AN OBSERVABLE DATA CACHE MODEL FOR FPGA PROTOTYPING

PARTHASARATHY RAVISHANKAR

A Thesis
in
The Department
Of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements for the
Degree of Master of Applied Science (Electrical and Computer Engineering)
at

Concordia University
Montréal, Québec, Canada

March 2013

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Parthasarathy Ravishankar

Entitled: "An Observable Data Cache Model for FPGA Prototyping"

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Rabin Raut

_____ Examiner, External
Dr. Benjamin Fung To the Program

_____ Examiner
Dr. Sofiene Tahar

_____ Supervisor
Dr. Samar Abdi

Approved by: _____
Dr. W. E. Lynch, Chair
Department of Electrical and Computer Engineering

_____March 2013_____ _____
Dr. Robin A. L. Drew
Dean, Faculty of Engineering and
Computer Science

ABSTRACT

An Observable Data Cache Model for FPGA Prototyping

Parthasarathy Ravishankar

This work presents design of a configurable and observable model of L1 data cache memory and a novel method for integrating the model into an FPGA prototype. Embedded system software designers use in-circuit emulation on FPGA platforms to validate the functionality and performance of embedded software. Data cache, particularly L1, has a major impact of system performance, yet remains unobservable during software debugging and analysis. Our solution is to model the data cache as an on-chip hardware peripheral that can be integrated into the processor system and can display the state of the data cache at any given time. The model is synthesized on Xilinx Virtex 5 FPGA and validated using several benchmarks. The experimental results show that the model can accurately track cache hits and misses and can estimate the run time of an embedded software application with an average error of only 5.4%, and a worst case error of only 13.7%.

ACKNOWLEDGEMENTS

Foremost, I would like to express my gratitude to my supervisor Dr. Samar Abdi, for his support, motivation and patience throughout my study and research. I thank him for his generous contribution of time and expertise which guided me to successful completion of Master's thesis.

Besides my supervisor, I would like to thank the rest of my thesis committee members, Dr. Sofiene Tahar and Dr. Benjamin Fung for their invaluable comments and questions. I also thank Ted Obuchowicz - Engineering specialist, who was helpful every time I had difficulties with my software.

I would also like to thank my fellow labmates: Zaid, Richard, Paul, Karim, Ali, Ehsan and Kazem for your suggestions and all the discussions we had. It was always fun to work with you guys. I thank all my greatest friends in Canada and KDs for always cheering me on.

Finally I would like to thank my wonderful parents, grandparents and brother for their unconditional love and continuous support. My parents have been my biggest source of inspiration and I cannot thank them enough for whatever they have done to me.

To my parents

CONTENTS

List of Tables

List of Figures

List of Acronyms

FPGA	Field Programmable Gate Array
SRAM	Static Random Access Memory
L1	Level 1
TLM	Transaction Level Modeling
ASIC	Application Specific Integrated Circuit
DDR2 SDRAM	Double Data Rate Synchronous Dynamic Random Access Memory
LRU	Least Recently Used
ILA	Integrated Logic Analyzer
CRCB	Configuration Reduction approach by the Cache Behavior
CSR	Control Status Register
FSM	Finite State Machine
RNW	Read Not Write
WT	Write Through
WB	Write Back
pCache	Peripheral Cache
ICache	Instruction Cache
DCache	Data Cache
R/W	Read or Write
BRAM	Block Random Access Memory
UART	Universal Asynchronous Receiver Transmitter
DLMB	Data Local Memory Bus
ILMB	Instruction Local Memory Bus
PLB	Processor Local Bus
DXCL	Data-Xilinx Cache Link
IXCL	Instruction-Xilinx Cache Link
EDK	Embedded Development Kit
JTAG	Joint Test Action Group
XMD	Xilinx Microprocessor Debugger
FSL	Fast Simplex Link

List of Principal Symbols

T_{pcache}	Run time in pCache based model
T_{est}	Estimated run time
$T_{built\text{-}in}$	Run time in built-in cache system
H	Number of hits
M	Number of misses
$T_{xcl\text{-}hit}$	Average time for hit via CacheLink
$T_{xcl\text{-}miss}$	Average time for miss via CacheLink
T_{plb}	Average time for access via PLB
RH	Number of read hits
RM	Number of read misses
NC_{wr}	Number of non-consecutive writes
C_{wr}	Number of consecutive writes
N_{rf_cyc}	Number of refresh cycles
T_{rd}	Average time for read via PLB
T_{wr}	Average time for write via PLB
T'_{rh}	Average time for read hit via XCL
T'_{rm}	Average time for read miss via XCL
$T'_{nc\text{-}wr}$	Average time for non consecutive writes via XCL
$T'_{c\text{-}wr}$	Average time for consecutive writes via XCL
WH	Number of write hits
WM_{nD}	Number of write miss not dirty
WM_{xD}	Number of write miss with x dirty bits
RM_{nD}	Number of read miss not dirty
RM_{xD}	Number of read miss with x dirty bits
T'_{wh}	Average time for write hit via XCL
$T'_{wm\text{-}nD}$	Average time for write miss not dirty via XCL
$T'_{wm\text{-}xD}$	Average time for write miss with *x* dirty bits via XCL
$T'_{rm\text{-}nD}$	Average time for read miss not dirty via XCL
$T'_{rm\text{-}xD}$	Average time for read miss with *x* dirty bits via XCL

CHAPTER 1

Introduction

The gap between processor performance and the time taken to access main memory keeps growing continuously as shown in Figure 1.1 [20]. This increasing performance gap is a major drawback in the overall computer system performance [20, 21].

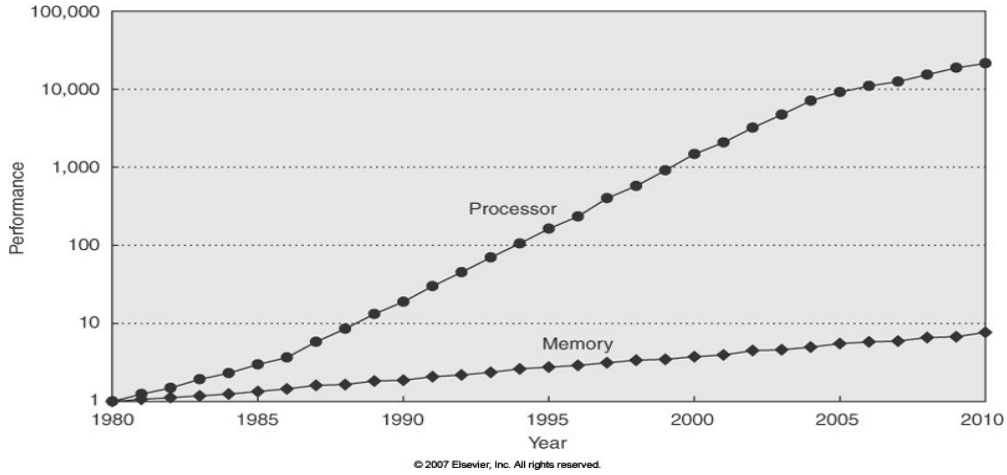


Figure 1.1 Processor-Memory gap [20]

In order to bridge the gap, cache memory was introduced between the processor and the memory. Cache is a small, fast, expensive memory made of SRAM, which reduces the average time to access memory. Since fast memory is expensive, the memory hierarchy is divided into multiple levels, such as registers, L1 cache, L2 cache, L3 cache and main memory. Each level in the hierarchy is smaller, faster, and more expensive per byte than the next lower level. The objective is to provide cost-effective and high performance memory system.

1.1 Motivation

The size and configuration of L1 data cache in embedded processors have a great impact on software performance. However, data caches are unobservable by the embedded software designer. As such, it is difficult for the software designer to observe the cache state and performance, for specific instances of code execution. Such feedback is useful to optimize the software for improving cache hits [1]. Moreover, during design space exploration, it is desirable to modify the cache configuration of a given processor core in order to evaluate cache design choices. In order to accomplish the above objectives, we need a fast, observable, configurable and timing accurate model of the data cache.

1.2 Embedded System Validation

Validation broadly refers to the process of determining that a design is functionally correct. The two most commonly used methods for early system level validation are software simulation and FPGA prototyping.

1.2.1 Software simulation

Software simulation refers to an event driven logic simulator that operates by propagating input changes through the design to simulate the operation of the digital circuit [9]. Software simulators use languages such as Verilog, VHDL and SystemC to describe the design and verification environment. Cycle accurate software models of processor cores and the memory hierarchy provide excellent observability. The cycle accurate models can also be easily configured to reflect different design choices. However, it becomes extremely difficult to model the processor subsystem, bus and the memory hierarchy altogether. Such cycle accurate model requires large amounts of computing resources and time. Abstract software

simulation models, such as *Transaction-level Models* (TLMs), are very useful for early system modeling, but they compromise cycle accuracy for greater simulation speed.

1.2.2 FPGA Prototyping

FPGA based prototyping refers to the process of prototyping SoC and ASIC design on FPGA for hardware verification and early software development. FPGA prototypes are created by instantiating the processor cores and other system components on an FPGA chip. This technique enables pre-silicon embedded software development and allows hardware and software co-development. Moreover, once the design process is over, the FPGAs are ready for production, while ASICs take much longer time to reach production. As such, it helps improve time to market window and avoids expensive silicon re-spin. FPGA prototypes are typically, several orders of magnitude faster than cycle accurate software simulation models, while still providing cycle accuracy. In-circuit emulation techniques can be used to debug software and observe the addressable memory in FPGA prototypes. However, L1 caches cannot be easily probed and observed in FPGA prototypes since they are not bus-addressable.

1.3 Rationale for modeling direct mapped L1 data cache

The rationale for modeling direct mapped L1 data cache is

- We do not target the instruction cache (ICache) to be modeled, because the performance of instruction cache is generally good. Instruction access pattern is sequential and less random when compared to data access pattern. The effect of change in configuration of ICache has less impact on the performance of software.

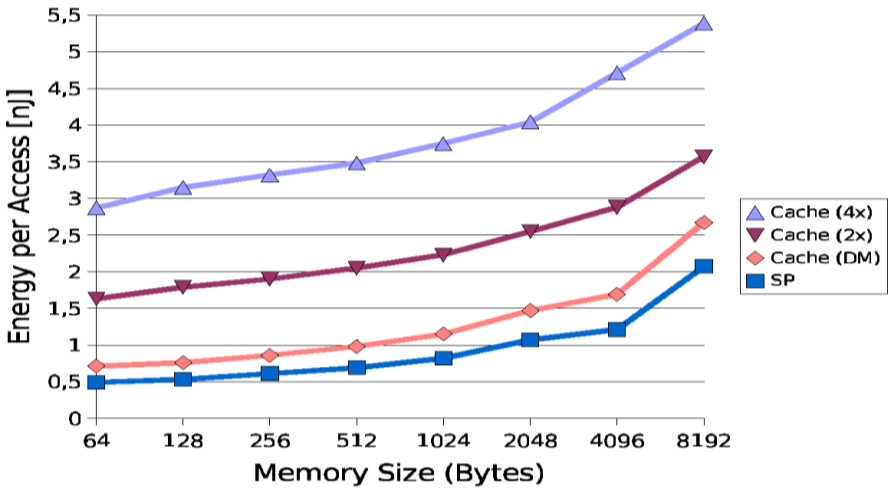
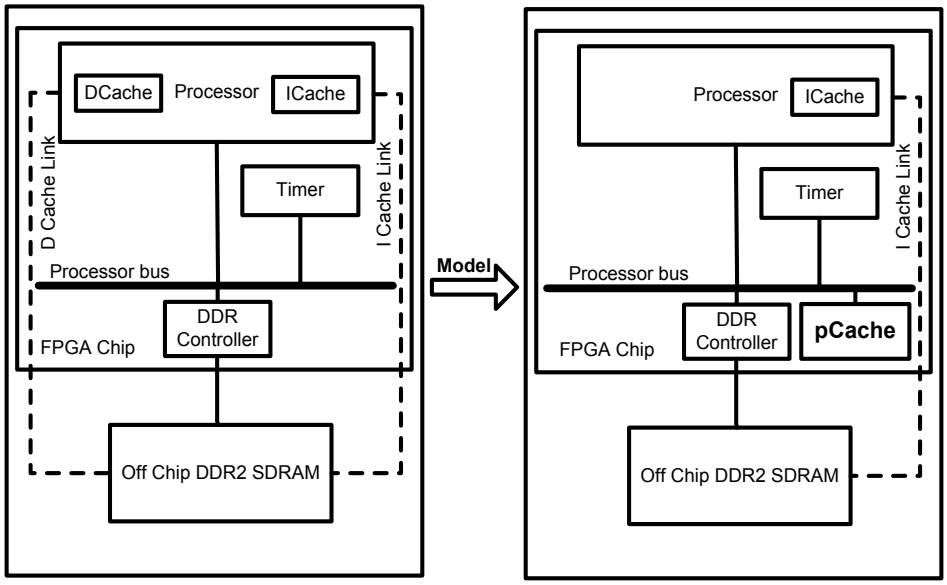


Figure 1.2 Influence of associativity of cache on energy [P. Marwedel et al., ASPDAC, 2004]

For example, increasing the block size of ICache will lead to increase in performance.

- Direct mapped cache is preferred to set associative cache because direct mapped cache is extremely quick to search and consumes less energy as shown by Figure 1.2 [33]. The set associative caches consume more energy which reduces the performance of embedded systems.
- Level 1 (L1) cache has the greatest impact on system performance. As such, embedded systems may have only L1 cache and ignore other hierarchy. Hence, the pCache focuses on modeling L1 cache.



(a) Processor system with built-in cache (b) Processor system with pCache

Figure 1.3 Modeling methodology

1.4 Methodology

Figure 1.3 illustrates our modeling methodology. The target design of the processor system with built-in L1 data cache (DCache) is shown in Figure 1.3 (a). The peripherals such as timer and DDR controller are connected to the processor bus. There is also dedicated cache link connecting the processor to the off chip memory. Upon cache miss, the block of data corresponding to the missed address is fetched from the off chip memory over the dedicated cache link. Software debuggers, typically, can display only the processor internal registers and the contents of memories mapped on the processor bus. As such it is not possible to display the cache state at run time using software debuggers. In order to provide an observable cache to the embedded software designers, we model this system as shown Figure 1.3 (b).

The pCache-based model consists of the processor system without a built-in data cache. In the case of hard processor cores, the data cache can simply be disabled in software, since most embedded processors provide instructions to turn off the data cache. The processor can access data from off chip memory only through the processor bus. The pCache module is a custom hardware component that interfaces to the processor using the processor bus. It monitors the processor address signals and Read/Write signals in the processor bus to track every data transaction targeted to the main memory, and to register a cache hit or miss. Memory mapped registers within pCache are used to store the cached memory addresses and are accessible using a software debugger. The run time of a software application in pCache system will be higher because of the absence of built-in data cache and the missing cache link. However, with the data from hit and miss counters we can derive the estimated run time of the software application in processor system with built-in cache. The pCache module is also configurable, enabling the designer to explore different cache architectures by changing its parameters.

The advantages of a pCache based model are

(i) **Observability**: an observable data cache model and a debugger can be used by embedded software developers to easily identify code with poor cache behavior and explore opportunities for code optimization.

(ii) **At-speed simulation**: since the cache model is built into the FPGA prototype, the embedded software is executed on the actual processor, implemented as a soft or hard core in FPGA. As such, there is no need to use a slow cycle accurate software simulation model of the processor to generate the memory trace.

(iii) **Easy reconfiguration**: The processor, if available as soft-core, can be treated as a black box. The cache model can be reconfigured without having to re-synthesize the processor in FPGA. If the processor core is implemented as a hard core on the FPGA/board, the built-in cache cannot be reconfigured. Therefore, for modeling purposes, the built-in cache can be turned off and the pCache model can be used.

1.5 Related Work

There are different ways to model and simulate cache memory behavior. It can be broadly divided into Trace driven simulation, Analytical model (trace analysis) and high level functional simulation.

1.5.1 Analytical model

Analytical model uses parameters extracted from address trace of programs to quickly predict the performance of cache. As such these models require a mechanism to collect address trace and to store them. Such mathematical approach requires various assumptions about the statistical properties of address traces and data use patterns to formulate analytical equations.

Several analytical models [26, 27 and 28] have been proposed which are limited mainly to analyzing perfect loop nests, with straight line assignments and no call statements in the program. Xue et.al. [3] proposed an analytical model of data cache which is applicable to whole programs, including loop nests. The model is applicable only to programs with compile time predictable memory accesses, and so is less accurate for programs with data dependent constructs.

An analytical cache model is proposed in [2] that analyses parameters extracted from the address trace to predict average cache miss rate. The miss rate calculated from the model is compared to the simulation results. Although the computation cost is low, the miss rate predictions by the model are consistently lower than simulated miss rates for caches of size 8K through 32K. The mean relative error in miss rate for certain cache configuration is quite significant at 23% and the worst case error in prediction is 39%.

In 2003, Ghosh et al. [19] proposed an analytical model of the cache combined with an algorithm to compute cache parameters satisfying desired performance constraints. Their objective was to obtain a set of optimal cache pairs (Depth, Associativity) for a given number of desired cache misses. However, to limit the number of design points in design space exploration, they do not consider the cache line size as a varying parameter. Likewise, they have assumed fixed Least Recently Used (LRU) and write back cache policies. As such, they have cache size and associativity only as varying configuration parameters.

1.5.2 Trace driven simulation

Trace driven simulation [25] has been the typical approach to evaluate cache performance. It requires the application software to be simulated once to generate memory reference trace (address trace). This address trace is then processed by a cache simulator for each cache configuration that needs to be evaluated. The advantage of trace simulation is that it is more accurate than the analytical model. However, processing only one configuration on each simulation pass during design space exploration can prove to be tedious and time consuming. Moreover, address traces that are very long requires large storage space requirement and longer simulation time.

One such simulator is Dinero IV [11], in which there is no notion of simulation time or cycles. It just gives information on hits and misses for an address trace. It also has the drawback of requiring repeated simulation runs resulting in lengthy design space exploration time. In 2012, Atanasovski et al [12] proposed a highly configurable trace driven cache simulator MMCacheSim. The simulator, implemented as a set of java classes predicts the performance of multiple levels of cache. In order to validate, they compare the average CPU cycles for memory access in the simulator and in real multiprocessors. The results show that the simulated values are not very accurate and there is no discussion on the actual error percentage. Moreover, the simulator is specific to matrix related applications and is not validated against standard benchmarks.

1.5.3 Single pass simulation

In order to overcome the disadvantage of repeated cache simulation in conventional trace driven simulation, the single pass simulation technique based on inclusion property [13] was first proposed by Mattson et al. An efficient Stack data structure was used to determine the performance of multiple cache architectures in one pass of the address trace. In 2010, Haque et al developed a single pass L1 cache simulator, SCUD [14]. It has a special data structure that is made up of Central look-up table, binomial tree and miss counter table to calculate the cache miss rate of an application trace. The central look-up table holds hit/miss information of memory addresses for all possible cache configurations. The SCUD uses several properties of its data structure like binary search and binomial tree to simplify the decision making process during simulation. Simulation times for various benchmarks are compared to the simulation time in conventional Dinero IV [11] cache simulator. It shows an average

speed up of 10 times over Dinero IV. However, for memory intensive workloads like Mpeg2, the SCUD simulation time is as high as 6.5 hours.

In order to further decrease the time complexity of simulation during design space exploration of caches, Janapsatya et al. [15] proposed a simulation algorithm based on Cache Inclusion properties. The basic idea of the approach in [15] is, given two caches with same associativity using LRU replacement policy, the following are applicable.

- Whenever a cache hit occurs, all caches that have larger set sizes will also guarantee a cache hit.
- Whenever a cache miss occurs, all caches that have smaller set sizes will also guarantee a cache miss.

The time complexity of finding the best configuration among all the cache configurations by this approach is reduced by skipping simulation based on the above assumptions. The results of this method are consistent with the Dinero IV [11] simulator, and have an average speed up of 45 times over the later. In 2009, Tojo et al. [6] proposed CRCB1 and CRCB2 algorithm to improve Janapsatya's [15] simulation approach. Both the cache simulation algorithms are based on Cache inclusion property. It reduces the number of hit/miss judgements that are required for simulating all the cache configurations compared to conventional full trace simulation approaches. The cache configuration with minimum total memory access time is obtained by analyzing the stored address trace only once.

1.5.4 Partial trace simulation

Many partial address trace simulation techniques to simulate the cache architectures are available. They do not simulate the entire length of address trace, leading to an increase in

simulation speed. Statistical trace sampling techniques to form smaller clusters of trace have been proposed. But it leads to the problem of cold start bias, in which there is no consistency in state of the cache from one cluster to other. The accuracy of such cache simulation techniques depends on the method used for repairing the state of the cache at the beginning of each sample.

To solve the above state repair problem, Conte et al. [4] proposed two techniques *fill flush* and *no state loss*. The first technique *fill flush* is based on Stone's approach [16] to the single pass technique. In *fill flush* approach, unique references with unknown cache state are removed from the address trace. The miss ratio for a cache configuration by this method is expressed as,

$$\rho = \frac{N - (R[B] - D[C,B,S])}{N - F[B]}$$

where [C,B,S] is the notation used to represent the dimension of a cache of size 2^C bytes, with a block size of 2^B bytes and 2^S blocks in each set. N is the total number of references and R[B] is the total number of recurrences (hits) for block size B. D[C,B,S] is the total number of dimensional conflicts (misses) for that particular configuration. F[B] is the number of references whose cache state is unknown. The *fill flush* method is fast, because only small clusters of trace are simulated. However, since the cache is flushed at the beginning of each cluster and references with unknown cache state are removed from calculating the miss ratio, the method is not very accurate. In the *no state loss* method, statistical sampling is applied only to the conflict metric D and R[B] and N are recorded for the whole trace. The miss ratio for this method is calculated as,

$$\rho = \frac{N - R[B]}{N} - \frac{D'[C,B,S]}{N_s\, L_s}$$

where N_s is the number of sampled clusters each of length L_s. The advantage of this method is the state of all the references within the sample is known since state of the cache is maintained throughout simulation. On average only 6% of the address trace is sampled in this method. Since most of the trace is processed, this method is accurate but very slow.

To overcome the state repair problem arising from sampling the address trace, X.Li et al. [17] proposed cache simulation using compressed traces. This technique is based on SEQUITUR algorithm [18] for trace compression and cache inclusion properties. The SEQUITUR algorithm identifies the repeating memory reference sequences present in the trace. Cache simulation for each repeating sequence is performed only once, leading to significant reduction in simulation time. This lossless compression scheme produces accurate cache hit/miss results. But the downside of this technique is that, the compression ratio will be very less if there are very few repetitive patterns.

Space sampling and time sampling techniques on address trace are propose by Chen et al. [7]. Although they reduce space and time requirements in trace driven simulation, the estimation of Miss per Instruction (MPI) by space sampling is not consistently accurate. For instance, Eqnott benchmark reports an error of 37.75%. The time sampling is applicable to loop iterations only and requires additional pre-processing to detect loops.

1.5.5 High level simulation

Transaction Level Modeling (TLM) [29] is the widely used approach for system modeling and simulation. This increased level of abstraction in system modeling can increase the

simulation speed by two or three orders of magnitude compared to conventional Instruction Set Simulator approach. In TLM, the source program is back annotated with execution delay information of the target processor. This annotated program is then compiled and executed on the host machine. The total execution time can be estimated by summing up the annotated timing numbers.

Early transaction level model [30] provides increased simulation speed, but do not model the behavior of cache during execution delay estimation. Such model compromises timing accuracy for greater speed. Fast Veri [31] a product of InterDesign Technologies, uses a high speed TLM model for hardware/software co-simulation. In order to maintain timing accuracy, the program is back annotated with delays from their cache model. However, their work is proprietary and not easily extendable for other transaction level models.

Pedram et al. [8] extended the transaction level model in [30], by integrating a cache model into a TLM. The address information obtained from synthesized target binary and the basic block timing information obtained from ISS is used in back annotated cache calls. The model dynamically updates its status and returns appropriate delay for each access. The disadvantage is the process of inserting cache calls is done manually. Experimental results for only a simple matrix multiplication program is provided. As such, the method has not been demonstrated on realistic benchmark applications.

Pieper et al. [7] performed high level cache simulation by back annotating the original trace with memory access delays. They used stack distance histogram which records the delay since last reference to an address and this data is used to simulate the behavior of cache. Since the delays for each memory address is too large for back annotation, they proposed

compression techniques to reduce the size required to store the data inside a histogram. However, the accuracy for large direct mapped cache is poor. The worst case miss prediction error for a 2KB direct mapped cache is 337%.

Lin et al. [32] proposed source level timing annotation for generating accurate TLM model. The source program is divided into basic blocks and annotated with timing information. Moreover, by analyzing the assembly code through a target processor model, they take into account features like pipeline, branch prediction and cache architectures. However, since the target assembly code is not executed, the exact address of data access is unknown. As such, they use average cycle number to model the delay of each data access. Hence, their data cache model does not precisely calculate the data cache access latency.

1.5.6 Novelty of pCache

The pCache based model is novel and provides significant improvement over existing models as explained below.

- Analytical models of cache extract parameters from the stored address trace for computation. Such models are fast, but the estimation is highly error prone. In contrast, the pCache model is fast, accurate and does not require storing address trace.
- Trace driven simulation is an accurate method to evaluate cache performance, but the simulation runs are very long. Since only one configuration can be processed in each simulation pass, it is prohibitively slow to be used in design space exploration. In contrast pCache is fast and can be easily used in cache design space exploration.

- Single pass simulation technique evaluates multiple cache configurations in single run. It is faster than conventional trace driven simulation but slower compared to pCache model.
- Partial trace simulation applies sampling technique to reduce the length of address trace to be simulated. But it leads to cache cold start problem that make such simulations inaccurate. In contrast, pCache can be used which is fast and accurate.
- Transaction level model of cache is fast compared to other simulation techniques, but is not cycle accurate. Alternatively, pCache is fast and cycle accurate.

