Operating System Support For Reconfigurable Cache

by

Vignesh Makeswaran

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Fakultät für Elektrotechnik, Informatik und Mathematik
Heinz Nixdorf Institut und Institut für Informatik
Fachgebiet Technische Informatik
Warburger Straße 100
33098 Paderborn

Operating System Support For Reconfigurable Cache

Master's Thesis

Submitted to the Computer Engineering Research Group
in Partial Fulfillment of the Requirements for the
Degree of
Master of Science

by
Vignesh Makeswaran
Böchinger Straße 1
76829 Landau in der Pfalz

Thesis Supervisor:
Prof. Dr. Marco Platzner
and
Jun-Prof. Dr. Christian Plessl

Paderborn, November 2015

Declaration

(Translation from German)

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Original Declaration Text in German:

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

City, Date

Signature

Acknowledgment

I would like to express my special appreciation and thanks to my advisers Mr.Nam Ho, for his guidance, support, and advice. I have greatly benefited from working with him. I would like to thank Professor Marco Platzner for giving me the opportunity to work in his research group, and for serving as main supervisor. I would also like to thank Dr. Christian Plessl for the inspiring guidance and for serving as an examinar. I would like to thank my father Makeswaran, my mother Banumathi and my aunt Annapooranam for all of the sacrifices that you've made on my behalf. Special thanks to Rüdiger Kern and Karin Kern who supported me during my study in Germany, and incented me to strive towards my goal. Finally, a very special thanks to my boy friend Lukas Kern who is the person behind all my success.

Contents

List of Figures

List of Tables

1 Introduction

In a computer system, memory plays a very important role in computing the given instructions. Memory is used to store the data, instructions, results, etc. Memory is used both for temporary storing of values during an execution or for storing values before and after the execution. For a processor to execute any given user application, the processor needs to load the user code from the memory.[10] There are different types of memory in a computer system like main memory, secondary memory, cache memory, etc. Usually the processor needs to access information from the main memory during execution. But processor directly accessing the main memory gives slow memory latency. Hence, cache memory concept is introduced to overcome the delay caused by the processor direct main memory access.[18] A cache memory replicates temporarily certain parts of the main memory which the processor can access during the execution. Basically the cache memory duplicates certain memory blocks of the main memory which will be used by the processor for the execution of the current process. This block of memory duplicated in the cache memory changes for every process that is being executed in the processor since each process may have different main memory blocks corresponding to them. Thus the information stored in the cache memory needs constant refreshing, flushing and mapping of the memory blocks. Generally the cache memory flushing and mapping takes place during context switching when the processor changes it's current process which is being executed.
A cache memory works by mapping the main memory contents. The cache memory is placed nearer to the processor for lower memory latency. The lower the memory latency the better.[18] The Figure 1.1 shows the bandwidth increases and the memory latency decreases as more the memory is closer to the processor. The informations located in cache memory are accessed in much less time than that are located in the main memory.[20] Thus, a central processing unit (CPU) with a cache memory needs to spend far less time waiting for instructions and operands to be fetched and/or stored[20]. There has been a huge improve in the performance of the processors due to technology development as stated by the Moore's law, but compared to that the memory performance has not seen much improvement. As each year passes by the perfromance gap between the processor and the memory widens. Many researchers are focused in decreasing the gap between the process performance and the memory performance. One of the lesser explored topic in improving memory performance is custom cache mapping.

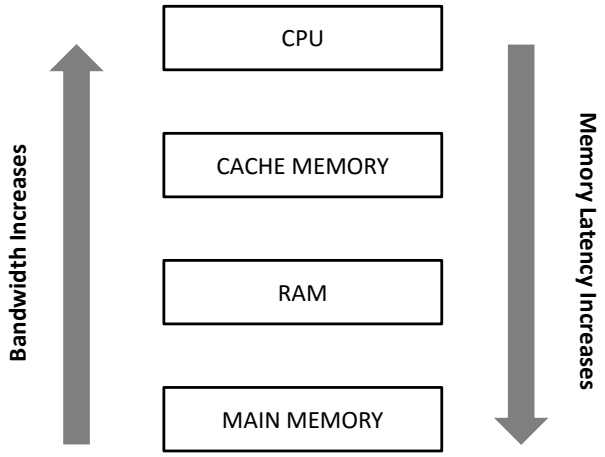

Figure 1.1: Bandwidth and memory latency in a computer system

In general systems, the application executed by the processor is unknown. The environment in which a general computer system is deployed is unknown. Also a very wide range of applications are executed by the processor. Hence, a general cache mapping function is supported by the operating systems to provide an fault free execution. But such general cache mapping function decreases the potential of the cache memory. Exploiting the utilization of cache memory potential performance can improve the memory performance. Though the idea is very hard to visualize for a general system due to huge range of applications, in embedded systems the applications executed is usually limited or known by the system. Thus the idea of improving the cache memory performance by using a custom cache mapping function for the specific application that is executed is more practical in an embedded system.
There are two major requirements for such cache memory performance improvement:
1. Customised cache mapping function for each application.
2. Operating systems which can reconfigure the customised cache mapping during run time.

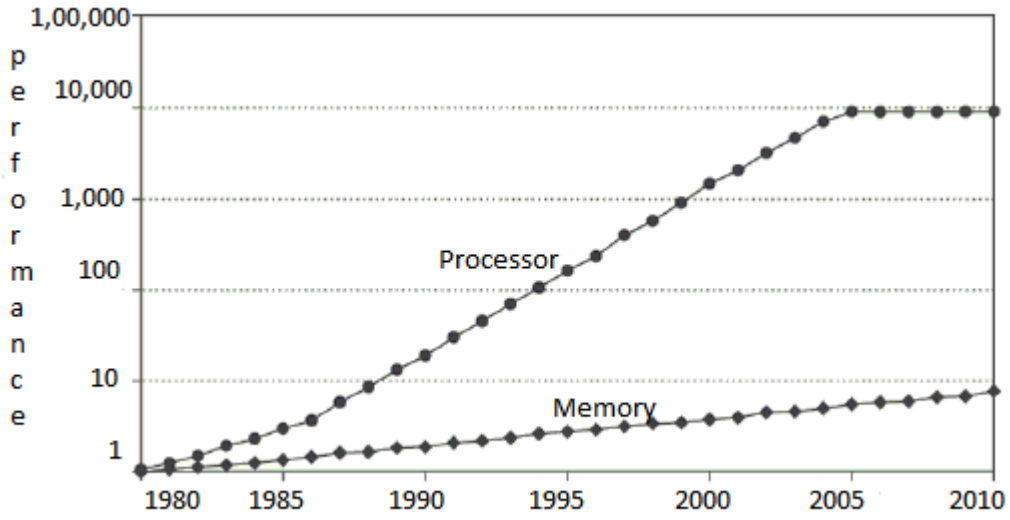

Figure 1.2: Performance estimation of memory and processor[11][6]

Developing a customised cache mapping is explored in detail in [15]. At this point, we have user applications and their corresponding customized cache mapping function. The requirement for the user is to have an operating system which uses the corresponding customised cache mapping function with respect to the process being executed in the processor. In other words, the user wants the operating system to reconfigure the cache mapping function according to the user application executed by the processor. We need to answer ***How the operating system can recognize and reconfigure the cache mapping function? This thesis provides an answer to How the operating system can reconfigure the cache mapping function.*** This thesis explores one of the successful ways to reconfigure the cache mapping function by the operating system during run time. The main concept of this thesis is that the operating system gets the customized cache mapping function as an input from the user to run one of the operating system feature, which is reconfiguring the cache mapping function. Generally, a user application gets the input from the user and the operating system runs the user application by executing the program. But in this scenario, the operating system gets the input from the user to replace an existing cache mapping function. By Linux kernel design, the user cannot directly communicate with the operating system internals, the kernel.[18] The user needs a mediator which is an application called user controller which provides channel for an indirect communication between the user and the kernel. The operating system has to recognize and change the cache mapping function, for which we had identified two players, they are:
1.*Frond end - User Controller*: User Controller gets the user inputs and provides it to the kernels. The user controller must know all the excepted inputs the user will be giving. And the user controller must extract the user input data from

the configuration files and convert them into the kernel understandable format and send it to the kernel. The user controller must also understand the channel through which it communicates to the kernel.
2. *Back end - Kernel Module*: The kernel module gets the custom cache mapping function from the user controller and stores in it. Before the respective application is being executed, the kernel must reconfigure the cache mapping function with the corresponding customised cache mapping function it has stored for that applciation.The operating system must have the knowledge of when to recofigure a cache mapping and what to reconfigure.
This thesis discusses one possible approach to support the reconfigurable cache mapping and provides a synchronous communication between the above mentioned two players user controller and kernel. Apart from the front end and back end players, the user plays an important role in configuring and controlling the reconfigurable cache mapping kernel module. The user must provide the input customised cache for an user application. The user must know the application name and the corresponding customised cache mapping. The user must also have the knowledge of communicating with the user controller. The user must be well informed about the format in which the inputs should be given to the user controller. The solution provided in this thesis is implemented and tested successfully in an embedded system environment. The thesis is contributed with six chapters. Introduction chapter, which is this chapter describes the idea and requirement for this thesis and the motivation. Next in the second chapter, Background, provides indepth information about the custom cache mapping functions, introduction to EvoCache, the hardware used in the thesis, the environment in which the thesis is implemented, detailed discussion about how the user applications are executed inside the Linux operating system and identified methods to implement the reconfigurable cache mapping function. In the third chapter, Reconfiguring cache mapping, gives a brief information about an optimized cache mapping method EvoCache hardware platform and explains in detail about the method chosen to reconfigure the cache mapping function. The fourth chapter, Design and System integration, describes in details the each main parts developed to support reconfigurable cache mapping and how they communicate. In Chapter five, Evaluation, explains how the evaluation and the experiments are carried out and the results obtained during the evaluation. Finally, in the last chapter, Conclusions, summarizing the thesis, and discussing the future works. And the appendix, has the user manual for the reconfigure cache mapping user application and different supported configuration file samples.

2 Background

This chapter briefs about cache mapping function, customized or optimized cache mapping function, the hardware setup, softwares and language used for implementation, how an user application is executed inside the operating system and specifically Linux and identified methods to implement the thesis.

2.1 Cache Mapping

In modern day systems, there is a growing gap between CPU and main memory performance. Hence, to minimize this performance gap, hierarchical memory concept is introduced. The hierarchical memory helps to increase the memory bandwidth and decrease the memory latency.[18][10] In this concept, the memory components are placed between the processor and the main memory. These memory components are called caches. There are different levels of caches L1, L2 and L3 as shown in the Figure 2.1.[18][20] In some systems there are even move levels of caches. The L1 caches are in the processor chip and the L2 cache is usually on the mother board and the L3 cache are slower caches than the L2, and can be in the motherboard. The L1 cache which is the closest to the CPU is placed on the chip for maximum performance. This L1 cache is divided into two parts:
1. Data cache, for the data fetched from the main memory.
2. Instruction cache, for the instructions fetched from the main memory.
The information stored in the caches are stored as cache lines. The cache lines are blocks of data copied from the main memory. When the processor needs instructions or data to read or write or execute, it searches for the memory block in the cache lines. If the memory block is found then the information is provided to the processor by the caches. If the memory block is not found then cache miss occurs. When the instruction required by the processor is not available in the cache then instruction cache miss occurs and similarly when the data required by the processor is not available in the cache then data cache miss occurs. When a cache miss occurs, the cache then fetches the requested block of memory from the main memory and replaces an existing cache line, which reduces the performance due to the low bandwidth and high latency by the main memory. This affects the performance of the system.[18][10][17]

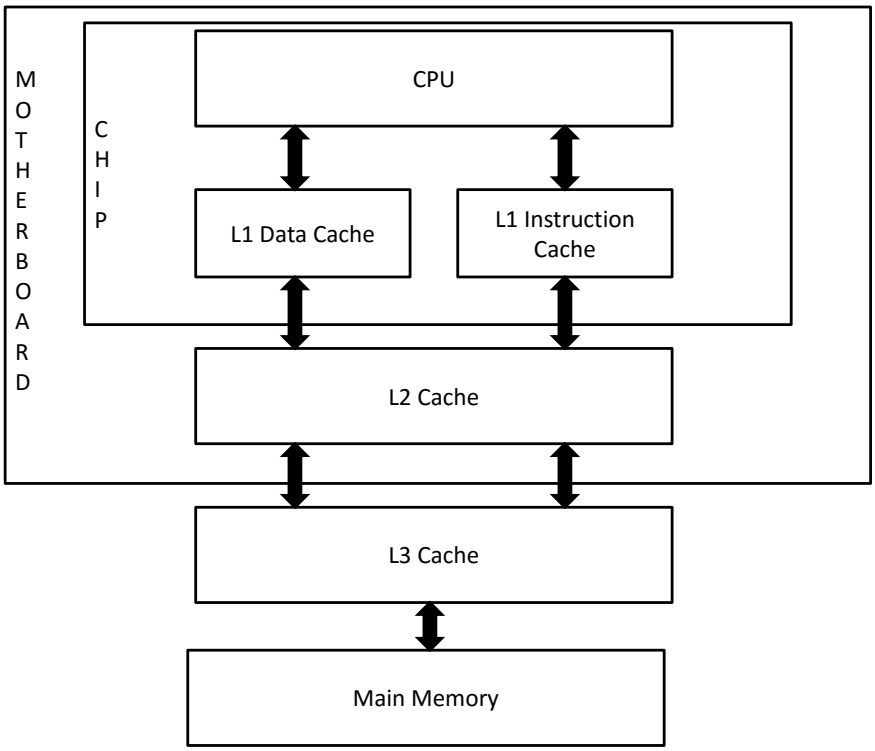

Figure 2.1: Interaction between different levels of caches and the CPU[17]

Cache mapping is the mechanism of mapping the main memory blocks into the cache. A good cache mapping technique decreases the cache miss rate and increases the performance. The cache mapping technique must decide which cache lines should be removed and add the newer cache lines. The more optimized replacement of the cache lines reduces the cache miss. Since the behaviour of an user application when executed is not know, using an optimized cache mapping technique is not possible in a general system. But in embedded system since the applications are limited and are known, it possible to customize the cache mapping for the given user application.

2.2 EvoCache

Customized cache mapping is having a customized cache mapping function specialized to improve the performance while executing the application for which it was developed. One of the methods for generating such customized cache mapping we discuss here is EvoCache. The EvoCahe (Evolutionary Cache), is a method of obtaining the optimized cache mapping function for a specific given application by using evolvable hardware and evolutionary optimization algorithm.[16]
The EvoCache uses an evolutionary optimization algorithm. The evolutionary optimization algorithm used by EvoCache is a hashing function to map part of

the main memory address needed for the application execution to the cache line index. Then this hash function is optimized to get an optimized cache mapping function for that application. The EvoCache is achieved by using Cartesian Genetic Programming, which is a FPGA hardware.[16]
A simplified representation of generating EvoCache is shown in Figure 2.2, the evolutionary optimization algorithm generates individual optimized cache mapping for each given user applications like jpeg. These optimized cache mapping are customized cache mapping technique which reduces the cache miss rate compared to a default common cache mapping techniques when the corresponding user application is executed by the processor.

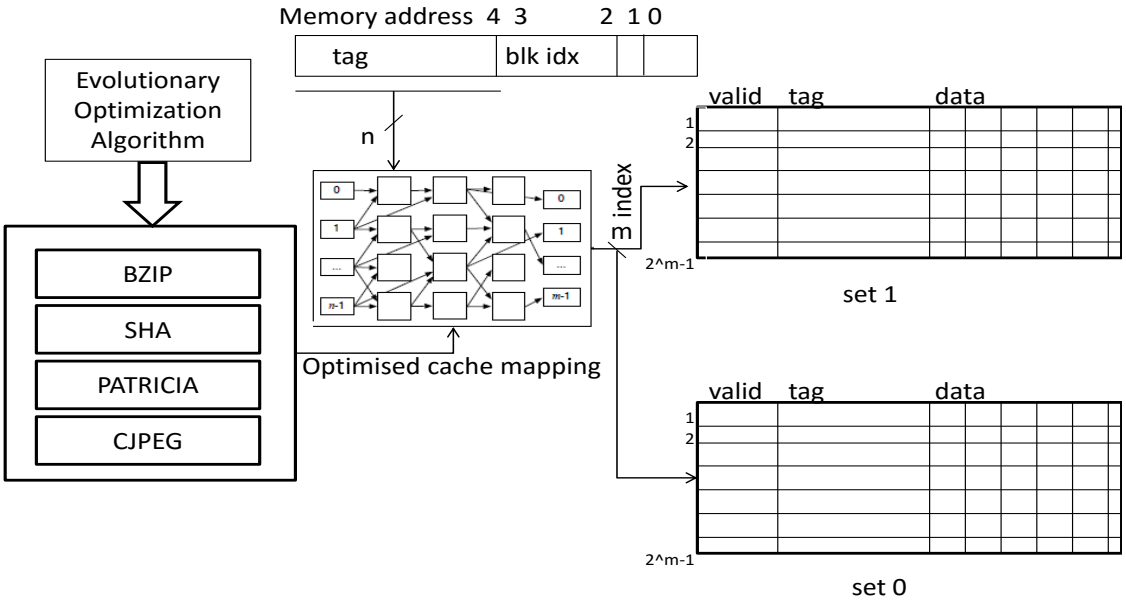

Figure 2.2: Evolutionary algorithm generates EvoCache for each application[16]

2.3 Hardware

Reconfigurable computing is achieved by bringing together the evolvable hardware and evolutionary algorithms. Evolvable hardware are a type of hardware, which are designed to build circuits according to the design specification without any manual engineering. The main concept of evolvable hardware is the ability of the hardware system to change the architecture by altering the structure. For the evolvable hardware to reconfigure its hardware, evolutionary algorithms are used.

