DIMSim: A Rapid Two-level Cache Simulation Approach for Deadline-based MPSoCs

Mohammad Shihabul Haque[1]
Roshan Ragel[2]
Angelo Ambrose[3]
S. Radhakrishnan[4]
Sri Parameswaran[5]

[1] National University of Singapore, Singapore
elemsh@nus.edu.sg
[2] University of Peradeniya, Sri Lanka
ragelrg@gmail.com
[3] University of New South Wales, Australia
ajangelo@cse.unsw.edu.au
[4] University of Peradeniya, Sri Lanka
swarna.radhakrishnan@gmail.com
[5] University of New South Wales, Australia
sridevan@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-201218
June 2012

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

It is of critical importance to satisfy deadline requirements for an embedded application to avoid undesired outcomes. Multiprocessor System-on-Chips (MPSoCs) play a vital role in contemporary embedded devices to satisfy timing deadlines. Such MPSoCs include two-level cache hierarchies which have to be dimensioned carefully to support timing deadlines of the application(s) while consuming minimum area and therefore minimum power. Given the deadline of an application, it is possible to systematically derive the maximum time that could be spent on memory accesses which can then be used to dimension the suitable cache sizes. As the dimensioning has to be done rapidly to satisfy the time to market requirement, we choose a well acclaimed rapid cache simulation strategy, the single-pass trace driven simulation, for estimating the cache dimensions. Therefore, for the first time, we address the two main challenges, coherency and scalability, in adapting a single-pass simulator to a MPSoC with two-level cache hierarchy. The challenges are addressed through a modular bottom-up simulation technique where L1 and L2 simulations are handled in independent communicating modules. In this paper, we present how the dimensioning is performed for a two-level inclusive data cache hierarchy in an MPSoC. With the rapid simulation proposed, the estimations are suggested within an hour (worst case on considered application benchmarks). We experimented our approach with task based MPSoC implementations of JPEG and H264 benchmarks and achieved timing deviations of 16.1% and 7.2% respectively on average against the requested data access times. The deviation numbers are always positive meaning our simulator guarantees to satisfy the requested data access time. In addition, we generated a set of synthetic memory traces and used them to extensively analyse our simulator. For the synthetic traces, our simulator provides cache sizes to always guarantee the requested data access time, deviating below 14.5% on average.

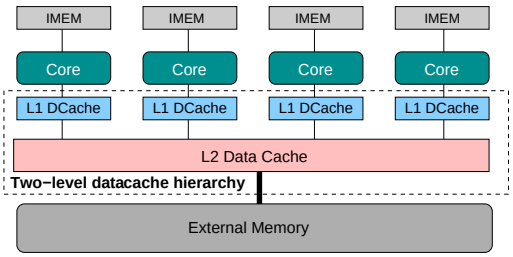

Figure 1.1: Two-level Cache Hierarchy in MPSoC Architecture

1 Introduction

Estimating performance in embedded systems, particularly in the face of many choices is difficult. Two separate aspects make it challenging. The first is that there are many differing hardware components which can be varied (for example, the processor, the co-processors, cache sizes and even instructions); and, the second is that the benchmarks which are executed on the system at design time might not be completely representative of the actual load that the system might endure during its lifetime (for example, in a multimedia system the processing load can vary significantly with the images encountered).

Designers tackle the first aspect by fixing as many components as possible; and the second is addressed by allowing a margin for worst case situations, unforeseen operating systems interventions and interrupts.

In this paper we examine a method of sizing L1 and inclusive L2 data caches for a multiprocessor embedded system. All other components are fixed (i.e. processor, instruction set, coprocessors, etc.). We first size L2 to meet the requirements needed by the system with the given trace (i.e. we assume that L1 cache does not exist). Then we size L1 caches to allow for the margin desired. In a ten processor system, it is possible that we could have 180^{11} configurations[1], and searching for an optimal scheme, or simulating through all of them is not possible. A trace for 100 micro seconds in a 10 processor system will yield 1 billion memory accesses at 1GHz. Thus an optimal method would be difficult, and thus we provide a fast heuristic to meet our provided miss rates and margins.

Cache simulation techniques can assist the designer in estimating the suitable cache configurations long before the system is implemented. Application trace driven simulation [16] is a well known approach to achieve such an estimation in a very short time. A single-pass simulator [25], which is robust, fast and resource generous, is a type of trace driven simulator. It traverses the memory access trace of an application one entry at a time and calculates the number of cache misses for different cache configurations. As the memory access time and therefore the execution time[2] of an application is influenced by the number of cache misses, this number is used in single-pass simulators to decide a suitable cache configuration to satisfy the application deadline and area budget.

Two-level inclusive[3] data cache hierarchies are widely utilised in MPSoCs [5, 18,

[1] assuming 180 different possible configurations for each cache

[2] when the execution time deadline of an application is given, it is possible to systematically derive the required/*requested memory access time*

[3] shared cache contains the superset of private caches as opposed to exclusive cache where the content

23] for sharing data among processors. Figure 1.1 depicts a commonly used two-level MPSoC architecture in contemporary chips [15], which is our target architecture. In our architecture, the processor cores include private L1 and shared L2 *inclusive data caches*, together with private instruction memories (note that similar to [7], our system has instruction memories local to each processor).

Utilising a single-pass simulator to simulate a cache hierarchy as in Figure 1.1 is challenging, nevertheless beneficial in quickly finding the right cache configurations required for a *set of communicating multiprocessor applications or tasks*. The two main challenges are:

1. *Maintaining coherency across processors, where multiple processors can access (read and write) a shared data memory space through a two-level inclusive cache hierarchy*. Current single-pass simulators do not consider such data sharing and therefore the data structure and the simulation strategy have to be adapted and extended to support coherency. Conflict scenarios are already covered for uniprocessors in two-level single-pass simulations [31] and can be directly applied in two-level cache hierarchies for MPSoCs;
2. *Maintaining scalability for both simulation time and storage space of the simulator with the increasing number of L1 private caches (or cores) in the system*. For a given cache hierarchy, such as the one shown in Figure 1.1, the cache simulator must select the suitable configuration for each and every cache memory used in the hierarchy. Single-pass simulators read the application trace only once and simulate all the cache configurations together. Therefore, a large combination of cache configurations have to be simulated together at once and this number will exponentially increase with the number of private L1 caches. Such a simulation would result in exponential growths in both the time needed and the memory storage required.

In this paper, for the first time, we address the challenges discussed above in selecting the cache configurations in a two-level *inclusive data cache hierarchy* to satisfy application deadlines in MPSoCs. The technique we propose takes a bottom-up approach (from L2 to L1), named "DIMSim" (Dual Independent Modular Simulation), that performs independent simulations on each level of the cache hierarchy. Given that the L2 simulation can be handled by the current single-pass simulators, our approach *reduces the complexity of DIMSim by first simulating the shared L2 cache* and then performing the private L1 cache simulations while addressing coherency. As detailed in Section 3, DIMSim uses its L2 module to satisfy the requested data access time constraint and L1 module to satisfy the requested headroom time (an overhead incurred by the system). It is worth noting that the modular approach we have devised is scalable for both simulation time and storage space with the increasing number of L1 private caches (i.e. the number of cores).

