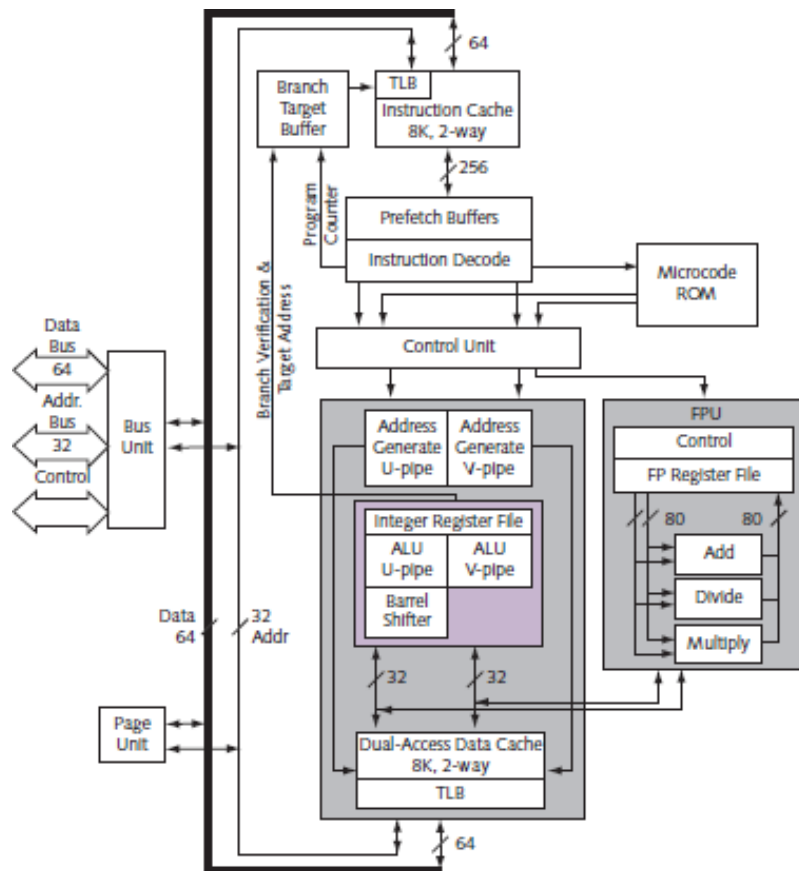


Figure A.1: The schematic layout of the P5 core.

SCC performance meter

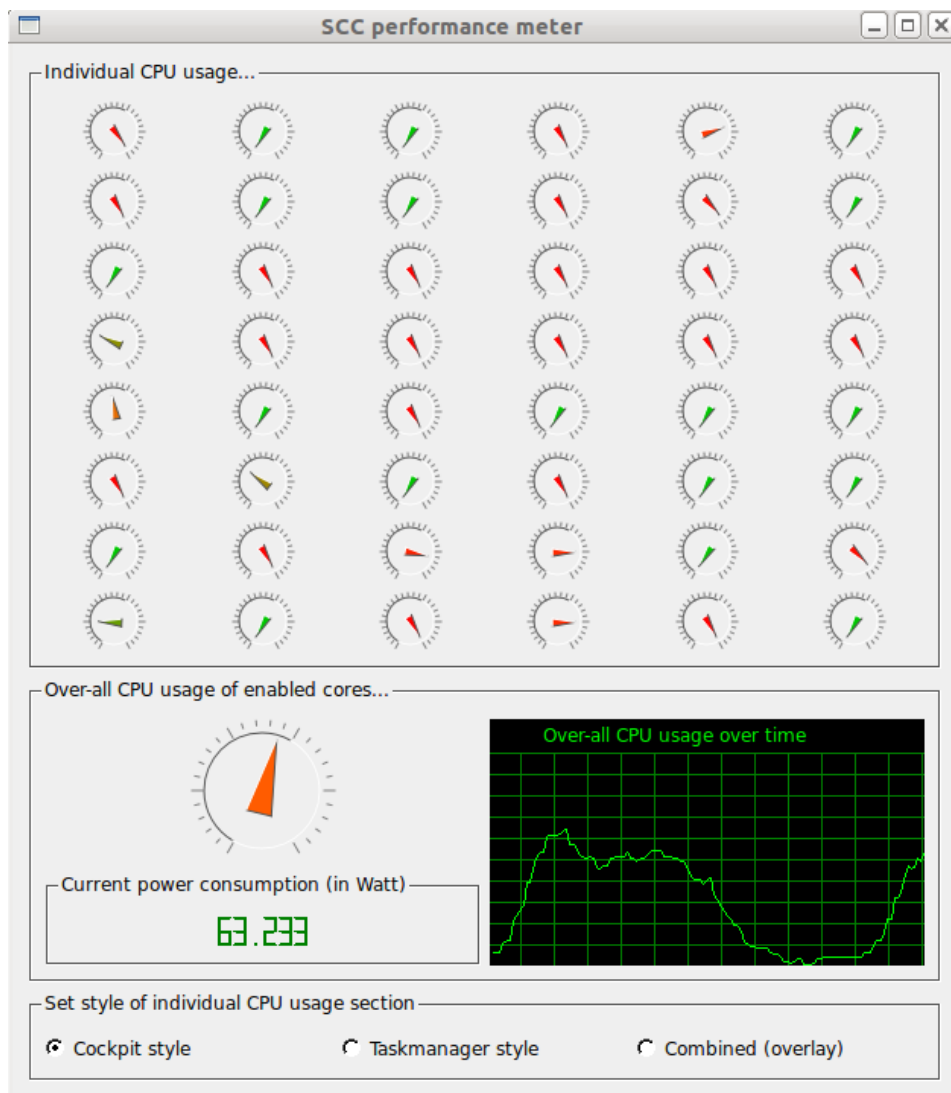


Figure B.1: The SCC performance meter shows the power usage, load per core, and total system load.

