Fast Process Migration on Intel SCC

using Lookup Tables (LUTs)

by

Vaibhav Jain

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved March 2013 by the
Graduate Supervisory Committee:

Partha Dasgupta, Chair
Aviral Shrivastava
Hasan Davulcu

ARIZONA STATE UNIVERSITY

May 2013

ABSTRACT

Process migration is a heavily studied research area and has a number of applications in distributed systems. Process migration means transferring a process running on one machine to another such that it resumes execution from the point at which it was suspended. The conventional approach to implement process migration is to move the entire state information of the process (including hardware context, virtual memory, files etc.) from one machine to another. Copying all the state information is costly. This thesis proposes and demonstrates a new approach of migrating a process between two cores of Intel Single Chip Cloud (SCC), an experimental 48-core processor by Intel, with each core running a separate instance of the operating system. In this method the amount of process state to be transferred from one core's memory to another is reduced by making use of special registers called Lookup tables (LUTs) present on each core of SCC. Thus this new approach is faster than the conventional method.

DEDICATION

To my parents, family, friends, and colleagues.

ACKNOWLEDGEMENTS

I would like to thank Dr. Partha Dasgupta, Dr. Aviral Shrivastava, and Dr. Hasan Davulcu to provide me an opportunity to work under their guidance on this thesis. I would also like to thank Dr. Stefan Lankes from RWTH Aachen University, Germany for his guidance and valuable suggestions. I am thankful to the members of MARC community for their help in answering my queries related to Intel SCC. Special thanks to Michael Ziwisky for helping me in understanding his Baremichael framework.

TABLE OF CONTENTS

APPENDIX Page

LIST OF TABLES

Table Page

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Process migration is the act of transferring a process between two processors and restoring the process from the point it left off. While load balancing across networked nodes has been a major motivation for process migration, it has a number of other applications including fault tolerance, power management and data access locality. Several implementations have been built for different operating systems including MOSIX[1], V[2], Accent[3], Sprite[4] and Mach[5]. Process migration involves transferring the state of a process from one machine to another. The state includes virtual address space (code, stack and data), registers, open files, message buffers and environmental data such process-id, username etc. The cost of moving all this state information is a major difficulty in process migration. The major portion of this cost is the cost of moving the virtual address space as it forms the largest unit of the process state [6].

This research looks into process migration on Intel Single-Chip Cloud computer (SCC) [7] which is a 48-core experimental processor prototype created by Intel Labs as a platform for many-core software research. The aim of the research is to propose a new technique for migrating a process from one core of SCC to another which is faster than the conventional technique of moving the complete state of the process from one core to another. This technique makes use of special sets of registers present on each core of SCC called Lookup tables (LUTs) such that only a portion of the state is transferred thereby reducing the cost of migration. Apart from performance there are many other complex issues associated with process migration such as transparency, naming, scheduling etc. but these are not in the scope of this research.

The SCC has 48 Pentium-1 (P54C) cores placed in a tile formation with two cores per tile. The tiles are connected by a 6X4 mesh network. Each P54C core is 32-bit processor and can address 4GB of memory called the core-physical address space. The SCC can support up to 64 GB of memory called the system-physical address space. The system-physical address space includes the memory of four DDR3 memory controllers (supporting a maximum of 16 GB each), on-chip memory for message passing between the cores (MPB) and memory-mapped configuration registers. The translation of core-physical addresses to system-physical addresses is done through lookup tables (LUTs). Each core has a private LUT which maps every core-physical address to a system-physical address which can belong either to the off-chip DRAM, the MPB or the configuration registers. The LUTs get assigned default values at boot time. The default LUT configuration divides the off-chip DRAM into memory private to each core and memory shared by all cores. However the LUT entries can be changed by a core dynamically. A core can change LUT entries not only of its own but also of other cores. These properties of LUTs can be used to migrate a process running on a core to another core without physically moving the complete state of the process from one core's private memory to another. The idea is to make a portion of the private memory of one core a part of the private memory of another core by changing a few LUT entries on both the cores.

To demonstrate the proposed method of migrating a process a small kernel has been written. The experiment involves running an instance of this kernel on two cores and migrating a process from one core to another. The kernel on the source core starts a user process and migrates it to the destination core using LUTs. The experiment also involves migrating the process using the conventional method in which the entire state of the process is copied from source core's memory to

destination core's memory using shared memory. The performance from both the approaches has been compared.

OUTLINE

The rest of the report is organized as follows. Chapter 2 gives an overview of Process Migration. Chapter 3 describes the SCC architecture and programming model. Chapter 4 describes the experiment and evaluation. Chapter 5 mentions related work.

CHAPTER 2

PROCESS MIGRATION

A process is an operating system abstraction of a program in execution. Each process has a state associated with it. The state of a process refers to all the information that is required to resume the execution of the process. The components of the process state may vary from machine to machine but typically include the following [8]:

- virtual address space – code, data and stack segment
- open files - file id, read-write pointers, buffers etc
- message buffers – sending and receiving messages and buffers
- machine state – registers , program status word etc.
- environment data – process-id , user-id, references to child , parent processes etc.

Process migration refers to the act of suspending a process running on one machine in the middle of execution and transferring to another machine where it resumes execution from the point at which it was suspended. For the process to resume execution properly the entire state of the process may need to be transferred.

Process Migration has a number of applications [9]:

- Dynamic Load Balancing - Processes can be moved from a heavily loaded node to a lightly loaded node. This would increase the system throughput and reduce the average response time.
- Resource Sharing - If a process requires access to a resource such as a special hardware available on a remote node it can be moved to that node resulting in better resource utilization and reduced network communication.

- Fault tolerance – If a node experiences partial failure or is likely to fail all the processes running on it can be moved to a different node thus ensuring availability.
- Improving System Administration – If a machine is temporarily shut down all the long-running processes executing on it can be moved to another machine and then migrated back when the machine is up again.
- Improving communication Performance - If a process is frequently communicating with another process on a different node the communication cost can be reduced by moving one of the processes to the same node as the other or to a nearby node.
- Mobile Computing – Users may use different devices at different times and may want to continue using the same programs while remaining connected to the network.

There are many different implementations of process migration. All the existing process migration implementations may vary in some aspects but have the following basic steps in common [10]:

- A decision is made to migrate the process.
- A remote node is selected and a migration request is sent to the node.
- The process is suspended on the source node. All the arriving messages are queued up to be delivered to the destination node after migration.
- A skeleton process is created on the destination node and space is allocated for process state. This is not activated until sufficient state has been imported into it.
- The process state (registers, virtual memory, open files, message channels etc.) is extracted from the source node and transferred to the destination node and imported into the skeleton process.

- The process is resumed on the destination node.
- The state information remaining on the source node is removed. Some information may still be retained for e.g. to redirect the incoming messages.

Designing a process migration facility is a complex task and a number of issues need to be considered. Following are some of the issues to be considered:

- Virtual memory transfer – The cost of transferring virtual memory is the dominant factor in the cost of transferring the entire state [6]. Transferring complete address space at once results in long freeze time (time for which process is not running during migration) for the process to be migrated. Different approaches have been adopted in different implementations to reduce the cost of moving virtual memory [8][11]:
  a. Eager (all) - Entire address space is transferred at the time of migration. This is the simplest approach. However it can prove to be expensive if the address space is large and the migrated process is not going to need most of it.
  b. Eager (dirty) - Only modified (dirty) pages are transferred during migration while the rest of the pages are only transferred on demand. This reduces the cost of transfer during migration. However it requires the source node to continue maintaining the state of the process.
  c. Copy-on-Reference - Pages are transferred only on reference. This has the lowest initial cost.
  d. Precopy – Pages are transferred while the process is still executing on the source node. This reduces the freeze time of the migrated process.
  e. Flushing – The dirty pages are flushed to the disk and accessed on demand from the disk. Thus the source node does not need to maintain pages in memory.