DIMSim considers the most widely used First-In-First-Out (FIFO) replacement policy (e.g., Intel XScale [1], ARM9 [3], ARM11 [2] and Tensilica Xtensa LX2 processors [29]) in our architecture. To the best of our knowledge,caches for embedded systems are designed using area and power constraints and the applications are mapped in a post-production stage. We take a different (and we believe a better) route to this approach by providing a notion of application performance during the design phase in an attempt to further optimise the total cache size.

---

will only be present in either level of the cache hierarchy

Problem Statement: Provided the application trace of (cache-less) memory accesses of an MPSoC with shared data, find the cache configurations (set size, associativity and cache line size), for each private L1s and shared L2, to satisfy an allowable number of cache misses.

Layout: The rest of the paper is organised as follows: Section 2 describes the related work, while Section 3 details the DIMSim simulation approach. The experiments and the results are presented in Section 4 and Section 5 respectively. We finally conclude with Section 6.

2 Related Work

Due to power and performance critical nature of modern computer systems, the necessity to find the optimal (or near optimal) cache configuration has become an important issue in computer system design. State-of-the-art cache design space exploration techniques can be broadly categorised into: 1) System Simulations [30], 2) Instruction Set Simulations (in SystemC) [21] and 3) Trace driven simulations [25]. Available system simulation tools such as [30] are difficult to customise and hard to program with respect to cache configurations. These tools incur significant simulation time (even several days) just to simulate a single cache configuration for a specific application in MPSoC. The instruction set simulators [21] are faster to simulate a specific cache configuration, but still simulates one cache configuration at a time and therefore consumes significant amount of time. Trace driven cache simulations [25], however, do not require architecture related information, and are the fastest amongst all, providing cache dimensions well before the design phase. Furthermore, trace driven simulators are capable of simulating multiple cache configurations at a time. Memory footprint of the application in the form of traces is the only input necessary.

Due to higher abstraction with respect to the actual architecture, various types of speedup mechanisms can be applied to trace driven simulation methods. Four of the popular speedup mechanisms applied to trace driven simulations are: 1) Compressed trace simulation, 2) Parallel simulation, 3) Sample-based simulation and 4) Single-pass simulation. *Compressed trace simulation* [20, 22, 26, 27] prunes redundant information to compress the application's memory access trace, resulting in a reduced simulation time. *Parallel simulation techniques* perform the simulation of a group of cache configurations in parallel on multiple processors [4, 10, 13]. *Sample based simulation* [6, 17, 19, 28] combines simulation and sampling such as getting the memory trace for a fixed number of cycles to reduce simulation time overhead. In *single-pass simulation*, the application trace is read only once and various cache configurations are simulated together. Unlike parallel simulation, single-pass simulation utilises one processing unit as optimally as possible and can be combined with parallel or compressed trace simulation methods for further speedup [25]. Single-pass simulation approaches usually exploit cache inclusion properties [11, 12] and custom-tailored data structures (such as binomial tree [16, 24]) to reduce processing time without the help of additional hardware.

To find the most appropriate cache configurations for an MPSoC system like the one in Figure 1.1, most suitable configurations must be found for all the private and shared cache memories. However, to simulate the behaviour of one processor's private/shared cache memory, other private and shared cache memories in the system must also be simulated to maintain coherency. Therefore, when single-pass simulation methods are

utilised in finding the appropriate caches in MPSoCs, a large combination of cache configurations has to be simulated by reading the application trace once. As a result, space and time consumption becomes a critical concern. To the best of our knowledge, no single-pass method has ever been adapted to simulate cache hierarchy of an MPSoC and therefore none of them ever had to address coherency issues. In this paper, we focus on designing a fast and space efficient cache simulation technique for *two-level inclusive data cache hierarchies* in MPSoCs.

Therefore, the **contributions** of DIMSim are as follows:

- We propose the first ever cache simulator (DIMSim) for MPSoCs that adapts the *single-pass strategy*. DIMSim will not only simulate a two-level data cache hierarchy (private L1s and shared L2) rapidly, but also work with deadline constraints. Therefore, DIMSim addresses the two main challenges in simulating a data cache hierarchy for deadline constrained MPSoCs: (1) coherency; and (2) scalability.
- The scalability is addressed (1) by breaking the simulator into two *independent* communicating modules (L1 and L2 simulators) and (2) by enabling the private cache simulations to be performed in parallel for each processor (more detail in Section 3).
- The coherency issue is addressed through a novel bottom up (L2 first and then L1s) simulation with *communicating* modules. As detailed later in the paper, the information passed through the interaction was used effectively to handle coherency.

3 Methodology

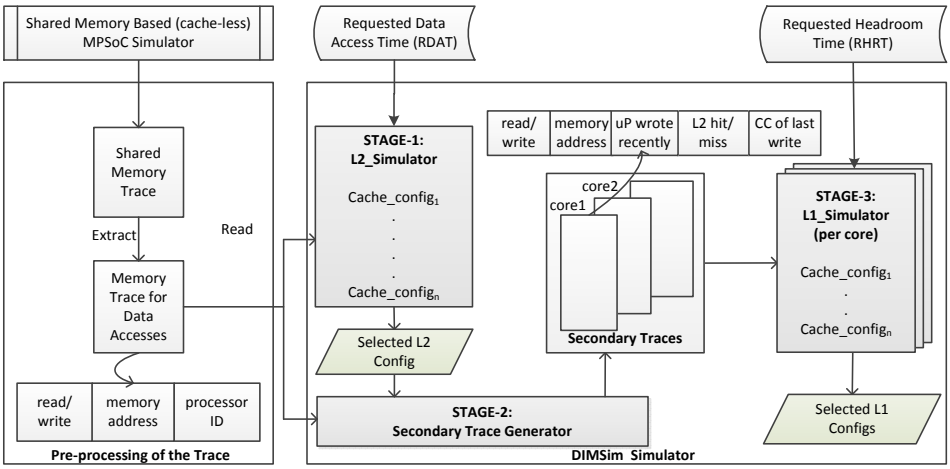

Figure 3.1: DIMSim Simulation Flow

Given that DIMSim is expected to take the application deadline (also known as *requested execution time*, RET) as the input and to decide of the suitable cache configurations, we could formulate Equation 3.1 to represent RET. In Equation 3.1, RET refers to the requested execution time and AET refers to the actual execution time. $TTDM$ is to the total time spent on data memory access and $TTNM$ is the total

time spent on non data memory operations. $TTOH$ refers to the total time spent on overheads such as the time spent for handling the scheduling and operating system overheads which are system specific and are not covered by the application trace. Our timing model as in Equation 3.1 is formulated such that introducing a cache hierarchy or changing cache sizes would only effect $TTDM$ and not $TTNM$ or $TTOH$.

$$RET \geq AET = TTDM + TTNM + TTOH \tag{3.1}$$

The ***RET*** is provided by the user, considering the application(s) throughput and latency constraints (e.g., JPEG encoder application has to encode an image within 20ms for the K7 Pentax Camera). The user will also provide a requested headroom time, $RHRT$. DIMSim addresses to satisfy both $TTDM$ (via its L2 estimation) and $TTOH$ (via its L1 estimation).

Figure 3.1 depicts the DIMSim flow. Pre-processing of the trace to create the shared memory trace and add tags (explained later) takes place before the simulation. The DIMSim simulation itself is broken down into three stages: (1) L2 simulation; (2) Secondary trace generation; and (3) L1 simulation. Pre-processing is performed to prepare the trace for the simulator. While the MPSoC embedded system (without caches) is executing an application as communicating tasks or multiple applications, memory accesses are observed and captured at the memory controller hence the order of access is correct. The data accesses from the trace are then extracted and annotated (read/write, memory-address, processor-ID) as depicted.