Generally, evolvable hardware was introduced for adapting the systems by using evolutionary algorithms.[14]
The concept of reconfigurable computing using evolvable hardware was evolved from the field-programmable gate array (FPGA). FPGA provided run time reconfigruation of the hardware which forms the basis of reconfigurable computing. FPGA are reprogrammable logical circuits with a grid of logic gates. And these logic gates are manipulated to reconfigure the hardware. This thesis uses the FPGA for reconfiguring the cache mapping.[8]
In this thesis we use Virtex6 device ML605 Evaluation Kit by Xilinx. The kit provides Virtex6 XC6VLX240T1FFG1156 FPGA.[3]

2.4 Environment

In this thesis, we use Leon3 SoC platform, the open source platform. The Leon3 processor core is a 32-bit processor compliant with the SPARC V8 architecture. The core uses AMBA 2.0 AHB bus for interfacing and supports the IP core plug & play method provided in the Gaisler Research IP library (GRLIB) [2]. Leon3 contains two levels of cache structure, a separate instruction and data cache. And also allows to reconfigure both the caches.[2] The thesis is implemented in Linux kernel version 1.0.6. and implemented entirely using C language.

2.5 Application Execution Design in a Operating System

Linux operating system has different levels of communication as shown in Figure 2.3. These different levels of communication are to protect the system from memory and data corruption and for better fault tolerance. There is a user space, which is used for executing all the user related tasks like running a user application. The kernel space is used for executing the operating system internals and device drivers. For a application in user space to communicate with the kernel, we need an inter process communication like netlink, IOCTL(input/output control).[10]

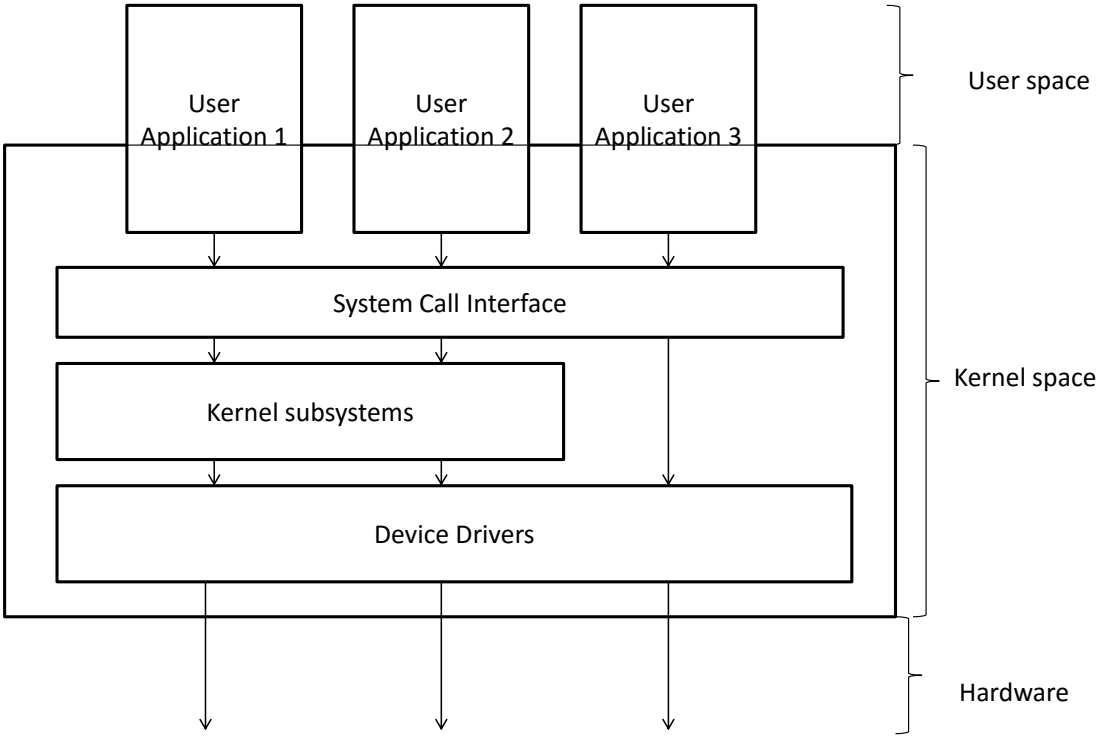

Figure 2.3: Interaction between user space, kernel space and hardware[18]

When an user application is executed in Linux, it means a process is created for that user application and is placed in a *run queue*. The processor gets the next process to be executed from it's run queue. The run queue contains the process which are ready to be executed. Each processor in a system has it's own run queue. The processes which are going to be executed by the a process are placed in their corresponding run queue. A processor becomes idle when it's run queue is empty. Usually the run queue is 140 queues each queue has different priority level. The process from the highest priority queue is executed first and when the queue becomes empty then the processes from the next highest priority queue will be executed. Thus the processor executes all the process in the 140 queues. There are two sets of 140 queues, one set is active and the other is expired. A processor has an expired queue for each of the 140 run queues and after expiration of the allotted execution time for that process, the process is sent to it's expired queue. The processor thus empties the active queues and sends them to their respective expired queues. When all the active queues are empty, the expired queues and active queues are inter changed and the processor starts executing the processes from the highest priority active queue. The highest priority process is given the longest run time in the process.[19]

Figure 2.4 shows the process life cycle state diagram.[18]
1. *New* - a new process is created by an existing task, in this case by executing the user application.
2. *Ready* - the process is ready for execution and is placed in the run queue. Processes goes to ready state from a new state after creation, a current running process can be sent back to ready state by another higher priority task and also from a wait state when the waiting event has occurred.
3. *Running* - the process is currently running in the processor.
4. *Terminate* - the execution of the process is completed.
5. *Wait* - the process cannot continue execution because it is waiting for an request to be satisfied or a event to occur.

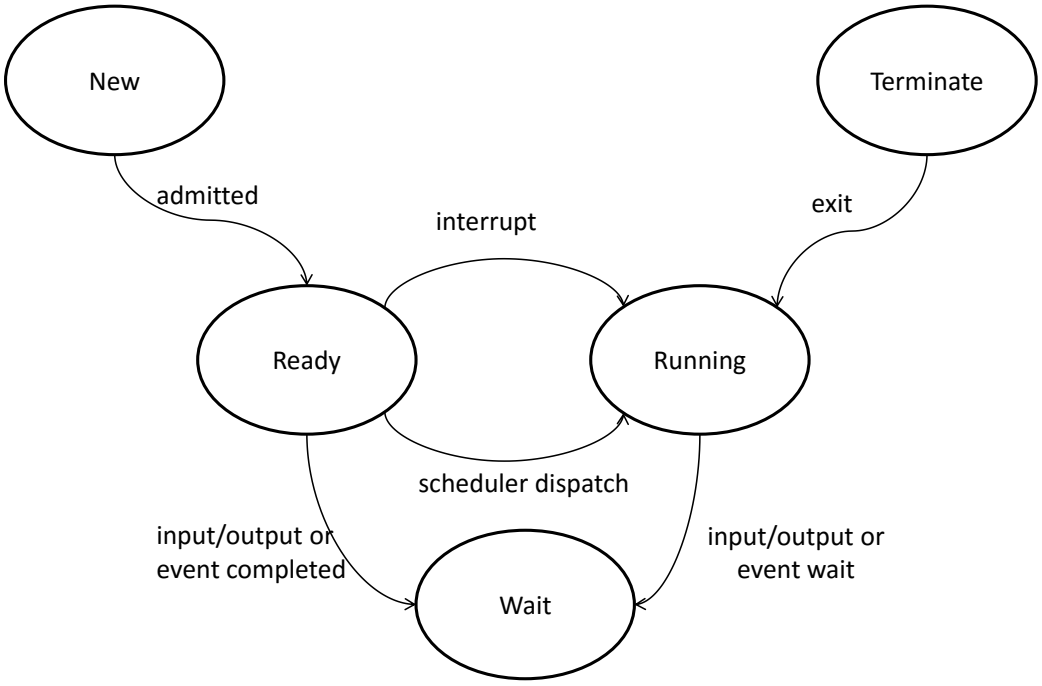

Figure 2.4: Different Process states in a process lifetime[18]

There is also another queue which is called as *wait queue*. Wait queue contains the processes which are in wait state. When a process is waiting for an event, then that process is sent to the wait queue to sleep. The processes in the wait queue is woken up by waking up the wait queue. The processes must then check for the condition whether the event is happened or not and if not happened, then it goes

back to sleep. If the condition is satisfied, then it goes to run queue.[19][10]
When the processor finishes executing a process for the allocated time, the processor removes the process and loads the next process to execute. This switching between processes by the processor is called as context switching. During context switching, the current state of the executed process is saved to it's Process Control Block(PCB). And the loads the state of the next process from the PCB of the next process. A PCB is a data structure which contains all the information about the corresponding process which is required by the processor to execute. The PCB has the identifier for that process, the state of the process and control information of that process.[19][10][18]
The Figure 2.5 shows how a normal context switching takes place between two processes.

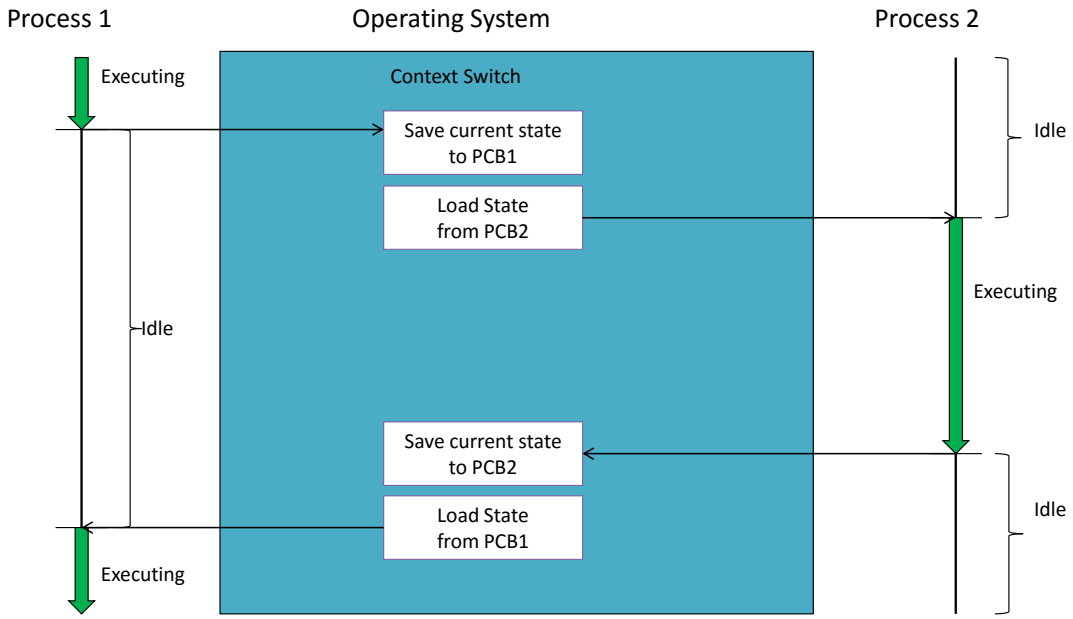

Figure 2.5: Normal context switching in an Operating system[18][10]

In the Figure 2.5, there are two processes Process 1 and Process 2 , the Process 1 is currently executing, when the allocated time slice is completed, the processor saves the current status of the Process 1 in Process 1 PCB and the Process 1 is placed in the run queue. The next process, Process 2 is loaded from the run queue

using its PCB by the processor. The processor, then executes the Process 2 for the given time slice and upon completion, it saves the Process 2 status to the Process 2 PCB and the Process 2 is placed in the run queue. Then the processor gets the next process , in this case it is Process 1 again. The switching from Process 1 to Process2 and then from Process 2 to Process 1 is called as the context switching.

2.6 Application Execution in Linux

A user application is an executable file. [18] One of the commonly supported executable file in Linux is ELF(Executable and Linking Format). An elf file has a elf header, a program header table , section header tables and data. The elf header contains the general information about the file. The program header tables contains the information where the code and data are located in the file and later where in the executable image. The section table header entries links to the data in the elf file for linking or relocation or junk data to identify which data is located in which section. [1]
When an ELF file is executed, one of the execution function `execve` is called by the Linux with the address space and string of variables. The page frames are allocated for the user application and `do_execve` function is called.[7]
Here the data structure is allocated, the file is checked whether executable and scheduling takes place. The scheduler decides which CPU to execute this file according to the current load in all the CPUs. Then `prepare_binprm` function is called. It again validates the file for executable and allocates the user id and group id. Checked for valid format of the file. If there is a failure in any of the above checks performed, the error code is returned and the allocated page frames are freed. Upon successful checks, `execve` terminates with valid address spaces, data structures and descriptors. Now the executable file continues the execution but with the new content in the address space changed by `execve`, executable image.[7]
In next step, dynamic linking is done. The mmap function is used to map the identified shared libraries and functions to the pages. After the dynamic linking is completed, the entry point of the executable file is pointed and the code is executed one by one.[7]

2.7 Modifying Linux

To support reconfiguring cache mapping, two methods are identified.
1. Method I - Inserting the optimized cache mapping values in to the elf file in a separate unique section. While context switching , before executing a process we check for the section in its elf file and get the optimized cache mapping values and

reconfigure the operating system. Advantages of this method is that each user application contains their optimized cache mapping function values inside their elf files. No separate handling of the optimized cache values. Disadvantages of this methods are, both the elf file and the kernel modules are modified. During execution, we need to get the optimized cache mapping function values from the user space to the kernel space, might affect the system performance. Many kernel modules needed to be modified context switching, accessing the run queue to get to the elf file, which results in additional kernel code executions.
2. Method II - ELF files are not modified, a separate user application for reconfiguring cache is created. The user application names and their corresponding customized cache mapping function values are given to this reconfigurable cache user application. The user application contacts the kernel and provides the information given by the user, the kernel stores this information. During context switching the kernel checks whether to reconfigure the cache mapping according to the stored information and reconfigures if needed. Advantages of this method are only one kernel module, context switching is modified, no ELF file modification, optimized cache mapping function values are stored inside the kernel. Disadvantages user must know how to give the input to the reconfigurable cache user application.

3 Reconfigurable Cache Mapping

This chapter explains about the hardware platform to reconfigure the cache mapping and method used to reconfigure the cache mapping function in this thesis.

3.1 Cache Mapping Hardware Platform

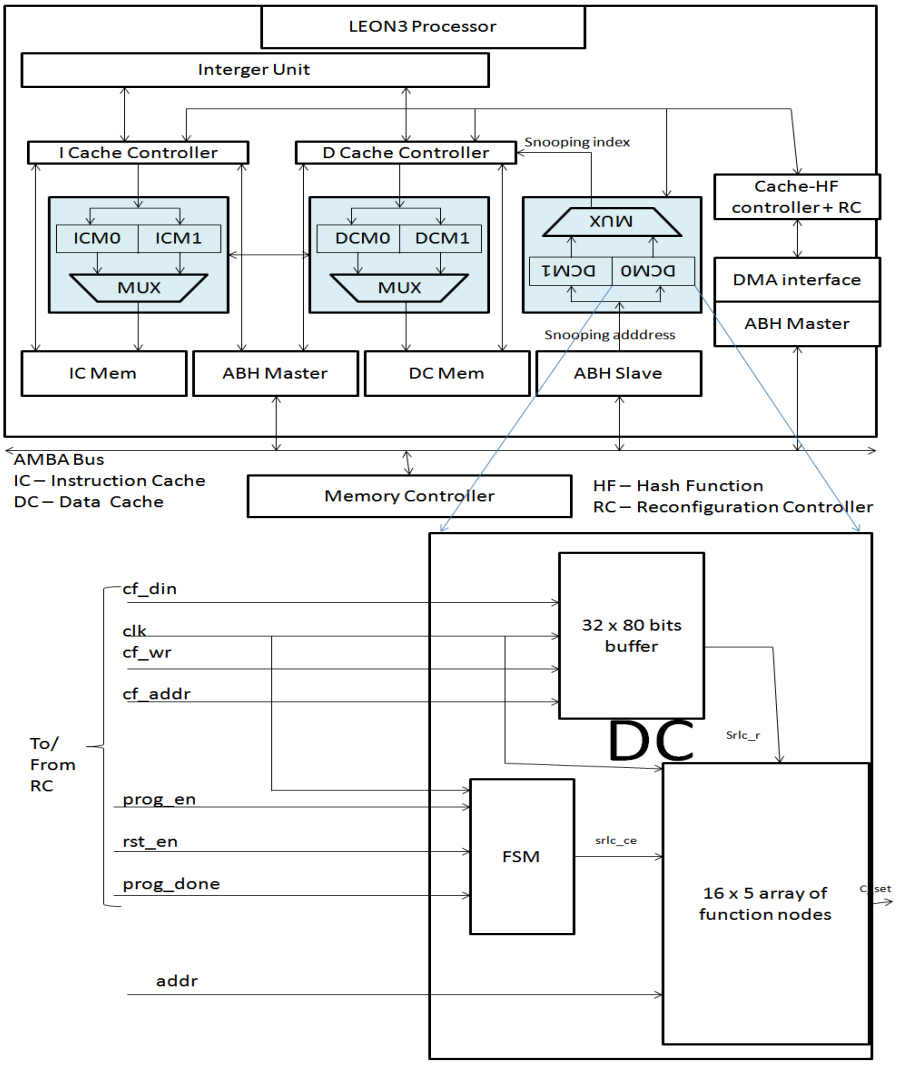

Figure 3.1: CGP architecture with reconfigurable cache mapping.[6]

Figure 3.1 shows the architectural block diagram of Cartesian Genetic Programming(CGP) with reconfigurable cache mapping. The overall system is in a single FPGA with the reconfiguring cache mapping unit is implemented in between the Cache memory and the cache controller. The reconfiguration controller in the unit performs the cache mapping functions of loading the memory blocks from the main memory and controlling the cache memory. The Cache-HF controller controls the removing of the cache mapping function and inserting new cache mapping function. The function nodes in the Figure 3.1 is where it has to be reconfigured with the values of the optimized cache mapping. Since Leon3 processor support reconfiguring the instruction and data caches, using the controllers mentioned above the caches are reconfigured in the hardware level. The operating system communicates with the hardware using device drivers to provide the values of the optimized cache mapping function. [16][6]

3.2 Operating System Modifications

As briefed in section 2.7, the operating system needs to be modified to support reconfiguring the cache mapping. The method we chose to implement in this thesis is method II mentioned in section 2.7, that is, having a reconfigurable cache user application and a reconfigurable cache kernel module. The reason to choose this method are:
1. Guarantees minimum operating system flow modification.
2. Avoiding unnecessary complicated interaction with run queue.
3. Performance for this method is expected to be better since the whole process of reconfiguration is handled within the kernel space.
4. Only a single function call can carry out the reconfiguration of cache mapping.
5. No ELF file modification required.
The reconfigurable cache module communicates with the corresponding device driver functions to change the cache mapping function in the underlying Leon3 instruction and data caches.
For an operating system to support reconfiguring cache mapping, it has to include the reconfiguring of cache mapping function in it's normal flow of action. As shown in Figure 3.2, the idea is to include the reconfiguration of cache mapping as the initial step in the context switching. It is designed to interact with the minimum default kernel modules as possible to keep the control flow change to as low as possible. The whole reconfiguration of the cache mapping function is completed in the first step of the context switching itself.