- File migration [12] – File migration is an important issue in process migration. The state associated with an open file consists of file descriptor, read/write pointers and cached blocks. Migrating open files transparently can be challenging. Different approaches have been implemented. One of them is to close the files on the source node and reopen them on the destination node. Another approach is to have all the file related system calls redirected to the source node[4]. The approach may vary if a distributed file system is available.
- Process relationships – The presence of child and parent processes adds more complexity to process migration. It may be decided to keep the related process on the source node or to move them to destination node.

This research focuses on improving performance of process migration on Intel SCC by reducing the amount of state transferred from one core to another. The components of process state that are considered are register and virtual address space. All the other issues such as migration of open files, devices, parent and child processes etc. are kept out of the scope of the research.

CHAPTER 3

INTEL SINGLE CHIP CLOUD COMPUTER

The Single Chip Cloud Computer (SCC) is an experimental processor created by Intel labs as a platform for many-core research. It is the second processor in Intel's Tera-scale research program [13]. The Tera-scale program envisions creating many-core platforms capable of handling terabytes of data and providing teraflops of computing performance. There are two main motivations for SCC [14] – to promote many-core research and to promote parallel-programming research. Intel has made SCC available to several researchers in both industry and academia as a result of which there is a lot of ongoing research in parallel and many-core computing using SCC as the platform. Many applications and operating systems have been ported to SCC [15][16][17] and new applications and operating systems are being developed to run efficiently on SCC.

The SCC has 48 IA (Intel Architecture) cores on a single die which according to Intel is the maximum number of IA cores that have ever been integrated on a single chip[18][18]. Each IA core is based on Pentium (P54C) core. The P54C was chosen because it is simple in design, cheap and smaller in area as compared to newer processors [17][19]. This made it possible to put 48 cores on a single die in a cost-effective way. It also made it easy to modify existing applications to run on SCC. The cores are placed in a tile formation. There are 24 tiles in total with two cores on each tile. The tiles are connected by a high performance 6x4 2D mesh network. SCC has four on-die DDR3 memory controllers. The tiles can communicate with each other or with the memory controllers by sending packets over the mesh network. Figure 1 illustrates SCC architecture.

The SCC currently does not have any hardware devices such as keyboard, monitor or disk. The only way to communicate with the SCC is through a

Management Console PC (MCPC) which is connected to it through PCI/e interface. The MCPC is a 64-bit machine running a version of Linux. It has software provided by Intel labs to manage SCC. The software on MCPC can be used to load operating systems and programs on some or all of the cores of SCC, to modify the SCC configuration registers, to read from and write to the memory and more [20].

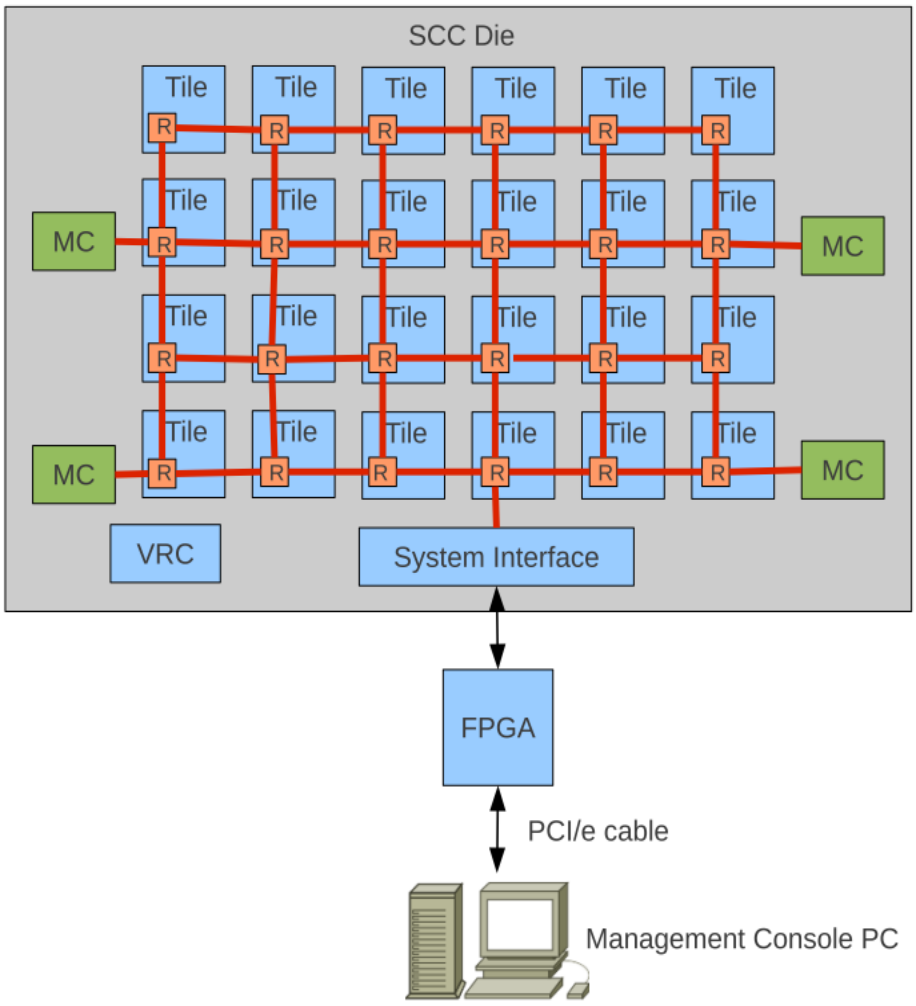

Figure 1. SCC Architecture. Reprinted from [17]. Reprinted with permission.

TILES

The SCC consists of 24 tiles connected by mesh network. Figure 2 shows the architecture of a tile. Each tile has two cores. Each core is a P54C [24] processor

with an L1 cache and an L2 cache. An important difference in the design of standard multi-core systems and SCC is that the caches in SCC are non-coherent. The reason behind this decision is that as the number of cores increases cache-coherency becomes a bottleneck in the performance of the system [21]. So to make the platform scalable to large number of cores cache-coherency has been left for the software to manage.

Each tile has small on-chip memory called the Message Passing Buffer (MPB). The MPB supports faster reads and writes than the off-chip DDR memory. Although the MPB is local to a tile, any core can access the MPB assigned to any other core. The primary purpose of MPB is to enable passing between programs and operating systems running on different cores. A message sent from one message-passing program to another goes from L1 cache of the sending core to the MPB and then to the L1 cache of the receiving core. Thus message or data is transferred without using the off-die memory.

There is a router connected to each tile to route packets over the mesh network. Each tile also contains a Mesh-Interface Unit (MIU) [23]. The MIU is the interface between the tile and the router. The MIU packetizes the data going out to the mesh and de-packetizes data coming in from the mesh. The MIU catches cache misses and translates 32-bit core addresses into 46-bit system addresses using Lookup tables (LUTs) present on the tile. The translation between core and system address is explained in detail later. The tiles are organized into four regions, each of which maps to a particular memory controller. This mapping is also determined by LUTs.