DIMSim approach consists of the following three stages:

STAGE-1: Finding the suitable shared L2 data cache configuration that satisfies the requested data access time

As depicted in Figure 3.1, in STAGE-1, DIMSim takes the whole data access trace of the application and the requested data access time $RDAT$ ($= RET - TTNM$) and estimates the shared L2 data cache configuration. In STAGE-1, DIMSim assumes that the system has only a shared memory and there exist no L1 caches. As the total time spent on data memory (when only L2 cache is present ($TTDM_{L2}$)) of an application will increase with the number of cache misses, the acceptable number of cache misses in L2, called "ML", that satisfies the $RDAT$ is calculated using Equation 3.2. As shown in the equation, when a data is requested by the processor, the L2 is checked first and if not found, the memory is checked. Hence, every miss in L2 will incur T_M[1] seconds. On the other hand, the data which is found in L2 is served by the L2 in T_{L2} [2] seconds.

Considering that the total number of accesses is A by all the processors, the total hit time is $(A - ML) * T_{L2}$ seconds.

$$RDAT \geq TTDM_{L2} = ML * (T_M + T_{L2}) + (A - ML) * T_{L2} \tag{3.2}$$

By reading the trace file once, DIMSim simulates all available cache configurations for L2. The result of this simulation would be a set of cache configurations that satisfies the ML. Among them, the smallest one in size is chosen as the suitable L2 shared cache to minimise the area and therefore to minimise the power consumption. The L2 simulation is implemented similar to CIPARSim [11], where pruning is applied to avoid simulating cache configurations which cannot support ML.

[1] T_M includes (the worst case) time to send a request from the L2 cache to memory, search time in memory, and time to send data from the memory to the L2 cache

[2] T_{L2} includes (the worst case) time to send a request from the processor to L2 cache, time to handle the coherency protocol, search time in L2, and the time to send data from L2 cache to processor

STAGE-2: Secondary trace generation to be used by the L1 simulator

When an L2 cache is shared among all the processor cores, coherency is the first challenge that the simulator must handle in finding the suitable private L1 caches. We propose, both independent and communicating, L1 and L2 simulators, passing information from L2 simulator to L1 simulator. The interaction between the simulators is enabled through a secondary trace file that is derived by instrumenting our original trace depending on the selected L2 cache in STAGE-1.

As depicted in Figure 3.1, for each memory block address, a secondary trace file records: 1) operation code of the data access (e.g., data read or write), 2) the requested memory address, 3) the name of the processing core that wrote in that particular memory block most recently, 4) whether the block was a hit or miss in the selected L2 cache, and 5) clock cycle when the memory block was last written in the shared L2 cache. All the extra information (compared to the original trace) recorded in the memory block address help in implementing cache coherency protocols for MPSoCs. The next stage describes the process of coherency protocol implementation.

STAGE-3: Finding suitable L1 cache configurations for the selected shared L2 cache

As shown in Figure 3.1, once the secondary trace file is generated, it is divided into smaller trace files by grouping the accesses from each processing core into a separate file. To find the suitable private L1 cache for a particular processor core, its respective smaller trace file is utilised. Dividing the secondary trace file into smaller trace files allows our simulation process to be parallelised and therefore the L1 simulation can be made faster if desired.

DIMSim makes use of the fact that adding L1 caches will speed up the system to satisfy the second timing requirement, $RHRT$, provided as another input (apart from the requested execution time). The $RHRT$ requirement is distributed among cores proportional to the number of data memory accesses by each core. The suitable L1 cache configurations are selected to satisfy the $RHRT$ requirement of each core. It is worth noting that DIMSim ignores the fact that the individual cores can access their private L1 caches simultaneously as the input trace does not have this information. Therefore, DIMSim might potentially satisfy the $RHRT$ requirement with a larger margin.

By reading the corresponding smaller secondary trace file once, DIMSim simulates all available cache configurations to find the suitable private L1 per core. For every cache configuration simulated per core, the total number of memory access requests that cannot be served by the particular cache configuration (L1 misses) is recorded at runtime. The DIMSim L1 simulation component reads each smaller trace per core and scans through every entry which is fed into Algorithm 1 as explained later in this section.

To calculate the total number of cache misses incorporating coherency, DIMSim records the last content update time in each cache line in the simulated cache configurations. Any memory address which is indicated as a L2 miss in the secondary trace file will also cause a miss in the private cache memory during simulation due to the inclusive property. When a requested memory address content is found in shared cache (L2) as well as in the private cache memory (L1), the last update time written in the private cache line, which is holding the content, and the L2 update time written on the secondary trace file is compared. If it matches, cache hit is declared; otherwise, cache miss is recorded. Cache miss is also recorded if the requested memory address content

is missing in the private cache configuration. On a cache miss, the requested content is placed in a cache line in the simulated cache configuration along with its last update time in the shared L2 cache memory. In this way, DIMSim approach implements coherency protocol without additional storage and time consumption.

Algorithm 1: AddressEvaluation(RequestedAddress(RA))

```
1  if (RA was not available in L2 cache) then
2      HandleCacheMiss(RA);
3  else
4      Search RA in the look-up Table (CLT);
5      if (RA is not available in CLT) then
6          HandleCacheMiss(RA);
7      else
8          select the root tree level L = 0 (for cache set size S = 2^L);
9          For both data read and write:
10         while 2^L is not larger than the largest set size do
11             A = the smallest associativity;
12             while A is not larger than the largest associativity do
13                 if (RA is available in the selected cache configuration in the tree) then
14                     if (RA writing time does not match with L2 writing time) then
15                         Record a cache miss in the cache configuration and update the simulation tree;
16                         Attach the writing time with RA in the tree;
17                         Update the records of CLT to indicate tag insertion;
18                     else
19                         Record a cache hit for the cache configuration;
20                 else
21                     Record a cache miss in the cache configuration and update the simulation tree;
22                     Attach the most recent writing time with RA in the tree;
23                     Update the records of CLT to indicate tag insertion;
24                 A = Next larger associativity;
25             L = L + 1; (continue to the next level)
```

Algorithm 1 reads every entry in the smaller trace and tries to evaluate whether it is a hit or a miss in each and every simulated cache configurations. The algorithm receives the address entry (i.e., Requested Address RA) from the secondary trace file (for every private L1 there will be one secondary trace file). Every entry in the secondary trace file is tagged with L2 hit or miss information as shown in Figure 3.1. Due to the inclusive property of the L1 and L2 caches, a miss in L2 will cause a miss in L1 as well. As shown in Algorithm 1, Function HandleCacheMiss is called when the RA is not available in the L2 cache (the L2 hit for that address in the secondary trace file indicates that the entry is available, likewise an L2 miss for that entry indicates that the address is unavailable). If the RA is available in L2 (else statement in Line 3), search in the L1 data structure (i.e., Central Look-up Table (CLT)) whether the RA is available. The Function HandleCacheMiss is called if RA is unavailable in CLT, which indicates a miss in that particular L1. A hit in L1 (i.e., RA is available in CLT; else statement starting at Line 7) requires recording information in the simulation tree. Starting from the root of the simulation tree all the levels in the tree (the simulation tree is explained in detail in Section 3.1) are traversed, until we reach the level with the largest set size. It is worth noting that misses can arise due to a data read or write. For each level (i.e., set size), iterate over all the associativities, starting from the smallest (this iteration is considered as analysing a cache configuration). If RA is found in the selected cache configuration in the tree, check whether the writing time matches with the L2's writing time. A match suggests an L1 data cache hit, hence will be updated as hit for that configuration in the simulation tree. A mismatch in timings denotes an L1 cache miss.