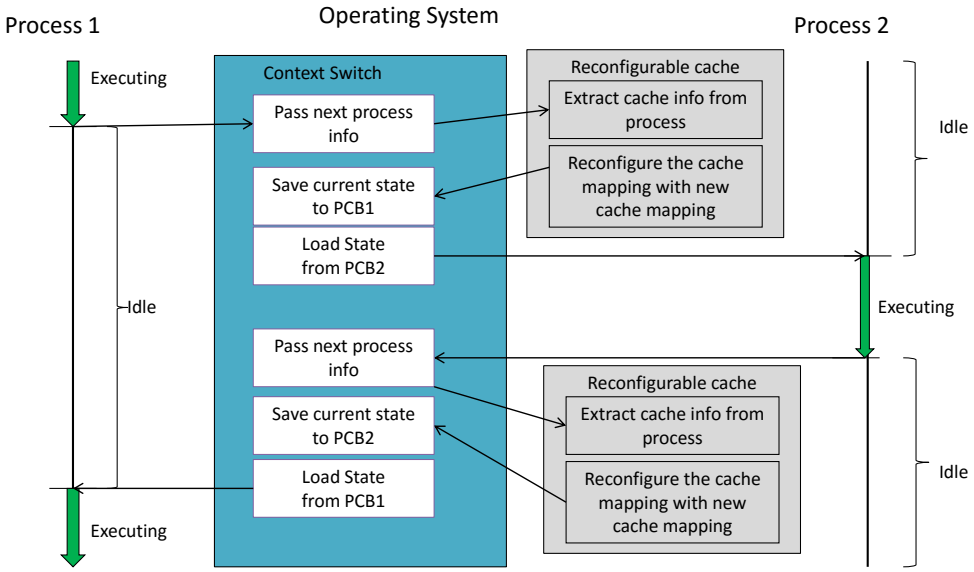

Figure 3.2: Context switching with reconfigurable cache in Linux

Our goal is not to have a huge increase in the context switching time due to the reconfiguration of the cache mapping. There will be an increase in the context switching time but we focus that the increase in context switching time is as minimum as possible. The context switching time is included in the system time during a process execution. Hence to benefit from reconfigurable cache mapping, the system execution time should also be maintained as minimum as possible. Here, the system time is the time spent on system code usually, the time spent in the kernel space.

4 Design and System Integration

In this chapter, we present the complete design and implementation details of this thesis.

4.1 Architecture

The aim is to be able to give the optimized cache mapping function of a user application as the input to the operating system. And the operating system should use that optimized cache mapping function when the corresponding user application is being executed by the processor. The input is given by the user to the operating system and the output is processor execution with the given input customized cache mapping in the cache memory. Hence, the design of this thesis is divided into two main parts, backend - the kernel part and frontend - the user part. The block diagram Figure 4.1 shows the overview of the complete design.
The user part consists of the *Reconfigurable Cache User Controller (RCUC)* and the configuration files. And the kerenl part consists of the reconfigurable cache kernel module. The detailed explanation of each part, the connection between them and the data flow of the corresponding data path in the operating system as described in following sections.
In the Figure 4.1, the block *Reconfigurable Cache Kernel Module (RCKM)* is the newly developed kernel module which handles all the kernel operations regarding the reconfiguring cache mapping function. The scheduling represents the scheduling module which also handles the context switching. The reconfigurable cache kernel module is called only during the context switching.

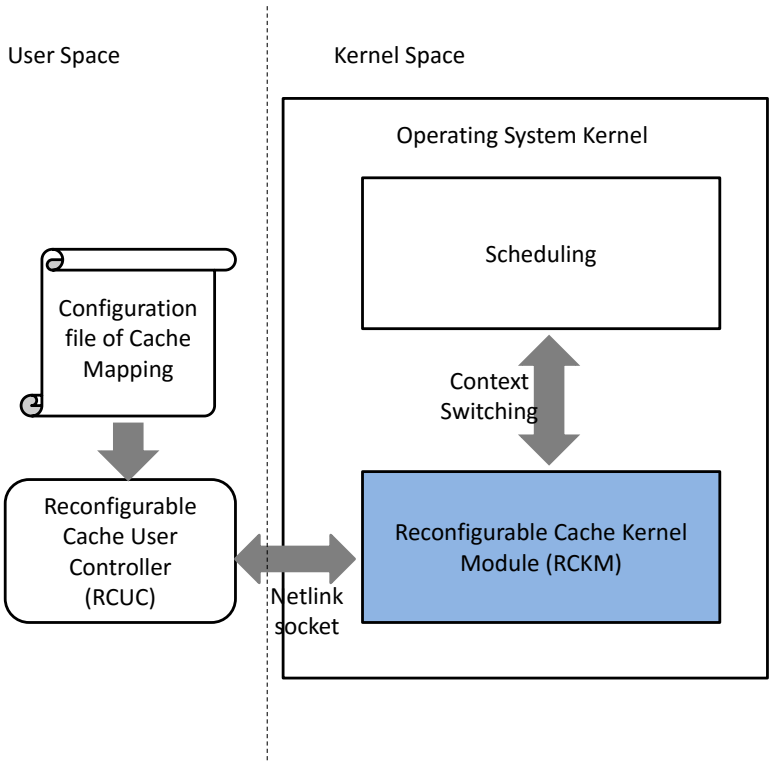

Figure 4.1: Simplified architectural block diagram of the overall system

4.2 Reconfigurable Cache Kernel Module (RCKM)

This section explains the design of the kernel part, that is the reconfigurable cache kernel module. The reconfigurable cache kernel module handles all the interactions with the RCUC. It is the only kernel module to which the RCUC interacts. The RCKM also interacts with other kernel modules and is responsible for reconfiguring the cache mapping function. Figure 4.2 shows the overview of the kernel module part.

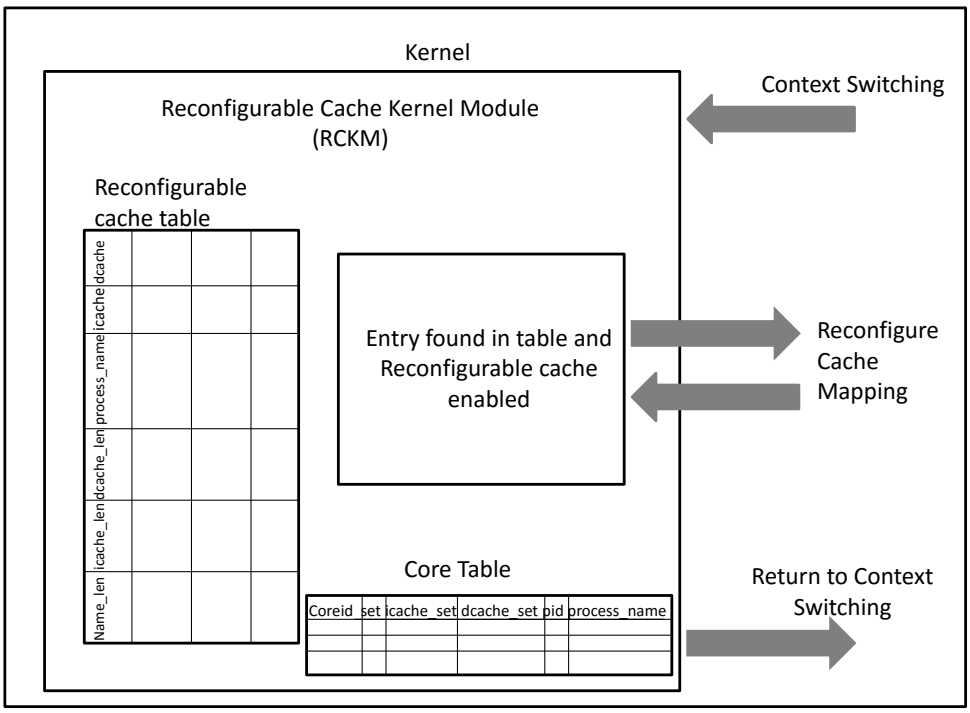

Figure 4.2: Reconfigurable cache kerenl module block diagram

The RCKM uses the netlink sockets to communicate with the RCUC. The netlink socket is preferred as the complete optimised cache data are sent by the RCUC to the RCKM. The RCKM stores the optimised cache data in a linked list table. This table is called as Reconfigurable cache table. A sample reconfigurable cache table is shown in Table 4.1. Each entry in the table has six fields, they are the `name_len, icache_len, dcache_ len, process_name, icache` and `dcache`.
`name_len` - this field contains the string length of the application or the process name.(integer)
`icache_len` - this field contains the length of the data of the evolved instruction cache mapping.(integer)
`dcache_len`- this field contains the length of the data of the evolved data cache mapping.(integer)
`process_name` - this field contains the name of the application or the process.(string)
`icache` - this field contains the evolved instruction cache mapping data in an array.(array)
`dcache` - this field contains the evolved data cache mapping data in an array.(array)

Table 4.1: Reconfigurable cache table with imaginary values

name_len	icache_len	dcache_len	process_name	icache	dcache
3	10	0	sha	0xAAAAAAAA 0xAAAAAABA 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAA3 0xCCCCCCCC 0xAAAACCCC 0xAAAAAAA3 0xBBBBBBBB 0xAAAAAABB	
5	15	15	cjpeg	0xAAAAAAAA 0xAAAAAAAA 0xBBBBBBB8 0xAACCAAAA 0xAAAAAAA9 0xCCCCCCCB 0xAAAAA8AA 0xBBBBBBB8 0xAACCAAAA 0xAAAAAAAA 0xCCCCCCCB 0xAAAAA8AA 0xAAAAAAA9 0xCCCCCCCB 0xAAAAAAAA	0xAAAAAABB 0xAAAAAA88 0xCCAAAAAA 0xBBAAAAAA 0xAAAAAA55 0xAAAAAACC 0xAAAAAABB 0xBBAAAAAA 0xBBAAAAAA 0xAAAAAABB 0xAAAAAACC 0xAAAAAABB 0xAAAAAA55 0xAAAAAACC 0xAAAAAAAA

The RCKM manipulates the reconfigurable cache table according to the netlink socket it receives from the user application. When the RCKM receives socket with *add* command, a new entry is added into the reconfigurable cache table if the application name is not already present in the table. When the RCKM receives socket with *delete* a single entry command, it searches the process name specified in the socket to be deleted and when found it deletes the entry. When the RCKM receives socket with *erase* the table, the module deletes all the entry in the reconfigurable cache table. When the RCKM receives socket with *print* command, the module prints all the process names currently in the reconfigurable cache table.

Apart from the above socket commands to manipulate the reconfigurable cache support by the operating system, additionally two other netlink socket commands are also supported by the RCKM. They are to *enable* and *disable* the operating system support for reconfigurable cache mapping. This value is stored in a global variable.

In addition to the reconfigurable cache table, this module also maintains another table called the core table. A sample core table is shown below in Table 4.2. Each entry in the core table has five fields, they are set, `icache_ set, dcache_set, pid` and `process_name`. The table is set of array the array index of each entry is the processor's core id. Hence the array index of 0, contains the entry for core id 0 and so on.
`set` - this field indicates whether the instruction cache or data cache for this core is reconfigured.(integer)
`icache_set` - this field indicates whether the instruction cache for this core is reconfigured.(integer)
`dcache_set` - this field indicates whether the data cache for this core is reconfigured.(integer)
`pid` - the process id of the last process that was executed in this core. (integer)
`process_name`- the process name of the last process that was executed in this core.(string)

Table 4.2: Core table - to track the previous process id and process name in each processor core with sample values

coreid	set	icache_set	dcache_set	pid	process_name
0	1	0	1	98	queens
1	0	0	0	0	
2	1	1	1	104	sha
3	1	0	1	117	queens

The core table is used by this kernel module to decide whether the cache mapping has to be reconfigured or not for the given core id.

When the operating system calls this kernel module with input data of a process name, the module flow is as shown in Figure 4.3:

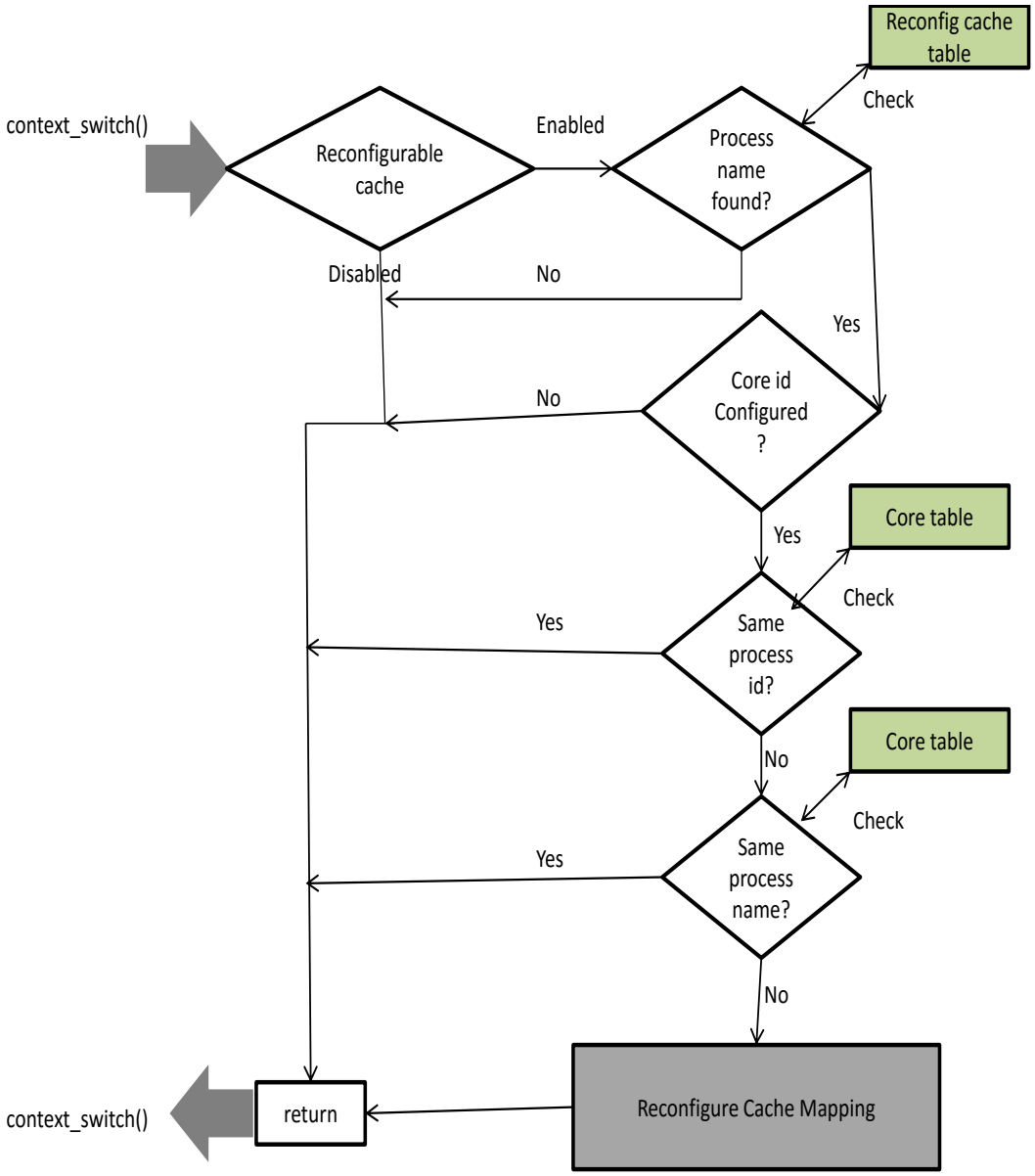

Figure 4.3: Reconfigurable cache kernel module flow diagram

The reconfigure cache mapping block in the flow diagram in Figure 4.3 is further expanded in Figure 4.4.