1.6 Thesis Contribution

In this thesis, we present an observable cache model called pCache that can be used in FPGA prototyping. A pCache based model combines the observability of software simulation with the speed and cycle accuracy of FPGA prototyping. The contributions of this work are,

- We designed and implemented a cycle accurate, observable and fast functional model of L1 data cache on the FPGA, which is a new technique not currently possible with bus analyzers. The module can be integrated into a prototype of a processor system on an FPGA.
- The model is several orders of magnitude faster than conventional software simulation. Examples show FPGA based model can simulate designs in milliseconds as opposed to several hours in software simulation.
- A highly configurable pCache model which allows us to create models of direct mapped, L1 caches of any size, of any block size, with write through and write back techniques.

- We developed a parameterized timing model of the cache for accurate performance estimation of embedded software, with different cache configuration.
- A methodology for cache design space exploration using the timing model.
- Demonstration of software optimizations such as loop splitting and fusion using the observable cache model.

1.7 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, we outline the design of the peripheral cache, called pCache, and its features. In chapter 3, the built-in cache based reference design and pCache based model is introduced. A complete functional validation of both the write policies, write through and write back, is also presented. A detailed timing analysis of the pCache based model is presented in Chapter 4. The parameterized cache timing model and equations to estimate overhead in pCache model are also presented. Timing validation against standard benchmarks is shown in section 4.5.

CHAPTER 2

Peripheral Cache

In this chapter, the architecture of the peripheral cache (pCache), its various features and interface are described in detail.

2.1 pCache - Interface

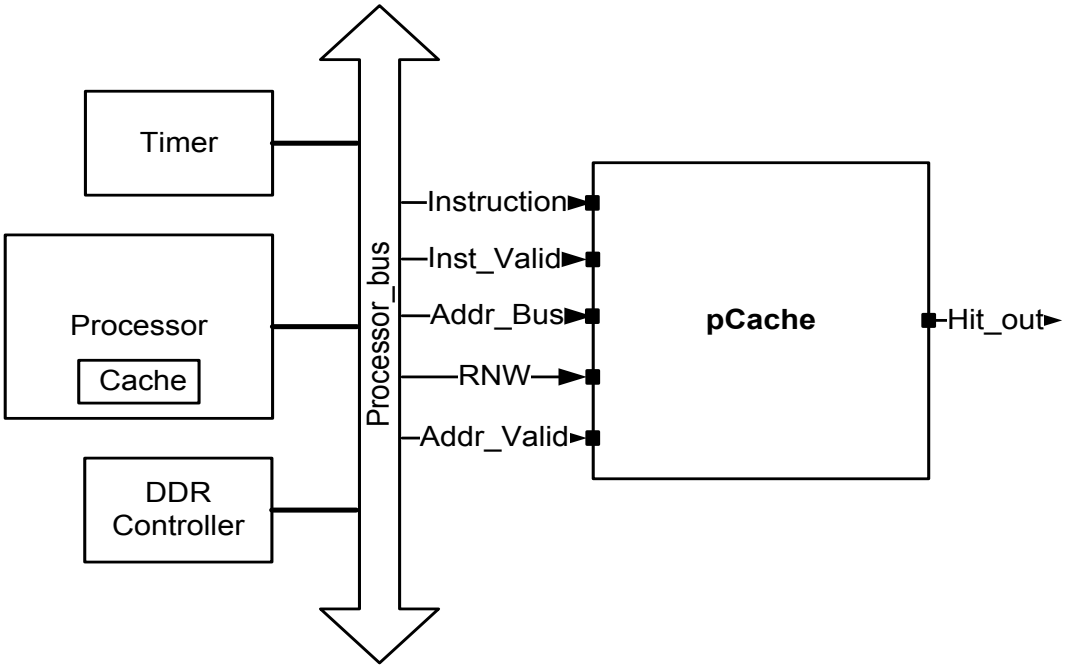


Figure 2.1 Block diagram of pCache - processor bus interface.

Figure 2.1 shows the interface between pCache and the processor bus. The pCache module is connected as a slave peripheral to the processor bus. The input signals to the pCache are Instruction bus, Instruction valid, Address bus, Read/Write and Address valid. The result of the data access (hit/miss) is shown by the Hit out output signal from pCache.

2.2 pCache Architecture

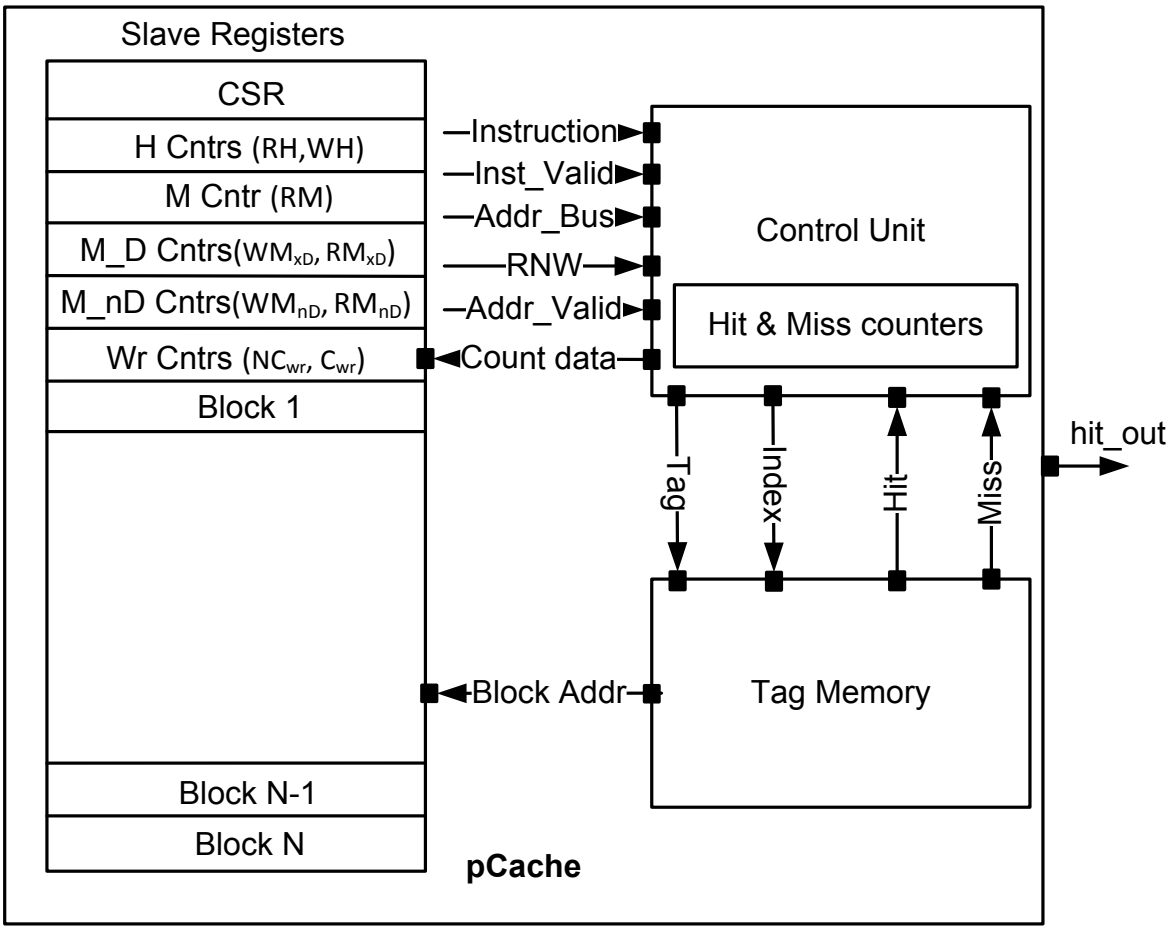


Figure 2.2 Block diagram of pCache.

Figure 2.2 shows the block diagram of direct mapped pCache. pCache consists of the following sub modules: Slave registers, Control Unit, Tag Memory, Control Status Register (CSR) and Hit and Miss Counters. CSR is a 32 bit control register for pCache that allows the processor to Reset, Enable and Disable pCache. Tag Memory is a 2D array of registers that store the tags of memory locations being accessed. The pCache model does not store the actual data being cached but stores only the address of the memory locations being accessed. Hence, we are not interested in tracking any of the data signals in the processor bus. In order to display the contents of the cache, we write the address of the first location of the block of

memory on to slave registers labeled Block 1 to Block N in Figure 3. Cache timing parameters are kept in specific slave registers: H Cntrs (hit counters), M Cntr (miss counter), M_D Cntrs (miss-dirty counters), M_nD Cntrs (miss-not-dirty counters) and Wr Cntrs (write counters). We will elaborate on the timing parameters in Section 4.2.

2.3 Write Policies

The write policies on write hit distinguish cache designs into Write Through and Write Back caches.

2.3.1 Write through policy

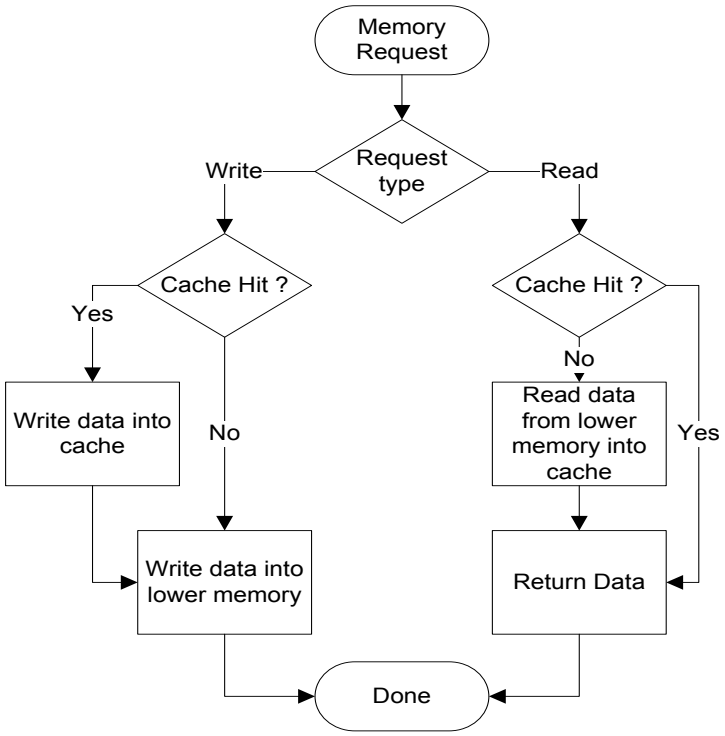


Figure 2.3 Flow chart describing write through policy.

The operation of write through policy is explained by the flow chart shown in Figure 2.3. When a write hit occurs, the data is updated both in the cache and main memory. On a write miss, the data is written directly into the main memory. On a read miss, the existing block in

cache is replaced by a new block from the main memory. The advantage of this method is that the main memory always has the most recent value of the data. But this makes write slower, since every write requires a main memory access.

2.3.2 Write back policy

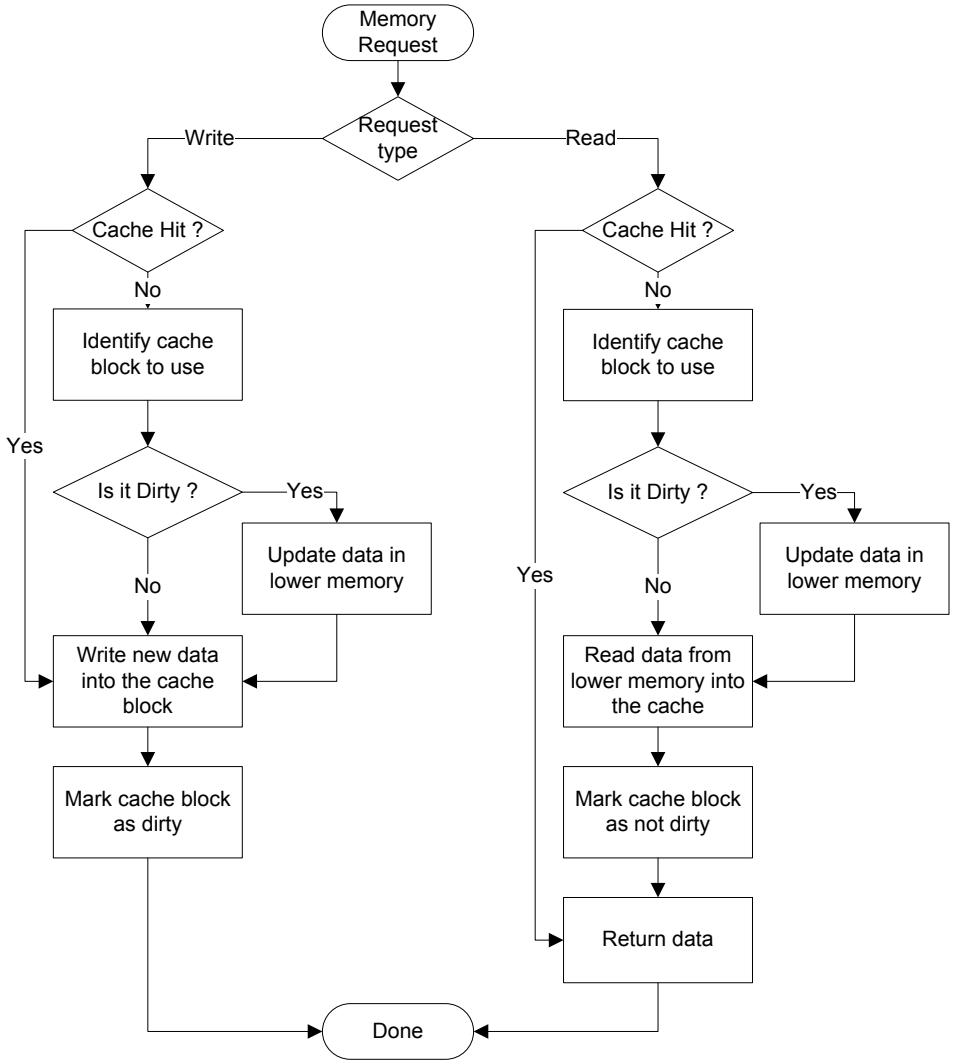


Figure 2.4 Flow chart describing write back policy.

Figure 2.4 describes the procedure for write back policy. On a write hit, the data is modified in the cache and the dirty bit is set high. On a write miss, the selected cache block's dirty bit is checked. If dirty, the modified block of data is updated in the memory and is replaced with

a new block. Similarly on a read hit, the data is read from the cache and on a read miss, the data is updated in the memory if dirty bit is high. Else it is replaced by a new block from the main memory and marked not dirty.

2.4 Control Unit

The pCache Control Unit models the write through and write back behaviors described in section 2.3. The Control Unit is responsible for tracking address and Read/Write signals in the processor bus and updating the pCache state. The operation of Control Unit is described using finite state machines shown in Figure 2.5 and 2.6. We define four states that the system may be in while running: *Check R/W*, *Read*, *Write*, and *Tag_mem update*. Control signals are assigned specific values in each of the states.

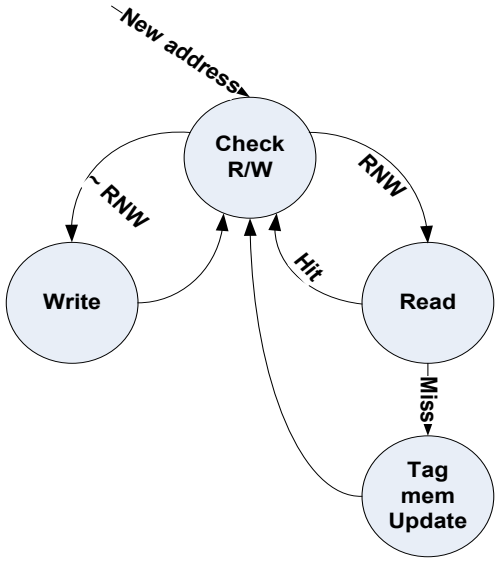


Figure 2.5 Control Unit FSM – Write through policy.

Figure 2.5 shows the FSM that implements write through policy. Upon receiving a new valid address, the controller checks the RNW signal. If RNW is '1' the controller goes to *Read* state where the tag memory is checked to know if the requested data is cached. The

controller waits until it receives hit/miss signal from the tag memory. Upon receiving miss, the miss counter is incremented, and the controller moves to *Tag_mem update* state. The tag memory is updated and the system returns to *Check R/W* state. Upon receiving hit, the hit counter is incremented and the system returns to *Check R/W* state. If RNW is logic '0' the controller goes to *Write* state. In this state, it waits for the arrival of hit/miss signal from the Tag memory. Upon receiving hit/miss result, the respective counters are incremented and the controller returns to *Check R/W* state irrespective of hit or miss.

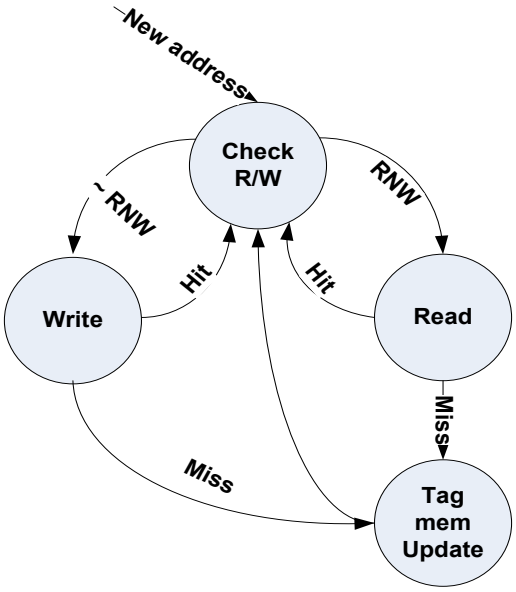


Figure 2.6 Control Unit FSM – Write back policy.

Figure 2.6 shows the FSM that implements write back policy. Upon receiving a new valid address, the controller checks the RNW signal. If RNW is '1' the controller goes to *Read* state where the tag memory is checked to know if the requested data is cached. Upon hit, the hit counter is incremented and the controller returns to *Check R/W* state. Upon miss, the miss counter is incremented and the controller moves to *Tag_mem update* state where the tag memory is updated. If RNW is logic '0', the system moves to *Write* state. The tag memory is

checked and the controller waits for the hit/miss result. Upon hit, the hit counter is incremented and the system returns to *Check R/W* state. Upon miss, the miss counter is incremented and the controller moves to *Tag_mem update* state. The tag memory is updated and the system returns to *Check R/W* state.

2.5 pCache – Features

The pCache module has user configurable parameters that can be used to describe the behavior of pCache model. It can be configured before synthesis using the parameters shown in Table 1. The parameters and its corresponding features are described below. The design of pCache makes it generic enough to be used with most buses. The configurability makes it flexible enough to model most direct mapped cache architectures used in embedded systems.

Table 1 The configuration parameters of pCache

Parameter	Description	Possible values	Default value
write_select	Write policy 1 = Write through 0 = Write back	0, 1	1
cache_line_size	Cache line length	4, 8	4
index_bits	Number of rows in cache	1 - 7	4
mem_addr_size	Size of DDR2 memory	6 - 28	14
mem_base_addr	Base address of DDR2 memory	0xXXXXXXXX	User defined

2.5.1 Write policy

The parameter *write_select* is used to select the write policy in pCache. When set to '1', write through policy is selected and when set to '0', write back policy is selected.

2.5.2 Line size

Each cache block in pCache can hold 4 or 8 words of data. The size of the block in pCache module can be configured by the parameter *cache_line_size*.

2.5.3 Cache size

The size of the pCache (number of rows) can be modified using *index_bits* parameter. The pCache can hold 2^{index_bits} rows. To demonstrate the operation of pCache, we have chosen the possible values to this parameter to be in the range of 1 to 7. This translates to cache size of 64 Bytes to 2K Bytes. However, the pCache can also have size more than 2KB.

2.5.4 Cacheable address range

The parameters *mem_addr_size* and *mem_base_addr* are used to determine the cacheable address range of the main memory. The parameter *mem_base_addr* represents the base address assigned to the off-chip DDR2 memory.

CHAPTER 3

Functional Validation

In this chapter, we validate the functional correctness of the pCache model manually by comparing its cycle accurate simulation to that of the reference design with built-in cache. Chipscope Pro Analyzer with an ILA (Integrated Logic Analyzer) core was used to validate and debug the pCache model.

3.1 Built-in cache based reference system

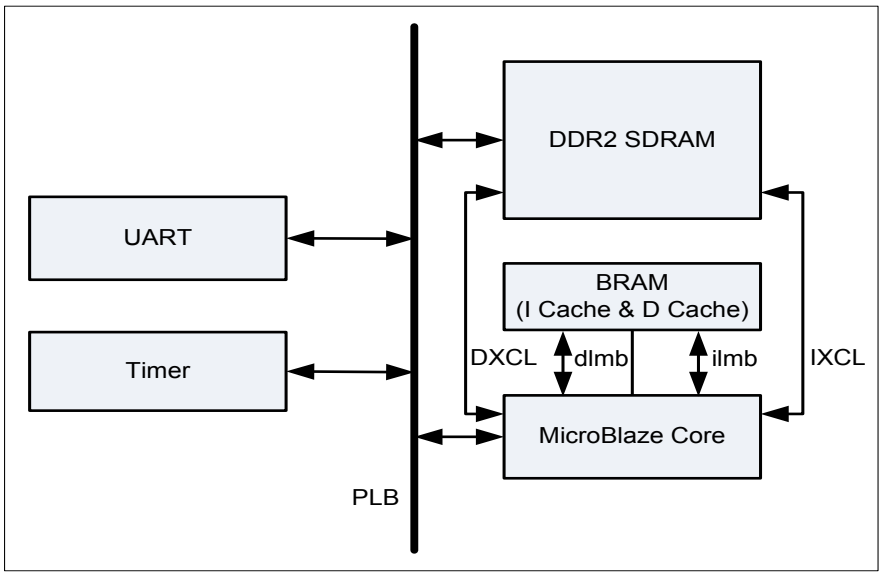


Figure 3.1 Block diagram of MicroBlaze core with built-in data cache

Figure 3.1 is the design of MicroBlaze based system with built-in data cache and instruction cache, similar to the one shown in Figure 1.3 (a). On-chip Block RAM connected to the MicroBlaze processor by data local memory bus (dlmb) and instruction local memory bus (ilmb) is utilised to implement the cache. When a new data is requested, the built-in cache

controller performs a lookup on the tags to check if the requested data is currently cached. On a cache miss, the built-in controller requests the new data block over data CacheLink (DXCL). The Universal Asynchronous Receiver Transmitter (UART) connects to the PLB and provides the interface for asynchronous serial data transfer between the system on FPGA and the HyperTerminal. The timer connected to PLB is used to measure the runtime of the embedded software executed by the processor.

3.2 pCache based model

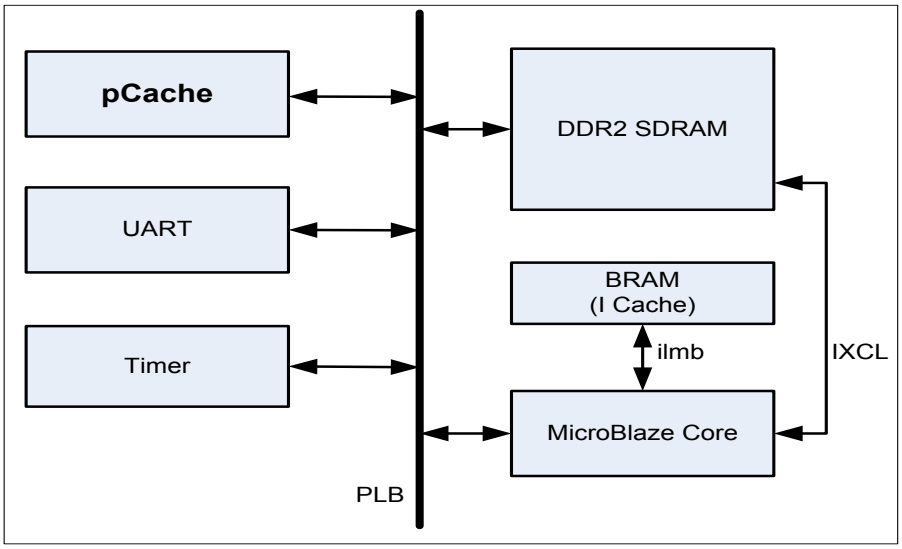


Figure 3.2 Block diagram of MicroBlaze core with pCache module.

Figure 3.2 is the model with pCache attached to the PLB as a peripheral, similar to the one shown in Figure 1.3 (b). Since the data cache is disabled, there is no DXCL bus in this model. The MicroBlaze processor accesses data from the main (DDR) memory via PLB. Instructions are fetched by the processor from the main memory via IXCL bus. Figure 3.3 shows the actual block diagram of datapath with MicroBlaze processor and the slave peripherals including pCache, timer and the UART.