Hence the cache configuration in the simulation tree is updated, including the writing time and a record in CLT to indicate tag insertion. If the cache configuration does not include the RA, record a cache miss in the configuration and update the simulation tree, while attaching the most recent writing time for the RA and an update in CLT to indicate tag insertion. The inner loop is iterated for each associativity and the outer for set size.

Function HandleCacheMiss(RA)

1 Record a cache miss in all the L1 cache configurations
2 Update the simulation tree and attach the most recent writing time with each RA tag in the tree;
3 Update CLT for RA;

The Function HandleCacheMiss is called when the RA is either not in L2 cache (i.e., checked using the secondary trace file) or not in L1 (i.e., checked in CLT). All the cache configurations will be recorded as cache miss for the RA. The simulation tree and CLT are updated with the most recent writing time of the RA and RA entry respectively.

3.1 DIMSim Data Structure

The data structures used in DIMSim are influenced by CIPARSim [11], and the single cache configuration simulator DineroIV [8]. DIMSim utilises a central look-up table (CLT), as shown in Figure 3.2. Simulated cache configurations are represented using a binary simulation tree which is used to update the look-up table. In DIMSim, the binary simulation tree that is utilised in [11] is modified to accommodate the last writing time of each memory address tag, to handle coherency.

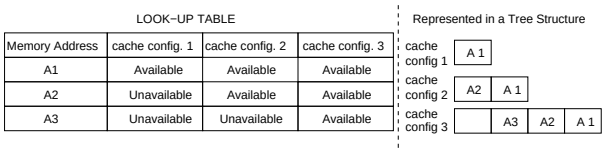

Figure 3.2: Central Look-up Table in DIMSim

For each cache line size, both the L2 and L1 simulation components in DIMSim utilises a simulation tree to represent cache memories and a Central Look-up Table (CLT), such as the one presented in Figure 3.3 and Figure 3.2 respectively, to determine cache hit/misses quickly. Each entry in the CLT records the availability of a memory block content in simulated cache configurations. For example, the memory block address A2 is available in cache configurations 2 and 3, but is unavailable in configuration 1. As the look-up table entries are sorted by memory block addresses, binary search is applied to quickly figure out the availability of a particular memory block in different cache configurations. All the cache configurations are included in the look-up table. In a simulation tree, each level represents a cache configuration. Each tree node represents a cache set in the cache configuration.

Figure 3.3 illustrates a simulation tree[3] representing three cache configurations with set size two, four and eight. The tree starts with a FIFO cache configuration containing two cache sets. The top left node, named '0', represents the cache set with index 0 in

[3]the simulation tree is developed based on CIPARSim [11]

the cache with set size 2. Similarly, the second node '1' refers to cache set 1. At the second level of the two trees, there are a total of four nodes stamped '00', '10', '01' and '11'. Thus the second level represents a FIFO cache with set size of four, and the numbering within the nodes represent the respective cache sets as shown in Figure 3.3. Similarly, the third level (illustrated as the bottom level in Figure 3.3), will represent a FIFO cache with eight sets. More caches with bigger set sizes can be represented by expanding the tree further, which is not presented in this paper. It is assumed that the traditional mapping from memory address to cache sets is performed by taking the lower bits of the memory address to determine the set. Thus, the elements that map to any node of a tree will be mapped to its child nodes in the next level and only its child nodes.

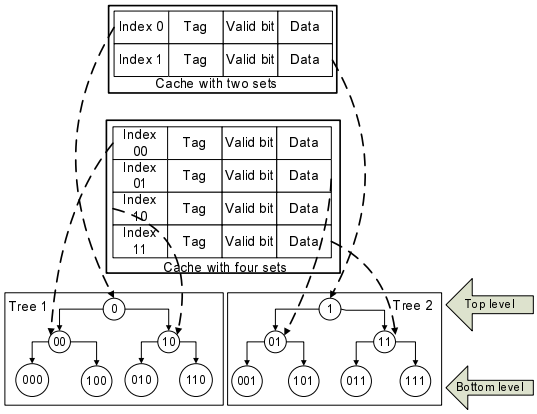

Figure 3.3: The Simulation Tree

Lists with varying number of nodes are attached with each tree node to represent different associativities. Each node inside a list represents a cache line and will have a pointer to the memory block address inside a look-up table set to indicate which memory block resides there. Each node in a simulation tree also stores the most recently inserted (MRI) memory block address of associativity two when the smallest associativity is two. The contents inside the associativity lists are replaced as in the FIFO caches. A bit called "Track_Flag" is also stored with each tree node when caches with associativity two are simulated. Whenever a new memory block is inserted into the list for associativity two, the Track_Flag is set to false (or 0). When an existing memory block of the associativity two in the selected tree node is re-accessed, the Track_Flag is set to true. A cache hit in the FIFO associativity two with Track_Flag set to true indicates that the memory block content is available in all the larger FIFO cache configurations in the same simulation tree. If the smallest associativity is larger than two, a bit "Intersection_Flag" per larger associativity is associated with each cache line in the smallest associativity list of a simulation tree node. Whenever a new memory block tag is inserted in a cache line in the smallest associativity list of a tree node, the Intersection_Flag is set to true if the same memory block tag is at least $((2 \times A_X) - 3)$ (where A_X is the smallest associativity) elements away from the replacement pointers in the other larger associativity lists in the same tree node. When a memory block content in the smallest associativity in a tree node is re-accessed, simulation can be avoided in the larger associativities if the Intersection_Flag is found true. To simulate direct mapped caches, Most Recently Accessed (MRA) tag must be saved for each simulation tree

node.

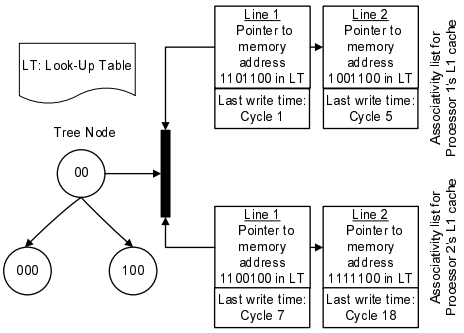

Figure 3.4: Associativity Lists

In Figure 3.4, an example tree node '00' from Figure 3.3 is presented with two FIFO associativity lists illustrated (namely associativity=2 and associativity=4) (this example is from [11]). Node '00' is from the second level in the tree of Figure 3.3. The second level of the tree of Figure 3.3 represents a FIFO cache with set size 4. The first cache line of the list for associativity 2 has a pointer to the memory block address "1101100" in the look-up table of DIMSim. Using these pointers, DIMSim can update the look-up table's bit arrays when the address "1101100" will be evicted from the associativity 2's list in tree node '00' due to a miss for that node. In the L1 simulator component of DIMSim, each cache line also stores the last writing time of the containing memory address tag to handle coherency.