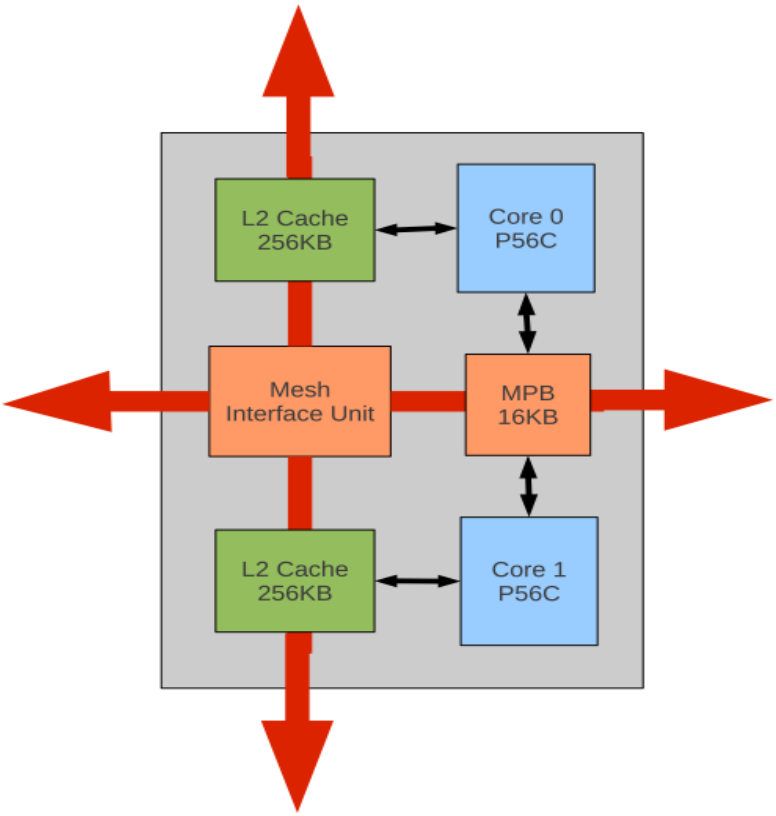

Figure 2. Tile design architecture of SCC. Reprinted from [17]. Reprinted with permission.

Each tile contains a set of configuration registers that can be used by programs running on the cores to control operating modes of different hardware components of the tile. The registers are located in the MIU in a Control Register Buffer (CRB). They are accessed using memory-mapped IO. The configuration registers consists of two Lookup tables (LUTs), two Test-and-set registers, a TileID register, two L2 cache configuration registers, two core configuration registers, a global clock configuration register and two sensor registers[22]. Table 1 lists the different configuration registers. There are two Lookup tables on each tile, one for each core. The registers LUT0 and LUT1 are the first entries in the two lookup tables. The TileID register MYTILEID is a read-only register and contains the (x,y) coordinates of tile. It also contains the core-ID of the core that read it. There are two core configuration registers – CLCFG0 and CLCFG1 on a tile. They can be used to

send inter-processor interrupts from one core to another. The L2 cache configuration registers – L2CFG0 and L2CFG1 are used to enable or disable L2 caches of the cores on the tile.

Table 1. SCC Configuration Registers

Register	**Description**	**Access**
LUT1	LUT register core 1	Read/Write
LUT0	LUT register core 0	Read/Write
LOCK1	Atomic Test and Set Lock Core 1	Read/Write
LOCK0	Atomic Test and Set Lock Core 0	Read/Write
MYTILEID	Tile ID	Read
GCBCFG	Global Clock Unit Configuration	Read/Write
SENSOR	Thermal Sensor Value	Read
SENSORCTL	Thermal Sensor Control	Read/Write
L2CFG1	L2 Cache Core 1 Configuration	Read/Write
L2CFG0	L2 Cache Core 0 Configuration	Read/Write
GLCFG1	Core 1 Configuration	Read/Write
GLCFG0	Core 0 Configuration	Read/Write

Note. Reprinted from [17]. Reprinted with permission.

There are three IDs associated with each core – tiledD , processorID and coreID[20]. The coredID identifies the core within a tile and is either 0 or 1.Since the tiles are arranged in a 6X4 2D array, each tile has an (x,y) coordinate. The tileID is and 8-bit value where the first four bits are for the y coordinate and and the last four are for the x coordinate. As mentioned earlier the tileID can be obtained by reading the TildeID register. The processorID goes from 0 to 47.

MEMORY

The SCC has features of shared memory systems as well as distributed memory systems [21]. It has off-chip DRAM and on-chip SRAM (MPB). The off-chip DRAM is accessed through four on-chip memory controllers attached to four tiles at the positions – (0, 0), (0, 5), (2, 0) and (2, 5) in the 6X4 grid. Each controller has two DDR3 banks. The banks can have capacity of 2GB, 4GB or 8GB. Thus a total of 64GB of memory can be supported by SCC.

Each core has a 16KB L1 data cache and a 16KB L1 instruction cache. There is also a unified 256 KB L2 cache. The SCC does not provide any cache coherence and it is up to the software to manage coherence. To enable the programmer to manage cache coherence the SCC provides two features [22]. A new memory type or tag called MPBT has been provided that identifies the cache lines that hold MPB data. The MPBT data bypasses the L2 cache. In addition a new instruction called CL1INVMB has been added to the P54C instruction set to invalidate all the cache lines marked as MPBT in the L1 cache. Invalidating the cache lines would force the data in the L1 cache to be update from MPB on the next read. The instructions WBINVD and INVD can be used to respectively flush and invalidate the L1 cache completely. However there is no hardware instruction to flush the L2 cache [19].

LUT

Each core on SCC is a 32-bit processor and can address up to 4GB of memory called the core-physical address space. The total memory supported by SCC is 64GB called the system-physical address space. The system address space is divided into different memory regions – off-chip DRAM, on-chip MPB and memory-mapped configuration registers. The translation of core physical address to system physical address is done by using lookup tables or LUTs. An LUT is a set of configuration

registers within the CRB (Configuration register buffer). Each core has its own LUT. The LUT determines if a core physical address refers to the off-chip memory, on-chip memory or configuration registers. When a program accesses a 32-bit virtual address it is first translated to a 32-bit core-physical address by the core's Memory Management Unit (MMU) using page tables. The core physical address is then translated to system physical address through the LUT. Figure 3 shows the translation process. The LUT has 256 8-byte entries. Each entry maps to a 16MB segment (an LUT page) of the core physical address space thus making a total of 4GB. The upper 8 bits of the core physical address are used to index one of the 256 LUT entries. Each LUT entry contains 22 bits of information. The lower 10 bits are prepended to the lower 24 bits of the core physical address to form a 34 bit address. The upper 12 bits of the LUT entry provide the routing information. The routing information consists of an 8-bit Destination ID, a 3-bit Subdestination ID and a Bypass bit. The Destination ID is the destination Tile ID (y, x). The Subdestination ID determines the type of memory to be accessed – DDR3, MPB or CRB. The Subdestination ID specifies the port at which the packet would leave the destination router - North, East, West, South. It can also specify one of the other destinations - Core0, Core1, MPB and CRB. The four memory controllers are either located at the West port or the East port of a router. Table 2 lists the different values of Subdesination ID and the destinations they indicate. The Bypass bit is used to bypass the MIU when the destination address is local to the tile thus allowing faster access.

For loading and running an operating system (or a baremetal application) a file containing LUT mappings is required. The tool sccMerge provided by Intel on the MCPC is used to generate a loadable image of the operating system to be booted along with the default LUT mappings. APPENDIX A shows the default LUT mappings

generated when the total system memory is 32 GB. The LUTs can be modified dynamically after booting. This can be done either through a program running on the Management Console or through a program running on a core. The default LUT configuration divides the off-chip DRAM into regions private to a core and regions shared by the cores. The memory private to a core is mapped only by the LUT of that core. The memory shared by the cores is mapped by LUTs of all the cores thus making it accessible to all the cores. However since the LUT entries can be changed dynamically, it is possible to make a portion of the private memory of one core visible to other cores by modifying a few LUT entries. This has the effect of moving data without physically copying it.