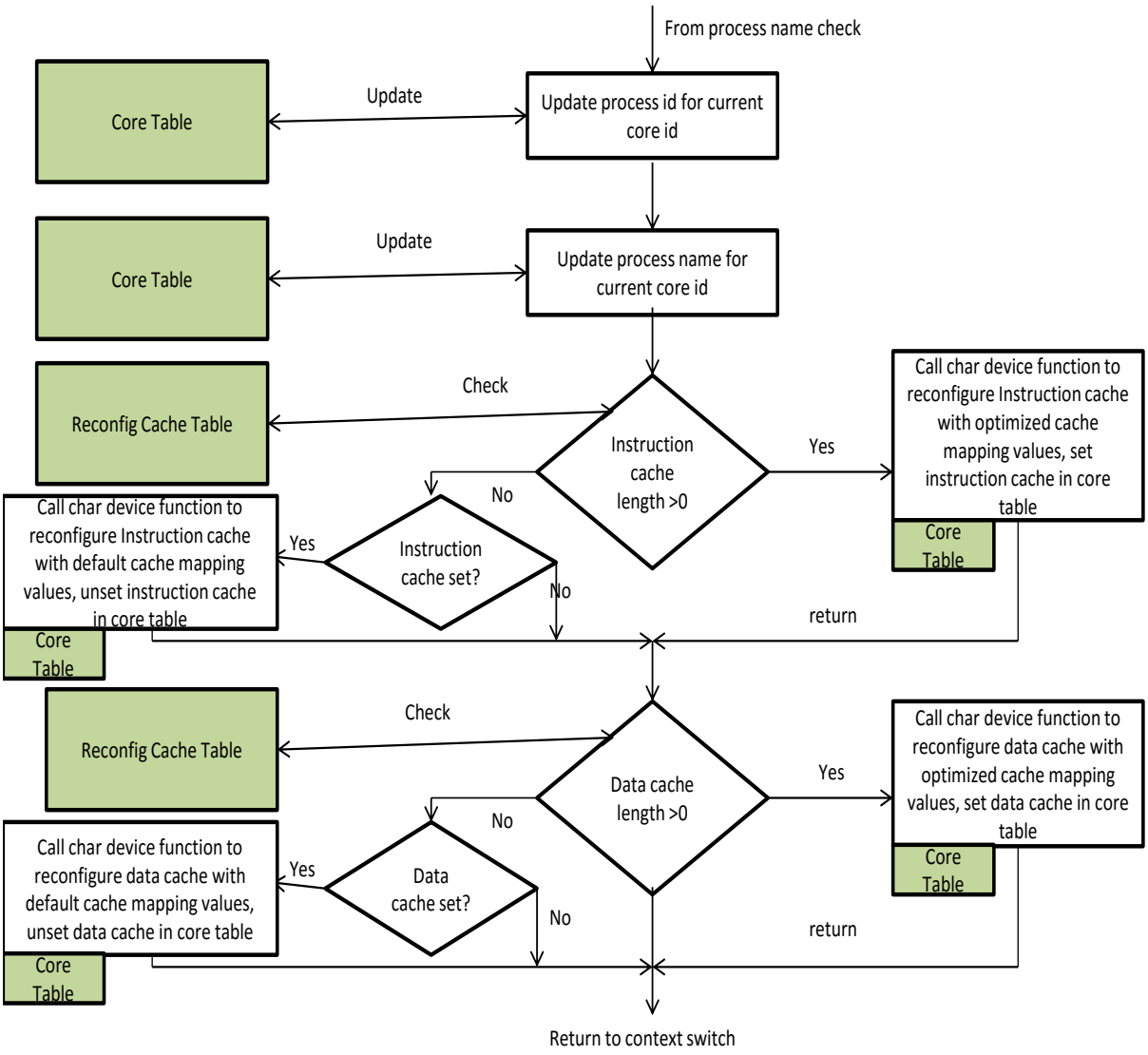

Figure 4.4: Expanded Reconfigure cache mapping block from Figure 4.3

The RCKM receives the next process task structure. From the task structure, RCKM retrieves the process name and the core id. If the retrieved process name is present in the reconfigurable cache table, then the RCKM checks the core table whether that core is already reconfigured for the retrieved process name. If not then it reconfigures the optimized cache mapping for the retreived process name. If only one of the instruction or data cache has a reconfigurable cache value in the reconfigurable cache table, then only that cache whose value is present is reconfigured with optimized cache mapping values and the other cache is reconfigured to

be in default cache mapping if it is already reconfigured to some other optimized cache. The way the RCKM interacts with other Linux kernel modules and the RCUC is explained in detail under section 4.6.

4.2.1 Protocols to Reconfigure Cache Mapping Function

The following protocols are followed during reconfiguring cache mapping function:
1. If there is an entry available in the reconfigurable cache table only then we consisder reconfiguring cache mapping.
2. The processes for which there is no entry in the reconfigurable cache table, uses the cache mapping function currently present in that core it could be the default cache mapping function or any reconfigured cache mapping function.
3. If the process name or process id is same as the last executed process in that core, then nothing is done. This is to avoid unnecessary reconfiguring the same cache mapping function already present in the core. Aslo both process id and process name is validated for either one is true since process id comparison is faster. So if the process id is same nothing is done. But if it is different we check for the process name if the process name is same nothing done. This method is followed to improve the performance.

4.3 Reconfigurable Cache User Controller (RCUC)

This section explains the design of the user part, that is the RCUC. The RCUC acts as a bridge between the users and the RCKM. The below Figure 4.5 shows the overview of the user part. The RCUC has two parts:
1. User input validation part, which is responsible to check the user input command and the input cache mapping configuration file.
2. Netlink communication part, which is responsible for the netlink communication between RCUC and the RCKM.

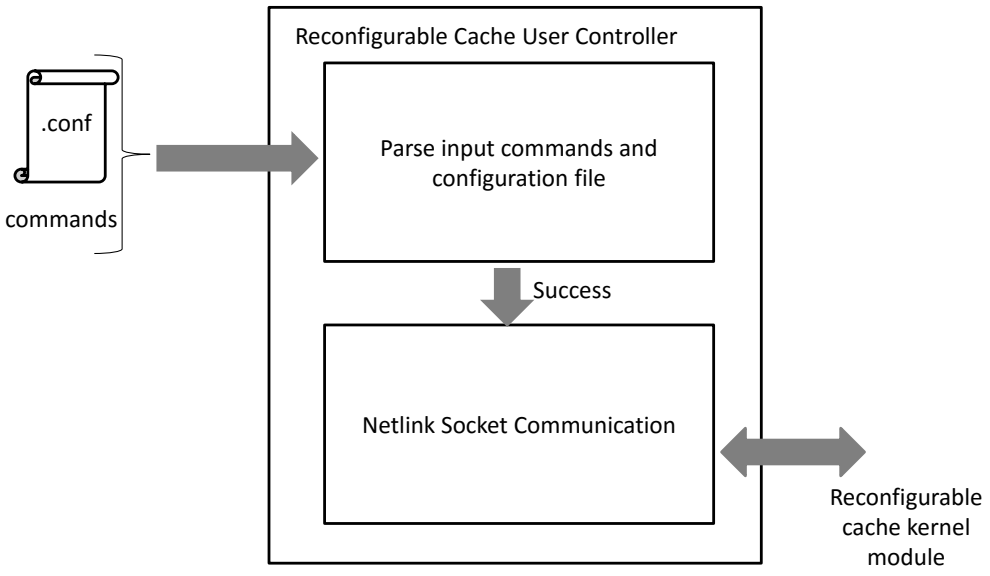

Figure 4.5: Reconfigurable cache user controller block diagram

The RCUC has two sets of communications:
1. Between the user and the RCUC.
2. Between the RCUC and the RCKM.

The user communicates with the RCUC through commands and also optionally with the configuration file. The RCUC communicates with the RCKM using the netlink sockets. The user input commands are validated by the RCUC if it is valid then a netlink socket is created which consists of the user executed command in kernel understandable format and the required data for the RCKM to execute the command. The RCUC then waits for a successful reply from the RCKM to confirm the communication.

4.4 User and Kernel Space Communication

The RCUC and the RCKM both needs to communicate with each other to exchange the optimized cache mapping function values. But the user application is in user space and the kernel module is in kernel space, the communication between these two spaces is possible using one of the default methods for communication provided by Linux. There are several methods like files, pipe, semaphore, etc,.

The method we choose for this thesis is sockets. We use netlink sockets to communicate between the user space and kernel space. In this method, we use network sockets to send and receive data between the user and kernel space. The reason for using the netlink method is:[13] [12]
1. Well defined protocols and simple socket Application Program Interface, (APIs).
2. Asynchronous communication using queues, with user defined port values.
3. Both the kernel and the user space can initiate the communication.
4. No Kernel modules dependencies.
5. Easy to transfer large amount of data.

4.4.1 Netlink

The following simple socket APIs are used in our thesis:
User space:
1. `socket` - used to create a socket. A socket is a communication endpoint which listens to a port using port number.
2. `bind` - to bind address to the socket created.
3. `sendmsg` - sends the data using the binded socket.
4. `recvmsg` - receives the data using the binded socket.
5. `close` - closes the descriptor of the socket.
Kernel space:
1. `netlink_kernel_create` - creates the kernel netlink socket with a callback function determined to process the received messages, when the kernel module is initialized.
2. `netlink_kernel_release` - releases the socket upon the kernel module exit.
3. `nlmsg_put` - puts the netlink data into the kernel socket buffer.
4. `nlmsg_unicast` - unicasting the socket to the user application listening to the specified port.
[13] [12]
The flow of netlink communication in the thesis is shown in Figure 4.6.

4.5 Linux Kernel Modifications

The only modification to the Linux kernel is during the context switching. The `context_switch` function in the Linux kernel file `sched.c` calls the RCKM as the first step. When the function call returns back to `context_switch` from RCKM the cache mapping function is already configured if applicable.

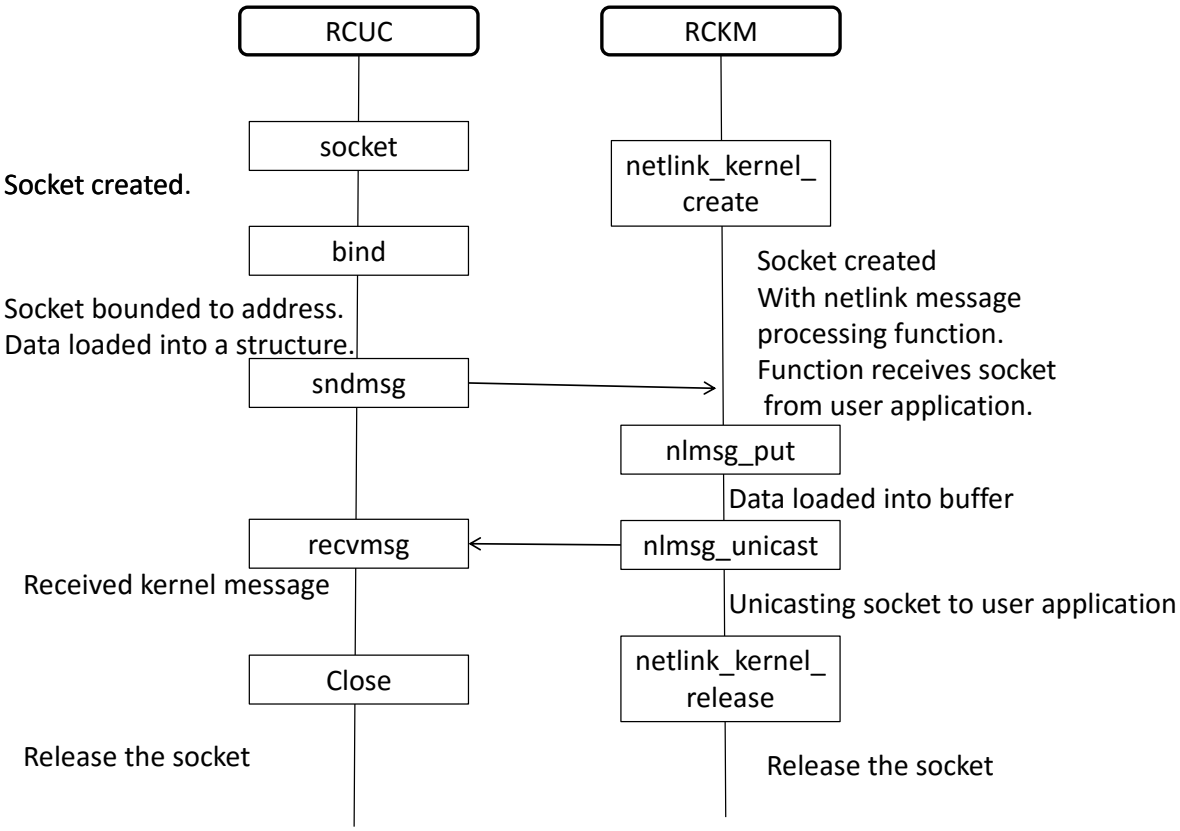

Figure 4.6: Netlink communication flow in this thesis

4.6 System Integration

As shown in the overview of the system in Figure 4.1 as a block diagram. The RCKM has three set of interactions:
1. Interaction with the RCUC. This interaction is carried out through netlink sockets. Both the RCUC and the RCKM has their own netlink communication function which handles the interaction.
2. Interaction with other kernel module. This interaction is carried out through a function call. The scheduling module is the only Linux kernel module with interacts with the RCKM. The communication is always initiated by the scheduling module. This function call from the scheduling module is called from inside the context switching function. In the context switching function, the first step is to call the function to reconfigure the cache mapping with the task structure of the next process as the input.
3. Interaction with the character device. This interaction is also carried out through a function call. The function call is always initiated by the reconfigurable cache module. The reconfigurable cache module call the character device to reconfigure the cache mapping with the cache mapping data, core id and which cache mapping to reconfigure (instruction cache or data cache) as input to the function.

A character (char) device is a device driver which is accessed as a stream of bytes and the data can only be accessed sequentially. The char devices are accessed by file system nodes.[9] A character device driver is created to reconfigure the cache mapping. This driver is used to handle the hardware module for the board ML605 Xilinx. It receives the function call input from the RCKM and manipulate the cache mapping. The cache mapping register is first flushed and then the new cache mapping values are written to the registers.

4.7 System Flow

This section explain the scenario of context switching that happens inside the operating system, with the reconfigurable cache mapping supported.
The Figure 4.7 shows an overall simplified system flow.

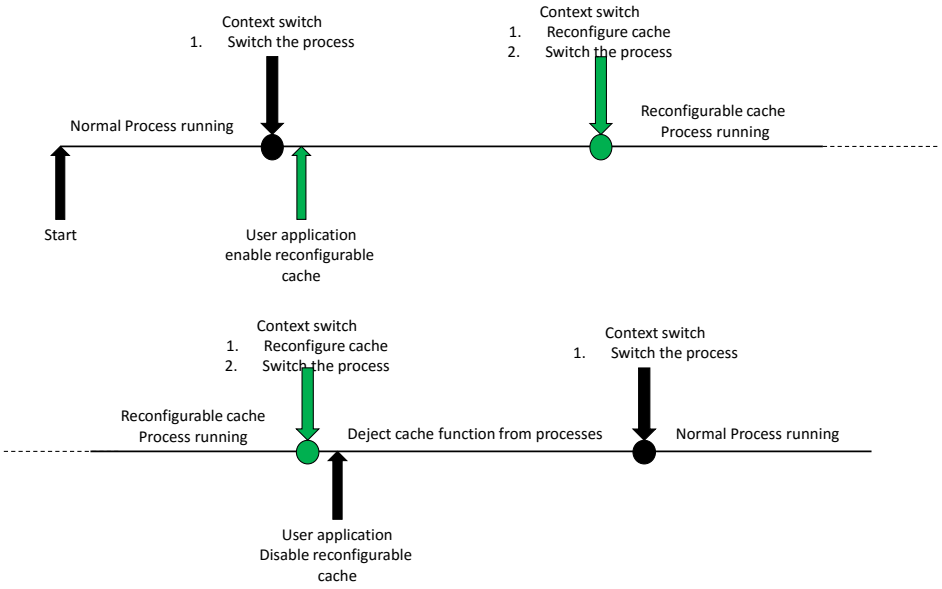

Figure 4.7: Simplified flow of the overall system

Initially the system is normal, support for reconfigurable cache is disabled by default. Now when the context switching takes place, no change in the operating system behavior. We use the RCUC to enable the reconfigurable cache mapping support. After enabling the reconfigurable cache mapping, when the context switching takes place, first reconfiguring the cache mapping takes place and then the normal context switching takes place. This reconfiguring the cache mapping before the context switching is always done when the reconfigurable cache mapping support is enabled. We again use the RCUC to disable the reconfigurable cache mapping support, all the reconfigured cache mapping are configured back to default cache mapping. Now when the context switching takes place, no reconfiguring of cache mapping takes place, just the normal context switching process.

4.7.1 RCUC and RCKM

As previously mentioned in Section4.2, the RCUC and the RCKM communicates using netlink sockets with different commands supported. Below we see all the commands supported in this specific netlink communication. The communication is always initiated by the user application for any netlink communication between the RCUC and the RCKM.
1. *Enable* and *disable* commands:

When the RCUC receives the enable command from the user, the RCUC creates the netlink socket with *enable* reconfigurable cache command. Upon receiving this command the RCKM checks whether the reconfigurable cache is enabled. If it is enable nothing is done. If it is not enabled, then the reconfigurable cache mapping enable value is set. The RCKM replies the netlink socket with the successful receiving of the netlink socket from the RCUC.
When the RCUC receives the disable command from the user, the RCUC creates the netlink socket with *disable* reconfigurable cache command. Upon receiving this command the RCKM checks whether the reconfigurable cache is disabled. If it is disable nothing is done. If it is not disabled, then the reconfigurable cache mapping disable value is set. The RCKM then searches the core table to see any core is reconfigured previously one by one. For all the cores whose cache mapping are reconfigured, they are reconfigured back to the default cache mapping and the core table for that core is updated as not reconfigured. The RCKM replies the netlink socket with the successful receiving of the netlink socket from the RCUC. Hence, the disable reconfigurable cache command not only stop the cache mapping reconfiguring but it also reverts back all the cores to the default cache mapping in case they are reconfigured.
The communication flow for the enable and disable command is shown in Figure 4.8.