4 Experiments

We created synthetic data memory access traces to represent inter-communicating applications, mapped on different processors (a four core MPSoC), to perform an extensive analysis in our simulator to cover a significant portion of the design space. The synthetic traces are generated to ensure a better coverage of the data sharing among cores, thus representing coherency and conflict scenarios extensively. As explained in methodology, the memory traces are generated assuming a cache-less system and then the target is to estimate the cache sizes (we assumed a four-core system with private L1s and a shared L2) for different sharing scenarios. The traces are categorised based on the following features:

1. The number of processors that uses the shared memory: we assumed that a selected number of processors out of all four use shared data. Therefore, the rest of the processors use private data. When core1 and core2 sharing data and core3 and core4 accesses private data, we named this scenario **case 1**. When core1, core2 and core3 share data and core4 uses private data, we named this scenario **case 2**. When all four cores share data, we named this scenario **case 3**.

2. Data usage: the percentage of shared data accesses out of the total data memory accesses in the trace. We have used the following percentages in our experiments: 25%, 50%, 75% and 100%. The unshared data will be considered for (and will influence) conflict misses.

To further analyse our simulator on real applications, we implemented a six core (more number of cores compared to the synthetic experiments to verify scalability and performance) cache-less multiprocessor implementation using the Tensilica tool set [29] and executed JPEG encoder and H264 encoder (only the motion estimation kernel) to generate traces for different image and video benchmarks. Both the applications are partitioned into multiple communicating/sharing tasks which are mapped on separate processors. We chose JPEG and H264 since they represent real time applications which are widely used in embedded systems and have distributed sharing of data across all the cores.

For our simulation experiments, we executed the DIMSim simulator on a machine with a dual core Opteron64 2GHz processor, 8GB of main memory and 1MBytes L2 cache pre-distributed among the processing cores.

Cache Set Size=2^i	$0 <= i <= 14$
Line Size=i Bytes	$i \in 4, 16, 64$
Associativity=2^i	$1 <= i <= 4$

Table 4.1: Cache Configuration Parameters

In our experiment, each private L1 or shared L2 cache has 180 possible FIFO configurations as shown in Table 4.1. We used *TL* = 5 ns and *TM* = 15 ns (based on Xtensa processor [29]), assuming that all the applications are mapped on a 1GHz processor with one clock cycle L1 cache latency.

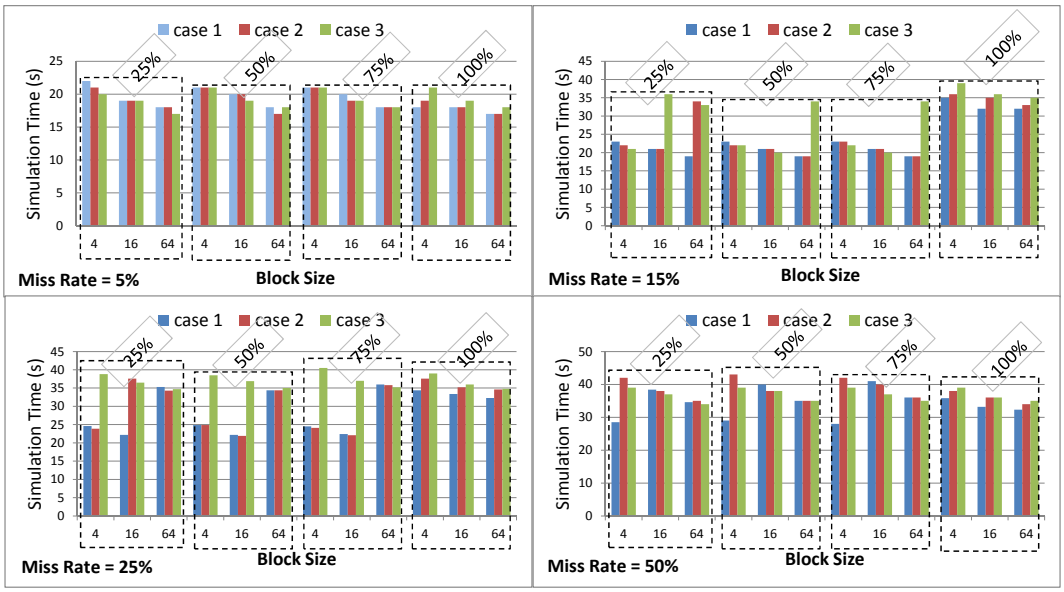

Figure 4.1: Simulation Time in DIMSim for Synthetic Applications

5 Results

We evaluated our DIMSim simulator using two different metrics: 1) simulation time, which denotes the time consumed by the DIMSim simulator to produce a solution; and