Table 2. Address Translation Fields

Sub-Destination	**subdestID (3 bit)**	**Comment**
Core0	0x0	Not a destination for memory R/W
Core1	0x1	Not a destination for memory R/W
CRB	0x2	Configuration Register
MPB	0x3	Message Passing Buffer
E_port	0x4	@'(0,5) is DDR3 MC
S_port	0x5	@'(0,3) is SIF, @(0,0) is VRC (presuming y,x order)
W_port	0x6	@'(0,0) is DDR3 MC
N_port	0x7	Nothing is on this port of any edge router

Note. Reprinted from [23]

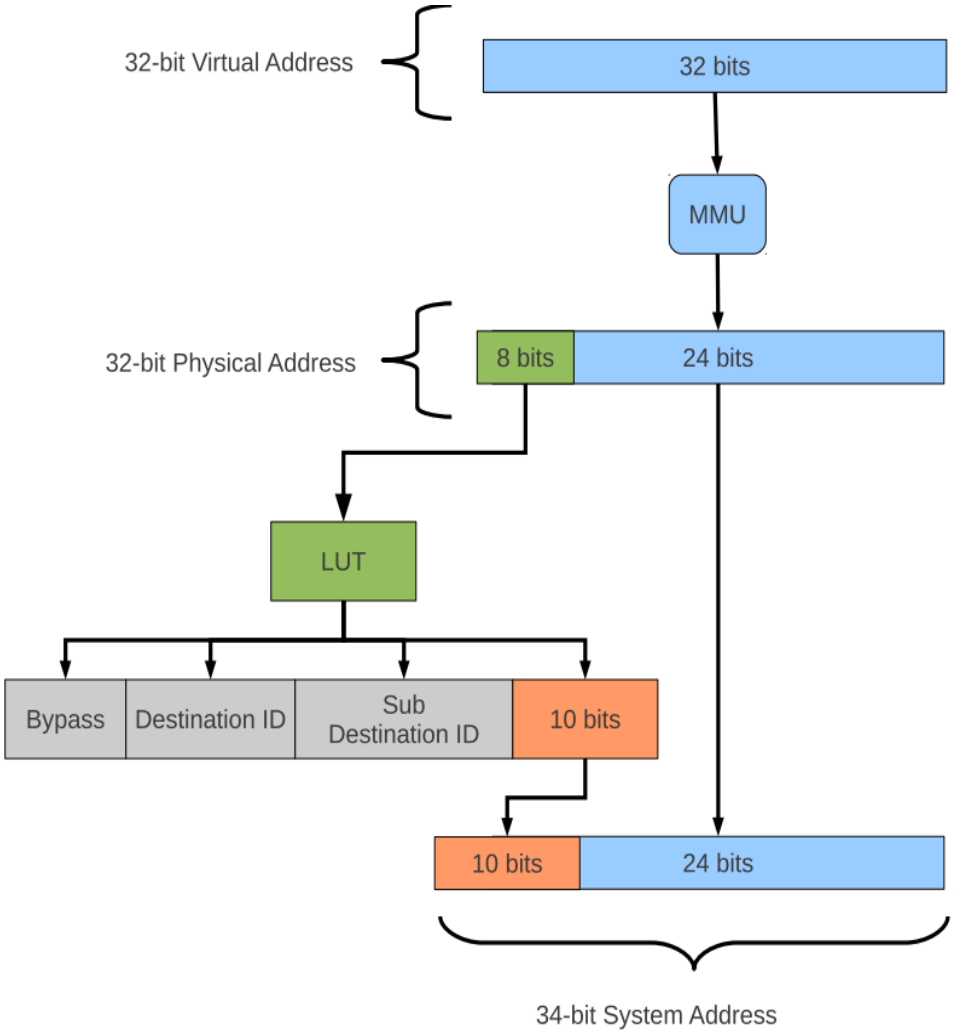

Figure 3. Address translation using Lookup Table. Reprinted from[17]. Reprinted with permission.

ON-CHIP MESSAGE PASSING

Each tile has 16KB of message passing buffer (MPB). By default 8KB assigned to each core. The MPB of each core is accessible is by all the cores. The LUT of a core maps the MPBs of all the tiles. The LUT also contains an entry for the core to access its own MPB. To support message passing new features have been added to the SCC core. A new memory type called Message Passing Buffer Type (MPBT) has been introduced. The MPBT data bypasses the L2 cache. Each cache line has a new status

bit that identifies the contents of the cache line as MPBT. A reserved bit in the page table marks the page as MPBT data. A new instruction called CL1INVMB has also been added to the instruction set of the P54C core. This instruction invalidates all the MPBT cache lines in the L1 cache thereby avoiding reading of stale data from the cache. To provide support for message passing through MPB Intel has provided a library called RCCE [25]. It moves the data from the private memory of the sending core through the L1 cache to the MPB and then to L1 cache of the receiving core.

MANAGEMENT CONSOLE PC

The MCPC is a 64-bit computer connected to the SCC over a PCIe interface. It runs some version of Ubuntu Linux. Intel provides a set of tools called sccKit on the MCPC to manage SCC. The MCPC also contains RCCE, a modified Linux image to load on to the cores and tools to enable developers to build custom Linux images. The sccKit provides both a graphical interface (called sccGui) and a command-line interface. The programs in the sccKit can be used to load an operating system on the cores, to read and write memory and configuration registers, to monitor core performance and more. Following are some of the useful commands [20]:

- sccBmc - This command is used to send commands to the Board Management Controller (BMC). It can be used to initialize the SCC platform or read some status information. This command when used with –i option re-initializes the SCC.
- sccBoot - This command is used to boot an operating system on a range of cores. The operating systems can either be the Linux image provided by Intel or a custom operating system. When used with –s option it lists the cores that can be reached by ping.

- sccReset - This command is used to reset a range of cores. It gives options for holding the reset and releasing the reset. It also has option to list the cores that are in reset mode.
- sccDump - This command reads the contents of the off-chip DRAM, the configuration registers, the MPB and the LUTs. The contents can be dumped to a file. It is a useful tool for debugging.
- sccMerge - This command is useful for loading custom operating system on the cores. It takes as an input a bootable image and create a merged image to be loaded on all the memory controllers along with the default LUT mappings for all the cores. The optional arguments include the size of memory at each controller in GB (-m) and the number of cores assigned to a memory controller (-n).

BAREMETAL

There are two ways in which applications can be run on SCC –

- On top of an operating systems such as Linux
- Directly on the hardware without the support of an operating system.

Intel has provided a modified version of Linux called sccLinux which can be loaded on each core separately using the tools present on the MCPC. It provides a number of useful features for application programming [20]. However, sometimes it is desirable to run the programs directly on hardware. This can help to achieve greater speed of execution and to avoid operating system overheads. It also provides greater freedom in programming for e.g. running programs on the highest privilege level. This way of programming is referred to as Baremetal programming. There are currently three frameworks available that can help in running baremetal applications – ETI SCC framework [26], Microsoft framework and Baremicahel framework [27]. Of these three frameworks only Baremichael is open-source.

Baremichael enables developers to load and launch C and assembly programs on SCC cores. It includes a subset of the C standard library and some SCC-specific helper functions and macros. It also provides a pseudo-terminal called MikeTerm that shows the output from applications running on the cores. The experiment conducted for this thesis makes use of some of the code from Baremichael.

CHAPTER 4

EXPERIMENTATION AND EVALUATION

As described earlier, Intel SCC has 48 cores and four memory-controllers which can support up to a maximum of 64 GB of off-chip memory.