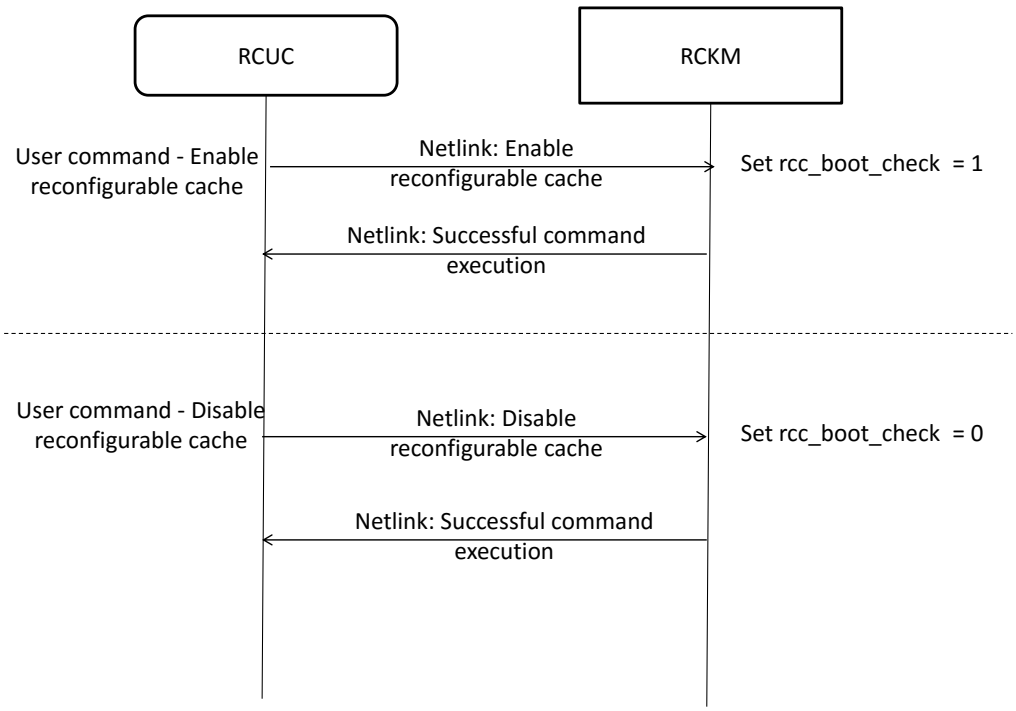

Figure 4.8: Communication between reconfigurable cache user controller and kernel module part 1

2. *Add* and *delete* commands:

When the RCUC receives the add command with a configuration file from the user, the RCUC validates the configuration file. Upon successful validation of the configuration file, the RCUC extracts the input data from the configuration file. The input data are process name, instruction cache or data cache or both and the cache mapping values. Then the RCUC creates the netlink socket with *add* command. Upon receiving this add command the RCKM checks whether the process name is already present in the reconfigurable cache table. If it is present, then an error message is sent to user application first delete the entry before adding. This error message is displayed to the user by the RCUC. If the process name is not present in the table, then a new entry is added to the reconfigurable cache mapping table with the input data from the netlink socket. And the RCKM replies the netlink socket with the successful receiving of the netlink socket from the RCUC.

When the RCUC receives the delete command with a process name from the user, the RCUC creates a netlink socket with *delete* command with the process name as the data. Upon receiving this delete command the RCKM checks whether the process name is present in the reconfigurable cache table. If it is present, then the entry is deleted. If the process name is not present in the table, then nothing is

done. The RCKM replies the netlink socket with the successful receiving of the netlink socket from the RCUC.
The communication flow for the add and delete command is shown in Figure 4.9.

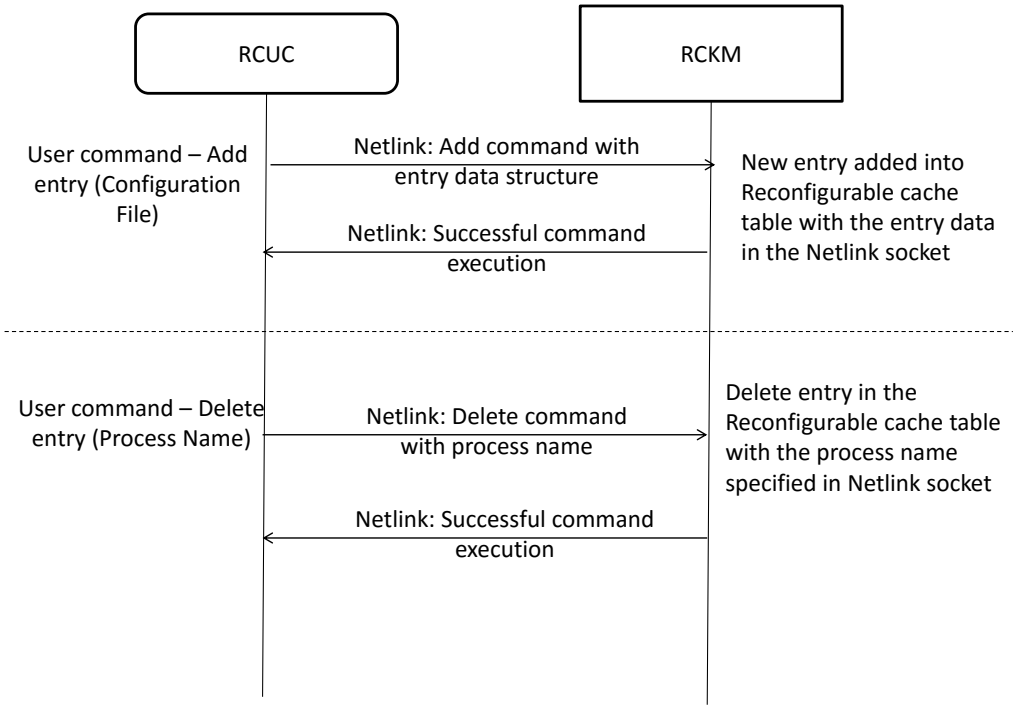

Figure 4.9: Communication between reconfigurable cache user controller and kernel module part 2

3. *Erase* and *print* commands:
When the RCUC receives the erase command from the user, the RCUC creates the netlink socket with *erase* command. Upon receiving this erase command the RCKM removes all the entry from the reconfigurable cache table. And the RCKM replies the netlink socket with the successful receiving of the netlink socket from the RCUC.
When the RCUC receives the print command from the user, the RCUC creates a netlink socket with *print* command. Upon receiving this print command the RCKM creates a netlink socket reply with data as all the process names present in the reconfigurable cache table. The RCUC then prints the process names to the user one by one.
The communication flow for the erase and print command is shown in Figure 4.10.

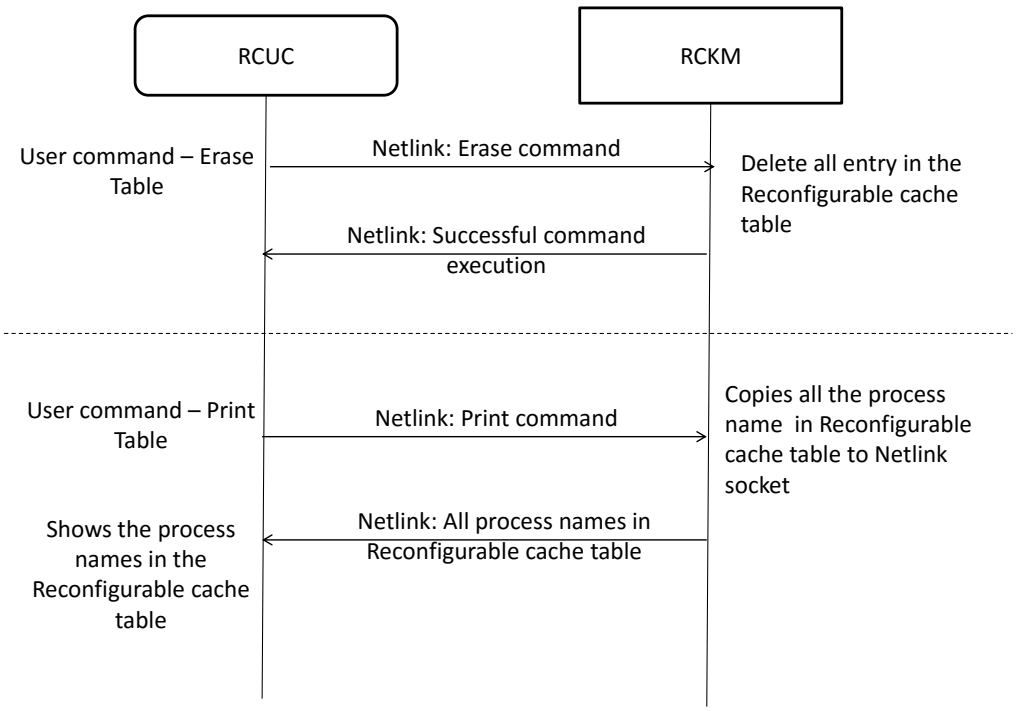

Figure 4.10: Communication between reconfigurable cache user controller and kernel module part 3

4.7.2 Context Switching and RCKM

As previously mentioned in Section2.5, when a new process is executed in the Linux operating system, the process is queued into a run queue of the corresponding core in which the process is going to be executed. Given the process is in ready state and is the next element in the queue, when the core completes its current execution, it gets the new process form the queue. Here the context switching takes places. The function context switch in the scheduling module is called. Before further executing the context switching, the context switching calls the RCKM to reconfigure the cache mapping for the new process if possible.
The reconfigurable gets the new process task structure from which it gets the core id value and the process name. The reconfigurable cache module reconfigures the cache mapping if the reconfigurable cache mapping is enabled and the process name is present in the reconfigurable cache table. The cache mapping is reconfigured only when the core is not already reconfigured for the same cache values the last time. In case if only one of the data or instruction cache needs to be reconfigured, then the cache for which there is no value, we reconfigure to default cache mapping function. That is, If both the instruction and the data cache in a core is reconfigured for a previous process. Now the next process needs we have

values for only the instruction cache, then we reconfigure the instruction cache to the values in the reconfigurable cache table and the data cache we reconfigure to the default cache mapping function the Linux operating system normally uses.
The communication flow between the context switching and the reconfigurable cache module is shown in Figure 4.11.

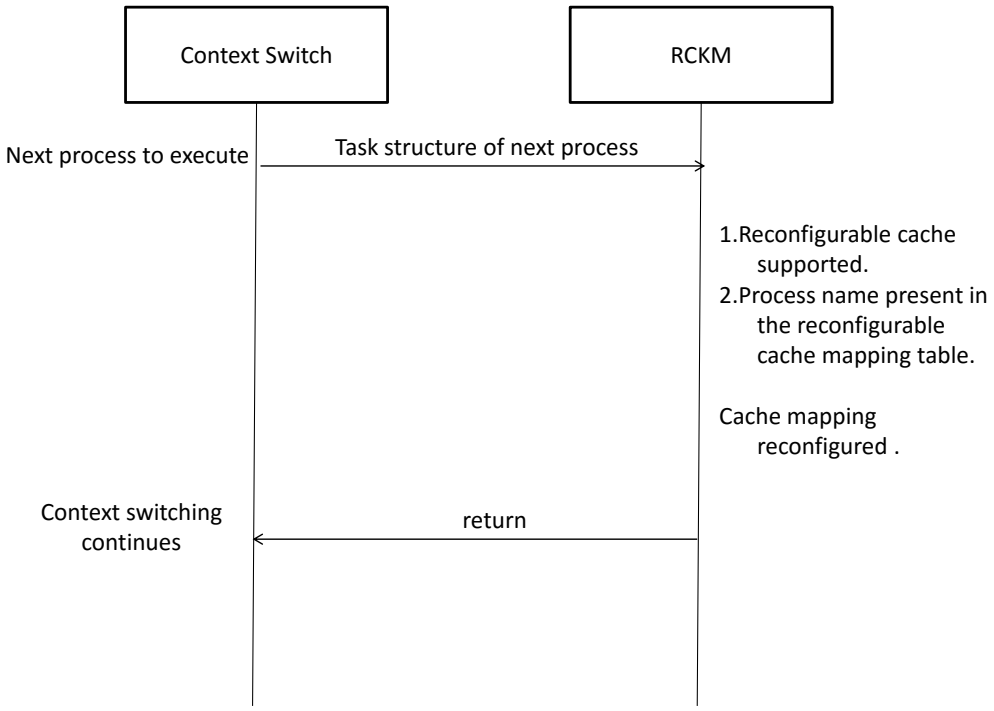

Figure 4.11: Communication between reconfigurable cache kernel module and context switching

4.7.3 Character Device and Reconfigurable Cache Mapping Kernel Module

As previously mentioned in Section4.6, when all the conditions are satisfied to reconfigure a cache mapping, then the reconfigurable cache mapping module calls the character device to reconfigure the cache mapping with the core id, type of cache and the cache values.
The character device then flushes the cache registers to remove the previous cache mapping and writes the new cache mapping received from the reconfigurable cache mapping module into the physical registers.
The communication flow between the reconfigurable cache module and the character device is shown in Figure 4.12.

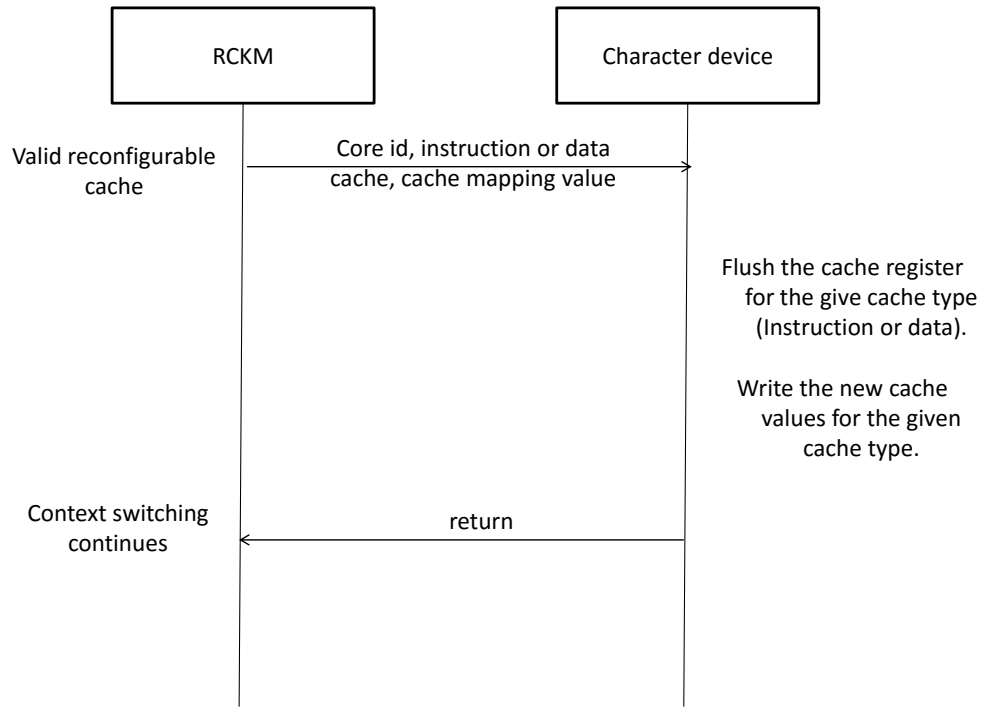

Figure 4.12: Communication between character device and reconfigurable cache kernel module

5 Evaluation

This section describes about how the system is evaluated, the metrics that are considered to measure the performance of the systems, the tools used for the evaluation, how these tools are used to measure those metrics, scenarios of evaluation and the results.

5.1 Tools for Evaluation

There are two Linux applications used for evaluation. They are :
1. Time application.
2. Perf.

5.1.1 Time

Time is a simple command to give resource usage. [5].
This time application is used for measuring the time taken by the evaluated applications to execute. The execution time is measured in three different variations as user time, system time and execution time. The user time is the time spent on the user mode code during the application execution. The system time is the time spent on the kernel during the application execution. The execution time is combining the user time and the system time, which is the total time spent for the application to execute.
Using the system time we can find the increase in time by the operating system when the reconfigurable cache mapping support is enabled. Using the user time we can find the decrease in time by the process when the reconfigurable cache mapping support is enabled.
Initially we execute the time application without the operating system support for reconfigurable cache mapping, then we repeat the same with reconfigurable cache mapping support for each type of cache mapping.

5.1.2 Perf

Perf is a monitoring tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple

command line interface.[4].
Using perf application we measure the following Hardware events for each reconfigured cache mapping types and for the default cache mapping.
1. cpu-cycles or cycles.
2. Instructions.
3. L1-dcache-loads.
4. L1-dcache-load-misses.
5. L1-icache-loads.
6. L1-icache-load-misses.
7. L1-dcache-storess.
8. L1-dcache-store-misses.
9. cpu-migrations or migrations.

Using cycles and instructions values, we measure the instructions per cycles. The L1-dcache-loads and L1-icache-loads values are used to verify that during each measurement with different cache types almost similar number of cache loads happens. The L1-dcache-load-misses and L1-icache-load-misses values are used to measure the performance increase or decrease for different cache mapping types. We follow the below calculations from the obtained Hardware event values using perf.

$$InstructionCacheMissRate = \frac{InstructionCacheLoadMisses}{InstructionCacheLoads}$$

$$DataCacheStoreMissRate = \frac{DataCacheStoreMisses}{InstructionCacheStores}$$

$$DataCacheLoadMissRate = \frac{DataCacheLoadMisses}{DataCacheLoads}$$

$$DataCacheMissRate = DataCacheLoadMissRate + DataCacheStoreMissRate$$

$$InstructionandDataCacheMissRate = InstructionCacheMissRate + DataCacheMissRate$$

5.2 Applications Evaluated

The following are the applications which are used to evaluate the performance of the system:
1. Sha.
2. Patricia.
3. Cjpeg.

4. Bzip.

5.3 Hardware Platform

The following Table 5.1 shows the hardware setup being used for the evaluation:

Table 5.1: Hardware setup for evaluation

Clock Frequency	50 MHz
Integer Unit	Yes
Floating Point	Software
Instruction Cache	2-way associative
Data Cache	2-way associative

5.4 Cache Reconfigured

For the evaluation we applied the following cache reconfiguring scenarios:
1. Instruction Cache (IC) only.
2. Data Cache (DC) only.
3. Both Instruction and Data Cache (IDC).