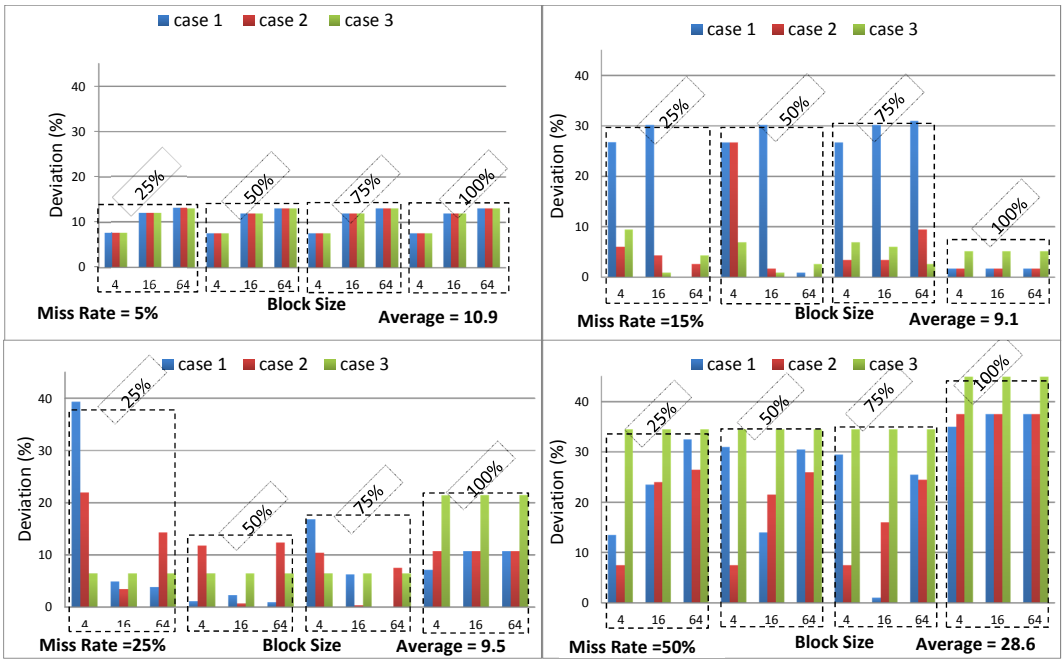

Figure 4.2: Deviation in Synthetic Applications

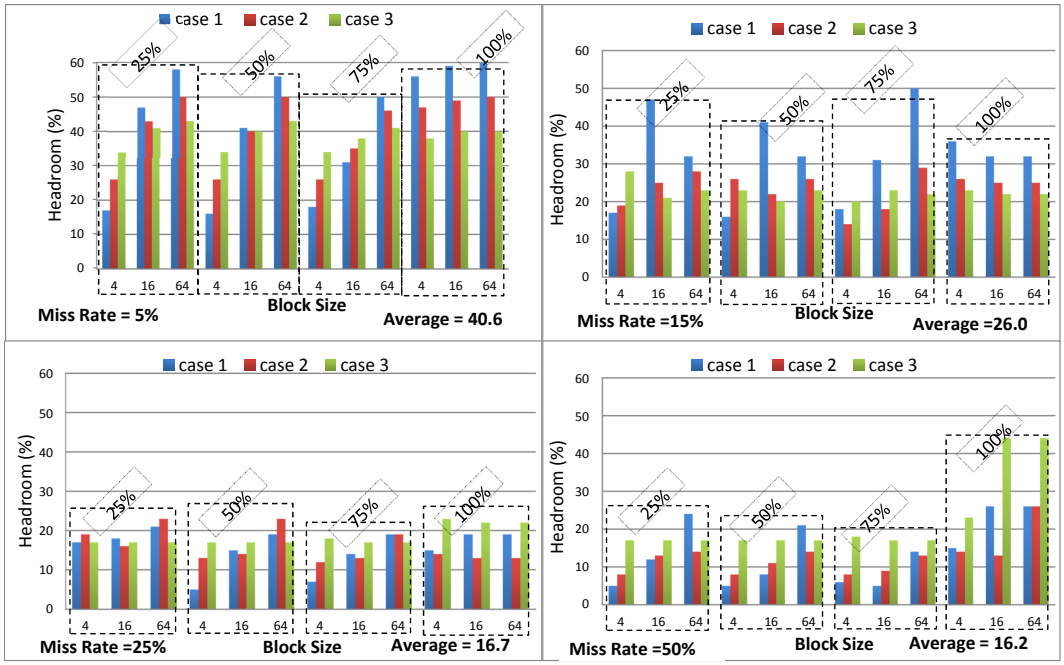

Figure 4.3: Maximum Allowable Headroom in Synthetic Applications

2) deviation, which indicates the difference between the *requested data access time* ($RDAT$) and the time supported by the selected (i.e., suitable) L2 cache configuration

Apps.	Blk. Size (B)	Miss Rate (Directly related to $TTDM$)																	
		5%	10%	15%	20%	25%	50%	5%	10%	15%	20%	25%	50%	5%	10%	15%	20%	25%	50%
		Simulation time (Sec)						Headroom (%)						Deviation (%)					
JPEG																			
graph.	4	1752	1800	1779	1778	3251	3351	23	19	19	19	12	9	6	4	14	22	7	20
	16	2944	3015	2911	3017	3090	3183	29	29	29	19	19	10	6	17	26	6	14	14
	64	2852	3017	2806	2854	2898	2816	37	37	37	22	22	25	11	21	29	2	11	38
lowg	4	220	243	227	235	451	455	22	20	20	20	13	10	0	4	14	22	7	20
	16	392	406	398	405	416	424	32	32	23	23	23	12	6	17	5	14	21	18
	64	375	376	351	386	399	416	41	41	41	25	25	9	11	21	29	9	16	3
lena	4	1735	1815	1780	1790	3217	3371	23	19	19	19	12	9	6	4	14	22	7	20
	16	2936	3023	2909	2995	3064	3171	29	29	29	19	19	10	6	17	26	6	14	14
	64	2821	3027	2817	2857	2888	2886	37	37	37	22	22	25	10	21	29	2	11	38
criss	4	1758	1855	1782	1781	3259	3331	23	19	19	19	12	9	6	4	14	22	7	20
	16	3015	3153	2912	3013	3055	3113	29	29	29	19	19	10	6	17	26	6	14	14
	64	2838	3011	2828	3002	2943	2892	37	37	37	22	22	25	10	21	29	2	11	38
photo1	4	1722	1763	1797	1830	3289	3318	23	19	19	19	12	9	6	4	14	22	7	20
	16	2985	3004	2998	3120	3086	3169	29	29	29	19	19	10	6	17	26	6	14	14
	64	2797	2835	2826	2880	2902	2842	37	37	37	22	22	25	10	21	29	2	11	38
photo2	4	1751	1788	1822	1824	3303	3316	22	19	19	19	12	9	6	4	14	22	7	20
	16	2967	2944	3038	3067	3073	3140	29	29	29	19	19	10	6	17	26	7	15	14
	64	2799	2798	2804	2889	2901	2864	37	37	37	22	22	25	10	21	29	3	11	38
H264																			
bluesky	4	252	1384	1435	1525	1534	1561	NS	7	6	6	5	5	NS	1	1	6	1	28
	16	1204	1217	1331	1339	2361	2284	17	16	14	13	12	6	1	3	4	5	8	3
	64	1102	1140	1219	2186	2200	2153	20	20	20	14	14	8	0	12	21	3	11	13
river	4	290	1440	1516	1472	1614	1593	NS	7	6	6	5	5	NS	1	1	6	1	28
	16	1228	1269	1295	1215	2347	2365	17	16	14	13	12	6	1	3	4	5	8	3
	64	1156	1140	1203	2043	2278	2261	20	20	20	14	14	8	0	12	21	3	11	13
station	4	277	1432	1488	1422	1549	1550	NS	7	6	6	5	5	NS	1	1	6	1	28
	16	1220	1160	1253	1230	2345	2282	17	16	14	13	12	6	1	3	4	5	8	3
	64	1142	1488	1134	2219	2159	2200	20	20	20	14	14	8	0	12	21	3	11	13
pedest.	4	303	1470	1492	1436	1604	1605	NS	7	6	6	5	5	NS	1	1	6	1	28
	16	1262	1323	1312	1234	2364	2306	17	16	14	13	12	6	1	3	4	5	8	3
	64	1151	1207	1134	2008	2198	2232	20	20	20	14	14	8	0	12	21	3	11	13
rushhr	4	281	1457	1507	1419	1567	1574	NS	7	6	6	5	5	NS	1	1	6	1	28
	16	1250	1267	1256	1236	2353	2396	17	16	14	13	12	6	1	3	4	5	8	3
	64	1160	1167	1155	2222	2210	2243	20	20	20	14	14	8	0	12	21	3	11	13
tractor	4	279	1431	1475	1449	1544	1582	NS	7	6	6	5	5	NS	1	1	6	1	28
	16	1231	1262	1285	1249	2388	2427	17	16	14	13	12	6	1	3	4	5	8	3
	64	1157	1151	1137	2115	2155	2173	20	20	20	14	14	8	0	12	21	3	11	13

Table 4.2: DIMSim Performance Analysis for JPEG and H264 Benchmarks

($TTDM_{L2}$). Since our focus is on data caches, we use $RDAT$ instead of the *requested execution time* (RET).

However, our method can be easily extended for RET using the equations provided in Equation 3.1. DIMSim is experimented for different block sizes (only 4, 16 and 64 are shown due to lack of space). It is worth nothing that the block sizes are fixed across L1 and L2 cache configurations for a particular experiment. We have selected so to avoid violating the two-level inclusion property as discussed in [14].

As detailed in Section 3, $TTOH$ in Equation 3.1 is accommodated into DIMSim by selecting the suitable L1 cache sizes. DIMSim allows its user to specify the $TTOH$ demand from the system as a parameter ($RHRT$) and uses the speedup gained by introducing suitable L1 caches to satisfy the demand. We have selected the smallest possible L1 caches and computed the maximum allowable headroom time in our experiments and the results are reported here (in addition to the simulation time and the deviation).