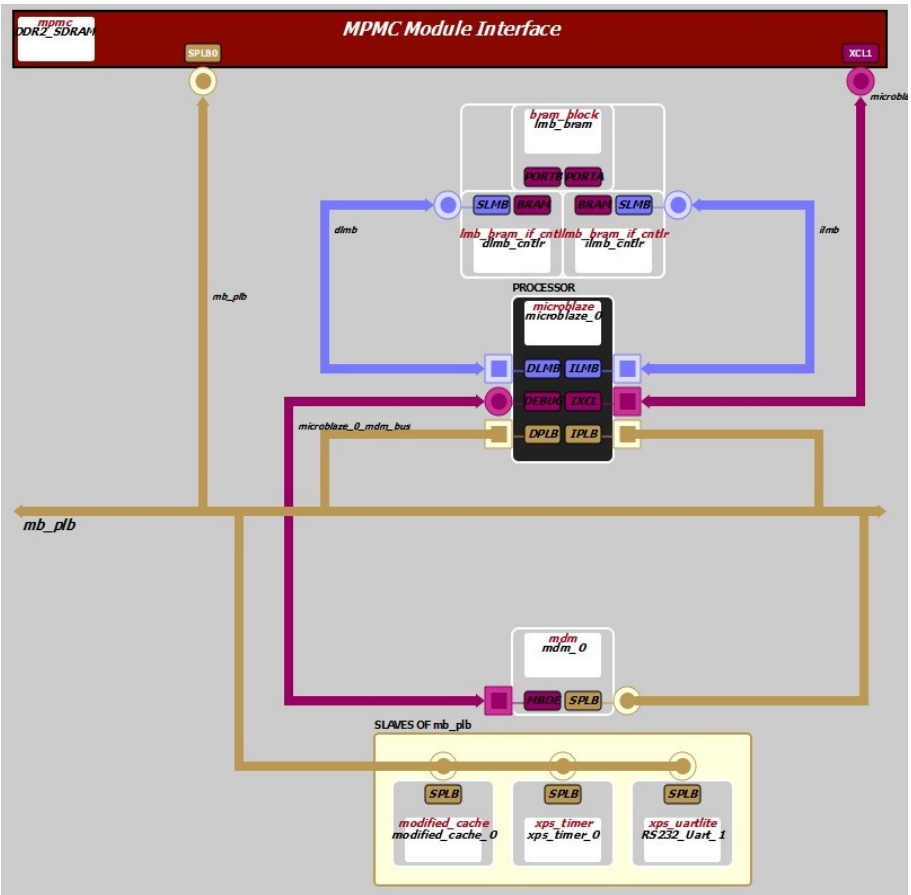


Figure 3.3 Block diagram of datapath with pCache module

The pCache design was implemented on a Xilinx Virtex 5 XC5VLX110T FPGA. The Xilinx Embedded Development Kit (EDK) supports debugging of a program on MicroBlaze processor running on an FPGA. The JTAG interface on board is used to communicate with the design and to capture internal bus signals via Chipscope bus analyzer. The trigger setup in Chipscope was configured to capture specific data in the design. The captured data in Chipscope is used to validate the pCache model against the reference design. The time for completion of a transaction shown in the following section, is specific to the particular case under consideration. The average delays for different types of transactions are discussed in Chapter 4.

3.3 Validation of Write through mode

3.3.1 Read transactions

Listing 1. Sample code to test read miss in WT cache

```
XIo_In32(0x90001A20);                    //Read miss
      imm       -28672
      addik     r3, r0, 6688
      lwi       r3, r3, 0
XIo_In32(0x900019AC);                    //Read miss
      imm       -28672
      addik     r3, r0, 6572
      lwi       r3, r3, 0
```

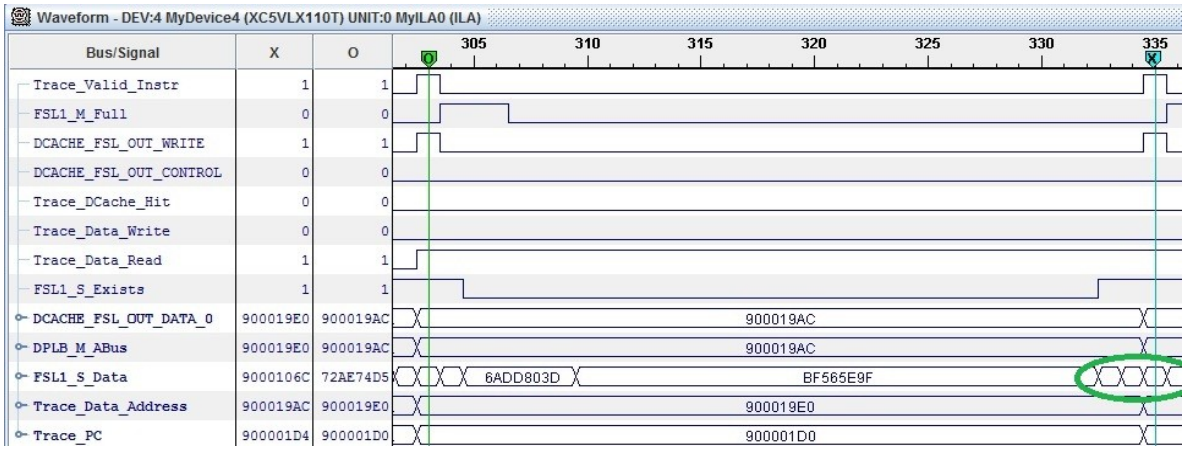


Figure 3.4 Read miss in 4 word WT built-in data cache

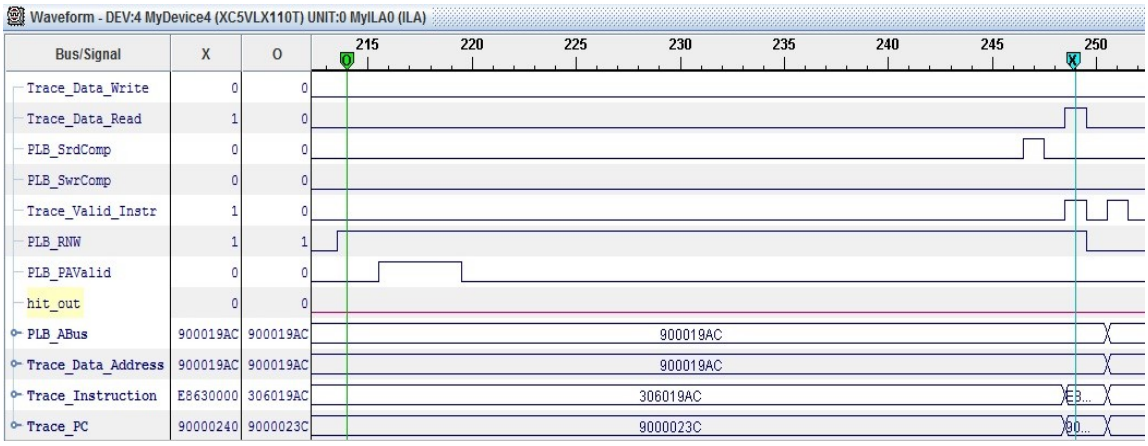


Figure 3.5 Read miss in pCache

Listing 1 shows the code to test read miss transaction in write through data cache. The *XIo_In32* and *XIo_Out32* macros read and write 32 bit data, respectively, into the memory. The assembly instructions that make up the macro are also shown in the listing 1. A 4 word data cache of size 256B is used for the test case. The first read operation to memory location 0x90001A20 results in a miss. The second read operation to memory location 0x900019AC, which is mapped to the same cache block as the previous address, also results in a miss. Figure 3.4 and Figure 3.5 shows the screenshot of read miss on data address 0x900019AC, in built-in data cache and pCache model respectively. The circled portion in Figure 3.4 shows 4 words of data being fetched over DXCL bus on a read miss. Pointers O and X point to the start and finish time of the data fetch. The missed data is read from off-chip memory in 32 clock cycles. Read miss in pCache is shown in Figure 3.5 by the highlighted signal *hit_out*. The hit signal remains low and takes 35 cycles to read data from main memory via PLB.

Listing 2. Sample code to test read hit in WT cache

```
XIo_In32(0x90005400);                    //Read miss
     imm        -28672
     addik      r3, r0, 21504
     lwi        r3, r3, 0
XIo_In32(0x90005408);                    //Read hit
     imm        -28672
     addik      r3, r0, 21512
     lwi        r3, r3, 0
```

Listing 2 shows the code to test read hit transaction in 4 word write through cache. The first read operation to memory location 0x90005400 results in a miss. The new block of data consisting 4 words is fetched into the data cache. The second read operation to memory location 0x90005408 results in a hit in data cache.

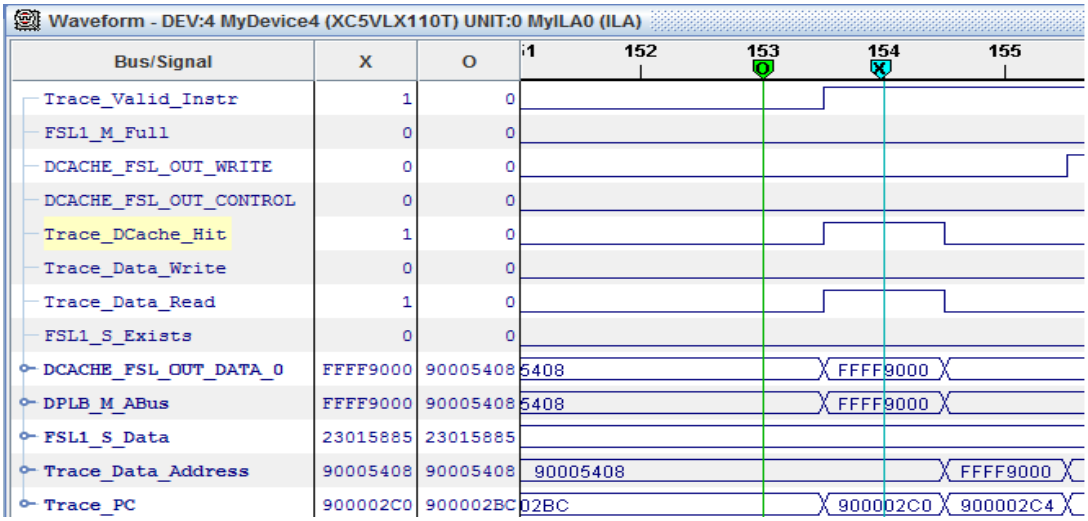


Figure 3.6 Read hit in built-in WT data cache

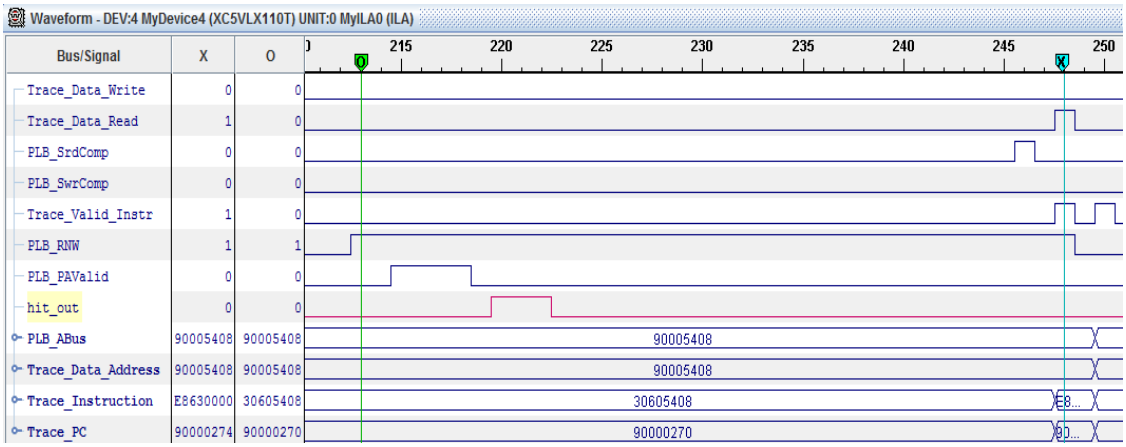


Figure 3.7 Read hit in pCache

Figure 3.6 and Figure 3.7 shows the screenshot of read hit on data address 0x90005408, in built-in data cache and the pCache model respectively. We can see the *Trace_Dcache_Hit* signal in built-in data cache and *hit_out* signal in pCache model go high for this memory transaction. We note, read hit takes 2 cycle delay in built-in data cache and in pCache model, it takes 35 cycles to complete the read from main memory via PLB.

3.3.2 Write transactions

Listing 3. Sample code to test consecutive writes in WT cache

```
asm("swi   r5, r19, 48");       //Write to 0x900019E8
asm("swi   r6, r19, 52");       //Write to 0x900019EC
asm("swi   r7, r19, 56");       //Write to 0x900019F0
asm("swi   r8, r19, 60");       //Write to 0x900019F4
```

Listing 4. Sample code to test non consecutive write in WT cache

```
XIo_Out32(0x9000540C, 0x12345FDA); //Write to 0x9000540C
      imm       -28672
      addik     r4, r0, 21516
      imm       4660
      addik     r3, r0, 24538       // 0x12345FDA
      swi       r3, r4, 0
```

Listing 3 shows the assembly code to test consecutive write operation in write through data cache. Since *XIo_Out32* macro consists of both arithmetic instructions and store instruction, we cannot use it to have consecutive write to main memory. Hence, we use the store instruction *swi* to generate four consecutive write to memory addresses 0x90019E8, 0x900019EC, 0x900019F0 and 0x900019F4.

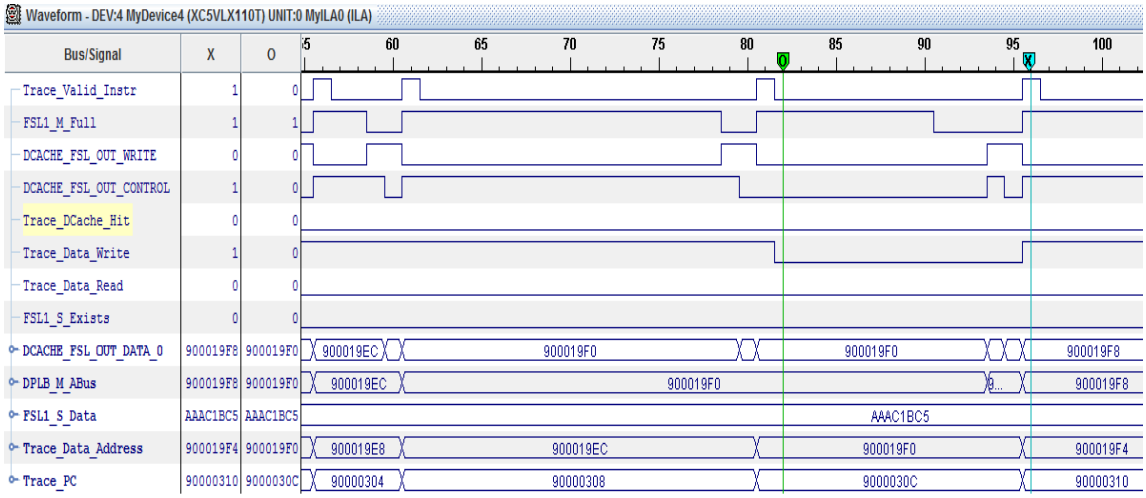


Figure 3.8 Consecutive writes in built-in WT data cache

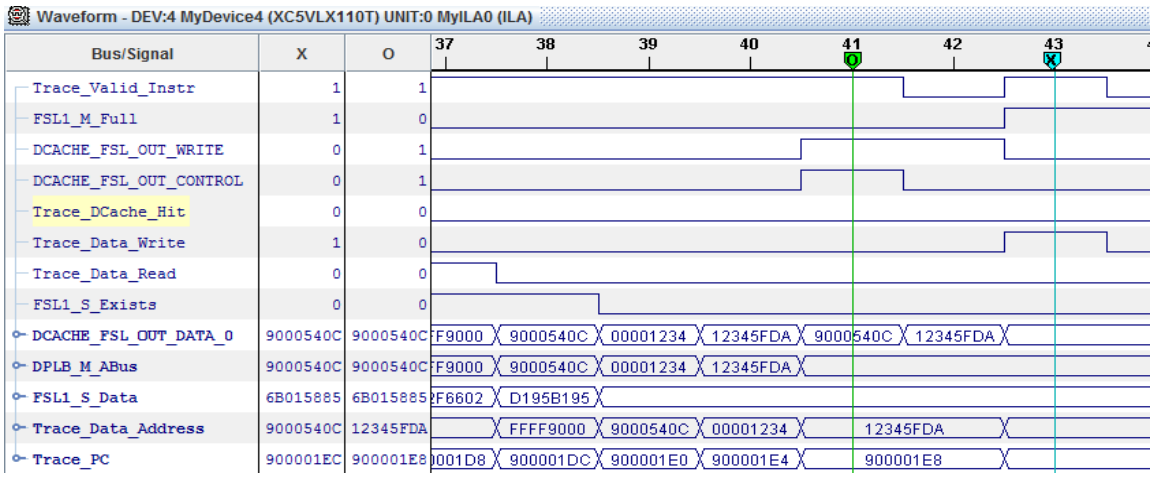


Figure 3.9 Non consecutive write in built-in WT data cache

Listing 4 shows the code to test non consecutive write in write through cache. The data 0x12345FDA is written to memory location 0x9000540C. The *swi* instruction in the macro follows an arithmetic instruction *addik*. Figure 3.8 and Figure 3.9 show the screenshot of consecutive and non–consecutive writes in built-in data cache respectively. Since write to the write through cache always result in write to the main memory, the bus protocol used in write through policy does not set hit signal to high even if it results in a hit. In the Figure 3.8, we can see a sequence of writes to main memory via DXCL bus. The pointers in Figure 3.8 show the write operation to memory address 0x900019F4, that has a delay of 14 clock cycles. The non consecutive write to memory address 0x9000540C shown in Figure 3.9 has 3 cycles delay. The reason for the difference in delay between consecutive and non consecutive write is discussed under the heading characterization of write through cache in section 4.3.

Listing 5 shows the code to test non consecutive and consecutive write in write through pCache. The first write operation to memory address 0x9000540C is a non consecutive write which follows an *addik* instruction. The second write instruction to

Listing 5. Sample code to test write in WT pCache

```
XIo_Out32(0x9000540C, 0x12345FDA); //Write to 0x9000540C
     imm        -28672
     addik      r4, r0, 21516
     imm        4660
     addik      r3, r0, 24538        // 0x12345FDA
     swi        r3, r4, 0

asm("swi  r5, r19, 48");            // Write to 900022AC
```

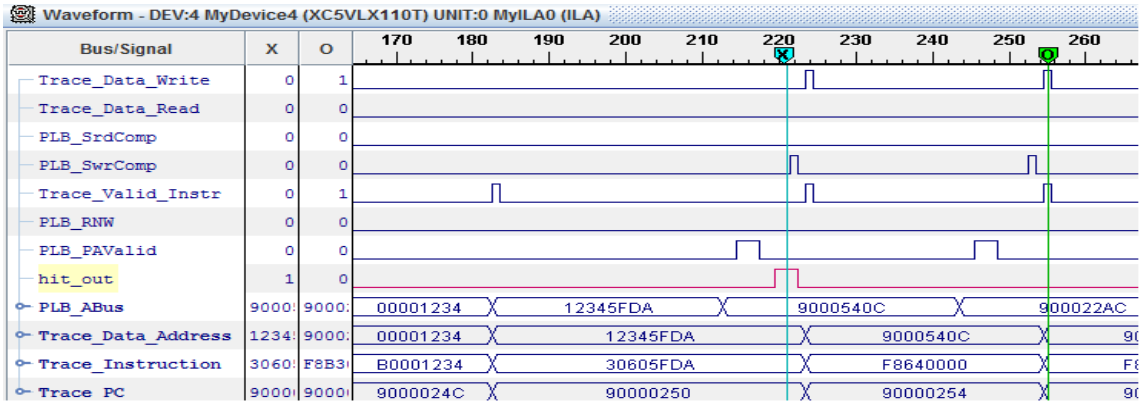


Figure 3.10 Non consecutive and consecutive write in pCache

memory address 0x900022AC is a consecutive write to main memory. Figure 3.10 shows the screenshot of non-consecutive and consecutive write in the pCache model. The pointers show a write transaction to memory address 0x9000540C. We can see, both the types take 11 cycles only to complete the write into main memory via PLB.

3.4 Validation of Write back mode

3.4.1 Read transactions

Listing 6. Sample code to test read hit in WB cache

```
XIo_In32(0x90006800);          //Read miss
XIo_In32(0x90006804);          //Read hit
```

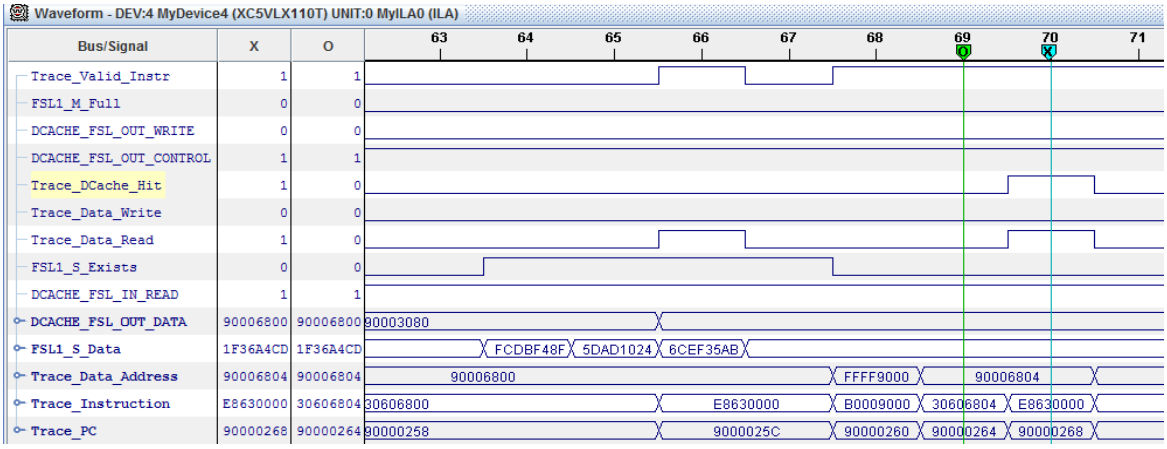


Figure 3.11 Read hit in built-in WB data cache

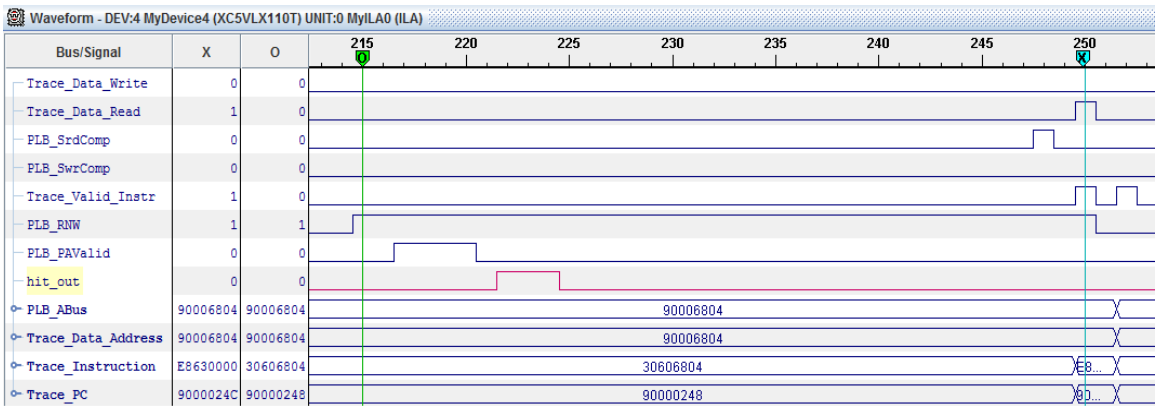


Figure 3.12 Read hit in pCache

Listing 6 shows the code to test read hit in 4 word write back cache. The first read operation to memory address 0x90006800 results in a miss. The corresponding block of data from main memory is fetched into the data cache. The second read operation to memory address 0x90006804 results in a hit in data cache. Figure 3.11 and Figure 3.12 shows the screenshot of read operation to memory address 0x9000540C, in built-in write back data cache and pCache respectively. The transaction results in a cache hit in both the models. The pointers X and O show the start and end of the read transaction. We note from the screenshot, the read hit in built-in data cache via DXCL bus takes 2 cycles only to be executed and 35 cycles via PLB in pCache model.

Listing 7. Sample code to test read miss (1D) in WB cache

```
XIo_In32(0x9000230C);                  //Read miss
XIo_Out32(0x90002300, 0x8544539A); //Write hit
XIo_In32(0x90005400);                  //Read miss 1D
```

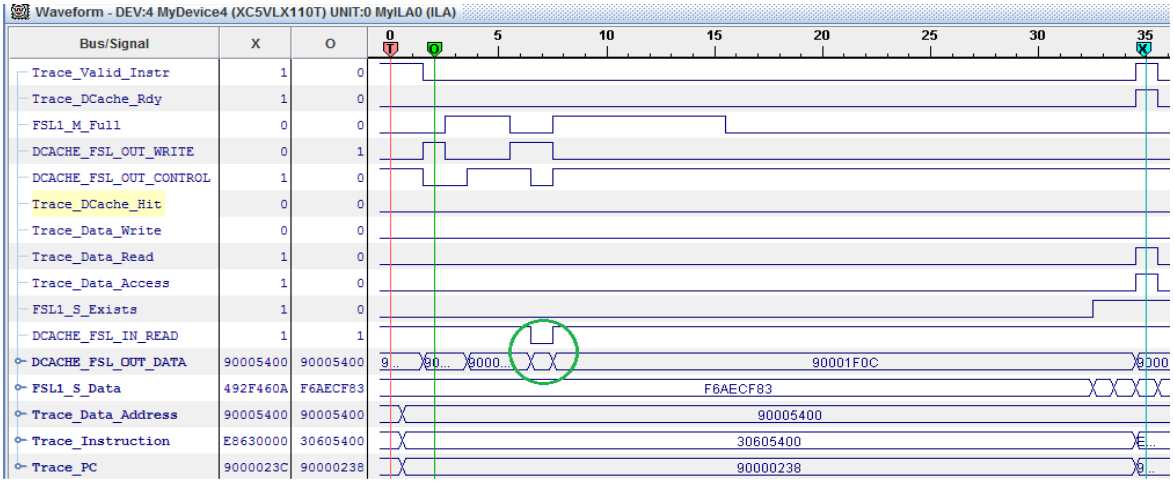


Figure 3.13 Read miss (1 dirty) in built-in WB data cache

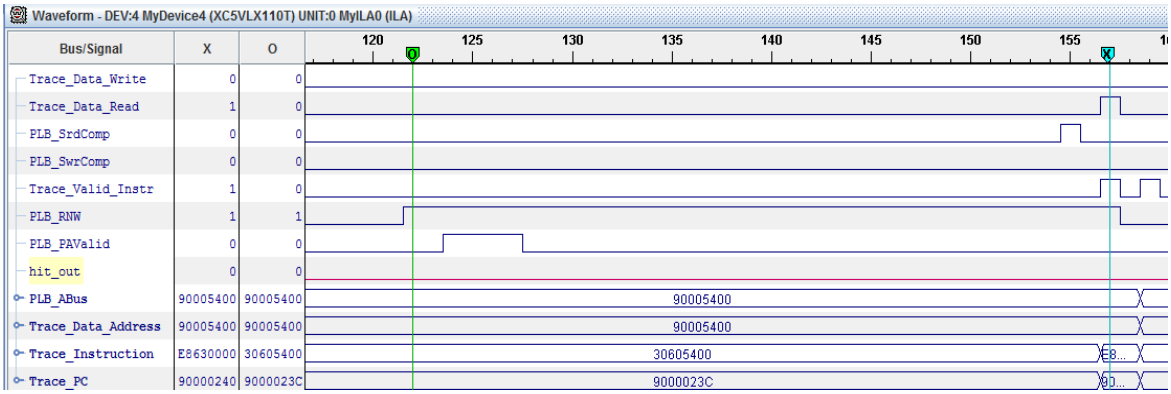


Figure 3.14 Read miss (1 dirty) in pCache

Listing 7 shows the code to test read miss with 1 dirty bit in 4 word write back data cache. The first read operation to memory address 0x9000230C results in a cache miss and a block of data is fetched into the data cache. The second write operation modifies the data in the cache block. The next read operation to memory address 0x90005400 results in a miss

with 1 dirty word. Figure 3.13 and Figure 3.14 shows the screenshot of read transaction in built-in write back data cache and pCache model respectively. The read to memory address 0x90005400 results in a miss with 1 dirty bit in both the models. Note the signal *hit_out* in Figure 3.14, which is low in the region between the pointers X and O. The circled portion in Figure 3.13 shows the modified data in the cache block being written back to main memory before it is replaced with a new block. We see the time for read miss with 1 dirty bit to be 33 cycles in built-in data cache and 35 cycles in pCache model.