5.5 Evaluation Scenarios

5.5.1 Single Core Evaluation

A single user application is executed at a time in a single core as shown in Figure 5.1. The core is isolated and only one application is executed at any given time. We isolate the core, so that no other process can run expect our test applications. We achieve this by setting CPU affinity to a specific core and we force our application to run in that specific isolated core. From this isolation we get a dedicated core for running our test application. Hence the values, we obtain in this scenario are by running an uninterrupted test application. This scenario is used to evaluate the user application performance difference. The metrics we evaluate in this scenario are:
1. Corresponding cache miss rate.
2. Time spent on user mode during execution - User Time.
3. Time spent on system mode during execution - System Time.
4. Total execution time - Execution Time.

These values are calculated for both with the default cache mapping and optimized cache mapping.

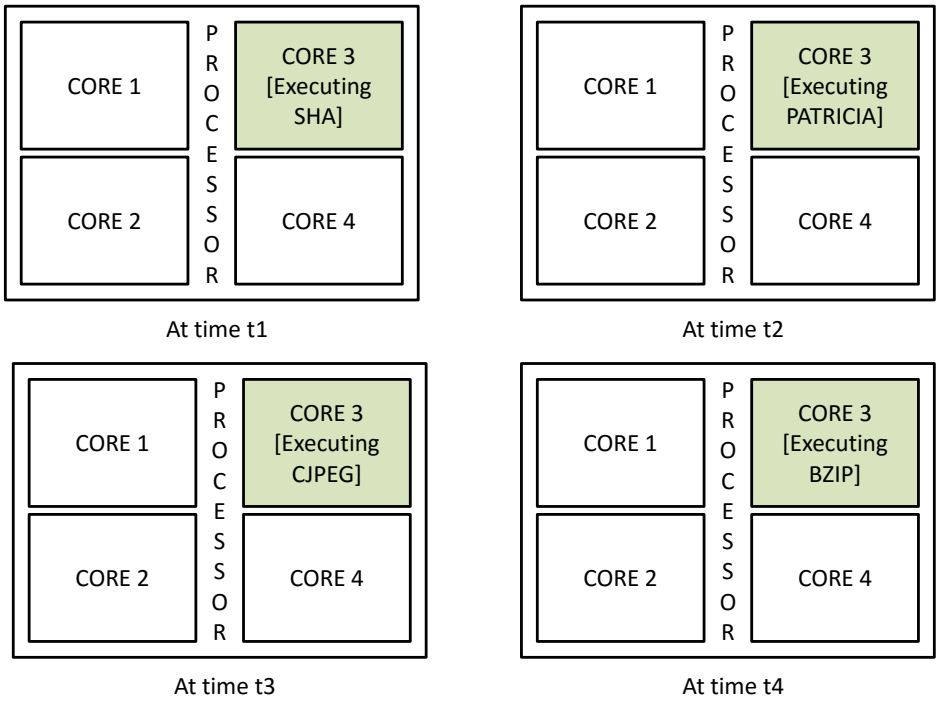

Figure 5.1: Only a single application is executed at any given time

5.5.2 Multi Core Evaluation with Different Application

More than one user application is executed at a time in more than one core as shown in Figure 5.2. The user applications used are different. We take two user application and fork them to run two time at the same time with in the isolated two cores. Here we force the isolated cores to be busy and force the system to make CPU migrations. For example, we execute two Patricia application and two Bzip application at the same time in cores 3 and core 4, noted core 3 and core 4 are both isolated cores. This scenario is used to evaluate the CPU migration effect on the performance. The metric we evaluate in this scenario is corresponding cache miss rate for both the default cache mapping and optimized cache mapping.

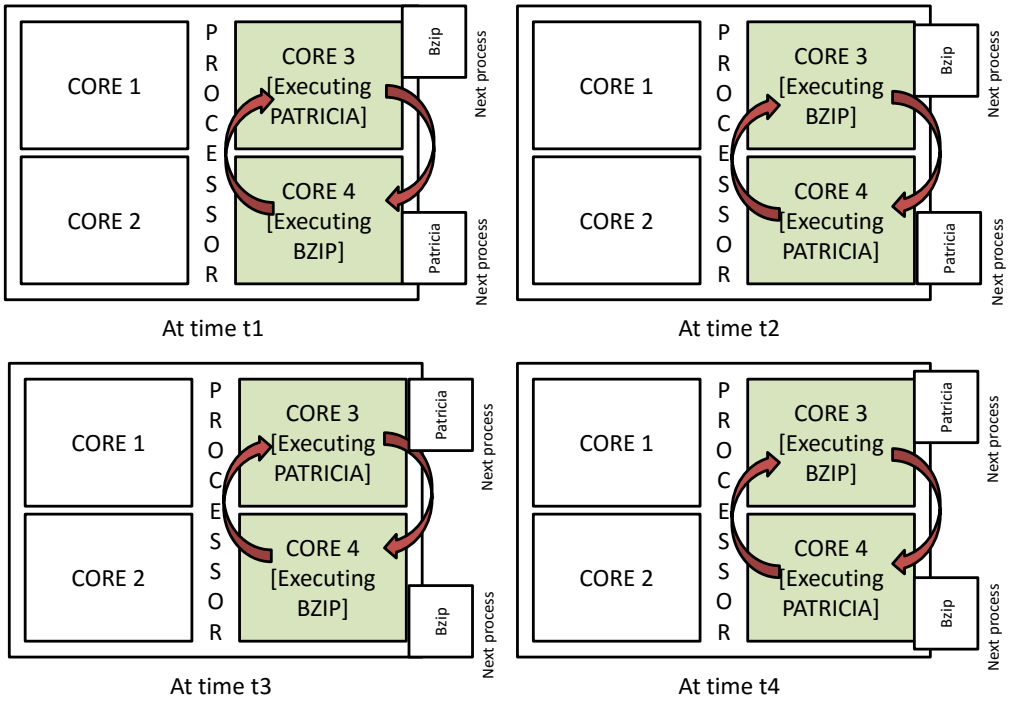

Figure 5.2: Two core executes two different applications

5.5.3 All Core with Different User Applications

All the cores, in our case four cores are made to run a user application. One user application is made to run in a core at a time as shown in Figure 5.3. No core is isolated, all the applications can run in any core. Here the cores execute the application along with other system processes and user processes. This scenario is used to evaluate the applications performance in a busy system. The metric we evaluate in this scenario is corresponding cache miss rate for both the default cache mapping and optimized cache mapping.

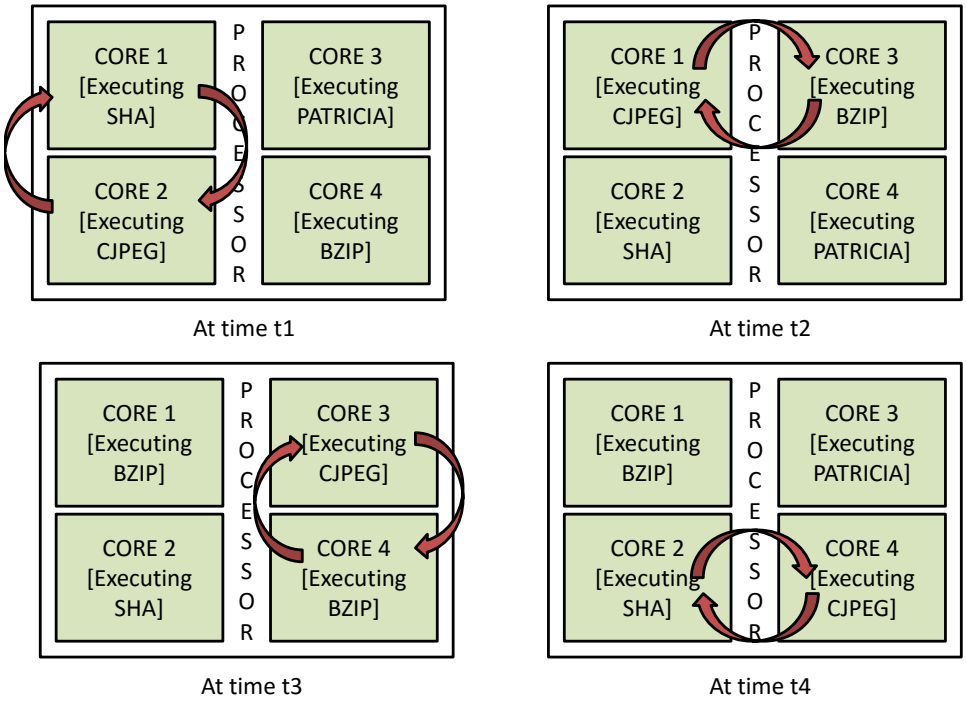

Figure 5.3: All core executes with different applications

5.6 Results

5.6.1 Single Core Evaluation Results

As discussed in section 5.1, we compared the values between a default cache mapping and optimized cache mapping supported by operating system during run time.

Results for SHA

Application Sha is evaluated under scenario as shown in Figure 5.1. We reconfigured IC during one set of evaluation and IDC during another set of evaluation. Figure 5.4 shows the cache miss rate comparison between default and reconfigured for Sha. From the results we observed that both the reconfigured IC and IDC cache miss rate is lesser than the default. The Sha application performed better during runtime in a reconfigurable cache mapping supported operating system.

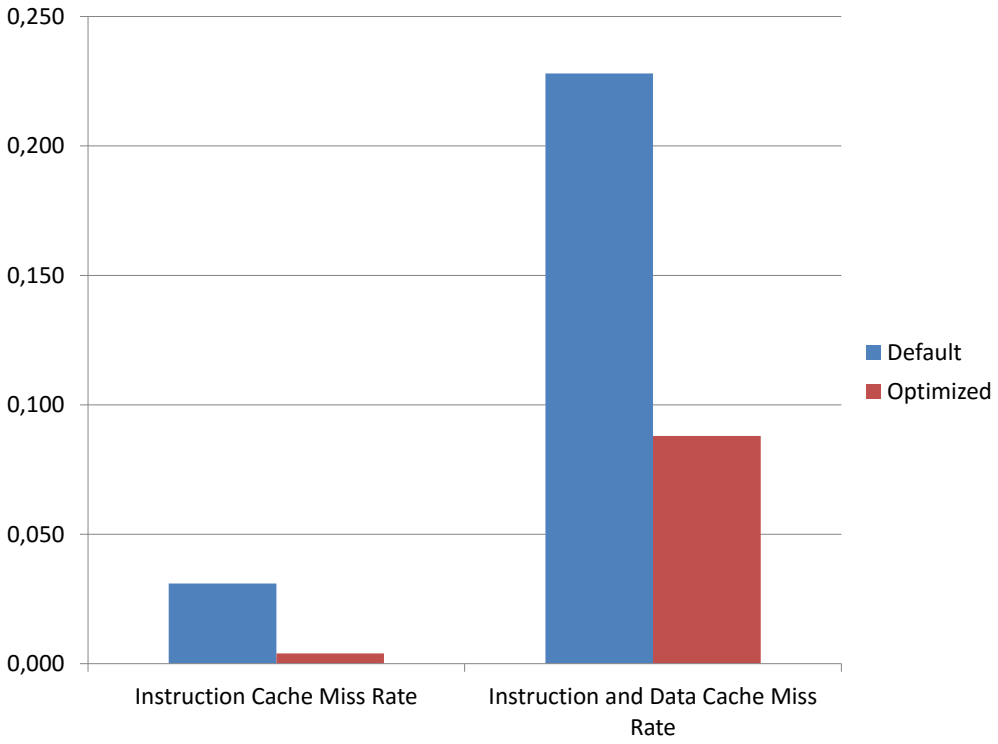

Figure 5.4: Instruction cache load misses for Sha application

Figure 5.5 shows the time taken for executing the Sha application. From the measurements we observe that the user time taken is more for default cache when compared with the reconfigured cache. But the system time is less for the default cache compared with the reconfigurable cache. Since the decrease in user time is huge and the increase in system time is very less during the reconfigured cache mapping, the overall execution time for both the reconfigured IC and IDC is less than the default cache.

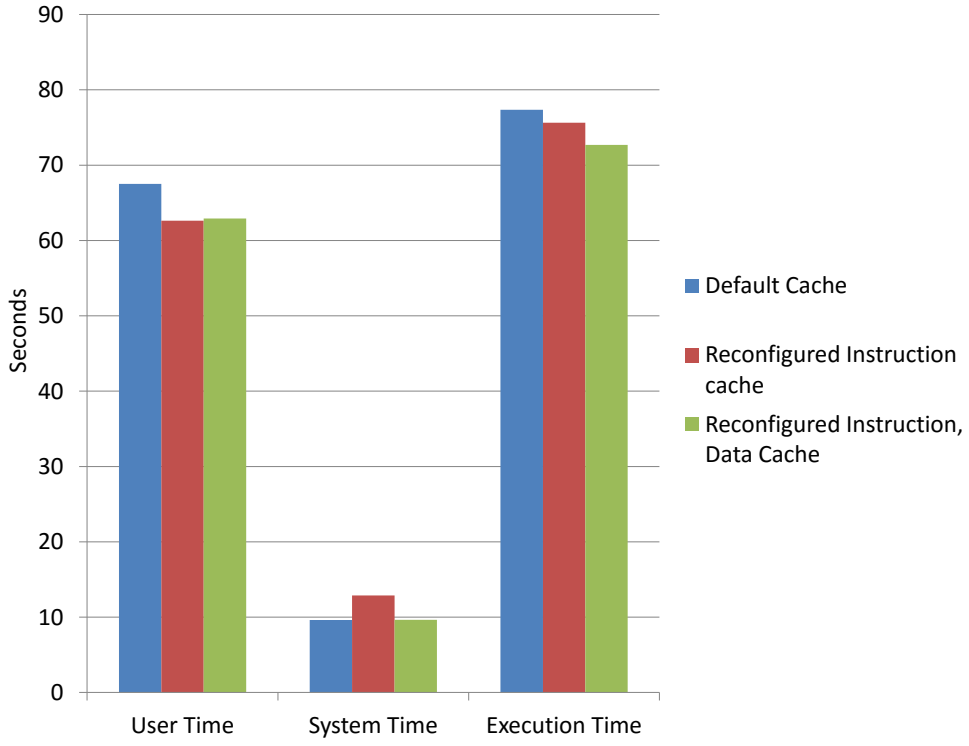

Figure 5.5: User time, system time and execution time for Sha application

From the above evaluation for Sha application, we observed that the application and the system performed better when we used reconfigurable cache mapping supported operating system. This observation is concluded due to lesser cache miss rate and lesser execution time.

Results for PATRICIA

Application Patricia is evaluated under scenario as shown in Figure 5.1. We reconfigured IC during our evaluation.
Figure 5.6 shows the cache miss rate comparison between default and reconfigured for Sha. From the results we observed that the reconfigured IC cache miss rate is lesser than the default. The Patricia application performed better during runtime in a reconfigurable cache mapping supported operating system.

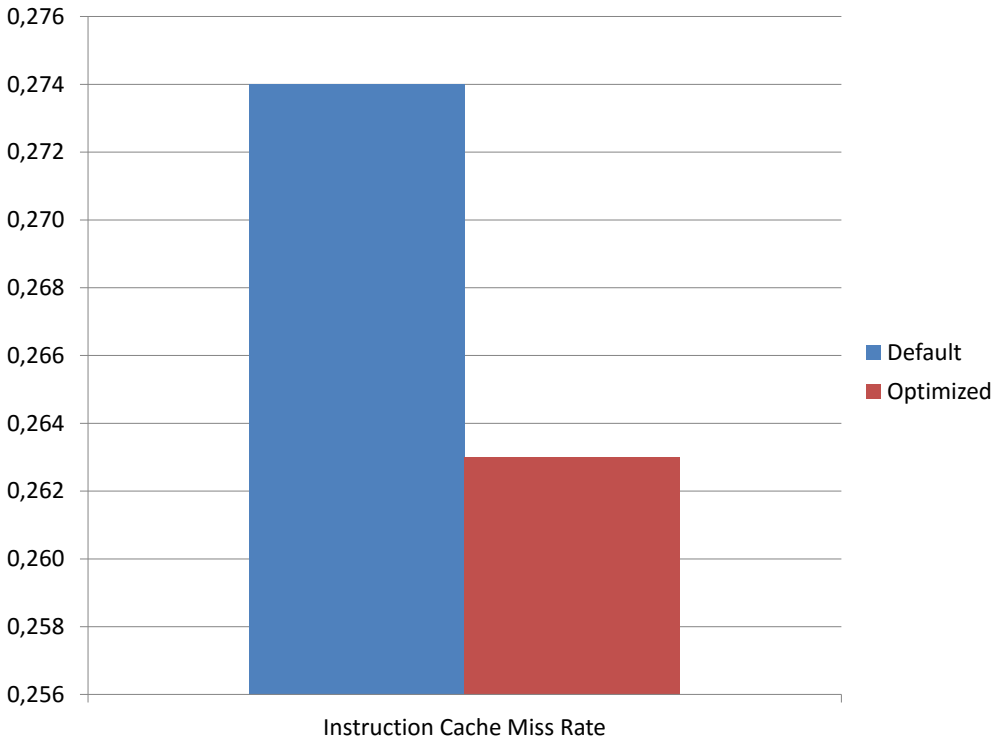

Figure 5.6: Instruction cache load misses for Patricia application

Figure 5.7 shows the time taken for executing the Patricia application. From the measurements we observe that the user time taken is more for default cache when compared with the reconfigured cache. But the system time is less for the default cache compared with the reconfigurable cache. Since the decrease in user time is huge and the increase in system time is very less during the reconfigured cache mapping, the overall execution time for the reconfigured IC is less than the default cache.