For all our experiments, we used a set of $RDAT$ values to see DIMSim's behaviour across different requested data access times. We use Equation 3.2 to calculate the number of misses in L2 (ML) when an $RDAT$ is known and the set of $RDAT$ is selected such that the miss rates[1] are 5%, 15%, 25% and 50% for synthetic traces

[1]The miss rate, in percentage, is computed as the number of misses in L2 (ML) divided by the total memory accesses (A).

and 5%, 10%, 15%, 20%, 25% and 50% for traces from JPEG and H264 benchmark applications. These miss rates are chosen (more analyzed but not reported due to lack of space) to cover a wide range of miss scenarios.

Figure 4.1 depicts the simulation time of DIMSim with the synthetic trace files. The simulation time (in seconds), on average, increases when the miss rate increases since more cache configurations are simulated for a higher miss rate. In other words, when the given miss rate is smaller, only a few large cache configurations can satisfy its respective ML, while other cache configurations are pruned away. This pruning point in the simulation is reached at a much earlier time for smaller miss rate hence the significant variations in the simulation time. The maximal simulation time of 42 seconds is observed at two places: (i) when 50% miss rate is imposed, at 25% sharing, in case 2 and block size of 4, and (ii) when 50% miss rate is imposed, at 50% sharing, in case 2 and block size of 4. The minimal simulation time of 17 seconds is observed at several places. Compared to the crude exhaustive approach [9] (i.e., simulating every single cache configuration at a time), a seven orders of magnitude in average speedup is estimated in synthetic applications across all the scenarios. The four core system with each core having a private L1 cache and a shared L2 cache will have around 2.3 billion possible cache configurations when each cache has 180 possible configurations.

$$Deviation = \frac{(RDAT - TTDM_{L2}) * 100}{RDAT} \tag{5.1}$$

Figure 4.2 depicts the deviation between the requested data access time ($RDAT$) and the total time spent on data memory accesses with only L2 ($TTDM_{L2}$). $Deviation$ is calculated, as per Equation 5.1, for each application trace on the suggested L2 cache from DIMSim using Equation 3.2. DIMSim suggested cache hierarchy's $TTDM_{L2}$ is *always smaller than* $RDAT$, hence $Deviation$ will be always positive. This is an indication that DIMSim suggested cache hierarchy never fails to satisfy the timing constraint of the target deadline constrained application on a given hardware platform. The deviation increases with the miss rate for an application until the number of misses result in a different cache configuration being chosen. Since the cache sizes are exponential, deviation incurs significant increase when miss rate increases, until the next (i.e., smaller cache to support more misses) cache configuration is chosen. A sharp decrease in deviation is observed with the next cache configuration. For example, the first bar in each miss rate plot (case 1, block size 4 and 25% sharing) indicates that the deviation increases from 8%, 27%, 38% for 5%, 15% and 25% miss rates respectively, and then drops to 14% for 50% miss rate. The chosen cache size was the same for 5%, 15% and 25% miss rates and then dropped to the next smaller cache for 50% miss rate, because of larger number of misses requested. The deviation fluctuates between 4% and 60% (only in 6% of the bars are over 50% in deviation). As mentioned above, when the miss rate is smaller, only a few large L2 caches will be simulated. An average deviation of 25% is recorded in synthetic applications across all the scenarios (best case 5% and worst case 60%). Variation in deviation is quite unpredictable since it is based on the number of misses and respective supporting cache configuration. Since the cache configuration sizes exponentially increase, a step to the next level cache will significantly reduce $TTDM_{L2}$.

$$Headroom = \frac{(TTDM_{L2} - TTDM) * 100}{TTDM_{L2}} \tag{5.2}$$

Figure 4.3 shows the maximum allowable headroom (for the smallest L1) observed in synthetic applications for different miss rates. The Headroom is calculated, as per Equation 5.2, for each application trace. In Equation 5.2, $TTDM_{L2}$ is the total data

memory access time when only L2 is present in the system. On average, the maximum headroom decreases with the miss rate. A maximum of 60% headroom is observed at 5% miss rate for 100% sharing and case 1, at block size 64. There was a minimum of 5% headroom occurred in several places at case 1. The headroom allows the designer to realize and choose the right cache configurations based on the extra overhead in the design.

The code of JPEG and H264 are partitioned into six different communicating tasks which share information via the shared memory. Table 4.2 depicts the Simulation time (in seconds), Deviation (in percentage) and Headroom (in percentage), for several JPEG and H264 benchmarks. The first column details the benchmark names for applications JPEG and H264, while the second column indicates the block sizes in bytes and the remaining columns depict the simulation time, deviation and the headroom for different requested miss rates.

The shortest simulation time for JPEG is 220sec and observed for the *lowg* benchmark with block size 4 bytes and 5% miss rate. A time of 3371 sec was the maximum simulation time noted at the *lena* benchmark with block size 4 bytes and 50% miss rate. The average simulation speedup across all the benchmarks of JPEG is 11.0 in $\log_{10}$ scale compared to the estimated (and crude) exhaustive simulation time. The six core system with each core having a private L1 cache and a shared L2 cache will have around 8.4 trillion possible cache configurations when each cache has 180 possible configurations. The maximum headroom was 41% at *lowg* benchmark. On the other hand, the minimum was 9% and observed in several benchmarks for 50% miss rate. On average, the headroom is 18.5%. For H264, the maximum simulation time is 2427 seconds and the minimum is 252 seconds, occurred at (50% missrate, *tractor*, B16) and (5% miss rate, *bluesky*, B4) respectively. The maximum, minimum and average speedups, in simulation time, for H264 benchmarks are 11.2, 10.9 and 11.0 respectively in $\log_{10}$, compared to the crude exhaustive simulation. The minimum for the allowable headroom is 5% and the maximum is 20%. A minimum of 0% deviation is observed at bluesky for block size 64, indicating the exact match of the cache configuration (i.e., $TTDM_{L2}$) with the requested time (i.e., $RDAT$). A maximum deviation of 28% has resulted at several places for H264 benchmarks.

	blk. size	Miss Rate					
		5%	10%	15%	20%	25%	50%
JPEG							
L1	4	6×(2,2)	6×(2,2)	6×(2,2)	6×(2,2)	6×(2,2)	2×(1,2),3×(2,2),(1,2)
	16	6×(2,2)	(1,2),(64,2),4×(2,2)	(1,2),(64,2),4×(2,2)	2×(1,2),4×(2,2)	2×(1,2),4×(2,2)	2×(1,2),3×(2,2),(1,2)
	64	(1,2),(16,8),4×(2,2)	2×(1,2),4×(2,2)	2×(1,2),2×(2,2),2×(1,2)	2×(1,2),4×(2,2)	2×(1,2),3×(2,2),(1,2)	(1,2),5×(2,2)
L2	4	(8,8)	(2,16)	(2,16)	(2,16)	(1,16)	(1,8)
	16	(1,16)	(1,16)	(1,8)	(1,8)	(1,8)	(1,4)
	64	(1,8)	(1,8)	(1,8)	(1,4)	(1,4)	(1,2)
H264							
L1	4	NS	6×(1,2)	6×(1,2)	6×(1,2)	6×(1,2)	6×(1,2)
	16	6×(1,2)	6×(1,2)	6×(1,2)	6×(1,2)	6×(1,2)	6×(1,2)
	64	6×(1,2)	6×(1,2)	6×(1,2)	6×(1,2)	6×(1,2)	6×(1,2)
L2	4	No Sol.	1024×16	64×8	16×16	4×16	2×16
	16	16×16	16×8	4×16	2×16	1×16	1×4
	64	4×16	4×16	4×16	1×8	1×8	1×4