Listing 8. Sample code to test read miss notD in WB cache

```
XIo_In32(0x90003120);          //Read miss
XIo_In32(0x9000302C);          //Read miss not Dirty
```

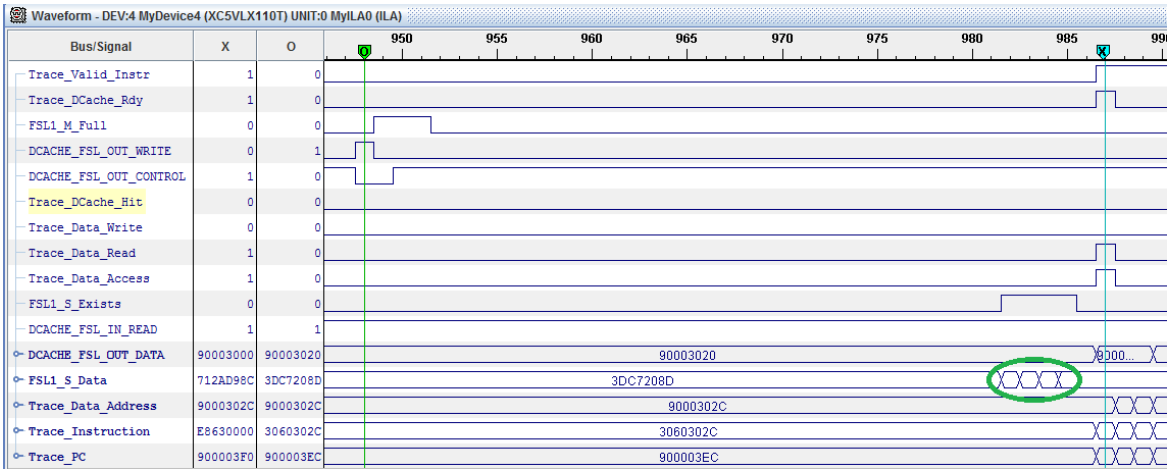


Figure 3.15 Read miss not dirty in built-in WB data cache

Listing 8 shows the code to test read miss no dirty in 4 word write back data cache. The first read operation to memory address 0x90003120 results in a read miss in data cache. The next read operation to memory address 0x9000302C which maps to the same cache block, results in a read miss with no dirty bit. Figure 3.15 and Figure 3.16 shows the screenshot of read transaction in built-in write back data cache and pCache models

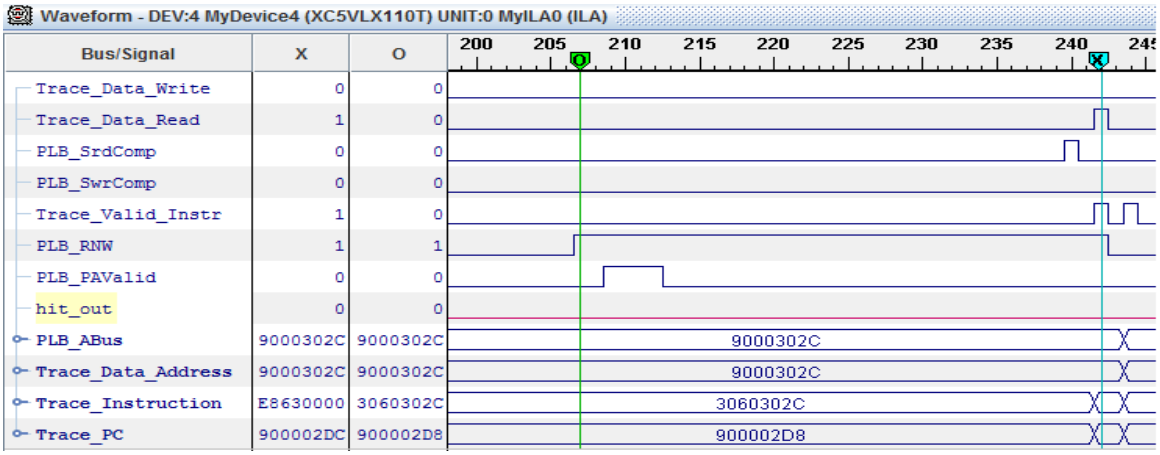


Figure 3.16 Read miss not dirty in pCache

respectively. The read to memory address 0x9000302C results in a miss with no dirty bits in both the models. Since the block of data that is replaced in the cache is not dirty, we do not find any data being written back to main memory in Figure 3.15. The new block of data that is being fetched from the main memory to data cache is shown by the circled region in Figure 3.15. We see the time for read miss with no dirty bits to be 39 cycles in built-in data cache and 35 cycles in pCache model.

3.4.2 Write transactions

Listing 9. Sample code to test write hit in WB cache

```
XIo_In32(0x90003004);                  //Read miss
XIo_Out32(0x90003000, 0x12825678); //Write hit
```

Listing 9 shows the code to test write hit in 4 word write back data cache. The first read operation to memory address 0x90003004 results in a miss in data cache. The corresponding block of data is fetched from main memory to data cache. The write operation to memory address 0x90003000 results in a hit in data cache. Figure 3.17 and Figure 3.18 shows the screenshot of write transaction in built-in write back data cache and pCache models

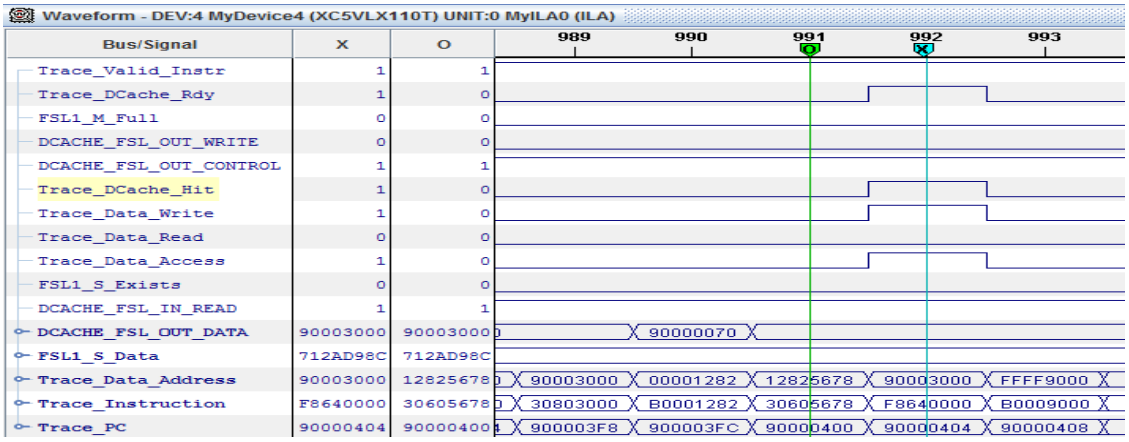


Figure 3.17 Write hit in built-in WB data cache

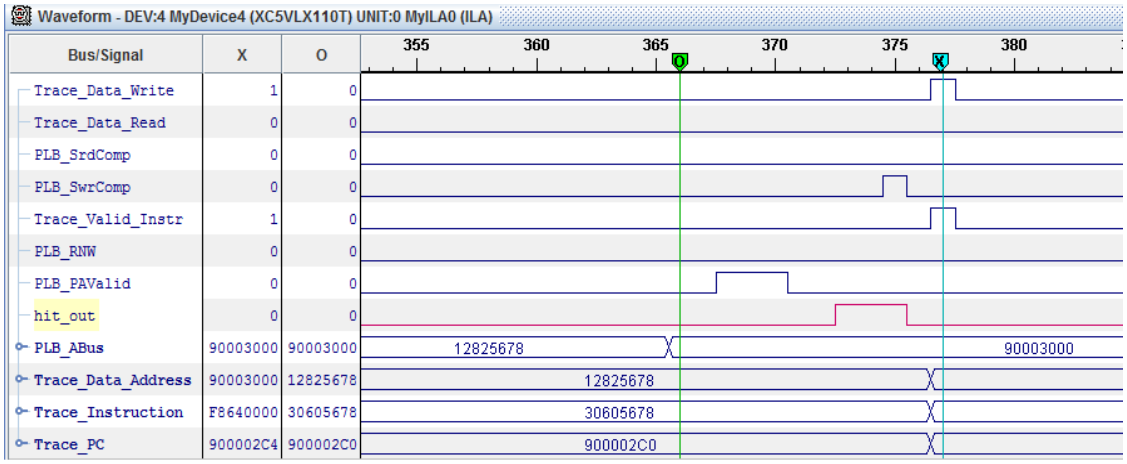


Figure 3.18 Write hit in pCache

respectively. The write to address 0x90003000 in main memory results in a hit in both the models. The hit signals, *Trace_Dcache_hit* and *hit_out* are set high in the region between the pointer X and O. We see the time for write hit to be 2 cycles in built-in cache and 11 cycles in pCache model.

Listing 10. Sample code to test write miss (1D) in WB cache

```
XIo_Out32(0x90005410, 0x12300678); //Write hit
XIo_Out32(0x90003010, 0x12300678); //Write miss 1D
```

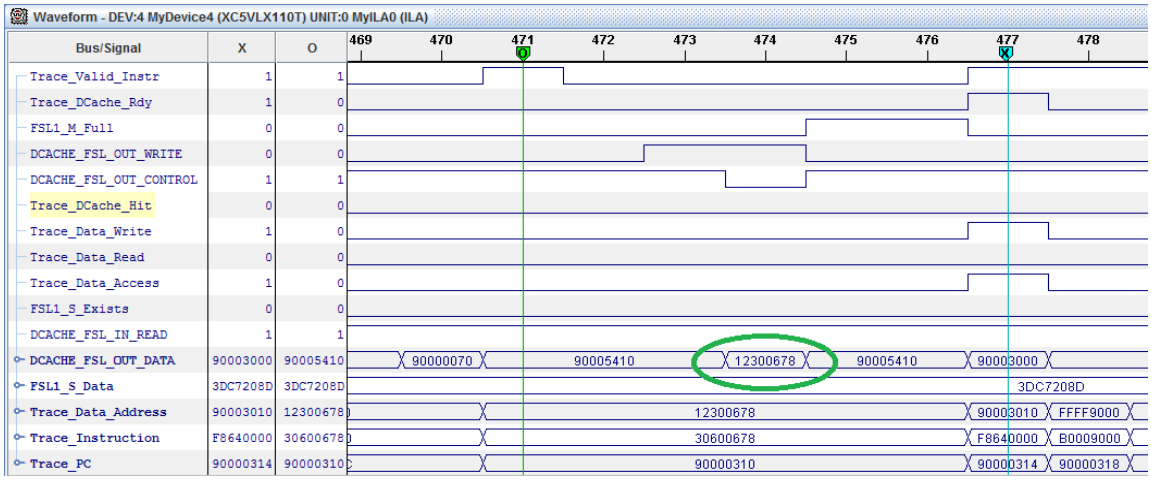


Figure 3.19 Write miss 1 dirty in built-in WB data cache

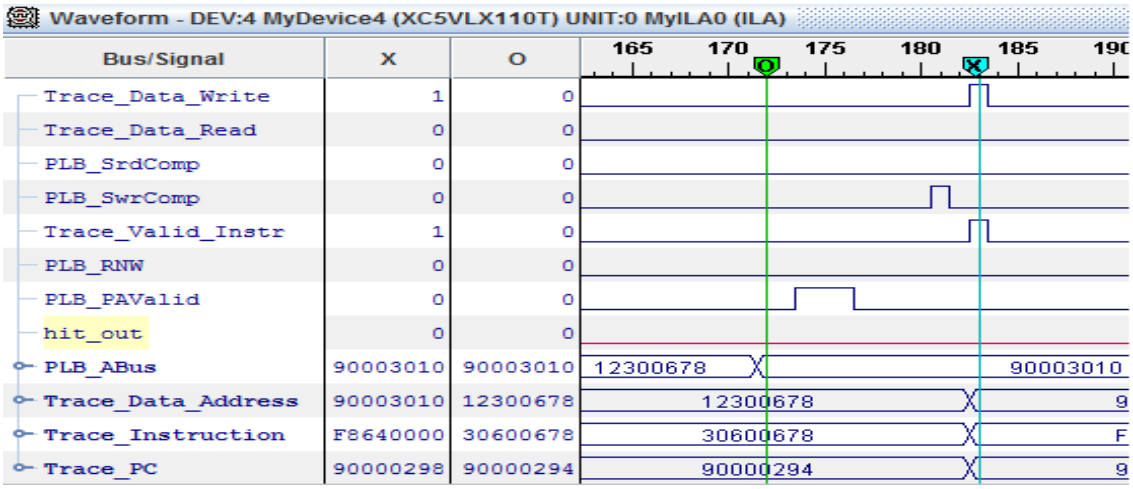


Figure 3.20 Write miss 1 dirty in pCache

Listing 10 shows the code to test write miss with 1 dirty bit in 4 word write back data cache. The first write operation to memory address 0x90005410 results in a cache hit and the data is modified in the cache. The second write operation to memory address 0x90003010 which maps to the same cache block, results in a miss with 1 dirty bit. Figure 3.19 and Figure 3.20 shows the snapshot of write transaction in built-in write back data cache and pCache models respectively. The write to address 0x90003010 in main memory results in a miss in cache with 1 dirty bit. The circled portion in Figure 3.19 shows the modified data in

cache being written back to main memory before it is replaced by a new block. The *hit_out* signal in the pCache model shown in Figure 3.20 is low in the region between the pointers X and O. We see the time for write miss with 1 dirty bit to be 7 cycles in built-in cache and 11 cycles in pCache model.

Listing 11. Sample code to test write miss notD in WB cache

```
XIo_Out32(0x90003180, 0x12300678); //Write miss
XIo_Out32(0x90003080, 0x12345FDA); //Write miss notD
```

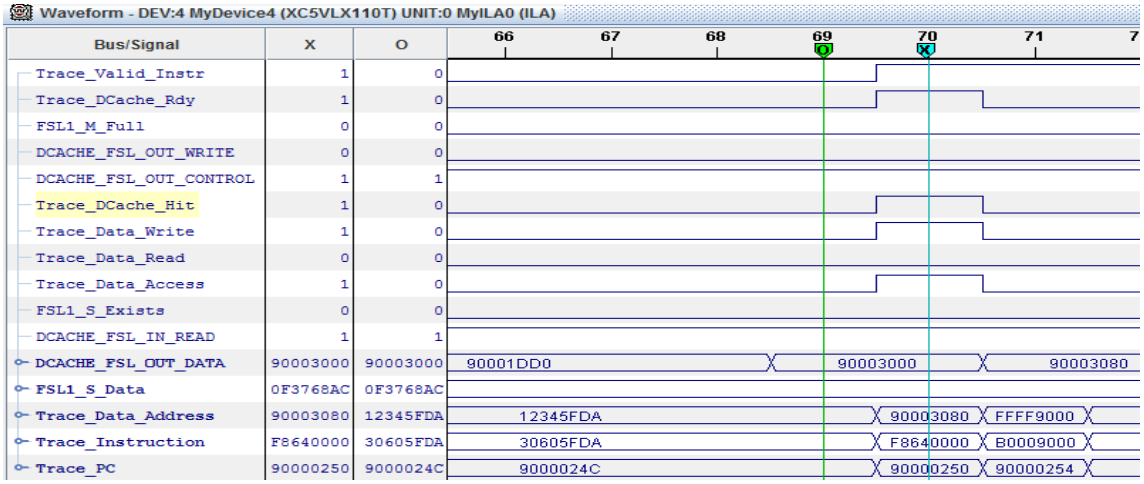


Figure 3.21 Write miss not dirty in built-in WB data cache

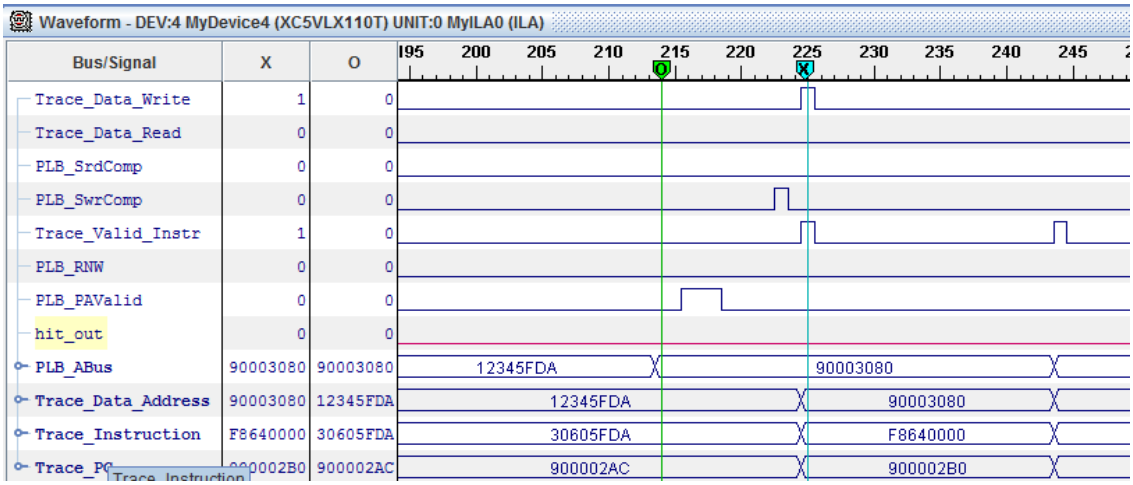


Figure 3.22 Write miss not dirty in pCache

Listing 11 shows the code to test write miss with no dirty bits in write back data cache. The first write operation to memory address 0x90003180 results in a miss in data cache. The write operation to memory address 0x90003080 which maps to the same cache block, results in a miss with no dirty bits. Figure 3.21 and Figure 3.22 shows the screenshot of write transaction in write back data cache and pCache models respectively. The write to address 0x90003080 in main memory results in a miss with no dirty bits in both the cache models. Upon write miss with no dirty bits, the bus protocol used in write back policy sets the hit signal *Trace_Dcache_hit* to high in the built-in data cache model. This can be seen in Figure 3.21shown by the pointer X. But in the case of pCache model, the *hit_out* signal is at logic 0 during similar scenario. We see the time for write miss with no dirty bits to be 2 cycles in built-in data cache and 11 cycles in pCache model.

3.5 Validation using Software debugger

The Xilinx Microprocessor Debugger (XMD) provided by EDK is used to debug programs running on the FPGA. We used the sample code shown in Listing 12 to validate the correctness of pCache model. Software breakpoints were set and the contents of memory mapped registers were tracked using the software debugger.

Listing 12. Sample code used in functional validation

XIo_In32(0x90005400);	//Read miss	(A)
XIo_Out32(0x9000540C, 0x12345FDA);	//Write hit	(B)
XIo_Out32(0x90005410, 0x12300678);	//Write miss	(C)
XIo_In32(0x90005408);	//Read hit	(D)
XIo_In32(0x90005420);	//Read miss	(E)
XIo_Out32(0x90005434, 0x12300678);	//Write miss	(F)

The first memory transaction A, reads address 0x90005400 which is not cached. This results in a cache read miss. Transaction B writes data 0x12345FDA into the address 0x9000540C. Since the cache block holds 4 words, (0x90005400, 0x90005404, 0x90005408 and 0x9000540C) transaction B results in a write hit. Transaction C writes data into memory location 0x90005410 which is not present in cache and results in a write miss. Transaction D results in a read hit on address 0x90005408, which is present in the cache. Transaction E and F reads memory address 0x90005420 and writes data into address 0x90005434 respectively, which are not cached resulting in read miss and write miss.

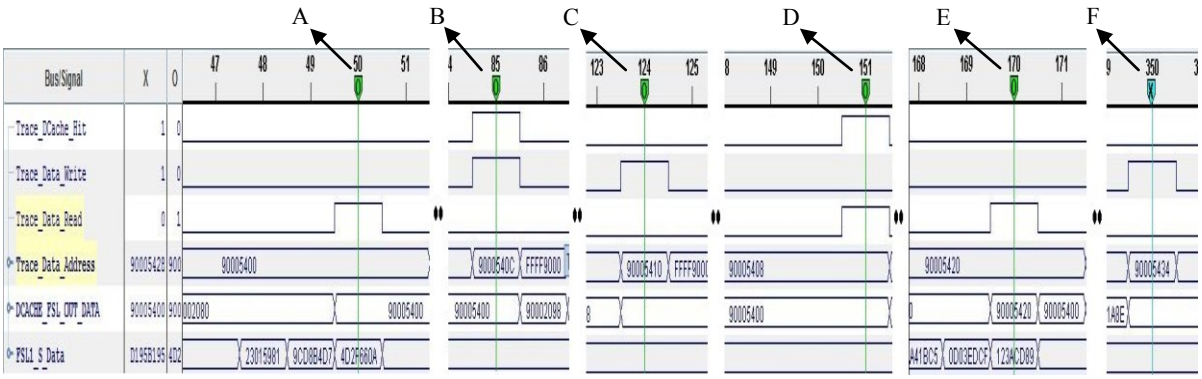


Figure 3.23 Result of the test code in 4 word built-in write back data cache model

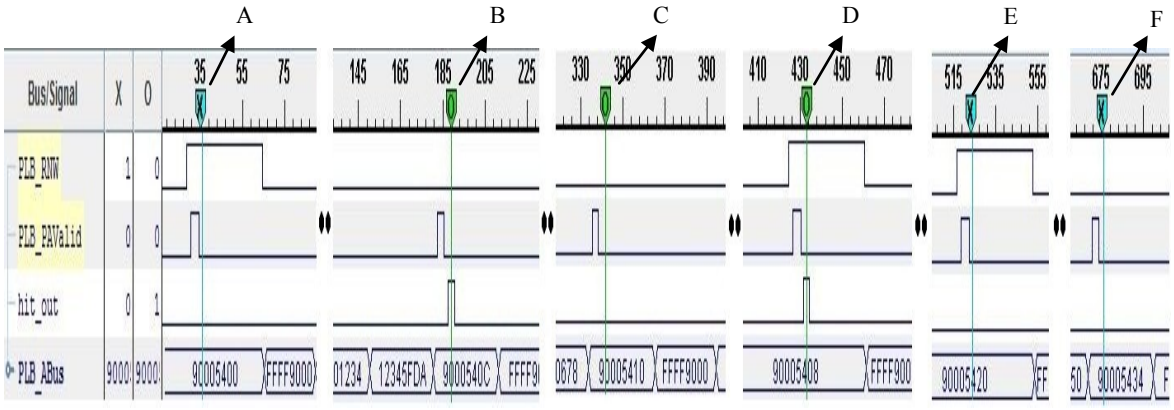


Figure 3.24 Result of the test code in 4 word write back pCache based model.

The cache hit signal *Trace_Dcache_Hit* in the reference built-in cache design was compared to the *Hit_out* signal of the pCache model. The screenshot of internal signals resulting from the execution of the test program in listing 1 is shown in Figure 3.23.

- Pointer A shows read miss at address 0x90005400
- Pointer B shows write hit at address 0x9000540C
- Pointer C shows write miss at address 0x90005410
- Pointer D shows read hit at address 0x90005408
- Pointer E shows read miss at address 0x90005420
- Pointer F shows write miss at address 0x90005434

Figure 3.24 shows the screenshot of signals viewed through Chipscope while simulating the pCache model. We note that the pointers A to F follow the same sequence as in the built-in cache. The hits in pCache are generated for the same memory addresses as in built-in cache design. This demonstrates the functional correctness of the pCache-based model.