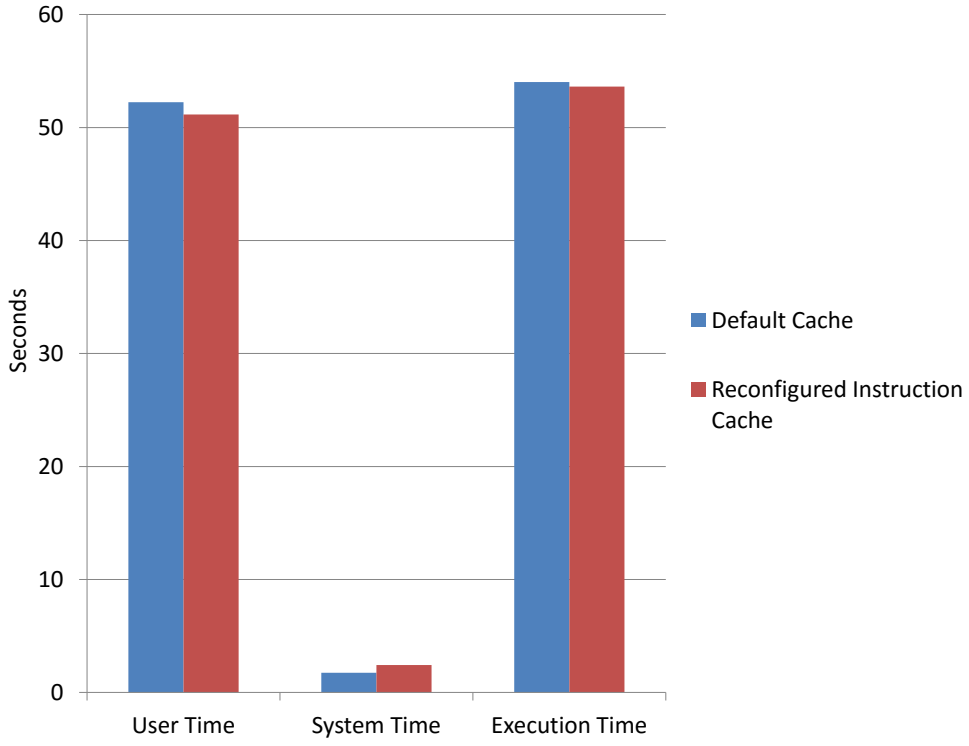

Figure 5.7: User time, system time and execution time for Patricia application

From the above evaluation for Patricia application, we observe that the application and the system performed better when we used reconfigurable cache mapping supported operating system. This observation is concluded due to lesser cache miss rate and lesser execution time.

Results for CJPEG

Application Cjpeg is evaluated under scenario as shown in Figure 5.1. We reconfigured DC during one set of evaluation and IDC during another set of evaluation. Figure 5.8 shows the cache miss rate comparison between default and reconfigured for Cjpeg. From the results we observed that both the reconfigured DC and IDC cache miss rate is lesser than the default. The Cjpeg application performed better during runtime in a reconfigurable cache mapping supported operating system.

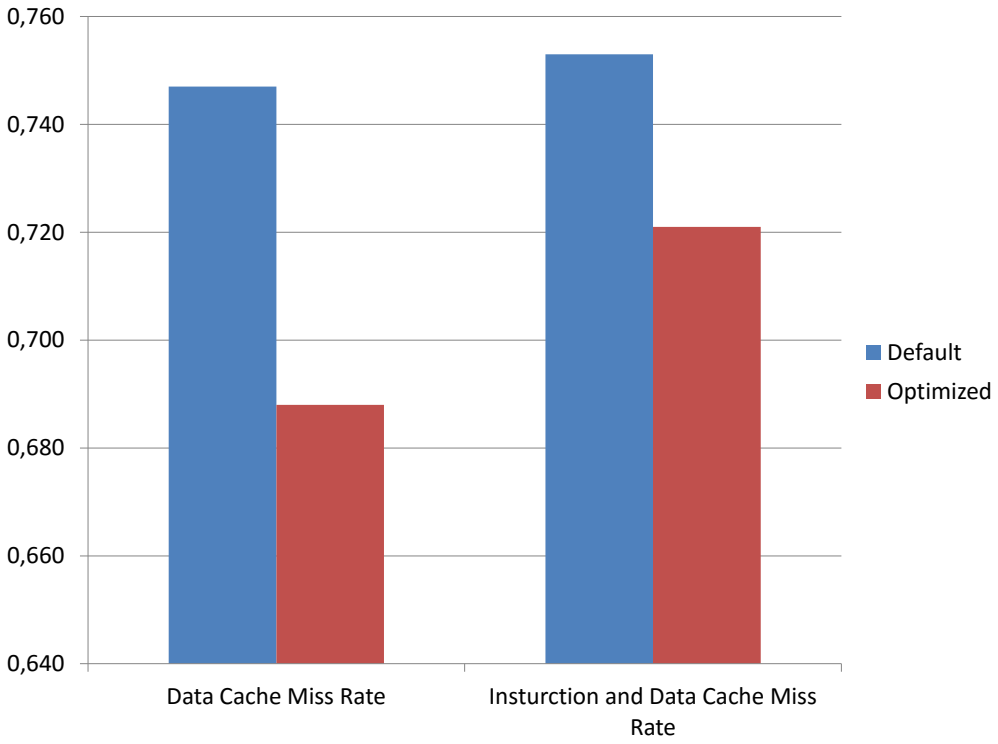

Figure 5.8: Instruction cache load misses for Cjpeg application

Figure 5.9 shows the time taken for executing the Cjpeg application. From the measurements we observe that the user time taken is more for default cache when compared with the reconfigured cache. But the system time is less for the default cache compared with the reconfigurable cache. Since the decrease in user time is huge and the increase in system time is very less during the reconfigured cache mapping, the overall execution time for both the reconfigured DC and IDC is less than the default cache.

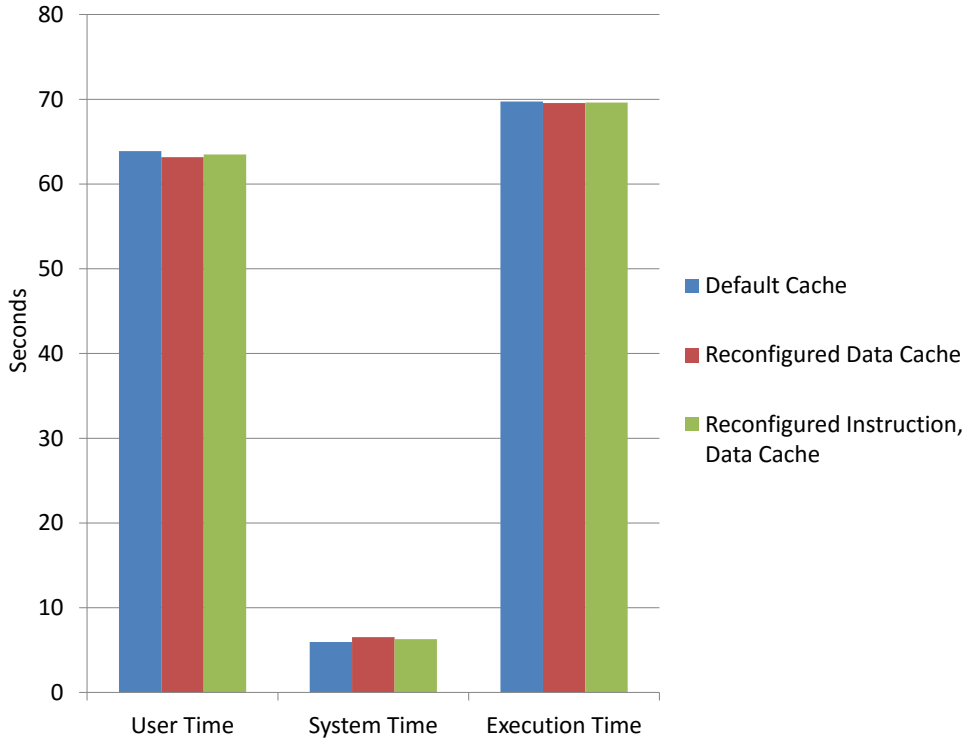

Figure 5.9: User time, system time and execution time for Cjpeg application

From the above evaluation for Cjpeg application, we observe that the application and the system performed better when we used reconfigurable cache mapping supported operating system. This observation is concluded due to lesser cache miss rate and lesser execution time.

Results for BZIP

Application Bzip is evaluated under scenario as shown in Figure 5.1. We reconfigured IC during one set of evaluation and IDC during another set of evaluation. Figure 5.10 shows the cache miss rate comparison between default and reconfigured for Bzip. From the results we observed that both the reconfigured IC and IDC cache miss rate is lesser than the default. The Bzip application performed better during runtime in a reconfigurable cache mapping supported operating system.

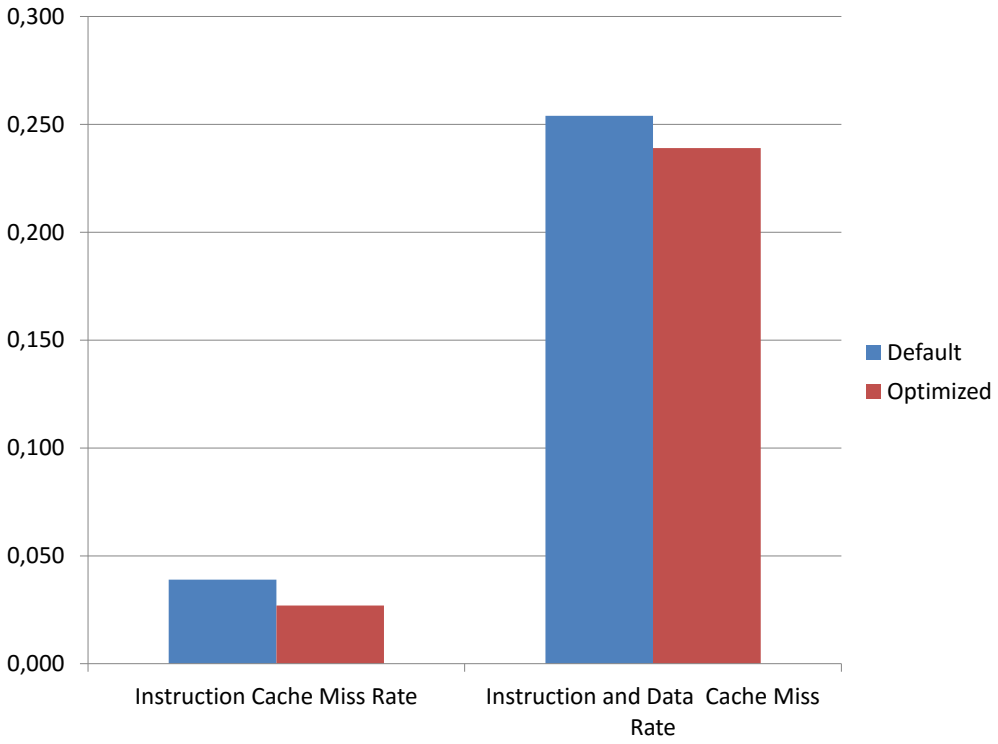

Figure 5.10: Instruction cache load misses for Bzip application

Figure 5.11 shows the time taken for executing the Bzip application. From the measurements we observe that the user time taken is more for default cache when compared with the reconfigured cache. But the system time is less for the default cache compared with the reconfigurable cache. Since the decrease in user time is huge and the increase in system time is very less during the reconfigured cache mapping, the overall execution time for both the reconfigured IC and IDC is less than the default cache.

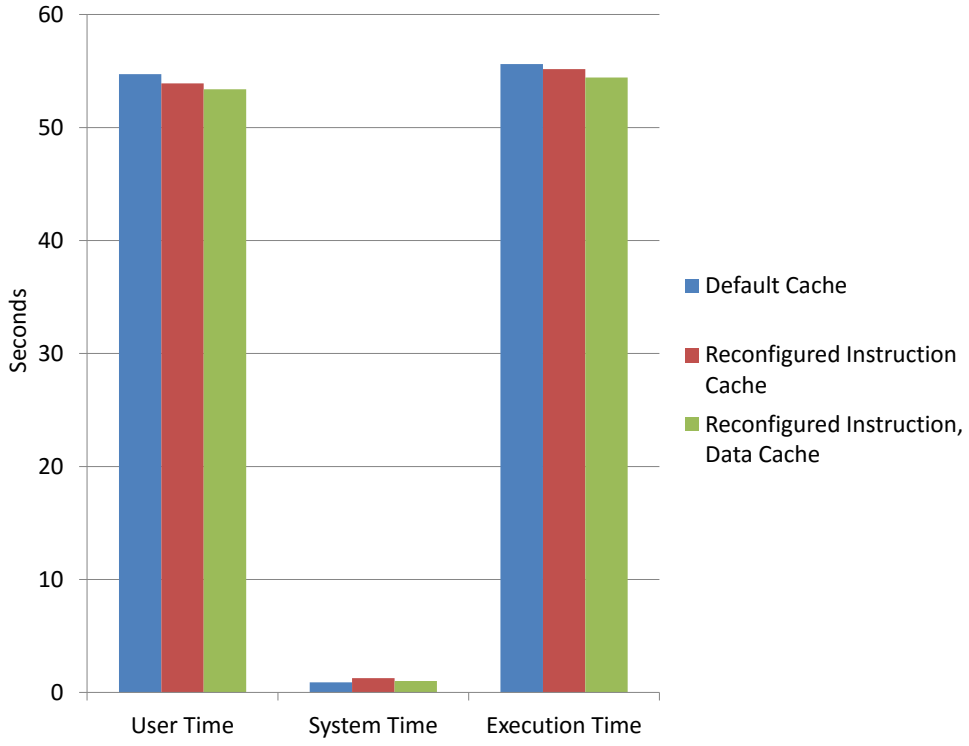

Figure 5.11: User time, system time and execution time for Bzip application

From the above evaluation for Bzip application, we observed that the application and the system performed better when we used reconfigurable cache mapping supported operating system. This observation is concluded due to lesser cache miss rate and lesser execution time.

5.6.2 Multi Core Evaluation Results

We compared the values between a default cache mapping and optimized cache mapping with operating system support.
Applications Patricia and Bzip are evaluated under scenario as shown in Figure 5.2. We reconfigured IC during the evaluation.
From the perf output we observed that CPU migration took place during this evaluation. Figure 5.12 shows the IC miss rate comparison between default and reconfigured for Patricia and Bzip for both the single core 5.1 and two core 5.2 scenario. From the results we observed that both the reconfigured IC miss rate is lesser than the default for two core evaluation. The isolated single core setup has less or the same reconfigured IC miss rate compared to the busy two core setup. The isolated single core setup less or the same default cache miss rate compared to the busy two core setup. Overall, the reconfigured IC miss rate in the busy two

core setup is much less than the default cache miss rate in an isolated core setup.

Results for Sha and Bzip

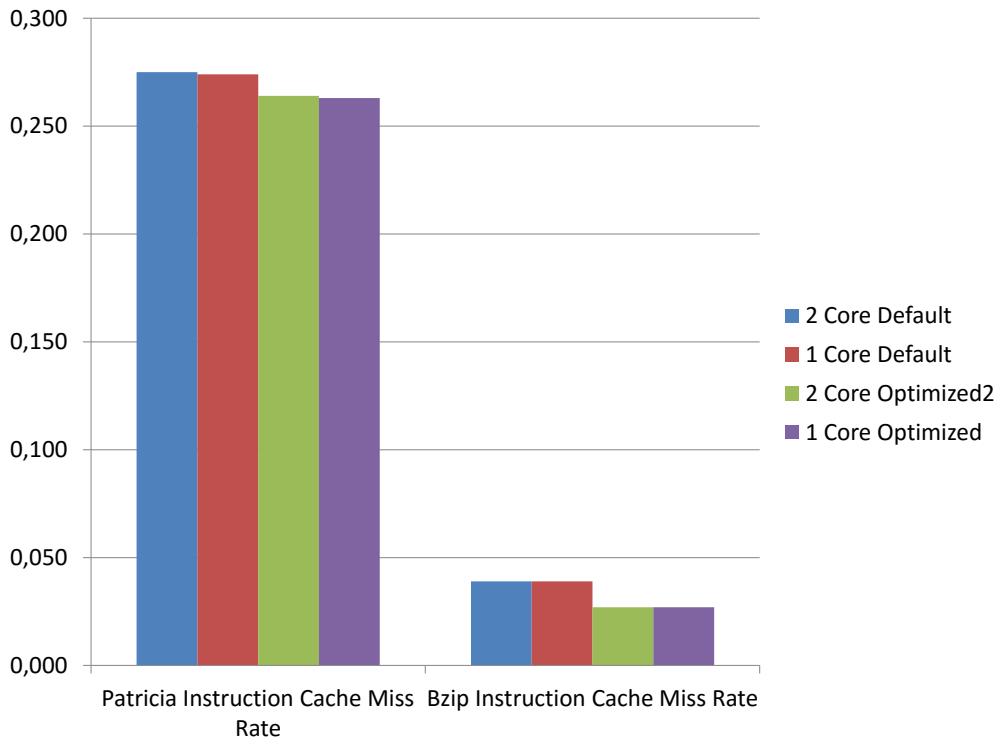

Figure 5.12: Patricia and Bzip executed in two core with CPU migration between the two cores

From the above evaluation we observe that the applications Patricia and Bzip are executed better in a reconfigurable cache mapping supported operating system with the busy two core setup. This observation is concluded due to lesser cache miss rate.

5.6.3 All Core Evaluation Results

The user applications are executed under scenario as shown in the Figure 5.3, at a given time all the core executing a process. During the evaluation we saw a few CPU migrations and a very high context switching confirming the busy cores setup. We observed the performance of the system for each application we reconfigured the cache mapping function and they are presented below.

All Core CJPEG

We reconfigured DC during the evaluation. Figure 5.13 shows the cache miss rate comparison between default and reconfigured for Cjpeg in the all core 5.3 and isolated single core 5.1 setup. From the results we observed that the reconfigured DC cache miss rate is lesser than the default in an all core set up. But the reconfigured DC miss rate in an all core setup is more than default cache in a single core setup. This observation confirms the system performance is also affected when many processes has to be executed. And sometimes the reconfiguring cache mapping alone cannot provide an overall system improvement, though it performs better than the default cache mapping under similar circumstances.