The memory at each memory-controller can be divided into memory private to each core and memory shared by all cores. This division is done through LUTs and is configurable. In the default LUT configuration, the tiles are divided into four regions with 12 cores each such that each region is served by a particular memory controller. Thus at each memory controller there are 12 private memory regions and one shared memory region shared by all the 48 cores. The most common way of programming SCC is to run it as a cluster. Each core is booted with an operating system and forms a node in the cluster. In such a setup if a process running on one core is to be migrated to another core, the typical approach would be to move the process state from the private memory of the source core to the private memory of the destination core by using shared memory. Copying all the state twice is expensive. This thesis therefore proposes a new approach to migrate a process. In the new approach a process is migrated by using LUTs such that minimal or no state needs to be copied. To demonstrate the new method of process migration a small kernel has been developed. The kernel implements a migrate() system call which can migrate the process that invokes it. There are two implementations of the migrate() system call. One that uses LUTs and the other that uses shared memory. To evaluate the performance improvement with the new approach, a process is migrated using both LUTs and shared memory and the results are then compared.

CUSTOM KERNEL

The custom kernel was first developed on a local Linux system on an emulator and its basic features were tested. It was then modified to run on SCC. It is a kernel with limited functionality. Following are some of the features of the kernel:

- Interrupt handling and system calls – The kernel implements an interrupt handling mechanism and some interrupt handlers. It also provides system calls to print text on the screen, to fork a child process and also to migrate a process (discussed later).
- Memory management – The kernel implements virtual memory through paging. It has a physical memory manager and a virtual memory manager. The physical memory manager keeps track of free frames using a bitmap. The virtual memory manager manages allocation and de-allocation of pages through page-directories and page-tables. The kernel also includes a heap allocator for dynamic memory allocation in kernel mode.
- Process management - A simple round-robin scheduler has been implemented using a queue of Process-control-blocks (PCBs). Whenever a timer interrupt occurs it causes a context-switch and the next process in the queue is executed. A PCB contains pointer to a memory block containing all the register values, a pointer to the page directory of the process, a process-id and a pointer to the next PCB in the scheduler queue. The address space of a process consists of a code segment, a data segment and a stack. The virtual addresses of the all three segments are currently fixed.

At the moment the kernel does not have a file-system. Therefore to execute a program, the hex-dump of the code and data sections of the executable needs to be copied into two arrays in the source code. To create a process the kernel reads the binary of the process from the arrays and copies it into virtual memory.

PORTING THE KERNEL TO SCC

The design of the SCC is different from a typical multi-core system in a number of ways. Various new architectural features have been incorporated for better scalability. Therefore to execute any application or operating system on SCC modifications are required. This section describes how the custom kernel has been ported to SCC and what modifications have been made to the kernel. The open-source Baremichael framework has been very helpful in porting the custom kernel to SCC.

The custom kernel was first developed on the local system using an emulator and grub bootloader was used to boot the kernel. However the SCC does not have BIOS due to which the booting process is different than a regular multi-core system with BIOS. The first step to execute a kernel (or application) directly on the SCC cores is to convert the binary of the kernel into an Intel specific ASCII format using a tool called bin2obj[28]. Since the cores start in 16-bit real mode, the binary of the kernel needs to include the boot-code to jump from real mode to protected mode. For the custom kernel the 16-bit boot-code is taken from Baremichael. After that the sccMerge script is used to create a loadable image of the kernel. sccMerge takes as input the output of bin2obj and a text file which mentions all the cores on which the kernel is to be loaded. sccMerge also creates a file containing the default LUT mappings. The loadable image and LUT mappings are then pre-loaded using sccBoot command and the cores can then be booted using sccReset.

The LUT of a core maps its physical address space to its private memory, shared memory, MPBs and CRBs. For the kernel to access these regions the physical address space needs to be mapped to virtual address space through page tables. Also, the kernel needs to access the shared memory, the MPBs and CRBs explicitly.

So the virtual addresses of shared memory, the MPBs and CRBs are explicitly set to be equal to the physical addresses which map to them through the LUT.

The kernel has been modified to add the functionality of sending inter-processor interrupt to another core. On the P54C core interrupts are handled by local APIC (Advanced Programmable Interrupt Controller). There are two ways to send inter-processor interrupt on SCC. One is to control the LINT pins of the local APIC of the destination core by writing to the Core Configuration register. This is the method that has been used in the kernel. The other method uses the Global Interrupt Controller. To handle the inter-processor interrupt the LINT pins of local APIC need to be configured.

To read the output from the kernel the Miketerm utility[27], which is a part of Baremichael, has been used. The utility works in conjunction with the printf() function provided by the framework. Therefore the printf() function implementation has been plugged into the kernel. The printf() function writes the data in a circular buffer in shared memory. Every core is allocated a different buffer in the memory. The Miketerm utility polls all the 48 buffers and displays the text found in them. It uses the sccDump command to read the contents of the buffers. Miketerm precedes all the ouput with core-identifier. The order in which output from different cores will be displayed is not guaranteed.

The kernel also includes some useful SCC-specific functions and macros that are taken from the Baremichael framework. For e.g. a function to get the core-id and macros for getting the address of CRB and LUT of the core based on the default LUT mapping.

PROCESS MIGRATION USING LUT

As explained earlier, process migration involves suspending a process on the source machine and resuming execution of the process on the target machine. This

requires the state of the process to be made available on the target machine. The conventional way of doing this is to copy the state from the source to the target. In the context of SCC the source and the target machines are the cores of the SCC each running a separate instance of an operating system. Each core has its own private memory which can belong to any of the four memory controllers. The cores can also share memory. The division between the shared and private memory can be configured through LUTs present on each core. A portion of the physical memory can be shared or private depending on the LUT entries of all the cores. If a memory area is mapped only by the LUT of one core, it is private to that core as no other core will be able to access it. On the other hand if a memory area is mapped by LUTs of more than one cores, it is shared by all those cores. In the default LUT configuration generated by sccMerge, the cores are divided into groups of 12 such that each core in a group has its private memory on the same memory controller. Also, on each controller there is some memory which is shared by each of the 48 cores. However, since the LUTs can be changed dynamically, it is possible for a program (or kernel) to make a portion of the private memory of a core visible to other cores by changing LUT entries. As each entry in an LUT maps to a 16 MB segment of physical memory, it requires changing only two LUT entries to move a 16 MB chunk of memory from one core's private memory to another without physically copying the memory. This leads to the idea that if entire state of a process can be located in 16MB chunks of memory then by modifying a few LUT entries the whole process can be migrated from one core to another. This would result in a smaller freeze time during migration as compared to the conventional approach. Figure 3 shows how changing an LUT entry causes redirection of memory read/write accesses.