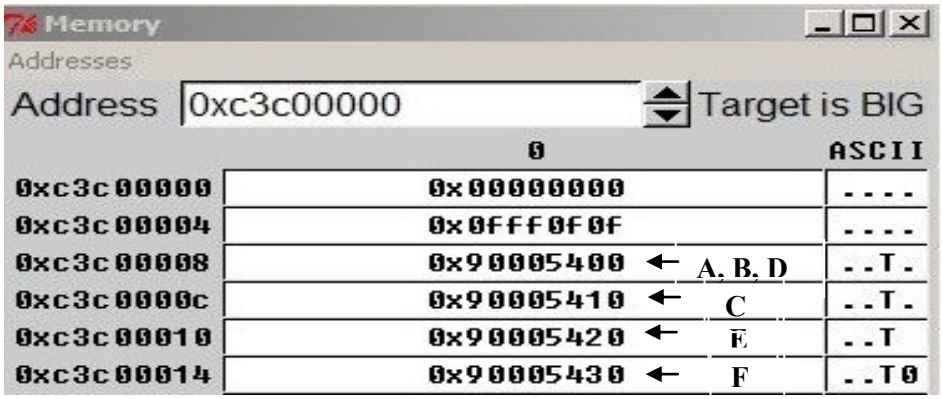


Figure 3.25 Contents of the pCache.

Figure 3.25 shows the contents of the data cache using XMD, after executing the test program in pCache model. Addresses of the first location of the block of memory being

accessed are written to the slave register dedicated to the corresponding cache block. The block address for the data being accessed in instructions A, B and D is 0x90005400. The block addresses for the data being accessed in instructions C, E and F are 0x90005410, 0x90005420 and 0x90005430 respectively. Theoretically, the bus analyzer can be used to determine the state of the cache. However, since the bus analyzer has small memory, which allows tracing for only upto 1024 cycles, the cache state must be computed manually for long runs of software execution. Moreover, the software developer needs to be familiar with hardware design and signal tracing, which is clearly impractical. Using pCache-based model, the entire content of the data cache is easily observable at run-time using a software debugger. The software developer does not need to be familiar with any hardware details.

CHAPTER 4

Timing Analysis

In this chapter, we present cache timing analysis using the example of a MicroBlaze based system [9].

The pCache based model has a longer software execution time due to two reasons.

- Accessing data from the off-chip main memory via PLB takes longer than access via CacheLink.
- The absence of an on-chip data cache.

We will define the number of additional cycles taken by the pCache-based model, compared to the built-in cache design, as the *overhead*. We also define T_{pcache} as the number of cycles taken for software execution in the pCache-based model. T_{pcache} can be easily measured using a timer peripheral. The timer is reset and started before a given block of code, and stopped after. The *overhead* cycles due to cache hit and miss are calculated, and subtracted from T_{pcache} to determine the estimated run time T_{est}.

$$T_{est} = T_{pcache} - \{H * (T_{plb} - T_{xcl-hit}) + M * (T_{plb} - T_{xcl-miss})\}$$

Equation 1

In Equation 1, T_{plb} is the average time to access a single word from main memory. $T_{xcl-hit}$ is the average time to access data in the built-in data cache and $T_{xcl-miss}$ is the average time to access data not in cache, over the XCL bus. The overhead due to a single cache hit in pCache is $T_{plb} - T_{xcl-hit}$. This factor is multiplied with the number of hits H in pCache to determine the total overhead due to cache hits. Similarly, the overhead due to each cache miss in pCache $T_{plb} - T_{xcl-miss}$ is multiplied with the number of misses M to determine the overhead due to cache misses.

4.1 Bus Characteristics

In order to model the estimated software execution time (T_{est}) in pCache based model, we need to understand the behavior of PLB and XCL buses. The following sub sections provide details about bus protocol and main memory access delay for both the buses. Chipscope Pro bus analyzer [10] is used to observe the bus activity and to measure the main memory access time.

4.1.1 PLB Behavior

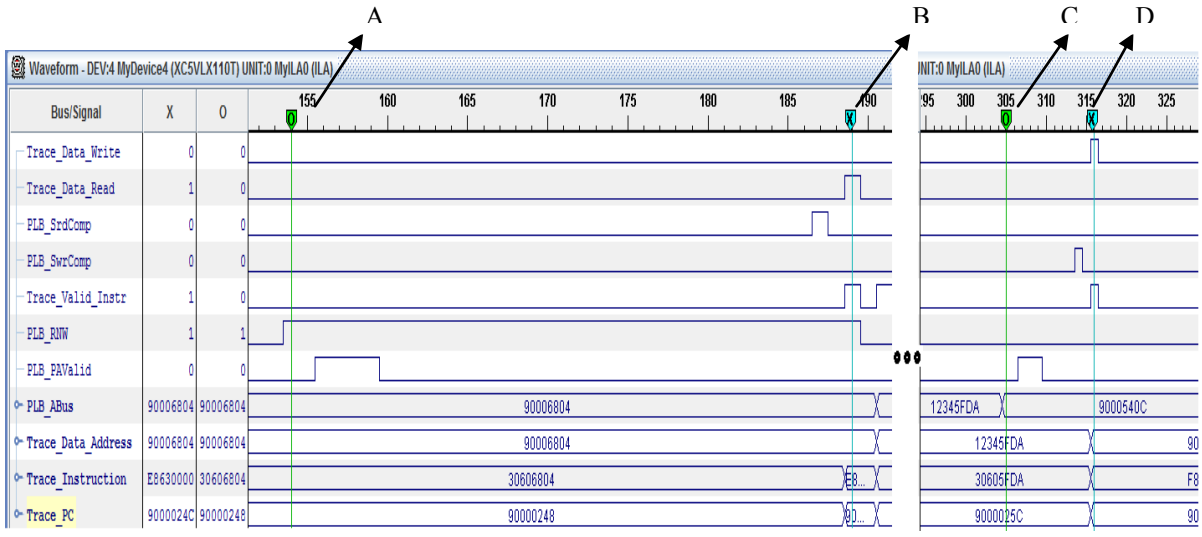


Figure 4.1 Read and write operations to off-chip DDR memory via PLB

Figure 4.1 shows snapshot of PLB transactions. It shows a valid read and write transaction to DDR main memory. To measure the number of cycles for an operation, we count from the start of *PLB_PAValid* the valid address signal (shown by pointers A and C) till where the instruction gets executed (shown by pointers B and D). The delay in main memory can vary due to factors such as bus turnaround time, column address latency and refresh rate. Therefore, the measured time for read/write operation is not always constant. We model the PLB access delay by averaging the read and write times for several transactions which gives us 35 cycles for a read and 11 cycles for a write transaction.

4.1.2 XCL Behavior

The XCL bus is a point to point FSL connection between the MicroBlaze processor and the DDR main memory. During data miss, the processor uses this connection to fetch data from the off-chip main memory. In data cache with write through policy, the cache link uses DXCL protocol [9] to communicate with the main memory. Upon read miss, the DXCL protocol follows critical word first method to read data from the main memory. By this method, the requested word is first read followed by the remaining 3 or 7 words. Upon read hit, the hit signal is asserted, however during write hit the hit signal is not set to high. The writes to data cache in write through mode will always result in a write over cache link regardless of whether there was a hit or miss as explained in section 2.3.

In the case of write back policy, the cache link uses DXCL2 protocol [9]. Upon read miss, the DXCL2 protocol follows linear fetch method to read the new block from main memory. In linear fetch, the data words are read in the order in which they are stored in the main memory. Write is performed only to the cache and the dirty bit is set high. Upon write miss the modified cache block is written back to the main memory before getting replaced.

4.2 Parameterized cache timing model

The parameters needed to calculate the overhead in write through cache timing model are as follows. The first four parameters are variables obtained from respective counters in the pCache module. The last six parameters are constants measured using the bus analyzer, used in overhead calculation. We will discuss the last six parameters in detail in section 4.5.

- RH : Number of read hits
- RM : Number of read miss
- NC_{wr} : Number of non-consecutive writes
- C_{wr} : Number of consecutive writes
- N_{rf_cyc} : Number of refresh cycles
- T_{rd} : Average time for read via PLB
- T_{wr} : Average time for write via PLB
- T'_{rh} : Average time for read hit via XCL
- T'_{rm} : Average time for read miss via XCL
- $T'_{nc\text{-}wr}$: Average time for non consecutive writes via XCL
- $T'_{c\text{-}wr}$: Average time for consecutive writes via XCL

Additional parameters are needed to calculate the overhead in a write back cache timing model as follows. The first five parameters are variables obtained from respective counters and the next five parameters are assigned constant values, based on measurements using the bus analyzer. The time for write/read miss in a write back cache depends on the number of dirty bits that are set, which we denote by *x*.

- WH : Number of write hits
- WM_{nD} : Number of write miss not dirty
- WM_{xD} : Number of write miss with x dirty bits
- RM_{nD} : Number of read miss not dirty
- RM_{xD} : Number of read miss with x dirty bits
- T'_{wh} : Average time for write hit via XCL
- $T'_{wm\text{-}nD}$: Average time for write miss not dirty via XCL
- $T'_{wm\text{-}xD}$: Average time for write miss with *x* dirty bits via XCL
- $T'_{rm\text{-}nD}$: Average time for read miss not dirty via XCL
- $T'_{rm\text{-}xD}$: Average time for read miss with *x* dirty bits via XCL

4.3 Characterization of the Write through Cache

Table 2 Average XCL delay in Write through cache

Memory Access	Delay (cycles)
Read hit	2
Read miss	32
Non-Consecutive Write	3
Consecutive Write	14

Table 2 shows the memory access type and its average delay via XCL in write through cache. The total overhead in the write through cache is computed by the following equation:

$$O_{WT} = \{[RH * (T_{rd} - T'_{rh})] + [RM * (T_{rd} - T'_{rm})] + [NC_{wr} * (T_{wr} - T'_{nc-wr})] + [C_{wr} * (T_{wr} - T'_{c-wr})] + N_{rf-cyc}\}$$

Equation 2

The first component on the RHS in Equation 2, $RH * (T_{rd} - T'_{rh})$, is the overhead due to read hits. The 2nd component, $RM * (T_{rd} - T'_{rm})$, is the overhead due to read miss. The 3rd and 4th components are the overhead associated with non-consecutive and consecutive writes respectively. The time for non-consecutive and consecutive writes are different because of delays such as write recovery time and write to write spacing time in DDR2 SDRAM. Hence, we have separate components for both the cases. The last component is the refresh cycles in the DDR memory. The datasheet mentions that the DDR memory, chosen in our design, is refreshed 8192 times within 64 ms at regular intervals of 7.8 us; and the length of each refresh cycle is 55 ns. This data is used by the pCache module to calculate N_{rf_cyc} for a given period.

4.4 Characterization of the Write back Cache

Table 3 Average XCL delay in 4-word Write back cache

Memory access	Delay (cycles)
Read hit	2
Read miss nD	34
Read miss D	36
Write hit	2
Write miss nD	2
Write miss D	7

Table 4 Average XCL delay in 8-word Write back cache

Memory access	Delay (cycles)
Read hit	2
Read miss nD	36
Read miss (1)D	39
Read miss (2-8)D	33
Write hit	2
Write miss nD	2
Write miss (1-7)D	7
Write miss (8)D	10

Table 3 and Table 4 shows the memory access type and its average delay in 4 word and 8 word write back caches respectively. The total overhead in write back cache is computed by the following equation,

$$O_{WB} = \{[RH * (T_{rd} - T'_{rh})] + [RM_{nD} * (T_{rd} - T'_{rm-nD})] + \sum[RM_{xD} * (T_{rd} - T'_{rm-xD})] + [WH * (T_{wr} - T'_{wh})] + [WM_{nD} * (T_{wr} - T'_{wm-nD})] + \sum[WM_{xD} * (T_{wr} - T'_{wm-xD})] + N_{rf-cyc}\}$$

Equation 3

The first component on the RHS, $RH * (T_{rd} - T'_{rh})$ is the overhead due to read hits. The 2nd component, $RM_{nD} * (T_{rd} - T'_{rm-nD})$ is the overhead due to read miss not dirty. The 3rd component, $\sum[RM_{xD} * (T_{rd} - T'_{rm-xD})]$ is the sum of overheads due to read miss dirty. The dirty bits range from 1 to 4 in cache with 4 word cache block and from 1 to 8 in 8 word cache block. The overhead due to write hit, write miss not dirty and write miss dirty are calculated similarly in the 4th, 5th and 6th components in Equation 3.

4.5 Timing Validation

In order to analyze the timing accuracy of pCache based model, we estimated the run time of several benchmarks. Equation 2 and Equation 3 are used to calculate the overhead in write through and write back caches respectively. Both the pCache based model and the reference design have a 32KB instruction cache. The error in timing estimation is calculated using Equation 4, where $T_{built\text{-}in}$ is the reference number of cycles for software execution with a built-in data cache.

$$Error\ \% = \frac{T_{est} - T_{builtin}}{T_{builtin}} * 100$$

Equation 4

To validate the timing of pCache based model, we run Dhrystone, Quicksort and JPEG benchmarks with different cache configuration. The result of timing estimation for each case is shown below.

Table 5 Performance estimation of Dhrystone in 4-word WT data cache

Cache Size	$T_{built\text{-}in}$	T_{pcache}	H_{wr}	H_{rd}	M_{rd}	M_{wr}	Over-head	T_{est}	Error %
64B	132,509	443,700	2,885	8,593	1,901	1,936	326,281	117,419	-11.4
128B	103,761	443,651	3,254	9,521	973	1,567	354,121	89,530	-13.7
256B	83,266	443,614	3,785	10,053	441	1,036	370,081	73,533	-11.7
512B	77,421	443,654	4,270	10,189	305	551	374,161	69,493	-10.2
1KB	73,853	443,643	4,342	10,280	214	479	376,891	66,752	-9.6
2KB	68,965	443,643	4,605	10,438	56	216	381,631	62,012	-10.1

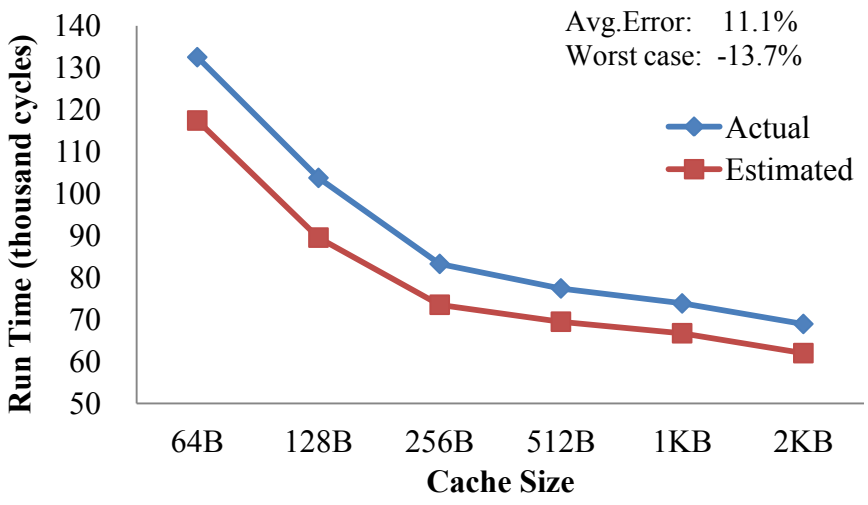


Figure 4.2 Plot of performance estimation of Dhrystone in 4-word WT data cache

Table 5 shows the result of running Dhrystone with different sizes of 4 word write through data cache. The average error in estimating the run time using pCache is 11.1% and has a maximum error of -13.7% for the data cache of size 128B. Figure 4.2 shows the plot of actual and estimated run time of Dhrystone in 4 word write through data cache.

Table 6 Performance estimation of Dhrystone in 8-word WT data cache

Cache Size	$T_{built-in}$	T_{pcache}	H_{wr}	H_{rd}	M_{rd}	M_{wr}	Over-head	T_{est}	Error %
64B	159,366	443,700	2,825	6,867	3,627	1,996	274,501	169,199	6.2
128B	110,576	443,651	3,226	8,448	2,046	1,595	321,931	121,720	10.1
256B	80,787	443,658	3,654	10,076	418	1,167	370,771	72,887	-9.8
512B	77,220	443,705	4,013	10,166	328	808	373,471	70,234	-9.0
1KB	74,334	443,662	4,146	10,239	255	675	375,661	68,001	-8.5
2KB	68,856	443,676	4,667	10,459	35	154	382,261	61,415	-10.8

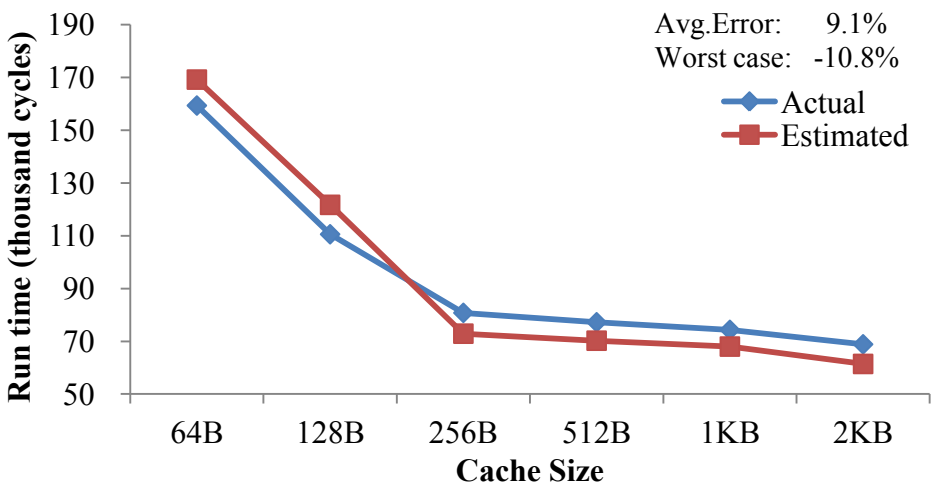


Figure 4.3 Plot of performance estimation of Dhrystone in 8-word WT data cache

Table 6 shows the result of running Dhrystone with different sizes of 8 word write through data cache. The average error in estimating the run time using pCache is 9.1% and has a maximum error of -10.8% for the data cache of size 2KB. Figure 4.3 shows the plot of actual and estimated run time of Dhrystone in 8 word write through data cache.

Table 7 Performance estimation of Quicksort in 4-word WT data cache

Cache Size	$T_{built-in}$	T_{pcache}	H_{wr}	H_{rd}	M_{rd}	M_{wr}	Over-head	T_{est}	Error %
64B	194,502	505,986	2,453	8,809	3,491	2,196	320,200	185,786	-4.5
128B	144,981	505,872	3,335	10,345	1,955	1,314	366,279	139,593	-3.7
256B	103,382	505,983	3,839	11,664	636	810	405,850	100,133	-3.1
512B	85,720	505,866	4,394	12,153	147	255	420,519	85,347	-0.4
1KB	84,091	505,872	4,467	12,189	111	182	421,599	84,273	0.2
2KB	84,031	505,950	4,475	12,191	109	174	421,660	84,290	0.3

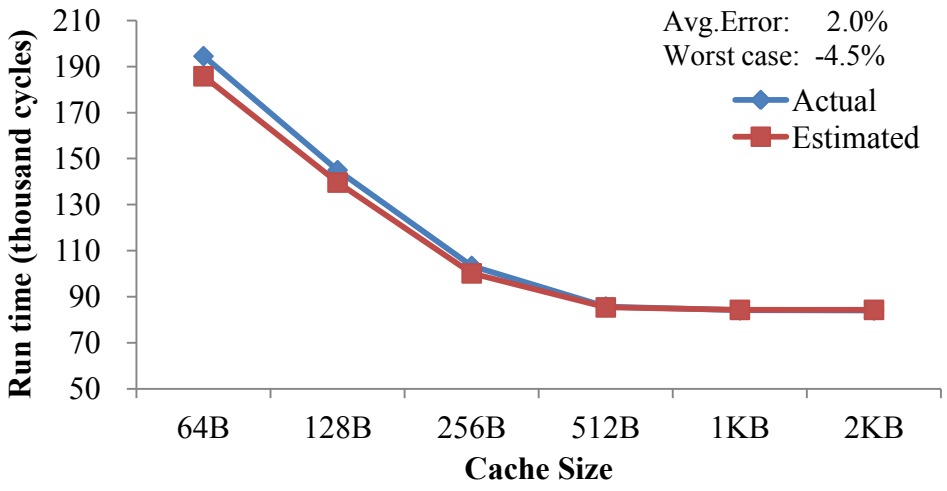


Figure 4.4 Plot of performance estimation of Quicksort in 4-word WT data cache

Table 7 shows the result of running Quicksort with different sizes of 4 word write through data cache. The average error in estimating the run time using pCache is 2.0% and has a maximum error of -4.5% for the data cache of size 64B. Figure 4.4 shows the plot of actual and estimated run time of Quicksort in 4 word write through data cache.

Table 8 Performance estimation of Quicksort in 8-word WT data cache

Cache Size	$T_{built-in}$	T_{pcache}	H_{wr}	H_{rd}	M_{rd}	M_{wr}	Over-head	T_{est}	Error %
64B	202,260	505,966	1,967	7,748	4,552	2,682	288,370	217,596	7.6
128B	143,522	505,866	3,451	10,061	2,284	1,198	357,894	147,972	3.1
256B	101,581	505,953	3,846	11,590	710	803	403,630	102,323	0.7
512B	85,243	505,953	4,451	12,209	91	198	422,200	83,753	-1.7
1KB	83,898	505,933	4,524	12,227	73	125	422,740	83,193	-0.8
2KB	83,854	505,828	4,533	12,230	70	116	422,829	82,999	-1.0

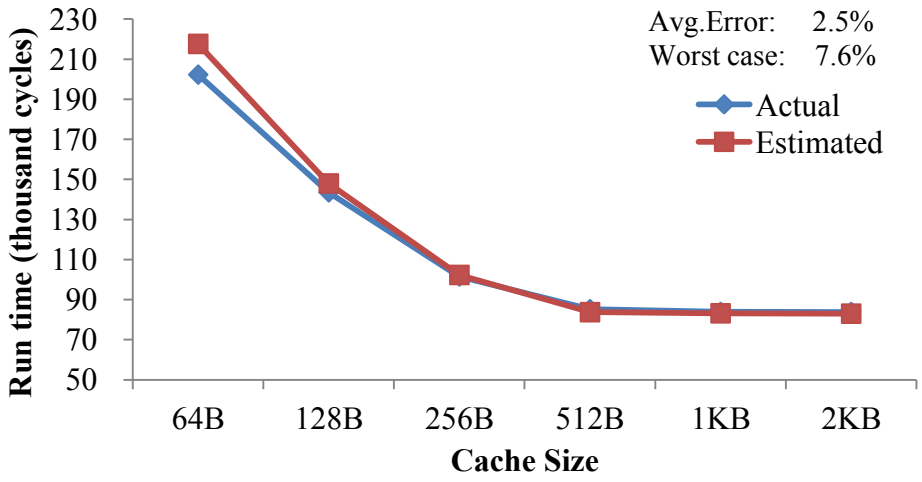


Figure 4.5 Plot of performance estimation of Quicksort in 8-word WT data cache

Table 8 shows the result of running Quicksort with different sizes of 8 word write through data cache. The average error in estimating the run time using pCache is 2.5% and has a maximum error of 7.6% for the data cache of size 64B. Figure 4.5 shows the plot of actual and estimated run time of Quicksort in 8 word write through data cache.

Table 9 Performance estimation of JPEG in 4-word WT data cache

Cache Size	$T_{built-in}$ (k cycles)	T_{pcache} (k cycles)	H_{wr}	H_{rd}	M_{rd}	M_{wr}	Over-head (k cycles)	T_{est} (k cycles)	Err %
64B	24,582.2	63,158.6	311,517	1,008,864	382681	251,025	38,890.8	24,267.8	-1.3
128B	18,497.2	63,158.5	410,002	1,191,142	200403	152,540	44,359.1	18,799.4	1.6
256B	16,191.5	63,159.0	439,216	1,261,630	129915	123,326	46,473.8	16,685.3	3.0
512B	14,227.0	63,158.3	472,034	1,319,342	72203	90,508	48,205.1	14,953.1	5.1
1KB	13,385.2	63,158.6	491,642	1,343,879	47666	70,900	48,941.3	14,217.4	6.2
2KB	12,487.6	63,158.6	526,984	1,371,391	20154	35,558	49,766.6	13,392.0	7.2

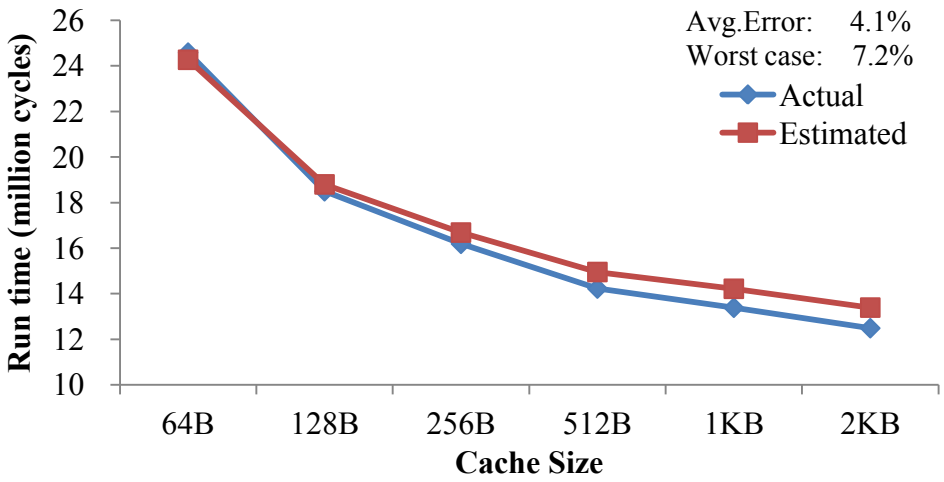


Figure 4.6 Plot of performance estimation of JPEG in 4-word WT data cache

Table 9 shows the result of running JPEG with different sizes of 4 word write through data cache. The average error in estimating the run time using pCache is 4.1% and has a maximum error of 7.2% for the data cache of size 2KB. Figure 4.6 shows the plot of actual and estimated run time of JPEG in 4 word write through data cache.