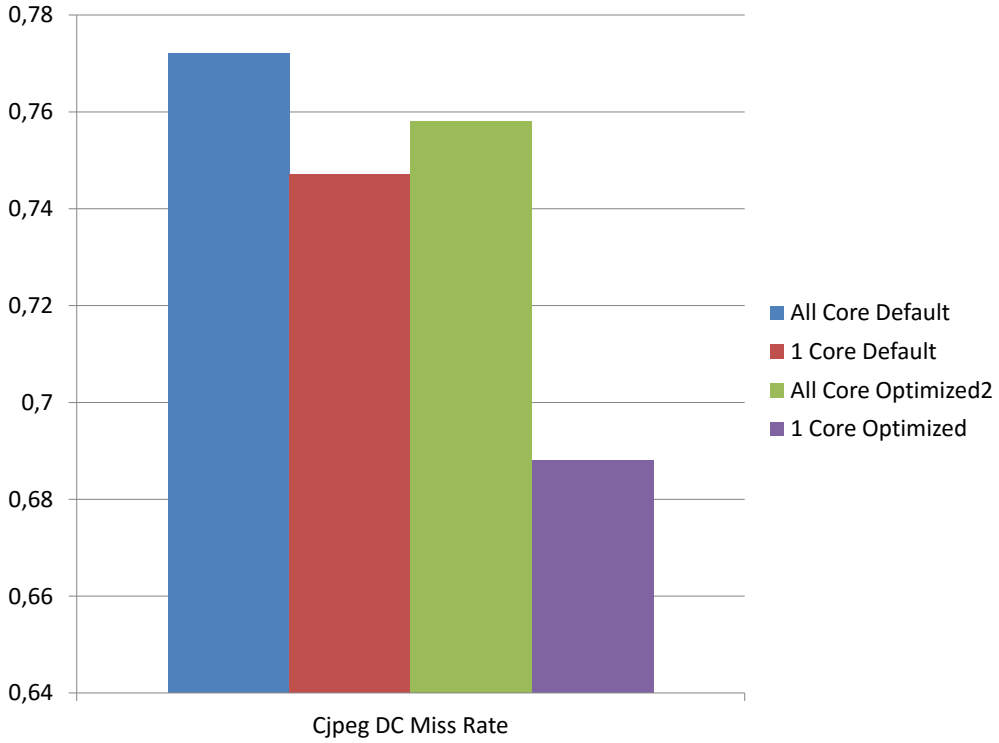

Figure 5.13: Cjpeg results during all core scenario

All Core BZIP

We reconfigured IC during the evaluation. Figure 5.14 shows the cache miss rate comparison between default and reconfigured for Bzip in the all core 5.3 and isolated single core 5.1 setup. From the results we observed that the reconfigured IC cache miss rate is lesser than the default in an all core set up. We also observed that the results of both the isolated core setup and all core setup are the same. It confirmed that the system might be able to provide the same performance some-

times for an application in a isolated core setup and all core busy setup.

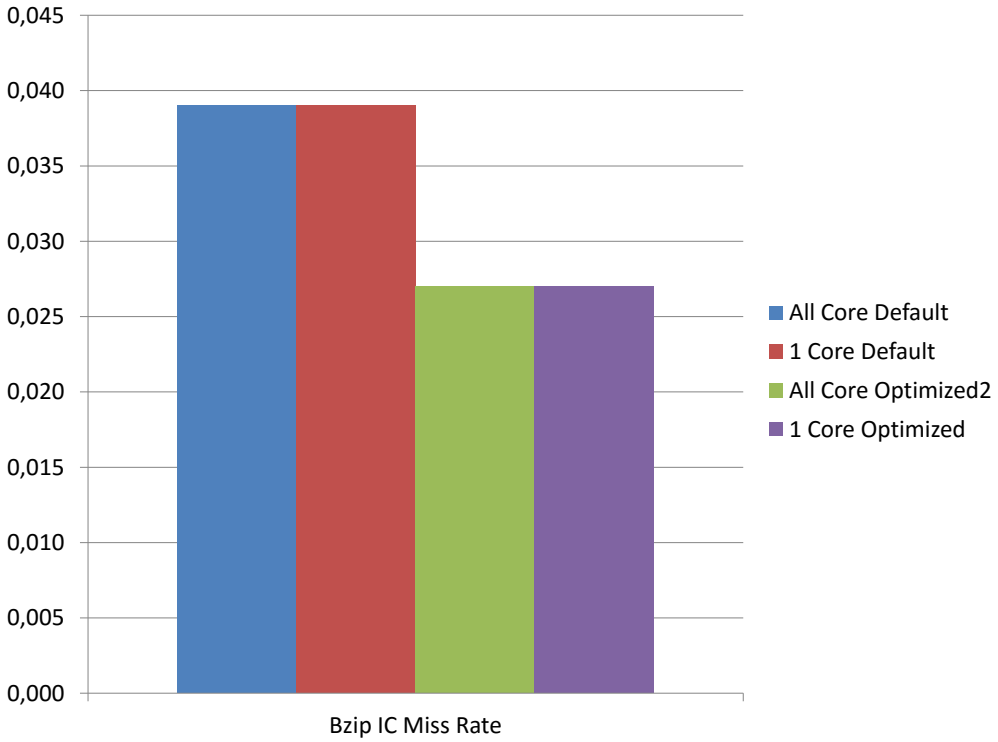

Figure 5.14: Bzip results during all core scenario

All Core PATRICIA

We reconfigured IC during the evaluation. Figure 5.15 shows the cache miss rate comparison between default and reconfigured for Patricia in the all core 5.3 and isolated single core 5.1 setup. From the results we observed that the reconfigured IC cache miss rate is lesser than the default in an all core set up and default isolated single core setup . But the reconfigured IC miss rate of all core setup is more than single core setup. This observation confirms the system performance can be affected by busy core. And sometimes the reconfiguring cache mapping can provide an overall system improvement.

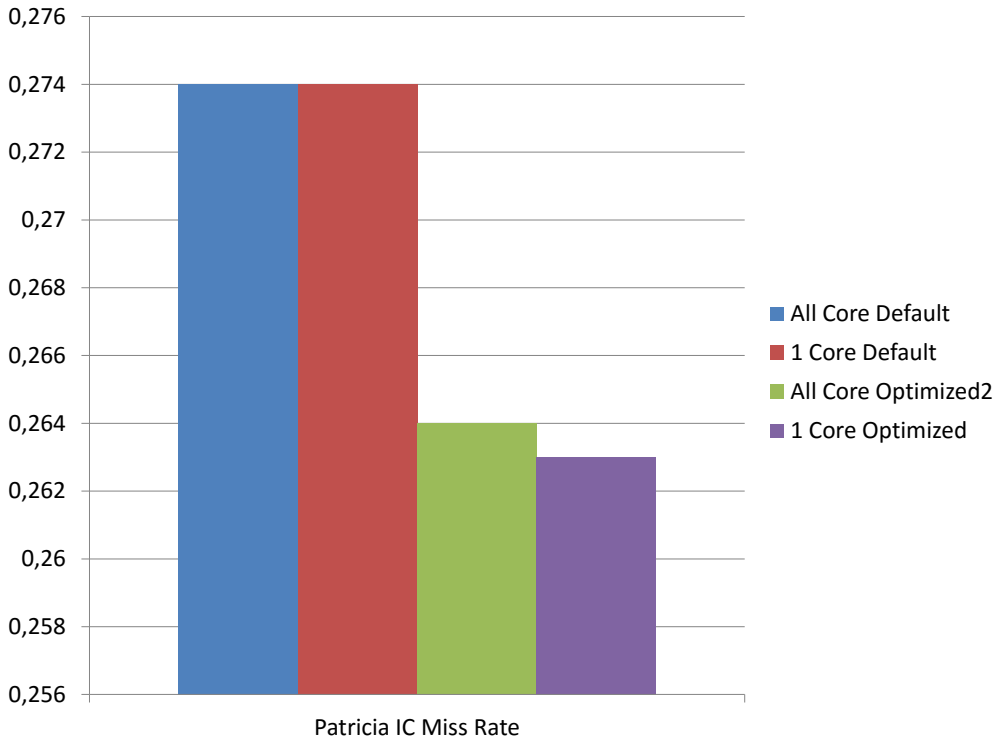

Figure 5.15: Patricia results during all core scenario

All Core SHA

We reconfigured IC during the evaluation. Figure 5.16 shows the cache miss rate comparison between default and reconfigured for Sha in the all core 5.3 and isolated single core 5.1 setup. From the results we observed that the reconfigured IC cache miss rate is lesser than the default in an all core setup, default isolated single core setup and reconfigured isolated single core setup. This observation which is rarely observed during the evaluation confirms the system performs sometime better for an application even during a busy core setup.

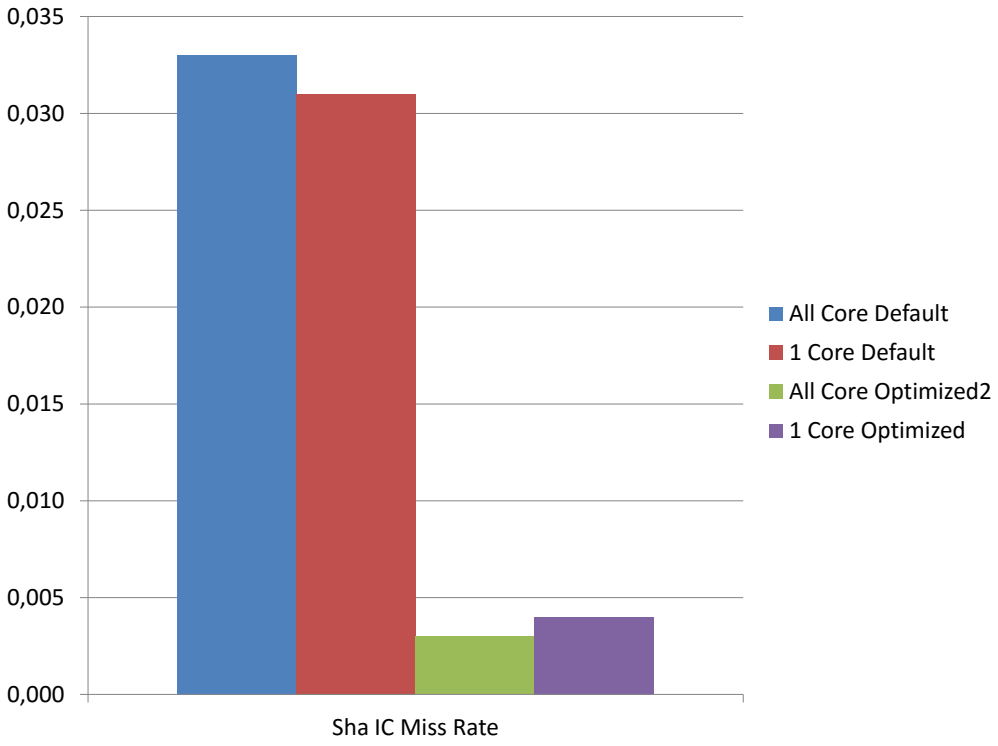

Figure 5.16: Sha results during all core scenario

From all the different setup evaluation we conclude that the system and application performance is better in run time with the reconfigurable cache supported operating system.

6 Conclusion

In this thesis, we designed and implemented a reconfigurable cache mapping kernel module which is supported by the Linux operating system during runtime. To control and configure the reconfigurable cache mapping kernel module we used the reconfigurable cache user application. The thesis was implementation in FPGA virtex 6 with Leon 3 platform as target architecture. This thesis serves as a related work of the another thesis Self-Optimizing Organic Cache [6]. Provided the optimized cache mapping for any user application, the thesis can reconfigure the cache mapping for that user application when it is being executed. The results were positive as the performance of the system and the user applications was much better with respect to hardware events and the execution time. The hardware event of cache load misses where decreased in real time for the corresponding reconfigured cache mappings when compared with the default cache mapping. We also noted that the execution time(user time + system time), of the user application also got decreased when we reconfigured with corresponding cache mapping function values. The system time is increased for a reconfigured cache mapping due to the additional computation and reconfiguring the cache mapping function. But there is a huge decrease in user time. Hence, the overall execution time is lower than a default cache mapping system.
The future works can address the following points:
1. Different values for instruction and data cache mappings. Till now for both the instruction and data cache has a single input value, needed to explore when we reconfigure both instruction and data cache at the same time with different values for each.
2. Using multi processor system. The current setup only supports the multicore system. The system need to be analyzed for a multiprocessor system. The challenge will be multi processors using multi core, expect a larger core table.
3. Exploring the possibility of optimising the operating system scheduling to essentially use the cache mapping reconfiguration. Helps to improve the system performance of busy systems and multi processor setup.

Appendix A

User Manual

NAME

reconf_cache - Controls the reconfigurable cache table

SYNOPSIS

reconf_cache [OPTION]... [FILE]...

DESCRIPTION

Controls and configures the information in the reconfigurable cache table inside the linux kernel.

-e, enables the operating system support for reconfigurable cache.

-d, disables the operating system support for reconfigurable cache.

-c [File Name], configures a new entry in reconfigurable cache table with the information inside the file.

-r [Application Name], removes an entry in the reconfigurable cache table which matches the application name.

-p, prints the application reconfigurable cache table.

-x, delete all the entries in the reconfigurable cache table.

-h, display all the command options.

EXAMPLES

To add an entry
reconf_cache -c [file name]

example: *reconf_cache -c queens.conf*

To remove an entry
reconf_cache -r [application name]
example: *reconf_cache -d queens*

To remove all the enrties in the table
reconf_cache -x

To print the whole table
reconf_cache -p

To enable operating system support for reconfigurable cache
reconf_cache -e

To disable operating system support for reconfigurable cache
reconf_cache -d

AUTHOR

Written by Vignesh Makeswaran.

REPORTING BUGS

Report bugs to ¡vigneshmakeswaran@gmail.com¿.

COPYRIGHT

SEE ALSO

The full documentation for the usage of this application with examples is described in the "Operating System support for Reconfigurable Cache" Master thesis report of Vignesh Makeswaran, Department of Computer Science, 2015, University of Paderborn.

Appendix B

Configuration File

B.1 Sample 1

```
#
# FileName reconfig_cache.conf
# Description This file represents Evo cache for corresponding applications for Linux,
# Reconfigurable Cache module
#

GROUP{
PROCESS = "queens"
ICACHE = "0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xCCCCCCCC"
}
```

B.2 Sample 2

```
#
# FileName reconfig_cache.conf
# Description This file represents Evo cache for corresponding applications for
Linux,
# Reconfigurable Cache module
#

GROUP{
PROCESS = "queens"
DCACHE = "0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xCCCCCCCC"
}
```

B.3 Sample 3

```
#
# FileName reconfig_cache.conf
# Description This file represents Evo cache for corresponding applications for Linux,
# Reconfigurable Cache module
#
GROUP{
PROCESS = "queens"
ICACHE = "0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xCCCCCCCC"
}
DCACHE = "0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xCCCCCCCC"
}
```

B.4 Sample 4

```
#
# FileName reconfig_cache.conf
# Description This file represents Evo cache for corresponding applications for
Linux,
# Reconfigurable Cache module
#
GROUP{
PROCESS = "queens"
DCACHE = "0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xCCCCCCCC"
}
ICACHE = "0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xAAAAAAAA,
          0xAAAAAAAA,0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xAAAAAAAA,0xCCCCCCCC,
          0xAAAAAAAA,0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,
          0xAAAAAAAA,0xCCCCCCCC,0xCCCCCCCC,0xCCCCCCCC"
}
```

Bibliography

[1] Executable and linkable format. online at: http://flint.cs.yale.edu/cs422/doc/ELF_Format.pdf.

[2] Leon3 processor. online at: http://www.gaisler.com/products/grlib/grlib.pdf.

[3] Ml605 hardware user guide. online at: http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf.

[4] Perf tutorials, linux kernel profiling with perf. online at: https://perf.wiki.kernel.org/index.php/Tutorial.

[5] Time(1) linux man pages. online at: http://man7.org/linux/man-pages/man1/time.1.html.

[6] Abdullah Fathi Ahmed. Self-optimizing organic cache. Master's thesis, The school where the thesis was written, University of Paderborn, Germany, 2 2015.

[7] Daniel P Bovet and Marco Cesati. *Understanding the Linux kernel, 3rd Edition.* " O'Reilly Media, Inc.", 2005.

[8] Stephen Brown and Jonathan Rose. Architecture of fpgas and cplds: A tutorial. *IEEE Design and Test of Computers*, 13(2):42–57, 1996.

[9] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux device drivers, 3rd Edition.* " O'Reilly Media, Inc.", 2005.

[10] Peter B Galvin, Greg Gagne, and Abraham Silberschatz. *Operating system concepts,9th Edition.* John Wiley & Sons, Inc., 2013.

[11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach.* Elsevier, 2011.

[12] Thomas F Herbert. *The Linux TCP/IP Stack: Networking for Embedded Systems.* Charles River Media, 2004.

[13] Glenn Herrin. Linux ip networking: A guide to the implementation and modification of the linux protocol stack. Technical report, DTIC Document, 2000.

[14] Tetsuya Higuchi, Tatsuya Niwa, Toshio Tanaka, Hitoshi Iba, Hugo de Garis, and Tatsumi Furuya. Evolving hardware with genetic learning: A first step towards building a darwin machine. In *Proceedings of the Second International Conference on From Animals to Animats 2 : Simulation of Adaptive Behavior: Simulation of Adaptive Behavior*, pages 417–424, Cambridge, MA, USA, 1993. MIT Press.

[15] Nam Ho, Abdullah Fathi Ahmed, Paul Kaufmann, and Marco Platzner. Microarchitectural optimization by means of reconfigurable and evolvable cache mappings. In *Adaptive Hardware and Systems (AHS), 2015 NASA/ESA Conference on*. IEEE, 2015.

[16] P. Kaufmann, C. Plessl, and M. Platzner. Evocaches: Application-specific adaptation of cache mappings. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pages 11–18, July 2009.

[17] Markus Kowarschik and Christian Weiß. An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for Memory Hierarchies*, pages 213–232. Springer, 2003.

[18] Robert Love. *Linux kernel development, 3rd Edition*. Pearson Education, 2010.

[19] Wolfgang Mauerer. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

[20] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3), 1982.