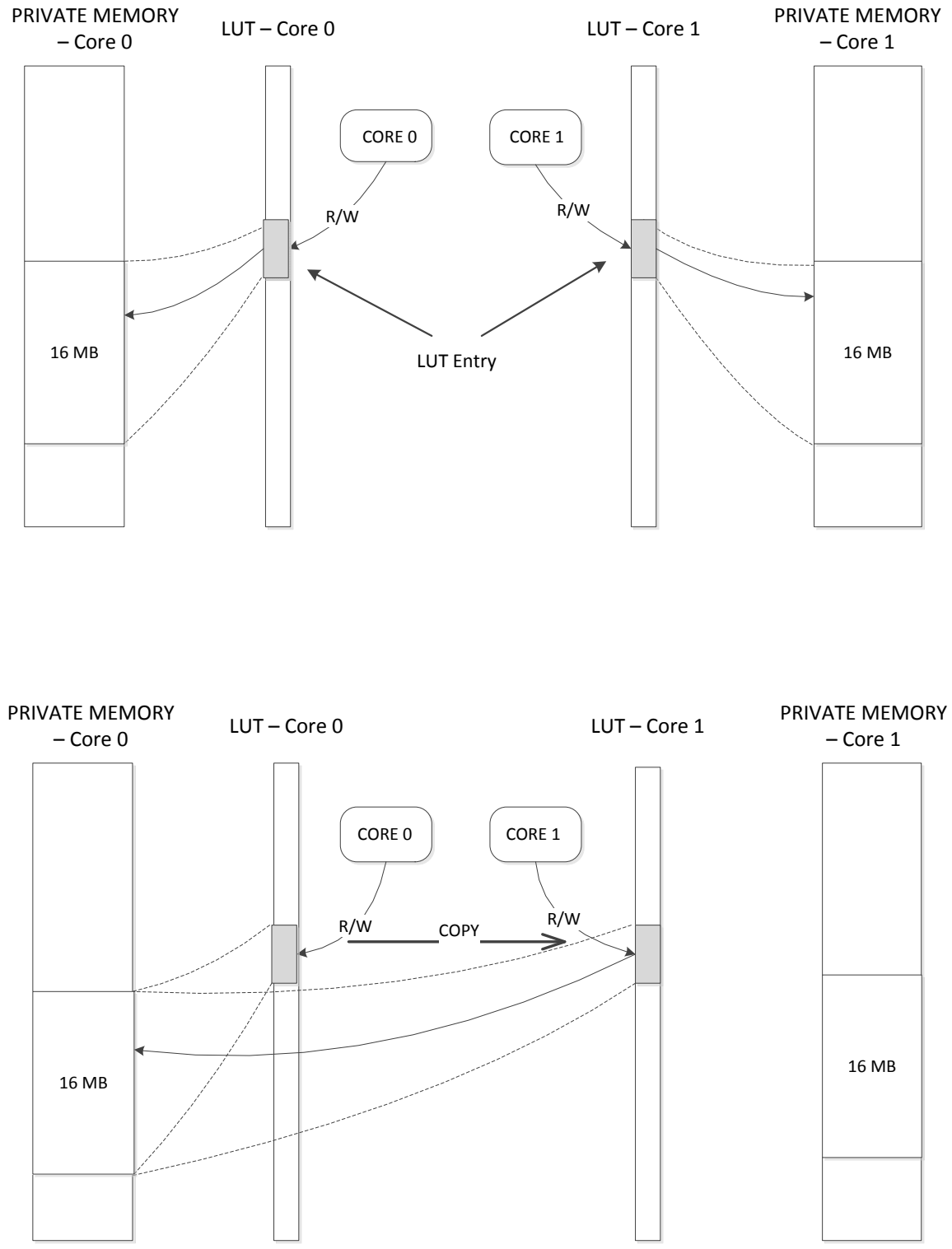

Figure 4. Effect of copying LUT entry on memory access

Following is an overview of how process migration is implemented in the custom kernel:

- create_process_LUT() - This method creates the user process to be migrated such that all of its state is located in a 16 MB segment of memory which is mapped by a single LUT entry. The state consists of PCB Page directory, page tables, code segment, data segment and stack segment. In the experiment a small process (code, data and stack segments each occupying one page each) is migrated. Therefore the process state only occupies a single LUT page.
- migrate() - This is a system call invoked by the user process to be migrated (created by create_process_LUT()). Invoking a system call causes the current context of the process to be saved in memory. The migrate() system call initiates the process of migration on the source core. It performs the following functions:
  - Suspends the process to be migrated by calling the remove_current_process() method.
  - Copies the contents of the LUT entry which maps the process state on the source core to the LUT entry on the destination core. This makes the process state visible to the destination core.
  - Sends an inter-processor interrupt to the destination core by writing to certain registers in the configuration register block of the destination core.
- remove_current_process() - This method is called by the migrate() system call as mentioned above. It removes the current process from scheduler's queue.
- ipi_handler() - This is the interrupt handler for inter-processor interrupt (IPI). When the destination core receives IPI from the source core this method

starts executing. It performs the second stage of process migration by calling receive_process().

- receive_process() - This method completes process migration by adding the process to be migrated to the scheduler's queue on the destination core. Since the process state is visible to the destination core after the LUT entry is copied, adding the PCB of the process to scheduler's queue just requires few pointer operations.

Following is a step-by-step description of how process migration is accomplished in the experiment conducted for this thesis:

1. Two cores of SCC are booted with the custom kernel and default LUT mappings.
2. As soon as the kernel starts running on the source core it creates the process to be migrated. For creating the process the kernel allocates memory for the state of the process such that entire state lies in a 16MB segment of physical memory. This segment maps to a single LUT entry. For simplicity the process is kept small enough to fit within 16MB of memory. Also, the source and destination cores are fixed.
3. When the process starts running, it displays the core-id of the core on which it is executing and then invokes the migrate() system call.
4. The migrate() system call removes the process from scheduler’s queue and copies its LUT entry mapping on the source core to the destination core. The indices of LUT entries on the source and destination core are kept same so that the destination core sees the same physical addresses as the source. This is required for the page-tables and page-directories to work as they refer to physical address of the pages. In the end the migrate() system call sends an inter-processor interrupt to the destination core.

5. On the destination core, the interrupt-handler for inter-processor-interrupt starts executing. The 16MB memory segment containing the process state is now visible to the destination core. The offset of the PCB of the process is fixed from the start of the segment. The interrupt-handler makes the process runnable on the destination core by adding the PCB to the scheduler queue.
6. When the process starts on the destination core it again prints the core id of the core on which it is running. This time it is destination core.

From the above description it can be seen that if the whole process state can within 16MB of memory it can be migrated just by changing one LUT entry. Even if it is required to manipulate multiple LUT entries it would still be better than copying the entire process state as LUT entries are just registers.

As mentioned before, this thesis concentrates only on one aspect of process migration which is transferring the virtual address space and processor state of the process. The other aspects of process migration such as migration of open files, devices, communication channels and messages and handling of parent-child processes have been kept out of the scope. Currently SCC does not have any devices connected to it and is managed by software present on the MCPC. When sccLinux is booted on the cores, a directory on the MCPC (/shared) is NFS mounted on all the cores and to create a persistent file it has to be created in this directory [34]. In such a setup, a simplistic way of dealing with open files during migration is to close files on the source core and reopen them on the destination core. However, a number of complexities can arise when factors such as deletion of files, caching of file blocks and sharing of files between parent and child processes are considered [4].

PROCESS MIGRATION USING SHARED MEMORY

The kernel contains a second version of both the migrate() system call and the IPI interrupt-handler to migrate a process using shared memory. The second version of migrate system call starts the migration by removing the process from the scheduler queue and then copies the state of process to shared memory including the Process Control Block, code, data and stack. The page directory and the page tables are not copied as they refer to physical addresses on the source core which will not be the same on the destination core after migration. The interrupt handler on the destination core copies the PCB and the address space of the process from the shared memory into the private memory of the destination core. It then creates a new process using the copied PCB and address space thus essentially creating a clone of the process to be migrated. This new process is then scheduled to run by adding the PCB to scheduler's queue.

EVALUATION

The proposed approach of migrating a process using LUTs has been compared with the conventional approach of using shared memory by measuring the time required to migrate a process using both the approaches. The experiment consists of migrating a process between two cores multiple times and averaging the total time over the number of the number of migrations. An instance of the custom kernel is booted on both the source and destination core. The source kernel creates a process to be migrated as soon as it starts. The process is small in size with only page each for code, data and stack segment. The code of the process consists of a loop that invokes the migrate() system call in each iteration. Whenever the migrate() system call is invoked on a core it causes the process to migrate to the other core. The process prints the value of the Global Timestamp counter before and after the loop. The Global timestamp counter is a 64-bit counter running at the frequency of 125

MHz and available in form of two 32-bit values in registers. Thus the total time taken by the loop in seconds is calculated by calculating the difference between the two values of the counter and dividing it by 125 X 106. This time divided by the number of iterations gives the time taken by a single migration. The experiment was repeated by selecting different cores as source and destination to observe the variation in the migration time due to the distance between the cores. In case of the shared memory approach, migration requires copying of entire state of process. Therefore the migration time is expected to be more when the source and destination cores are accessing different memory controllers as compared to when both the cores access the same controller. The following core pairs were selected for migration:

- Core0 and Core1 on Tile (0,0) – same memory controller.
- Core0 on Tile (0,0) and Core 0 on Tile (5,0) - horizontally opposite memory controllers.
- Core0 on Tile (0,0) and Core0 on Tile (5,2) - diagonally opposite memory controllers.