Table 10 Performance estimation of JPEG in 8-word WT data cache

Cache Size	**$T_{built-in}$** (k cycles)	**T_{pcache}** (k cycles)	**H_{wr}**	**H_{rd}**	**M_{rd}**	**M_{wr}**	**Over-head** (k cycles)	**T_{est}** (k cycles)	**Err %**
64B	28,630.9	63,159.0	288,024	940,611	450934	274,518	36,843.2	26,315.8	-8.1
128B	20,711.9	63,159.0	382,918	1,146,863	244682	179,624	43,030.8	20,128.2	-2.8
256B	17,202.7	63,158.5	432,826	1,242,049	149496	129,716	45,886.4	17,272.1	0.4
512B	14,553.2	63,158.8	469,796	1,316,503	75042	92,746	48,120.0	15,038.8	3.3
1KB	13,521.8	63,158.3	490,303	1,345,951	45594	72,239	49,003.4	14,154.9	4.7
2KB	12,475.9	63,158.8	522,755	1,374,198	17347	39,787	49,850.8	13,308.0	6.7

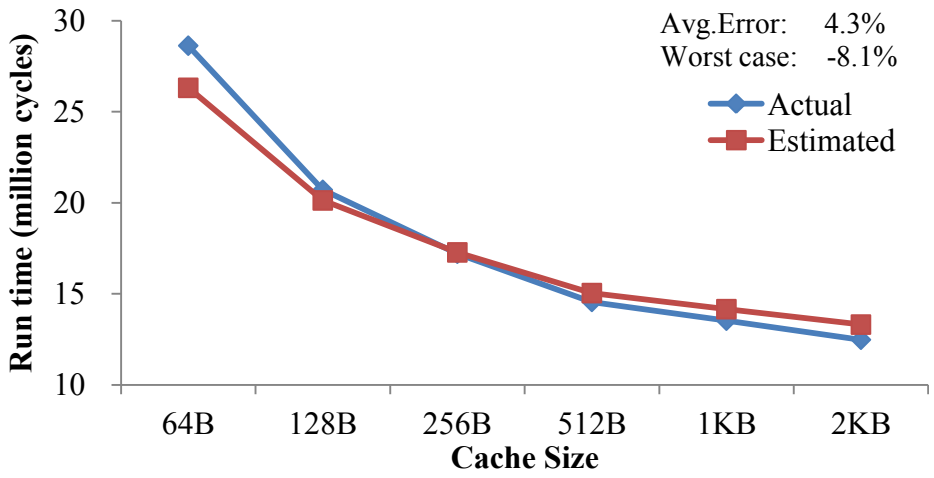


Figure 4.7 Plot of performance estimation of JPEG in 8-word WT data cache

Table 10 shows the result of running JPEG with different sizes of 8 word write through data cache. The average error in estimating the run time using pCache is 4.3% and has a maximum error of -8.1% for the data cache of size 64B. Figure 4.7 shows the plot of actual and estimated run time of JPEG in 8 word write through data cache.

Table 11 Performance estimation of Dhrystone in 4-word WB data cache

Cache Size	$T_{built\text{-}in}$	T_{pcache}	H_{wr}	H_{rd}	M_{rd}	M_{wr}	Over-head	T_{est}	Error %
64B	135,130	443,609	3,556	8,321	2173	1265	317,252	126,357	-6.5
128B	90,192	443,662	4,310	9,516	978	511	359,249	84,413	-6.4
256B	69,015	443,662	4,472	10,184	310	349	381,447	62,215	-9.9
512B	62,991	443,717	4,469	10,274	220	142	382,222	58,495	-7.1
1KB	59,581	443,609	4,679	10,359	135	124	388,015	55,594	-7.1
2KB	55,799	443,676	4,784	10,470	24	37	392,099	51,577	-7.6

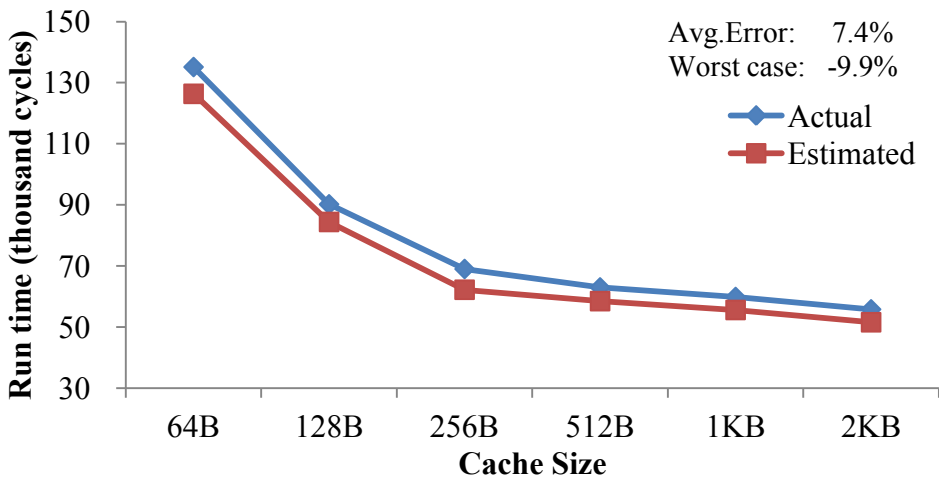


Figure 4.8 Plot of performance estimation of Dhrystone in 4-word WB data cache

Table 11 shows the result of running Dhrystone with different sizes of 4 word write back data cache. The average error in estimating the run time using pCache is 7.4% and has a maximum error of -9.9% for the data cache of size 256B. Figure 4.8 shows the plot of actual and estimated run time of Dhrystone in 4 word write back data cache.

Table 12 Performance estimation of Dhrystone in 8-word WB data cache

Cache Size	$T_{built-in}$	T_{pcache}	H_{wr}	H_{rd}	M_{rd}	M_{wr}	Over-head	T_{est}	Error %
64B	203,886	443,662	3,349	6,207	4287	1472	243,518	200,144	-1.8
128B	124,893	443,601	3,971	8,104	2390	850	310,247	133,354	6.8
256B	70,469	443,627	4,523	10,175	319	298	380,997	62,630	-11.1
512B	62,989	443,694	4,643	10,288	206	178	385,308	58,386	-7.3
1KB	61,304	443,700	4,672	10,326	168	149	386,567	57,133	-6.8
2KB	55,606	443,614	4,799	10,478	16	22	392,273	51,341	-7.7

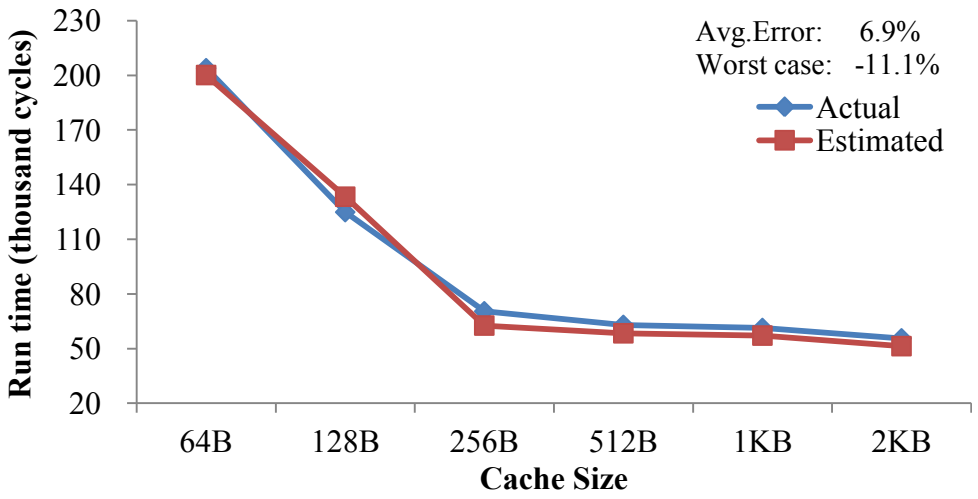


Figure 4.9 Plot of performance estimation of Dhrystone in 8-word WB data cache

Table 12 shows the result of running Dhrystone with different sizes of 8 word write back data cache. The average error in estimating the run time using pCache is 6.9% and has a maximum error of -11.1% for the data cache of size 256B. Figure 4.9 shows the plot of actual and estimated run time of Dhrystone in 8 word write back data cache.

Table 13 Performance estimation of Quicksort in 4-word WB data cache

Cache Size	$T_{built-in}$	T_{pcache}	H_{wr}	H_{rd}	M_{rd}	M_{wr}	Over-head	T_{est}	Error %
64B	146,104	505,899	3,411	9,470	2830	1238	335,585	150,314	2.9
128B	105,587	505,774	3,933	10,713	1587	716	397,403	108,371	-0.2
256B	71,754	505,968	4,283	11,898	402	366	436,890	69,078	-3.7
512B	60,025	505,895	4,567	12,225	75	82	448,849	57,046	-5.0
1KB	59,213	506,017	4,586	12,243	57	63	449,464	56,553	-4.5
2KB	59,177	506,011	4,587	12,244	56	62	449,508	56,503	-4.5

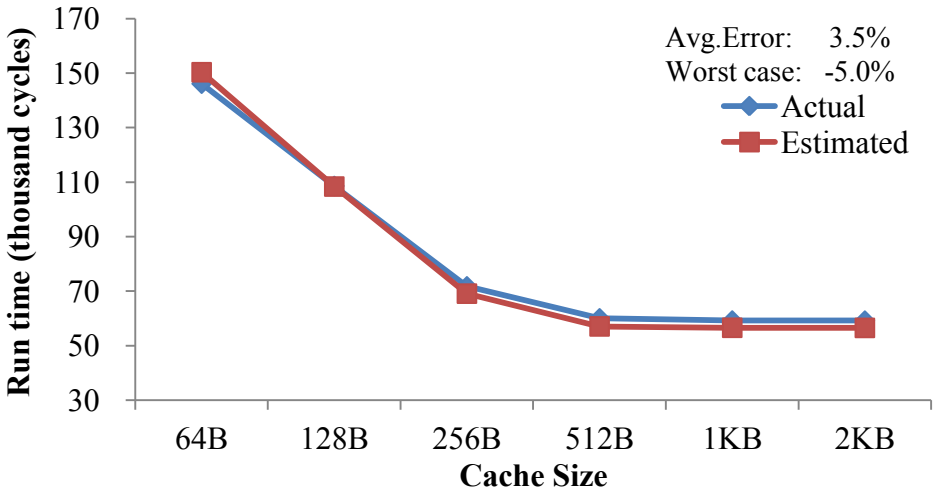


Figure 4.10 Plot of performance estimation of Quicksort in 4-word WB data cache

Table 13 shows the result of running Quicksort with different sizes of 4 word write back data cache. The average error in estimating the run time using pCache is 3.5% and has a maximum error of -5.0% for the data cache of size 512B. Figure 4.10 shows the plot of actual and estimated run time of Quicksort in 4 word write back data cache.

Table 14 Performance estimation of Quicksort in 8-word WB data cache

Cache Size	$T_{built-in}$	T_{pcache}	H_{wr}	H_{rd}	M_{rd}	M_{wr}	Over-head	T_{est}	Error %
64B	179,515	505,985	2,992	8,320	3980	1657	310,721	195,264	8.8
128B	120,183	505,986	4,089	10,270	2030	560	380,763	125,223	4.2
256B	75,450	505,795	4,370	11,818	482	279	434,293	71,502	-5.2
512B	60,356	506,011	4,599	12,251	48	50	449,689	56,322	-6.7
1KB	59,711	505,983	4,609	12,260	39	40	449,961	56,022	-6.2
2KB	59,626	506,018	4,611	12,261	39	38	449,999	56,019	-6.0

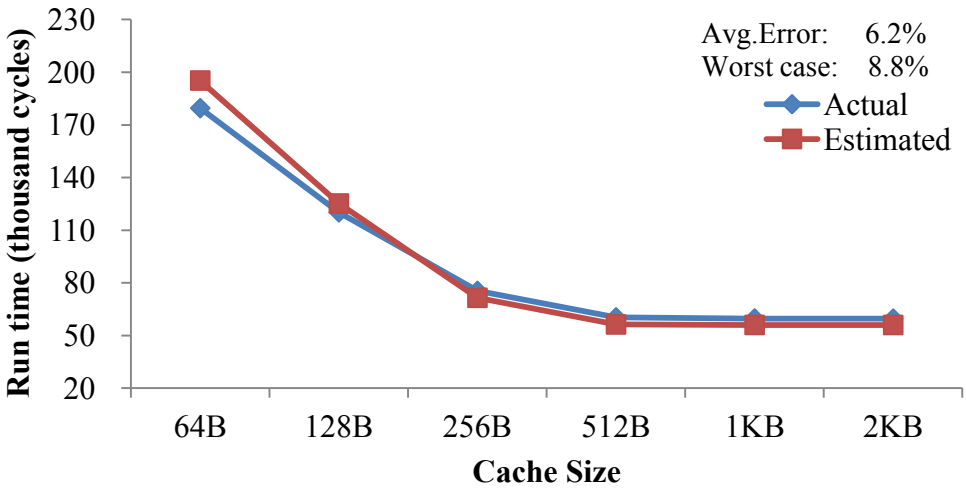


Figure 4.11 Plot of performance estimation of Quicksort in 8-word WB data cache

Table 14 shows the result of running Quicksort with different sizes of 8 word write back data cache. The average error in estimating the run time using pCache is 6.2% and has a maximum error of 8.8% for the data cache of size 64B. Figure 4.11 shows the plot of actual and estimated run time of Quicksort in 8 word write back data cache.

Table 15 Performance estimation of JPEG in 4-word WB data cache

Cache Size	**$T_{built-in}$** (k cycles)	**T_{pcache}**	**H_{wr}**	**H_{rd}**	**M_{rd}**	**M_{wr}**	**Over-head** (k cycles)	**T_{est}** (k cycles)	**Error %**
64B	24,377.7	63,643.4	365,455	1,014,369	389231	199138	38,878.9	24,755.5	1.7
128B	18,079.8	63,634.0	450,358	1,191,780	211820	114235	44,753.8	18,880.2	4.4
256B	15,622.4	63,643.1	487,000	1,263,167	140433	77593	47,031.6	16,602.5	6.3
512B	13,192.5	63,633.9	525,200	1,336,000	67600	39393	49,516.9	14,117.0	7.0
1KB	12,140.8	63,634.3	542,160	1,336,011	37589	22433	50,549.5	13,084.8	7.8
2KB	11,348.2	63,634.0	556,214	1,388,700	14900	8379	51,354.0	12,280.0	8.2

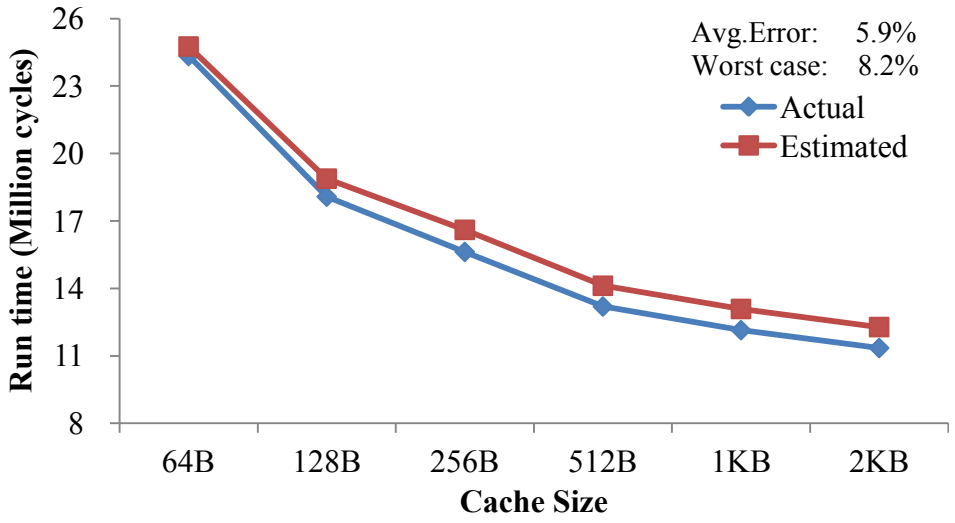


Figure 4.12 Plot of performance estimation of JPEG in 4-word WB data cache

Table 15 shows the result of running JPEG with different sizes of 4 word write back data cache. The average error in estimating the run time using pCache is 5.9% and has a maximum error of 8.2% for the data cache of size 2KB. Figure 4.12 shows the plot of actual and estimated run time of JPEG in 4 word write back data cache.

Table 16 Performance estimation of JPEG in 8-word WB data cache

Cache Size	**$T_{built-in}$** (k cycles)	**T_{pcache}**	**H_{wr}**	**H_{rd}**	**M_{rd}**	**M_{wr}**	**Over-head** (k cycles)	**T_{est}** (k cycles)	**Error %**
64B	31,984.2	63,634.5	328,649	933,290	470310	235944	34,909.0	28,725.5	-10.2
128B	22,473.4	63,634.6	427,044	1,131,421	272179	137549	42,059.4	21,575.2	-4.0
256B	18,123.6	63,634.3	478,377	1,225,056	178544	86219	45,418.6	18,215.7	0.5
512B	14,500.4	63,634.2	519,049	1,331,638	71959	45542	49,256.0	14,378.3	-0.8
1KB	12,679.5	63,634.1	546,800	1,364,071	39526	17791	50,444.6	13,189.5	4.0
2KB	11,472.7	63,634.5	558,448	1,388,734	14866	6145	51,348.5	12,286.0	7.1

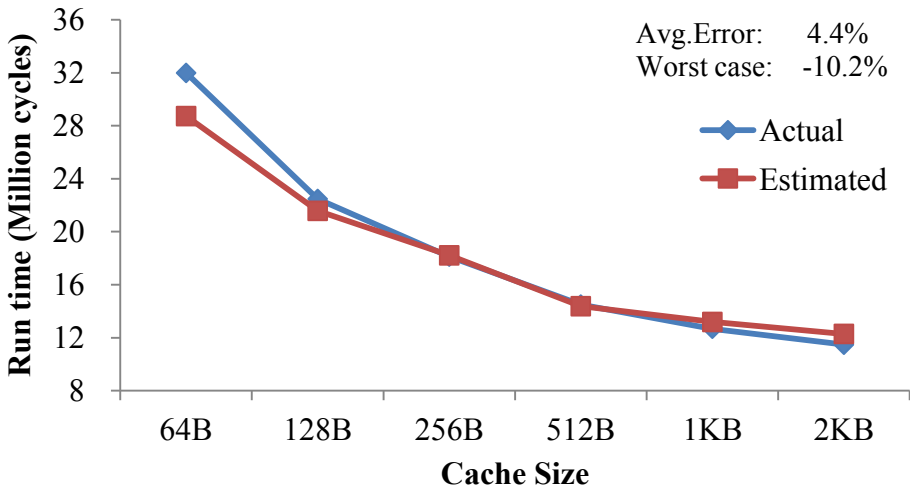


Figure 4.13 Plot of performance estimation of JPEG in 8-word WB data cache

Table 16 shows the result of running JPEG with different sizes of 8 word write back data cache. The average error in estimating the run time using pCache is 4.4% and has a maximum error of -10.2% for the data cache of size 64B. Figure 4.13 shows the plot of actual and estimated run time of JPEG in 8 word write back data cache.

4.6 Run-time Estimation Error

Table 17 shows the average error in estimating the run time of benchmarks using pCache based model. The overall average error in estimating the run time is only 5.4%. The worst case error of -13.7% is noted in the Dhrystone benchmark for a 4-word 128B write through cache shown in Table 5. Although the pCache is functionally identical to the built-in cache, the timing analysis is subject to certain errors that are discussed in the following section.

Table 17 Average error in timing estimation using pCache based model

Write Policy	Cache line size	Benchmark	Average Error %
Write Through cache	4 word	Dhrystone	11.1
		Quicksort	2.0
		JPEG	4.0
	8 word	Dhrystone	9.1
		Quicksort	2.5
		JPEG	4.3
Write Back Cache	4 word	Dhrystone	7.4
		Quicksort	3.5
		JPEG	5.9
	8 word	Dhrystone	6.9
		Quicksort	6.2
		JPEG	4.4
Overall Average Error %			5.4

4.6.1 Sources of Error

The timing model does not account for instruction cache misses. It must be noted that, in pCache, the instructions are fetched on the IXCL, independent of and concurrent with data reads and writes on the PLB. The processor waits for 27 cycles on an instruction miss. Figure 4.14 shows the impact of Instruction cache on Overhead calculation. There are 4 scenarios while estimating the overhead. In the case of an instruction hit, irrespective of data hit or data miss, the instruction fetch delays do not impact the overhead. In the case of simultaneous instruction miss and data miss, the time for data access is greater than that for instruction fetch. As such, the execution delay depends only on the data transaction delay. Therefore the instruction cache behavior has no impact on the overhead. As a result, in the above cases, our estimation is accurate.

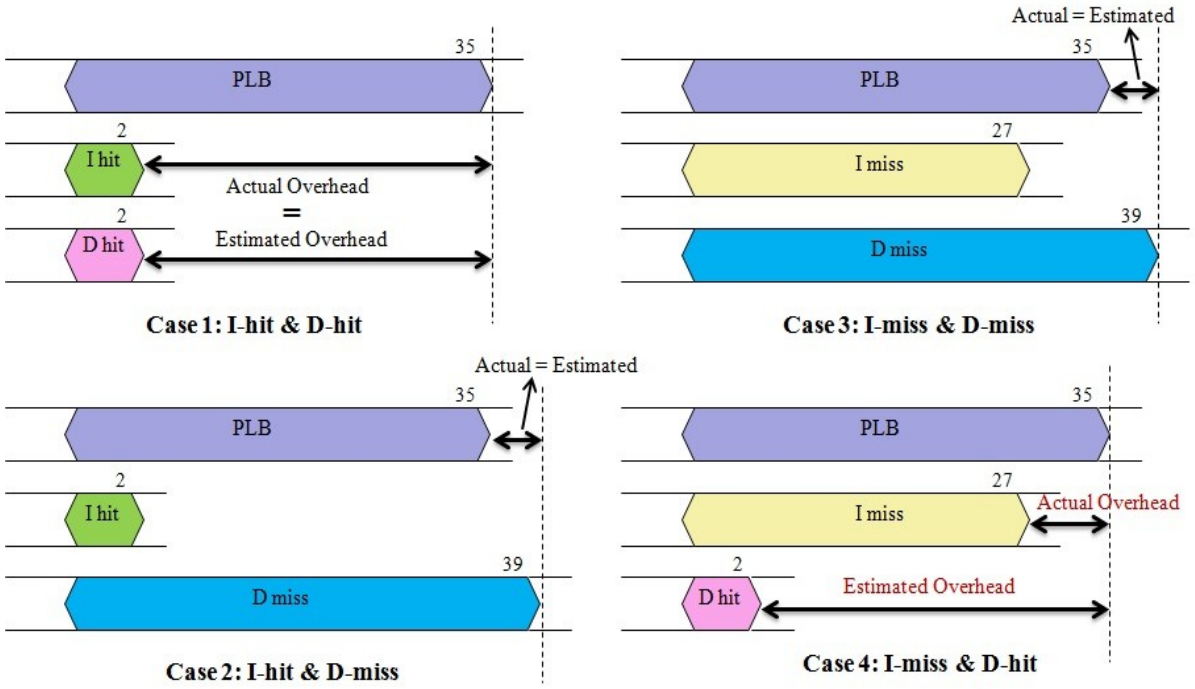


Figure 4.14 Impact of Instruction Cache on Overhead

However, in the case of a simultaneous instruction miss and data hit, there is an execution delay of 27 cycles. Therefore, the correct overhead is (35-27) cycles, since data read over PLB takes 35 cycles in the pCache-based model. However, since we do not model the instruction cache, and always assume an instruction cache hit, the overhead for the above case is calculated to be (35-2) cycles, given the 2 cycles delay for a data read hit. This source of error leads to underestimation of run time. But from Table 5 to 16 we note that, for some cases, the error is positive, implying an overestimation. This is because the constants assigned to the parameters in the cache timing model are averaged over several transactions. As such, the parameter values do not account for possible variations, and can lead to overestimation.

4.6.2 Error Reduction

We can mitigate the effects of instruction cache miss on our timing model by using a large instruction cache (32KB), which is expected to have a high hit rate. This will reduce the probability of the simultaneous instruction miss and data hit scenario. By reducing this scenario which was one of the sources of error, we could achieve more accurate estimates of the run time. Disabling the instruction cache is not an option since the run time will be dominated by the instruction fetch from main memory. Since instruction and data are fetched in parallel, the effect of changing the parameters of data cache will not be observed.

4.7 Speed

Table 18 Comparison of run time of JPEG application

Write Policy	Cache line size	$T_{built\text{-}in}$ (s)	T_{pCache} (s)	Software simulation
Write Through	4 word	0.10	0.51	6 hours
	8 word	0.10	0.51	
Write Back	4 word	0.09	0.51	
	8 word	0.09	0.51	

Table 18 shows the comparison of run time of our largest benchmark, the JPEG application in built-in data cache model, pCache model and cycle accurate software simulation. The average run time on a 2KB pCache model is 0.51 seconds. The average run time on a 2KB built-in data cache model is 0.095 seconds, which is faster than pCache model but provides no observability. The cycle accurate software simulation of JPEG in the reference design with 2KB built-in data cache, takes over 6 hours on an i7 desktop with 16GB RAM. As such, the pCache based model is observable and much faster than cycle accurate software simulation.

4.8 Performance optimization using observable cache

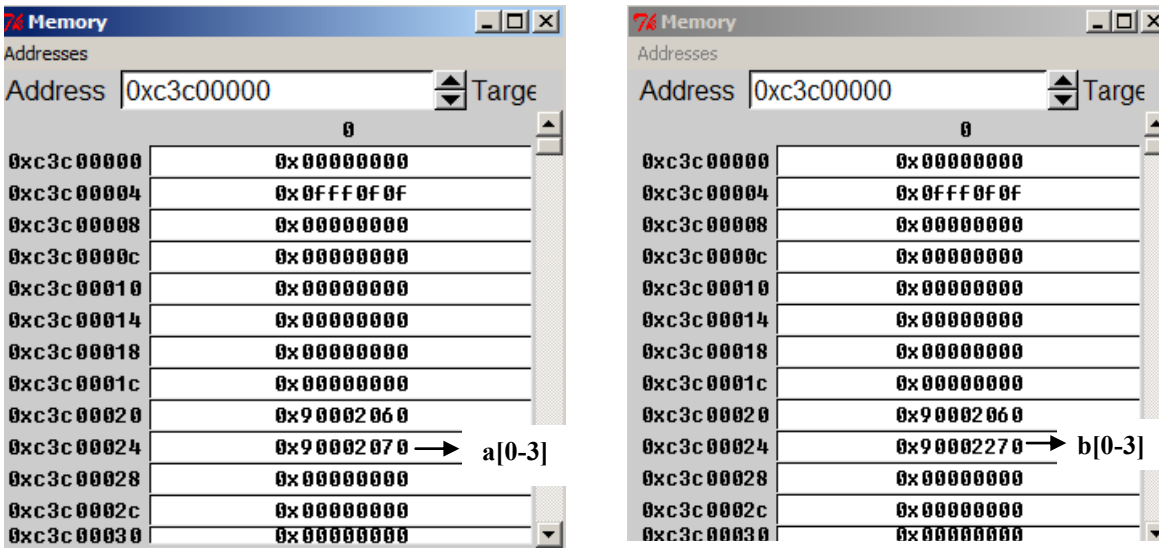
Listing 13. Sample code with single loop

```
int a[128], b[128];
int i=0;

for(i;i<128;i++)
{
    • a[i] = i;
    • b[i] = i+1;
}
```

Listing 14. Sample code with split loop

```
int a[128], b[128];
int i=0,j=0;

for(i;i<128;i++)
{
     a[i] = i;
}
for(j;j<128;j++)
{
     b[j] = j+1;
}
```

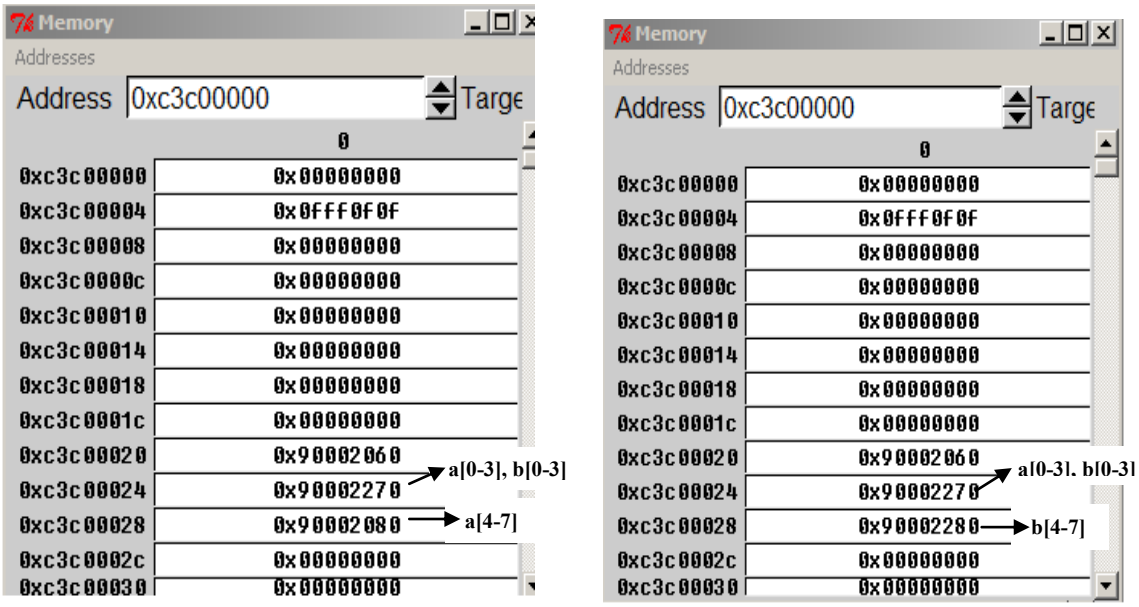
Listings 13 and 14 show the sample code to demonstrate performance optimization of software using the observable cache model, pCache. The address of the integer arrays *a* and *b* used in the above code maps to the same block in data cache. The sample code in Listing 13 shows single loop that modifies both the arrays *a* and *b*. To analyze the program behavior, we use pCache model of size 512 bytes, and insert two software breakpoints at positions shown by red bullets in Listing 13.



(a) Address of *a[0-3]* being cached (b) Address of *b[0-3]* being cached

Figure 4.15 Address conflict in data cache (1)

Figure 4.15 shows the contents of the data cache using the XMD, during the execution of the test program in pCache model. As mentioned earlier, the address of the first location of the block of memory being accessed is written to the slave register dedicated to the corresponding cache block. The block addresses for the elements *a[0-3]* and *b[0-3]* are mapped to the slave register 0xC3C00024. Figure 4.15 (a) shows the contents of data cache when the execution reaches the first breakpoint. The block address for the elements *a[0-3]*, 0x90002070 is written to the data cache. When the execution hits the second breakpoint, the block address for the elements *b[0-3],* 0x90002270 is written to the data cache as shown by Figure 4.15 (b).



(a) Address of *a[4-7]* being cached (b) Address of *b[4-7]* being cached

Figure 4.16 Address conflict in data cache (2)

Figure 4.16 shows the block addresses for the elements *a[4-7]* and *b[4-7]* which are mapped to the slave register 0xC3C00028. Figure 4.16 (a) shows the contents of data cache when the program reaches first breakpoint during second iteration. The block address for the elements *a[4-7]*, 0x90002080 is written to the data cache. In Figure 4.16 (b), we see the block address for the elements *b[4-7]*, 0x90002280 being written to the same block in data cache. Since both the arrays are accessed alternately in the same loop, we could see continuous misses in pCache. The estimated run time of the sample code in Listing 13 in system with pCache based model is 15,747 cycles.

In order to improve the software performance, we optimize the code as shown in Listing 14. We use loop splitting optimization to create two loops, and the arrays *a* and *b* are modified inside different loops. This method significantly improves the cache hit rate, and thereby reduces the execution time of the software. The estimated run time of the sample code in

Listing 14, using the pCache based model is 6186 cycles. Therefore, we get a performance improvement of almost 2.5X by applying the loop splitting optimization. It must be noted that loop splitting is not always desirable because it results in larger code and twice as many branches. If the arrays *a* and *b* did not conflict on the cache, loop splitting might result in poorer behavior. As such, loop splitting cannot be performed automatically by an optimizing compiler, and requires introspection by the software designer. Identification of cache conflicts due to a specific code sequence, during program execution, is not possible without an observable cache model. The pCache model, therefore, exposes such conflicts to the software designer and opens up opportunities for optimization.

For reference, we also executed the two sample codes with a built-in data cache of 512 bytes. The resulting run times for the single and split loop codes were 15,455 cycles and 5,896 cycles, respectively. The numbers translate to timing errors of 2% and 5% respectively. Therefore, not only does pCache expose optimization opportunities, software designers can rely on the predicted impact of the optimization.

4.9 Cache Design Exploration

A major challenge in the design of embedded systems is the many design possibilities that need to be evaluated. Identifying the best architecture configuration requires performance analysis, which is performed via time consuming cycle accurate software simulations. Instead, pCache can be used in design space exploration to identify the best configuration of data cache for a given application and cache size. Consider the following examples to find the *optimal* configuration of data cache of given size, for Dhrystone, Quicksort and JPEG applications. The run time, in cycles, (Y-axis) is plotted for various cache configurations (X-

axis), for both the reference built-in cache design and the pCache-based model. The run time in reference built-in cache design and pCache based model are denoted as *Actual* and *Estimated* respectively.

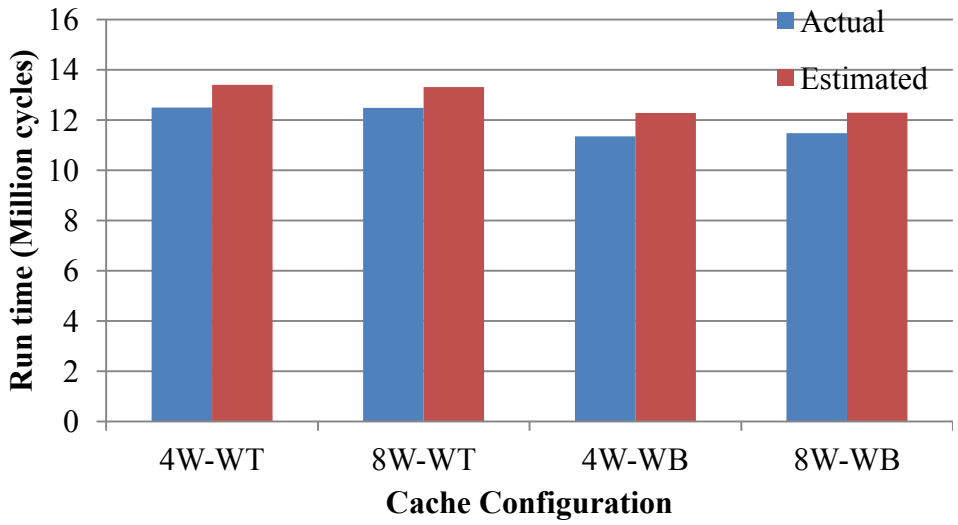


Figure 4.17 Design space exploration of 2KB data cache for JPEG

Figure 4.17 shows the comparison of run time of JPEG application for different configurations of 2KB data cache. The pCache based model indicates that 4-word write back cache configuration has the lowest run time (Estimated 12.28 million cycles). This is corroborated from the timing measurements in the reference design with built-in data cache. We find the same 4-word write back configuration to be the fastest (measured at 11.35 million cycles).

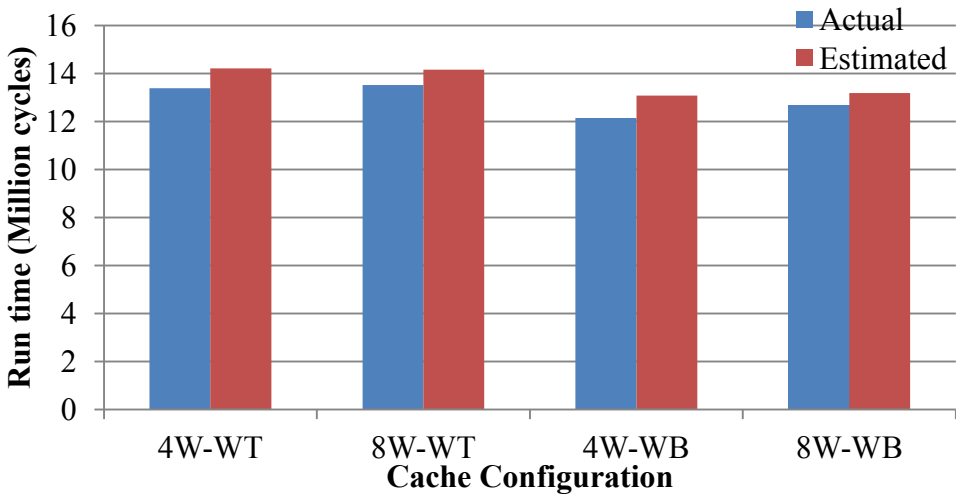


Figure 4.18 Design space exploration of 1KB data cache for JPEG

Figure 4.18 shows the comparison of run time of JPEG application for different configurations of 1KB data cache. The pCache based model shows that 4-word write back cache configuration has the lowest run time (Estimated 13.08 million cycles). This is verified from the timing measurements in the built-in data cache model, in which we find the same 4 word write back configuration to be the fastest (measured at 12.14 million cycles).

Figure 4.19 shows the comparison of run time of Dhrystone benchmark for different configurations of 2KB data cache. The pCache based model shows that 8-word write back cache configuration has the lowest run time (Estimated 51.3 thousand cycles). The timing measurements in built-in data cache model shows that the same 8-word write back configuration to be the fastest (measured 55.6 thousand cycles).

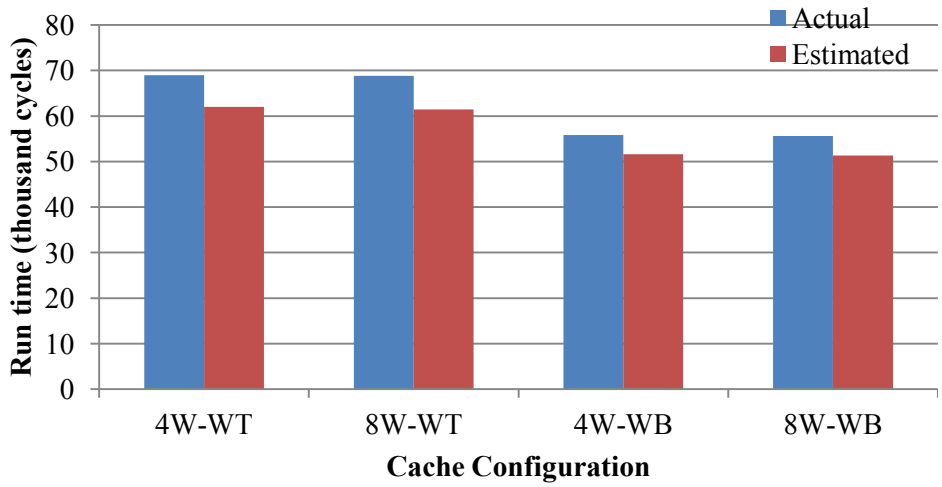


Figure 4.19 Design space exploration of 2KB data cache for Dhrystone

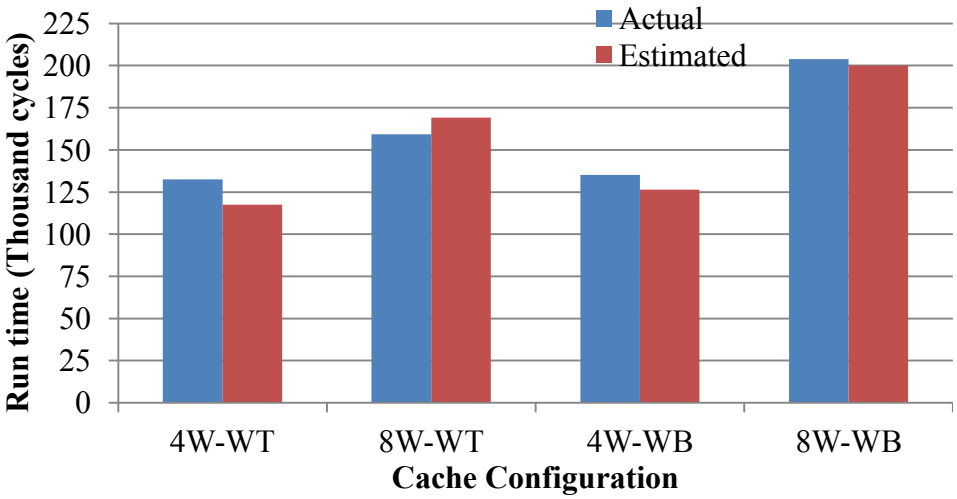


Figure 4.20 Design space exploration of 64B data cache for Dhrystone

Figure 4.20 shows the comparison of run time of Dhrystone benchmark for different configurations of 64B data cache. The pCache based model shows that 4-word write through cache configuration has the lowest run time (Estimated 132.5 thousand cycles). The timing measurements in built-in data cache model shows that the same 4-word write through configuration to be the fastest (measured 117.4 thousand cycles).

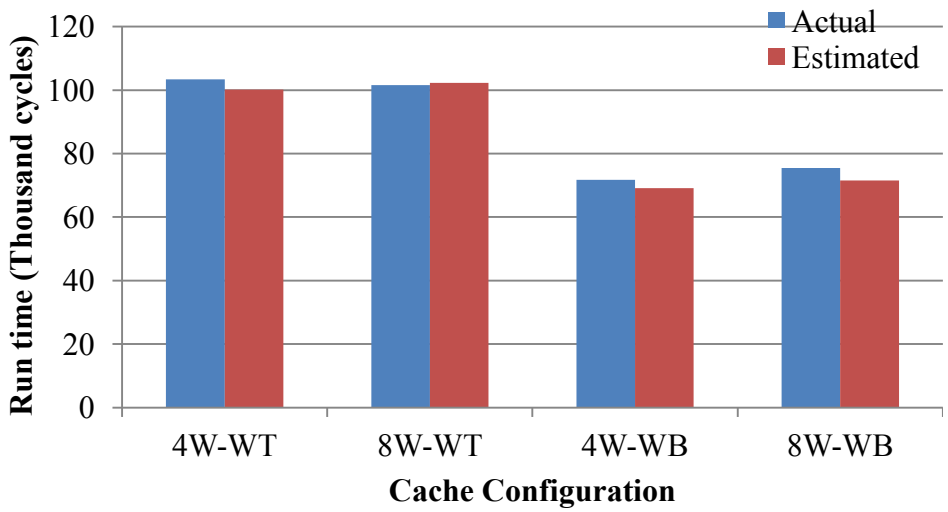


Figure 4.21 Design space exploration of 256B data cache for Quicksort

Figure 4.21 shows the comparison of run time of Quicksort benchmark for different configurations of 256B data cache. The pCache based model shows that 4-word write back cache configuration has the lowest run time (Estimated 69.1 thousand cycles). The timing measurements in built-in data cache model shows that the same 4-word write back configuration to be the fastest (measured 71.8 thousand cycles).

From the above examples, we find the optimal cache configuration can be different for different benchmarks and different cache sizes. The pCache model can be used to accurately select the optimal configuration among others, which can be corroborated by the results from built-in cache in every case. Hence, designers can use pCache model with high confidence to do early design space exploration.

CHAPTER 5

Conclusion and Future work

An observable data cache used in FPGA prototyping is of great help to embedded system and software designers for validating the performance of embedded software. In this thesis we presented our work on an on-chip hardware peripheral, called pCache, which models a data cache. The peripheral can be integrated into a processor system on an FPGA and can display the state of the cache at any given time during software execution. We also presented a parameterized timing model of the cache for accurate performance estimation during embedded software execution.

The pCache provides easy observation of the cache contents and accurate count of hits and misses. With such feedback, the embedded software designers can optimize the software for improving cache hits. We also demonstrated two software optimization techniques using the observable cache model. The embedded system designers need to evaluate many cache design choices during design space exploration in order to find an optimal configuration of cache for specific embedded applications. The pCache, being configurable, allows the designers to modify the cache configuration (size of the cache, cache write policy and cacheable address range) of a given processor core to evaluate the system performance.

In the future, we will extend pCache to incorporate the following,

- Instruction Cache – The pattern of instruction access is usually sequential and less random compared to data access. As such, the performance of ICache is good and

changing configuration has less impact on the performance. However, we will model the instruction cache to improve the accuracy in performance estimation. The challenge in the timing model will be to identify the occurrences of simultaneous instruction miss and data hit.

- Set associative cache – Set associative caches are complex and require many tags to be compared simultaneously. The extra hardware (Comparators and Multiplexers) leads to increased energy consumption, as mentioned earlier. However, set associative cache can solve the problem of cache pollution.