Following table shows the time per migration (rounded up) obtained for both the approaches with different source and destination cores.

Table 3. Time for single process migration

	Migration Using LUT	Migration using shared memory
Core0 to Core1 on Tile(0,0)	1 ms	9 ms
Core0 on Tile(0,0) and Core0 on Tile(5,0)	1 ms	165 ms

Core0 on Tile (0,0) and Core0 on Tile (5,2)	1 ms	165 ms

The table shows a clear improvement in performance when a process is migrated using LUT based approach as compared to shared-memory based approach. It also shows that unlike the shared memory based approach the performance of the LUT based approach does not get affected by the distance between the memory controllers.

The proposed method of process migration is also evaluated for a possible use-case of process migration on SCC- migrating a memory intensive process to core that is closer to the memory controller which the process is accessing. The SCC has four memory controllers and a core can access any memory controller depending on the LUT configuration. The default LUT configuration provides shared memory on all the four controllers. Therefore if a process is heavily using memory on a memory controller but is executing on a core far from the controller on the mesh, it can benefit by migrating to the core closer to it. To demonstrate this a process was started on a core on Tile (0,0). The process was just copying contents of one memory location to another in a loop with both the locations belonging to the memory chip close to the Tile (5,2). The time taken by the process to execute the read/write loop was measured. After that the process was again started on Tile (0,0) but this time it was migrated to the Tile (5,2) before the loop started and the time taken by the loop was measured. As expected the time to complete the loop was less when the process was migrated. For 2^{21} iterations of the loop, it took 1.006 secs when the process was executing on Tile (0,0) whereas it took 0.77 secs when the process was migrated to Tile (5,2).

CHAPTER 5

RELATED WORK

Process migration has been a topic of research for many years and several implementations have been developed, both for distributed systems such as Sprite[4], Amoeba[29], V[2], Accent[3], DEMOS/MP[30], Charlotte[31] and for Unix such as Condor[32] and MOSIX[1]. Process migration can be implemented either at user-level or at kernel-level. At user-level, process migration is easier to implement and maintain as it does not require any changes to the operating system. However, at user-level it is difficult to extract the kernel-dependent state and not all processes can be migrated. Also, user-level process migration is less transparent and less efficient as compared to kernel-level process migration. An example is Condor which migrates processes using checkpoint/restore mechanism. Each process is linked to a user-level library which provides the process the ability to save its state to stable storage. This saved state is later used to restart the process on a different node. Kernel-level process migration is complex but more efficient than user-level migration. It is easier to capture the state of the process at kernel-level. Charlotte[31], Sprite[4], DEMO/MP[30] and V[2] are examples which employ kernel-level process migration techniques. The transfer of virtual address space of a process is a dominating factor in the performance of a process migration system. This is the reason that a lot of research in process migration has focused on reducing the cost of virtual memory transfer. Accent[3] uses a technique called lazy copying or Copy on Reference to reduce the initial cost of migration. In Copy on Reference, virtual memory pages stay on the source until they are referenced on the target machine. Pages are copied to the target only when the migrated process references them. Thus the pages that are not used are never copied. In the V[2] system, a method called Pre-copy is used to reduce the time for which a process is suspended during

migration. In this method, the process is allowed to continue execution while the address space is being transferred. In Sprite[4], the source flushes all the dirty pages to a file server while the process is being migrated. All the requests for the pages from the target are then served by the file server instead of the source. This method is called Flushing. This thesis also focuses on reducing the cost related to transfer of process state during process migration. However, the technique used is SCC-specific. It proposes a novel use of Lookup tables which are typically used for memory mapping. A related use of Lookup tables that has been proposed in the past is for efficiently copying large blocks of memory from one core to another in SCC[33]. In this case however, the memory block is physically copied.

REFERENCES

[1] Barak, A., La'aren O.(1998).The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems,13.*Retrieved from : http://www.sciencedirect.com/science/article/pii/S0167739X9700037X

[2] Cheriton, D.R.(1988).The V distributed system. *Communications of the ACM*, *31*.

[3] Rashid, R. Robertson, G. (1981). Accent: A communication oriented network operating system kernel. *Proceedings of the 8th Symposium on Operating System Principles*

[4] Douglis, F., Ousterhout J.(1990) Transparent Process Migration: Design Alternatives and the Sprite Implementation*,Ssoftware—practice and experience, 21.*

[5] Rashid, R., Julin, D., Orr, D., Sanzi, R., Baron, R., Forin, A., Golub, A., Jones, M.(1989) Mach: A System Software Kernel, *Proceedings of the 1989 IEEE International Conf. COMPCON.*

[6] Zayas, E.R.(1987).Attacking Process Migration Bottleneck. *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles.*

[7] Howard, J., Dighe, S., Vangal, S.R., Ruhl, G., Borkar, N., Jain, S., Erraguntla, V., Konow, M., Riepen, M., Gries, M., Droege, G., Lund-Larsen, T., Steibl, S., Borkar, S., De, V.K., Van Der Wijngaart, R.(Jan 2011) A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling, *IEEE Journal of Solid-State Circuits,46.* doi: 10.1109/JSSC.2010.2079450

[8] Eskicioglu, M.R.(1995).Design Issues of Process Migration Facilities in Distributed Systems. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.2276

[9] Milojičić, D. S., Douglis, F., Paindaveine, Y., Wheeler R., Zhou S.(Sept 2000) Process migration, *ACM Computing Surveys (CSUR)*, *32* n.3, p.241-299. doi:10.1145/367701.367728

[10] Mathias, N.(2003)Comparative Evaluation of Process Migration Algorithms, *Diploma Thesis, Operating Systems Group, Dresden University of Technology*

[11] Stallings, W. (2008*) Operating Systems: Internals and Design Principles, 6/E.* : Prentice Hall.

[12] Alard, E. and Bernard, G. (1992) Preemptive process migration in networks of UNIX workstations *Proc. 7th Int. Symp. on Computer and Information Sciences*

[13] Held, J., Bautista, J., Koehl, S. (2006) Tera-scale Computing Research Overview. Retrieved from

http://www.intel.com/content/www/us/en/research/intel-labs-tera-scale-research-paper.html

[14] Arlt, A., Hendrik, J.S. and Richling, J.(2011) Meta-programming Many-Core Systems. *In Proc. of 3rd MARC Symposium*

[15] Verstraaten, M., Grelck, C., Tol, M., Bakker, R., and Jesshope, C.R. (2011) On Mapping Distributed S-NET to the 48-core Intel SCC Processor. *In 3rd MARC Symposium*.

[16] Partheymuller, M., Stecklina,J. and Dobel, B.(2011) Fiasco.OC on the SCC. *MARC Symposium with Proceedings at HPI, Potsdam, Germany*

[17] Linnenbank,M.(2011) Implementing MINIX on the Single Chip Cloud Computer. *Master's thesis, Vrije Universiteit Amsterdam*

[18] Held, J. (2009) Exploring programming models with the Single-chip Cloud Computer research prototype. Retrieved from http://blogs.intel.com/intellabs/2009/12/02/sccloudcomp/

[19] Roy, B. (2011) Exploring the Intel Single-Chip Cloud Computer and its possibilities for SVP. *Master's Thesis, University of Amsterdam*

[20] Intel Corp. SCC Programmers's Guide V 1.0

[21] Mattson, T.G.Intel, Wijngaart, R.,Riepen, M., Lehnig, T.,Brett, P., Hass, W., Kennedy, P., Howard, J., Vangal, S., Borkar, N., Ruhl,G., Dighe, S.(2010) The 48-core processor: the Programmer's view. *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*

[22] Intel Corp. SCC Platform Overview Revision 0.7

[23] Intel Corp. SCC extended architecture specification, November 2010. Revision 1.1.