Appendix

A.1 Cache Controller

```
module cachecontroller_fullyparam
(clk,rst,CSR,processorword,inst_r_w,hit,miss,rd_wr,index,tag,check,rst_cache,
inst,Data_range_low,D_bit,hit_rdcounter_out,miss_rdcounter_out,
hit_wrcounter_out,miss_wrcounter_out,check,miss_D_wrcounter_out,
miss_nD_wrcounter_out,miss_D_rdcounter_out,miss_nD_rdcounter_out,
msb_word_select,mem_AValid,write_inst_three_cycles_out,write_inst_four_cycles_out,
write_inst_five_cycles_out,write_inst_ten_cycles_out,read_miss_after_write_miss_out,
read_miss_after_write_hit_out,read_miss_after_read_hit_out,
read_miss_after_read_miss_out,read_hit_wb_thirty_cycles_out,
read_hit_wb_extended_thirty_cycles_out,rite_missD_wb_fortytwo_cycles_out,
read_missnD_wb_fiftythree_cycles_out,read_missD_wb_eightyfive_cycles_out,
Trace_Validinst,Trace_Inst);

parameter write_select =1;
parameter mem_address_bits = 14;
parameter index_bits = 4;
parameter cache_line_size = 4;

localparam word_select_bits = (cache_line_size % 3) + 1 ;
localparam tag_bits = mem_address_bits - index_bits - word_select_bits - 2;


input [0:31] processorword,CSR,Data_range_low,Trace_Inst;
input hit,miss,clk,rst,inst_r_w,D_bit;
input mem_AValid,Trace_Validinst;

output [0:1]rd_wr;
output [0:index_bits-1] index;
output [0:tag_bits-1] tag;
output [0:31] check,inst;
output rst_cache;
output [0:31] hit_rdcounter_out,miss_rdcounter_out,hit_wrcounter_out,
miss_wrcounter_out,miss_D_wrcounter_out,miss_nD_wrcounter_out,
miss_D_rdcounter_out,miss_nD_rdcounter_out;
output msb_word_select;
output [0:31] write_inst_three_cycles_out,write_inst_four_cycles_out,
write_inst_five_cycles_out,write_inst_ten_cycles_out;
output [0:31] read_miss_after_write_miss_out,read_miss_after_write_hit_out,
read_miss_after_read_hit_out,read_miss_after_read_miss_out;
output [0:31] read_hit_wb_thirty_cycles_out, read_hit_wb_extended_thirty_cycles_out;
output [0:31] write_missD_wb_fortytwo_cycles_out,read_missnD_wb_fiftythree_cycles_out,
read_missD_wb_eightyfive_cycles_out;
```

```
wire [0:31] processorword,CSR,Data_range_low;
wire hit,miss,clk,rst,inst_r_w,D_bit;

reg  [0:1]rd_wr;
reg [0:index_bits-1] index;
reg [0:tag_bits-1] tag;
reg [0:31] inst;
reg msb_word_select;

reg [0:mem_address_bits-1] temp;
reg [2:0] current_state;
reg [2:0] next_state;
reg [2:0] previous_state;
reg [0:31] check,processorword_old;
reg rst_cache,flag,stop;

reg flag_prev_access_write,flag_current_access_write;
reg [2:0]flag_current_access;          //000 Read Hit; 001 Read Miss; 010 Write Hit; 011 Write Miss; 100
Write Miss nD; 101 Write Miss D; 110 Read Miss nD; 111 Read Miss D;
reg [2:0]flag_prev_access;

reg [1:0] counter_wb_read_hit;

integer hit_rdcounter;
integer miss_rdcounter;
integer hit_wrcounter;
integer miss_wrcounter;
integer miss_D_wrcounter,miss_nD_wrcounter,miss_D_rdcounter,miss_nD_rdcounter;

integer counter_continous_write;
integer write_inst_three_cycles;
integer write_inst_four_cycles;
integer write_inst_five_cycles;
integer write_inst_ten_cycles;

integer read_miss_after_write_miss;
integer read_miss_after_write_hit;
integer read_miss_after_read_hit;
integer read_miss_after_read_miss;

integer read_hit_wb_thirty_cycles;
integer read_hit_wb_extended_thirty_cycles;
integer write_missD_wb_fortytwo_cycles;
integer read_missnD_wb_fiftythree_cycles;
integer read_missD_wb_eightyfive_cycles;
always @(posedge clk)
begin
```

```
if (write_select == 1)
begin
        if (rst==1'b1 || CSR == 32'hFFFFFFFF)
                begin
                    rd_wr= 2'b10;
                    index= {index_bits{1'b0}};
                    tag = {tag_bits{1'b0}};
                    current_state= 3'b111;
                    next_state= 3'b000;
                    previous_state= 3'b111;
                    rst_cache = 1'b1;
                    flag = 1'b1;
                    stop = 1'b1;
                    inst = 32'h00000000;
                    processorword_old = 32'h00000000;
                    hit_rdcounter = 0;
                    miss_rdcounter = 0;
                    hit_wrcounter = 0;
                    miss_wrcounter = 0;
                    check = 32'b0;
                    miss_D_wrcounter = 0;
                    miss_nD_wrcounter = 0;
                    miss_D_rdcounter = 0;
                    miss_nD_rdcounter = 0;
                    msb_word_select = 1'bz;
                    flag_prev_access_write = 1'b0;
                    flag_current_access_write =1'b0;
                    counter_continous_write =0;
                    write_inst_three_cycles = 0;
                    write_inst_four_cycles = 0;
                    write_inst_five_cycles = 0;
                    write_inst_ten_cycles = 0;
                    flag_current_access = 3'bzz;
                    flag_prev_access = 3'bzz;
                    read_miss_after_write_miss = 0;
                    read_miss_after_write_hit = 0;
                    read_miss_after_read_hit = 0;
                    read_miss_after_read_miss = 0;
                    counter_wb_read_hit = 2'bzz;
                    read_hit_wb_thirty_cycles = 0;
                    read_hit_wb_extended_thirty_cycles = 0;
                    write_missD_wb_fortytwo_cycles = 0;
                    read_missnD_wb_fiftythree_cycles = 0;
                    read_missD_wb_eightyfive_cycles = 0;
                end
        else if ((mem_AValid == 1'b1) && (processorword[0:3] == 4'b1001) && (CSR ==
32'h11111111) && (stop == 1'b1))
begin
```

```
        current_state= 3'b111;
         next_state= 3'b000;
         previous_state= 3'b111;
         stop = 1'b0;
         rst_cache = 1'b0;
end

else if ((Trace_Validinst == 1'b1) && (Trace_Inst[0:5] != 6'b111110))
begin
        flag_current_access_write = 1'b0;
        flag_prev_access_write = 1'b0;
        check = 32'hbaafbeef;
end

else if((rst == 1'b0) && (CSR == 32'h11111111) && (stop ==1'b0))
begin
        if((processorword[0:3] == 4'b1001) )
        begin
                previous_state= current_state;
                current_state= next_state;

case(current_state)
3'b000:
        begin //idle
                rst_cache = 1'b0;
                 if ((previous_state == 3'b010 ) || (previous_state == 3'b011) || (previous_state == 3'b100)
|| (previous_state == 3'b001) || (stop == 1'b1))
                    begin
                            next_state = 3'b000;
                            stop = 1'b1;
                            rd_wr = 2'b10;
                            flag_prev_access_write = flag_current_access_write;
                            flag_prev_access = flag_current_access;
                            if((flag_current_access_write == 1'b1) && (counter_continous_write == 1))
                              write_inst_three_cycles =  write_inst_three_cycles+1;
                            else if((flag_current_access_write == 1'b1) && (counter_continous_write == 2))
                              write_inst_four_cycles =  write_inst_four_cycles+1;
                            else if((flag_current_access_write == 1'b1) && (counter_continous_write == 3))
                              write_inst_five_cycles =  write_inst_five_cycles+1;
                             else if ((flag_current_access_write == 1'b1) && (counter_continous_write >= 4))
                              write_inst_ten_cycles = write_inst_ten_cycles +1;
                    end

                else
                    begin
                            flag =1'b0;
                            temp = processorword[32-mem_address_bits:31];
                            rd_wr= 2'b10;
```

```
                        if(inst_r_w) // check if read
                          begin
                                next_state= 3'b010;
                                flag_current_access_write = 1'b0;
                          end
                        else
                          begin
                                next_state= 3'b001;
                                check = 32'hccccdddd;
                                flag_current_access_write = 1'b1;
                                if(flag_prev_access_write == 1'b1)
                                   counter_continous_write = counter_continous_write+1;
                                else
                                   counter_continous_write = 1;
                        end
                end
        end

3'b010:
begin //read
        rd_wr= 2'b11;
        tag= temp[0:tag_bits-1];
        index= temp[tag_bits:mem_address_bits-word_select_bits-3];
        check = 32'hfabfabee;
        if(hit)
          begin
                next_state= 3'b000;
                hit_rdcounter = hit_rdcounter+1;
                check = 32'hfabf00ee;
                flag_current_access = 3'b000;
          end
        else if(miss)
          begin
                next_state= 3'b011;
                miss_rdcounter = miss_rdcounter +1;
                check = 32'hfabfab99;
                flag_current_access = 3'b001;
          end
end

3'b001:
begin //write
          rd_wr= 2'b11;
          tag= temp[0:tag_bits-1];
          index= temp[tag_bits:mem_address_bits-word_select_bits-3];
          check = 32'heeeeffff;
          if(hit)
          begin
```

```
                next_state = 3'b000;
                hit_wrcounter = hit_wrcounter+1;
                check = 32'ha2a2a2a2;
                flag_current_access = 3'b010;
         end
        else if (miss)
        begin
                next_state = 3'b000;
                miss_wrcounter = miss_wrcounter+1;
                check = 32'haaaabbbb;
                flag_current_access = 3'b011;
        end
end

3'b011:
begin
         rd_wr = 2'b00;
         tag= temp[0:tag_bits-1];
         index= temp[tag_bits:mem_address_bits-word_select_bits-3];
         next_state = 3'b000;
         inst = processorword;

         if(cache_line_size==8)
                msb_word_select = processorword[27];
         else
                msb_word_select = 1'bz;

         if(flag_prev_access == 3'b011)
                read_miss_after_write_miss = read_miss_after_write_miss+1;


         if(flag_prev_access == 3'b010)
                read_miss_after_write_hit = read_miss_after_write_hit+1;

         if(flag_prev_access == 3'b000)
                read_miss_after_read_hit = read_miss_after_read_hit+1;

         if(flag_prev_access == 3'b001)
                read_miss_after_read_miss = read_miss_after_read_miss+1;

end

3'b111:
begin
         rd_wr = 2'b10;
         next_state = 3'b000;
end
endcase
```

```
end

else
begin
        rd_wr= 2'b10;
        rst_cache = 1'b0;
        index= {index_bits{1'b0}};
        tag = {tag_bits{1'b0}};
        current_state= 3'b000;
        next_state= 3'b000;
        previous_state= 3'b000;
        msb_word_select = 1'bz;
end
end

else if ((CSR == 32'hDDDDDDDD))
begin
        rd_wr= 2'b10;
        index= {index_bits{1'b0}};
        tag = {tag_bits{1'b0}};
        current_state= 3'b000;
        next_state= 3'b000;
        previous_state= 3'b000;
        rst_cache = 1'b0;
        msb_word_select = 1'bz;
end
end


else
begin
          if(rst==1'b1 || CSR == 32'hFFFFFFFF)
                  begin
                        rd_wr= 2'b10;
                        index= {index_bits{1'b0}};
                        tag = {tag_bits{1'b0}};
                        current_state= 3'b111;
                        next_state= 3'b000;
                        previous_state= 3'b111;
                        rst_cache = 1'b1;
                        flag = 1'b1;
                        stop = 1'b1;
                        inst = 32'h00000000;
                        processorword_old = 32'h00000000;
                        check = 32'b0;
                        hit_wrcounter = 0;
                        hit_rdcounter = 0;
                        miss_D_wrcounter = 0;
```

```
                miss_nD_wrcounter = 0;
                miss_D_rdcounter = 0;
                miss_nD_rdcounter = 0;
                miss_rdcounter = 0;
                miss_wrcounter = 0;
                msb_word_select = 1'bz;

                flag_prev_access_write = 1'b0;
                flag_current_access_write =1'b0;
                counter_continous_write =0;
                write_inst_three_cycles = 0;
                write_inst_four_cycles = 0;
                write_inst_five_cycles = 0;
                write_inst_ten_cycles = 0;
                flag_current_access = 3'bzzz;
                flag_prev_access = 3'bzzz;
                read_miss_after_write_miss = 0;
                read_miss_after_write_hit = 0;
                read_miss_after_read_hit = 0;
                read_miss_after_read_miss = 0;
                counter_wb_read_hit =2'b00;
                read_hit_wb_thirty_cycles = 0;
                read_hit_wb_extended_thirty_cycles = 0;
                write_missD_wb_fortytwo_cycles =0;
                read_missnD_wb_fiftythree_cycles = 0;
                read_missD_wb_eightyfive_cycles = 0;
            end

else if ((mem_AValid == 1'b1) && (processorword[0:3] == 4'b1001)&& (CSR == 32'h11111111) &&
(stop == 1'b1) )
begin
        current_state= 3'b111;
        next_state= 3'b000;
        previous_state= 3'b111;
        stop = 1'b0;
        rst_cache = 1'b0;
end


else if((rst == 1'b0) && (CSR == 32'h11111111) && (stop ==1'b0))
begin
        if(processorword[0:3] == 4'b1001)
        begin
            previous_state= current_state;
            current_state= next_state;

        case(current_state)
```

```
3'b000:
begin //idle
      rst_cache = 1'b0;
      if ((previous_state == 3'b010 ) || (previous_state == 3'b011) || (previous_state == 3'b100)
      || (previous_state == 3'b001) || (stop == 1'b1))
      begin
          next_state = 3'b000;
          stop = 1'b1;
          rd_wr = 2'b10;
          flag_prev_access = flag_current_access;
          if((flag_current_access == 3'b000) && (counter_wb_read_hit == 2'b01))
                    read_hit_wb_thirty_cycles = read_hit_wb_thirty_cycles +1;
          if((flag_current_access == 3'b000) && (counter_wb_read_hit == 2'b10))
                   read_hit_wb_extended_thirty_cycles  =  read_hit_wb_extended_thirty_cycles
+1;
      end

      else
      begin
          flag =1'b0;
          temp = processorword[32-mem_address_bits:31];
          rd_wr= 2'b10;

          if(inst_r_w) // check if read
          begin
               next_state= 3'b010;
               check = 32'hbdabdaad;
          end

          else
          begin
               next_state= 3'b001;
               check = 32'hccccdddd;
          end
      end
end

3'b010:
begin //read
      rd_wr = 2'b11;
      tag= temp[0:tag_bits-1];
      index= temp[tag_bits:mem_address_bits-word_select_bits-3];
      check = 32'hadcdacaa;

      if(hit)
      begin
           next_state = 3'b000;
           hit_rdcounter = hit_rdcounter+1;
```

```
            check = 32'hbfdacffa;
            flag_current_access = 3'b000;

            if(flag_prev_access == 3'b000)
                  counter_wb_read_hit = counter_wb_read_hit +1;

             else if ((flag_prev_access == 3'b101) || (flag_prev_access == 3'b100))
                        read_hit_wb_extended_thirty_cycles                                    =
            read_hit_wb_extended_thirty_cycles +1;
              else
                        counter_wb_read_hit = 2'b00;

      end


      else if (miss)
      begin
            if(D_bit)
            begin
                  next_state = 3'b011;
                  miss_D_rdcounter = miss_D_rdcounter+1;
                  flag_current_access = 3'b111;

                  if((flag_prev_access == 3'b000) || (flag_prev_access == 3'b010) ||
      (flag_prev_access == 3'b101) || (flag_prev_access == 3'b100) || (flag_prev_access ==
      3'b110))
                    read_missD_wb_eightyfive_cycles = read_missD_wb_eightyfive_cycles +1;
            end

            else
            begin
                  next_state = 3'b011;
                  miss_nD_rdcounter = miss_nD_rdcounter+1;
                  flag_current_access = 3'b110;
                  if(flag_prev_access == 3'b000)
                    read_missnD_wb_fiftythree_cycles = read_missnD_wb_fiftythree_cycles+1;
            end
      end
 end

3'b011:
begin //read-write
      rd_wr = 2'b00;
      tag= temp[0:tag_bits-1];
      index= temp[tag_bits:mem_address_bits-word_select_bits-3];
      next_state = 3'b000;
      inst = processorword;
```

```
        if(cache_line_size==8)
            msb_word_select = processorword[27];
        else
            msb_word_select = 1'bz;
    end


    3'b001: //  write
    begin
        rd_wr = 2'b11;
        tag= temp[0:tag_bits-1];
        index= temp[tag_bits:mem_address_bits-word_select_bits-3];
        check = 32'hacbacbbb;

        if(hit)
        begin
            next_state = 3'b000;
            check = 32'habcabcab;
            hit_wrcounter = hit_wrcounter+1;
            flag_current_access = 3'b010;
        end

       else if((miss == 1'b1) && (D_bit ==1'b1))
       begin
            next_state = 3'b100;
            check = 32'habcdabcd;
            miss_D_wrcounter = miss_D_wrcounter+1;
            flag_current_access = 3'b100;

            if((flag_prev_access == 3'b101) || (flag_prev_access == 3'b010))
                write_missD_wb_fortytwo_cycles = write_missD_wb_fortytwo_cycles +1;
       end


        else if ((miss == 1'b1) && (D_bit ==1'b0))
        begin
            next_state = 3'b100;
            check = 32'hdcdcdcdc;
            miss_nD_wrcounter = miss_nD_wrcounter+1;
            flag_current_access = 3'b101;
        end
    end

    3'b100:
    begin  //write - write
        rd_wr = 2'b01;
        tag= temp[0:tag_bits-1];
        index= temp[tag_bits:mem_address_bits-word_select_bits-3];
```

```
                next_state = 3'b000;
                inst = processorword;

                if(cache_line_size==8)
                msb_word_select = processorword[27];

                else
                msb_word_select = 1'bz;
            end


            3'b111:
            begin
                rd_wr = 2'b10;
                next_state = 3'b000;
            end
            endcase
        end

        else
        begin
            rd_wr= 2'b10;
            rst_cache = 1'b0;
            index= {index_bits{1'b0}};
            tag = {tag_bits{1'b0}};
            current_state= 3'b000;
            next_state= 3'b000;
            previous_state= 3'b000;
            msb_word_select = 1'bz;
        end
    end

        else if ((CSR == 32'hDDDDDDDD))
        begin
            rd_wr= 2'b10;
            index= {index_bits{1'b0}};
            tag = {tag_bits{1'b0}};
            current_state= 3'b000;
            next_state= 3'b000;
            previous_state= 3'b000;
            rst_cache = 1'b0;
            msb_word_select = 1'bz;
        end
    end
    end

    assign hit_wrcounter_out = hit_wrcounter;
    assign hit_rdcounter_out = hit_rdcounter;
```

```
assign miss_D_wrcounter_out = miss_D_wrcounter;
assign miss_nD_wrcounter_out = miss_nD_wrcounter;
assign miss_D_rdcounter_out = miss_D_rdcounter;
assign miss_nD_rdcounter_out = miss_nD_rdcounter;

assign miss_rdcounter_out = miss_rdcounter;
assign miss_wrcounter_out = miss_wrcounter;

assign write_inst_three_cycles_out = write_inst_three_cycles;
assign write_inst_four_cycles_out = write_inst_four_cycles;
assign write_inst_five_cycles_out = write_inst_five_cycles;
assign write_inst_ten_cycles_out = write_inst_ten_cycles;

assign read_miss_after_write_miss_out = read_miss_after_write_miss;
assign read_miss_after_write_hit_out = read_miss_after_write_hit;
assign read_miss_after_read_hit_out = read_miss_after_read_hit;
assign read_miss_after_read_miss_out = read_miss_after_read_miss;

assign read_hit_wb_extended_thirty_cycles_out = read_hit_wb_extended_thirty_cycles;
assign read_hit_wb_thirty_cycles_out = read_hit_wb_thirty_cycles;
assign write_missD_wb_fortytwo_cycles_out = write_missD_wb_fortytwo_cycles;
assign read_missnD_wb_fiftythree_cycles_out = read_missnD_wb_fiftythree_cycles;
assign read_missD_wb_eightyfive_cycles_out = read_missD_wb_eightyfive_cycles;

endmodule
```

B.1 Tag memory

```
Module cache_fullyparam
(clk,rst,rst_cache,r_w,index,tag,hit,miss,dataout,indexout,writeout,inst_in,D_bit,msb_word_select);

parameter write_select =1;
parameter mem_address_bits = 14;
parameter index_bits = 4;
parameter cache_line_size = 4;

localparam word_select_bits = (cache_line_size % 3) + 1 ;
localparam tag_bits = mem_address_bits - index_bits - word_select_bits - 2;
localparam cache_depth = 1<< index_bits;
localparam zero_stuff = 32-1-tag_bits;
localparam zero_indexout_stuff = 8-index_bits;


input [0:index_bits-1] index;
input [0:tag_bits-1] tag;

input clk,rst,rst_cache;
input  [0:1]r_w;
input [0:31] inst_in;
input msb_word_select;


output [0:31] dataout,writeout;
output  hit,miss,D_bit;
output [0:7] indexout;

wire clk,rst;

reg  hit,miss,D_bit;
reg [0:31] dataout,writeout;
reg [0:7] indexout;

reg [0:31] mem [0:cache_depth-1];

reg [0:31] temp;
integer i;

always @ (posedge clk)
begin

        if(write_select == 1)
        begin
```

```
if (rst==1'b1 || rst_cache == 1'b1)
begin
        hit = 1'b0;
        miss = 1'b0;
        dataout = 32'b0;
        indexout = 8'b0;//{index_bits{1'b0}};
        writeout = 32'b0;
        D_bit = 1'bz;

        for( i=0; i<cache_depth; i=i+1)
            mem[i] = 32'b0;

end

else
begin

case (r_w)

2'b11:
begin
        temp = mem[index];
        if(temp[0])
        begin
                if(tag == temp[1:tag_bits])
                begin
                        dataout = temp;
                        hit = 1'b1;
                        miss = 1'b0;
                        indexout = 8'bz;//{index_bits{1'bz}};
                 end

        else
        begin
                dataout = 32'b00110000111100001111000011110000;
                hit = 1'b0;
                miss = 1'b1;
                indexout = 8'bz;//{index_bits{1'bz}};
                 D_bit = 1'b0;

         end
end

else
begin
        dataout = 32'b00000000111111111111111111110000;
        indexout = 8'bz;//{index_bits{1'bz}};
```

```
            hit = 1'b0;
            miss = 1'b1;
            D_bit = 1'b0;
    end
    end

    2'b00:
    begin
            hit = 1'b0;
            miss = 1'b0;
            temp = {1'b1, tag, {zero_stuff{1'b0}}};
            mem[index] = temp;
            dataout = 32'bz;
            indexout = {{zero_indexout_stuff{1'b0}},index};
            if((cache_line_size == 8) && (msb_word_select == 1'b1))
                writeout = {inst_in[0:27], 4'b0000} - 5'b10000;
            else
                writeout = {inst_in[0:27],4'b0000};

    end

    2'b01:
    begin
            hit = 1'b0;
            miss = 1'b0;
            indexout = 8'bz;//{index_bits{1'bz}};
            //writeout = 32'b0;

            dataout = 32'b00001111000010010000110000001111;
    end

    2'b10:
    begin
            hit = 1'b0;
            miss = 1'b0;
            dataout = 32'b00001111111111111000011110000 1111;
            indexout = 8'bz;//{index_bits{1'bz}};
            //  writeout = 32'b0;
    end
    endcase

    end
end

    else
    begin

            if (rst==1'b1 || rst_cache == 1'b1)
```

```
begin
        hit = 1'b0;
        miss = 1'b0;
        dataout = 32'b0;
        indexout = 8'bz;//{index_bits{1'bz}};
        writeout = 32'b0;
        D_bit = 1'bz;

        for( i=0; i<cache_depth; i=i+1)
                mem[i] = 32'b0;

 end

else
begin

case (r_w)

2'b11: //read
begin
         mem[index];
         if(temp[0])
        begin
                if(tag == temp[2:tag_bits+1])
                begin
                        dataout = temp;
                        hit = 1'b1;
                        miss = 1'b0;
                        indexout = 8'bz;//{index_bits{1'bz}};

                end

                else
                begin
                        dataout = 32'b00110000111100001111000011110000;
                        hit = 1'b0;
                        miss = 1'b1;
                        indexout = 8'bz;//{index_bits{1'bz}};
                        D_bit = temp[1];
                end
    end

    else
    begin
            dataout = 32'b00000000111111111111111111110000;
            indexout = 8'bz;//{index_bits{1'bz}};
            hit = 1'b0;
            miss = 1'b1;
```

```
                    D_bit = 1'b0;
                end
          end

        2'b00:      // read-write
        begin
                hit = 1'b0;
                miss = 1'b0;
                temp = {1'b1, 1'b0, tag, {zero_stuff-1{1'b0}}};
                mem[index] = temp;
                dataout = 32'bz;
                indexout =  {{zero_indexout_stuff{1'b0}},index};

                if((cache_line_size == 8) && (msb_word_select == 1'b1))
                    writeout = {inst_in[0:27], 4'b0000} - 5'b10000;
                else
                    writeout = {inst_in[0:27],4'b0000};
        end

        2'b01: // write-write
        begin
                hit = 1'b0;
                miss = 1'b0;
                temp = {1'b1, 1'b1, tag, {zero_stuff-1{1'b0}}};
                mem[index] = temp;
                dataout = 32'bz;
                indexout =  {{zero_indexout_stuff{1'b0}},index};

                if((cache_line_size == 8) && (msb_word_select == 1'b1))
                    writeout = {inst_in[0:27], 4'b0000} - 5'b10000;
                else
                    writeout = {inst_in[0:27],4'b0000};
        end

        2'b10:
        begin
                hit = 1'b0;
                miss = 1'b0;
                dataout = 32'b00001111111111111000011110000111 1;
                indexout = 8'bz;//{index_bits{1'bz}};
        end
        endcase
      end
end
end
endmodule
```

References

1. M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. 1998. Improving locality using loop and data transformations in an integrated framework. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture* (MICRO 31). IEEE Computer Society Press, Los Alamitos, CA, USA, 285-297.

2. A. Agarwal, J. Hennessy, and M. Horowitz. 1989. An analytical cache model. *ACM Trans. Comput. Syst.* 7, 2 (May 1989), 184-215. DOI=10.1145/63404.63407 http://doi.acm.org/10.1145/63404.63407

3. J. Xue, X. Vera. "Efficient and accurate analytical modeling of whole-program data cache behavior," *Computers, IEEE Transactions on* , vol.53, no.5, pp.547,566, May 2004 doi: 10.1109/TC.2004.1275296

4. T.M. Conte, M.A. Hirsch, W.-M.W. Hwu, "Combining trace sampling with single pass methods for efficient cache simulation," *Computers, IEEE Transactions on* , vol.47, no.6, pp.714,720, Jun 1998

5. T.F. Chen, "Efficient trace-sampling simulation techniques for cache performance analysis," *Simulation Symposium, 1996., Proceedings of the 29th Annual* , vol., no., pp.54,63, 8-11 Apr 1996

6. N. Tojo, N. Togawa, M. Yanagisawa, T. Ohtsuki, "Exact and fast L1 cache simulation for embedded systems," *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific* , vol., no., pp.817,822, 19-22 Jan. 2009 doi: 10.1109/ASPDAC.2009.4796581

7. J.J Pieper, A. Mellan, J.M. Paul, D.E. Thomas, F. Karim, "High level cache simulation for heterogeneous multiprocessors," *Design Automation Conference, 2004. Proceedings. 41st* , vol., no., pp.287,292, 7-11 July 2004

8. A. Pedram, A. Gerslauer, “Modeling Cache effects at the transaction level,” in Proceedings of IESS, 2009.

9. Xilinx Inc, MicroBlaze Processor Reference Guide, available http://www.xilinx.com

10. Xilinx Inc, ISE Concepts, Tools and Techniques, available http://www.xilinx.com

11. Dinero IV Trace-Driven Uniprocessor Cache Simulator, http://pages.cs.wisc.edu/~markhill/DineroIV/, 2012.

12. MMCacheSim: A Highly Configurable Matrix Multiplication Cache Simulator, ICT Innovations 2012.

13. R.L. Mattson, J.Gercsei, D.R. Slutz, and I.L. Traiger, “Evaluation Techniques for Storage Hierarchies,” IBM Systems J., vlo. 9, no. 2, pp. 78-117, 1970.

14. M.S. Haque, J. Peddersen, A. Janapsatya, S. Parameswaran, "SCUD: A fast single-pass L1 cache simulation approach for embedded processors with Round-robin replacement policy," *Design Automation Conference (DAC), 2010 47th ACM/IEEE* , vol., no., pp.356-361, 13-18 June 2010

15. A. Janapsatya, A. Ignjatovic, S. Parameswaran, "Finding optimal L1 cache configuration for embedded systems," *Design Automation, 2006. Asia and South Pacific Conference on* , vol., no., pp.6 pp., 24-27 Jan. 2006 doi: 10.1109/ASPDAC.2006.1594783

16. H.S. Stone, High-Performance Computer Architecture. New York: Addison-Wesley, 1990.

17. L. Xianfeng, S.N Hemendra, M. Tulika, R. Abhik, 2004. Design space exploration of caches using compressed traces. In *Proceedings of the 18th annual international conference on Supercomputing* (ICS '04). ACM, New York, NY, USA, 116-125.

18. C. G. Nevill-Manning and I. H. Witten. “Linear-time incremental hierarchy inference for compression". In Data Compression Conference(DCC'97), pages 3-11,1997

19. A. Ghosh, T. Givargis, "Analytical design space exploration of caches for embedded systems," *Design, Automation and Test in Europe Conference and Exhibition, 2003* , vol., no., pp. 650- 655, 2003 doi: 10.1109/DATE.2003.1253681

20. J.L. Hennessy, D.A. Patterson, *Computer Architecture: A Quantitative Approach,* fourth ed. Morgan Kaufmann, 2007.

21. C. Carvalho, "The Gap between Processor and Memory Speeds", Proc. ICCA 2002, 2002, pp. 51-58.

22. Coursera, Online course on Computer Architecture available at https://class.coursera.org/comparch-2012-001/wiki/view?page=coursedetails

23. J.L. Hennessy, D.A. Patterson, Computer Organization and Design, Fourth Edition: The Hardware/Software Interface, 2008

24. W. Stallings, Computer Organization and Architecture: Designing for Performance. 9780273769194, 2012 Pearson.

25. R.A. Uhlig and T.N. Mudge, “Trace-Driven Memory Simulation: A Survey,” ACM Computing Surveys, vol. 29, no.3, pp. 128-170, Sept 1997.

26. B.B. Fraguela, R. Doallo, E.L. Zapata, "Automatic analytical modeling for the estimation of cache misses," *Parallel Architectures and Compilation Techniques, Proceedings. International Conference on* , vol., no., pp.221,231, 1999 doi: 10.1109/PACT.1999.807544

27. S. Ghosh, M. Martonosi, and S. Malik, "Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior," ACM Trans. Programming Languages and Systems, vol. 21, no. 4, pp. 703-746, 1999.

28. J.S. Harper, D.K. Kerbyson, G.R. Nudd, "Analytical modeling of set-associative cache behavior," *Computers, IEEE Transactions on* , vol.48, no.10, pp.1009,1024, Oct 1999 doi: 10.1109/12.805152

29. R. Domer, "Transaction Level Modeling of Computation," *Center for Embedded Computer Systems, Technical Report* ,2006

30. G. Schirner, A. Gerstlauer, R. Domer, Abstract, multifaceted modeling of embedded processors for system level design. In: ASP-DAC, Yokohama, Japan (January 2007)

31. D. Araki, N. Ito, T. Shinsha, Y. Mori, High speed hardware/software co-verification with CPU model generator from software code. Technical report, InterDesign Technologies Inc (2006)

32. K.L. Lin, C.K. Lo, R.S. Tsay, "Source-level timing annotation for fast and accurate TLM computation model generation," *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific* , vol., no., pp.235,240, 18-21 Jan. 2010

33. M. Peter, W. Lars, V. Manish, S. Stefan, H. Urs, 2004. Fast, predictable and low energy memory references through architecture-aware compilation. In*Proceedings of the 2004 Asia and South Pacific Design Automation Conference* (ASP-DAC '04).