[24] Intel Corp. (1995) Pentium Processor Family Developers Manual Volume 3: Architecture and Programming Manual. Technical report, 2200 Mission College Blvd. Santa Clara, CA 94054-1549, USA.

[25] Mattson, T. and Wijngaart, R., (2010) RCCE: a Small Library for Many-Core Communication. Retrieved from http://www.intel.com/content/www/us/en/research/intel-labs-rcce-single-chip-cloud-brief.html

[26] Bare Metal OS for Intel's SCC "Cloud on a Chip". Retrieved from http://www.etinternational.com/index.php/projects/intel-scc

[27] Ziwisky, M., Byrlow D.,(2012) BareMichael: A Minimalistic Bare-metal Framework for the Intel SCC. *Proc. of the 6th MARC symposium, ONERA*

[28] sccAPI lib and docs. Retrived from http://communities.intel.com/message/93196#93196

[29] Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R., and van Staveren, H. (May 1990). Amoeba – A Distributed Operating System for the 1990s. *IEEE Computer, 23(5):44–53.*

[30] Miller, B. and Presotto, D., Powell, M (April 1987). DEMOS/MP: The Development of a Distributed Operating System. *Software-Practice and Experience, 17(4):277–290.*

[31] Artsy, Y. and Finkel, R. (September 1989). Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer, pages 47–56.*

[32] Litzkow, M., Tannebaum, T., Basney, J., and Livny, M.,(April 1997). Checkpoint and Migration of Unix Processes in the Condor Distributed Processing System. *Techincal Report#1346.Compute Sciences Department, University of Wisconsin-Madison*.

[33] Tol. M., Bakker, R., Verstraaten, M., Clemens Grelck and Jesshope, C.R. Efficient Memory Copy Operations on the 48-core Intel SCC Processor.*3rd MARC Symposium*

[34] Running programs with input files on SCC. Retrieved from http://communities.intel.com/message/146008#146008

APPENDIX A

DEFAULT LUTs

Table 4. Default LUT entries for 32 GB System Memory

LUT #	Physical Address	Maps to
255	FFFFFFFF - FF000000	Private
254	FEFFFFFF - FE000000	N/A
253	FDFFFFFF - FD000000	N/A
252	FCFFFFFF - FC000000	N/A
251	FBFFFFFF - FB000000	VRC
250	FAFFFFFF - FA000000	Management Console TCP/IP Interface
249	F9FFFFFF - F9000000	N/A
248	F8FFFFFF - F8000000	N/A
247	F7FFFFFF - F7000000	System Configuration Register -- Tile 23
246	F6FFFFFF - F6000000	System Configuration Register -- Tile 22
245	F5FFFFFF - F5000000	System Configuration Register -- Tile 21
244	F4FFFFFF - F4000000	System Configuration Register -- Tile 20
243	F3FFFFFF - F3000000	System Configuration Register -- Tile 19
242	F2FFFFFF - F2000000	System Configuration Register -- Tile 18
241	F1FFFFFF - F1000000	System Configuration Register -- Tile 17
240	F0FFFFFF - F0000000	System Configuration Register -- Tile 16
239	EFFFFFFF - EF000000	System Configuration Register -- Tile 15
238	EEFFFFFF - EE000000	System Configuration Register -- Tile 14
237	EDFFFFFF - ED000000	System Configuration Register -- Tile 13
236	ECFFFFFF - EC000000	System Configuration Register -- Tile 12
235	EBFFFFFF - EB000000	System Configuration Register -- Tile 11
234	EAFFFFFF - EA000000	System Configuration Register -- Tile 10
233	E9FFFFFF - E9000000	System Configuration Register -- Tile 09
232	E8FFFFFF - E8000000	System Configuration Register -- Tile 08
231	E7FFFFFF - E7000000	System Configuration Register -- Tile 07
230	E6FFFFFF - E6000000	System Configuration Register -- Tile 06
229	E5FFFFFF - E5000000	System Configuration Register -- Tile 05
228	E4FFFFFF - E4000000	System Configuration Register -- Tile 04
227	E3FFFFFF - E3000000	System Configuration Register -- Tile 03
226	E2FFFFFF - E2000000	System Configuration Register -- Tile 02
225	E1FFFFFF - E1000000	System Configuration Register -- Tile 01
224	E0FFFFFF - E0000000	System Configuration Register -- Tile 00
223	DFFFFFFF - DF000000	
:	:	:
216	D8FFFFFF - D8000000	
215	D7FFFFFF - D7000000	MPB in Tile (x=5,y=3)
214	D6FFFFFF - D6000000	MPB in Tile (x=4,y=3)
213	D5FFFFFF - D5000000	MPB in Tile (x=3,y=3)

LUT #	Physical Address	Maps to
212	D4FFFFFF - D4000000	MPB in Tile (x=2,y=3)
211	D3FFFFFF - D3000000	MPB in Tile (x=1,y=3)
210	D2FFFFFF - D2000000	MPB in Tile (x=0,y=2)
209	D1FFFFFF - D1000000	MPB in Tile (x=5,y=2)
208	D0FFFFFF - D0000000	MPB in Tile (x=4,y=2)
207	CFFFFFFF - CF000000	MPB in Tile (x=3,y=2)
206	CEFFFFFF - CE000000	MPB in Tile (x=2,y=2)
205	CDFFFFFF - CD000000	MPB in Tile (x=1,y=2)
204	CCFFFFFF - CC000000	MPB in Tile (x=0,y=2)
203	CBFFFFFF - CB000000	MPB in Tile (x=5,y=1)
202	CAFFFFFF - CA000000	MPB in Tile (x=4,y=1)
201	C9FFFFFF - C9000000	MPB in Tile (x=3,y=1)
200	C8FFFFFF - C8000000	MPB in Tile (x=2,y=1)
199	C7FFFFFF - C7000000	MPB in Tile (x=1,y=1)
198	C6FFFFFF - C6000000	MPB in Tile (x=0,y=1)
197	C5FFFFFF - C5000000	MPB in Tile (x=5,y=0)
196	C4FFFFFF - C4000000	MPB in Tile (x=4,y=0)
195	C3FFFFFF - C3000000	MPB in Tile (x=3,y=0)
194	C2FFFFFF - C2000000	MPB in Tile (x=2,y=0)
193	C1FFFFFF - C1000000	MPB in Tile (x=1,y=0)
192	C0FFFFFF - C0000000	MPB in Tile (x=0,y=0)
191	BFFFFFFF - BF000000	
:	:	:
132	84FFFFFF - 84000000	
131	83FFFFFF - 83000000	Shared MCH3 - 4MB
130	82FFFFFF - 82000000	Shared MCH2 - 4MB
129	81FFFFFF - 81000000	Shared MCH1 - 4MB
128	80FFFFFF - 80000000	Shared MCH0 - 4MB
127	7FFFFFFF - 7F000000	
:	:	:
85	55FFFFFF - 55000000	
84	54FFFFFF - 54000000	Private
49	31FFFFFF - 31000000	Private
48	30FFFFFF - 30000000	Private
:	:	:
1	01FFFFFF - 01000000	Private
0	00FFFFFF - 00000000	Private

Note. Reprinted from [23]