Table 5.1: Suitable Cache Configurations

Simulation times of JPEG are relatively larger than H264 due to the fact that the number of memory accesses in JPEG are higher than H264 (i.e., higher number of entries in the trace). It is worth to note that we only partitioned and experimented the motion estimation kernel of the H264, hence the less number of data accesses compared to the full-on JPEG encoder. In several cases, such as at bluesky and riverbed for 4 bytes block size and 5% miss rate, DIMSim is unable (*NS = No Solution*) to provide any

cache hierarchy for the requested miss rate. In these cases, DIMSim could not find an L2 configuration which satisfies the requested miss rate (or $RDAT$) from the provided list of L2s shown in Table 4.1. As a result, the simulation time is significantly low as no L1s are simulated. Even though the images tested for JPEG are of different sizes, each benchmark's simulation time and deviation per block size are quite similar to the other benchmarks. A similar variation is observed for H264. The underlying cause for such a behaviour is similar number of memory accesses and memory access patterns within the selected benchmarks of each application. At the end of the simulation, the smallest L2 and the smallest L1s are chosen as the suitable cache hierarchy from the provided set of solutions (as mentioned in Section 3).

Table 5.1 presents the suggested set associative cache configurations by DIMSim for all the block sizes and miss rates. The numbers, in the form of a tuple (set size, associativity), are divided for private L1s and shared L2. The L1s are tabulated in the sequence of cores. For example, for cache miss rate of 25% and block size of 64 bytes, the first two processors should have 2-way fully associative (i.e, 2×(1,2)) L1 caches, the next three processors should have 2-way set associative caches with 2 sets (i.e, 3×(2,2)) and the last processor can deploy a 2-way fully associative cache. The proposed shared L2 cache for this scenario is a 4-way fully associative cache.

6 Conclusion

We proposed a bottom-up cache simulation approach to provide estimations that satisfy the deadline constraints for a system containing two-level inclusive data cache hierarchies in deadline-based MPSoCs. For the first time, a trace based single-pass simulator is adapted for simulating a two-level cache hierarchy of an MPSoC. Our simulator addressed the two main challenges on simulating a two-level cache hierarchy in an MPSoC, namely coherency (that occurred due to shared data cache) and scalability (that occurred due to the large number of cache combinations to be simulated) for simulation time and storage space. Our simulator, in the worst case, took less than an hour to simulate an application (from the set of benchmarks considered) to estimate the L2 shared cache and L1 private caches for a two-level cache hierarchy. The cache configurations selected by our simulator deviates from the requested data access time by 16.1%, 7.2% and 14.5% in JPEG, H264 and synthetic applications respectively. The deviations are always positive meaning the deadline requirements are always satisfied. Therefore, our simulator is both fast and reasonably accurate in producing results, satisfying the deadline constraints.

7 Acknowledgement

The authors would like to thank Hong Chinh Doan and Su Myat Min Shwe from the University of New South Wales, Australia, for providing valuable suggestions in obtaining the JPEG and H264 shared memory traces.

Bibliography

[1] Intel xscale microprocessor data book. www.intel.com.

[2] Arm11 processor datasheet. In *www.datasheetarchive.com/ARM11-datasheet.html*, ARM Limited.

[3] Arm9 processor datasheet. In *www.datasheetarchive.com/ARM9-datasheet.html*, ARM Limited.

[4] L. Barriga and R. Ayani. Parallel cache simulation on multiprocessor workstattions. In *Proc. Int. Conf. Parallel Processing ICPP 1993*, volume 1, pages 171–174, 1993.

[5] J. Casazza. Intel core i7-800 processor serioes and intel core i5-700 processor series based on intel microarchitecture (nehalem). *Intel White Paper, Intel Corporation, USA*, 2009.

[6] T. Conte, M. Hirsch, and W.-M. Hwu. Combining trace sampling with single pass methods for efficient cache simulation. *Computers, IEEE Transactions on*, 47(6):714 –720, jun 1998.

[7] H. C. Doan, H. Javaid, and S. Parameswaran. Multi-asip based parallel and scalable implementation of motion estimation kernel for high definition videos. In *ESTImedia*, pages 56–65. IEEE, 2011.

[8] J. Edler and M. D. Hill. Dinero iv trace-driven uniprocessor cache simulator. http://www.cs.wisc.edu/markhill/DineroIV/, 2004.

[9] T. Givargis and F. Vahid. Platune: a tuning framework for system-on-a-chip platforms. *TCAD*, 2002.

[10] W. Han, G. Xiaopeng, and W. Zhiqiang. Cache simulator based on gpu acceleration. In *Simutools*, pages 1–6, ICST, Brussels, Belgium, Belgium, 2009.

[11] M. Haque, J. Peddersen, and S. Parameswaran. Ciparsim: Cache intersection property assisted rapid single-pass fifo cache simulation technique. In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pages 126 –133, nov. 2011.

[12] M. S. Haque, A. Janapsatya, and S. Parameswaran. Susesim: a fast simulation strategy to find optimal l1 cache configuration for embedded systems. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '09, pages 295–304, New York, NY, USA, 2009. ACM.

[13] P. Heidelberger and H. S. Stone. Parallel trace-driven cache simulation by time partitioning. In *WSC*, 1990.

[14] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*, pages 397–398. The Morgan Kaufmann Series in Computer Architecture and Design Series. Elsevier Science & Technology, 2011.

[15] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell. Exploring the cache design space for large scale CMPs. *SIGARCH Comput. Archit. News*, 33:24–33, November 2005.

[16] A. Janapsatya, A. Ignjatovic, and S. Parameswaran. Finding optimal l1 cache configuration for embedded systems. In *Design Automation, 2006. Asia and South Pacific Conference on*, page 6 pp., jan. 2006.

[17] R. Kessler, M. Hill, and D. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *Computers, IEEE Transactions on*, 43(6):664 –675, jun 1994.

[18] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded Sparc processor. *Micro, IEEE*, 25(2):21–29, march-april 2005.

[19] S. Laha, J. Patel, and R. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *Computers, IEEE Transactions on*, 37(11):1325 –1336, nov 1988.

[20] X. Li, H. S. Negi, T. Mitra, and A. Roychoudhury. Design space exploration of caches using compressed traces. In *ICS '04*.

[21] M. Loghi, M. Poncino, and L. Benini. Cache coherence tradeoffs in shared-memory MPSoCs. *ACM Trans. Embed. Comput. Syst.*, 2006.

[22] A. Milenković and M. Milenković. An efficient single-pass trace compression technique utilizing instruction streams. *ACM Trans. Model. Comput. Simul.*, 17(1):2, 2007.

[23] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4.5):505–521, july 2005.

[24] R. A. Sugumar and S. G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Trans. Comput. Syst.*, 13(1):32–56, 1995.

[25] N. Tojo, N. Togawa, M. Yanagisawa, and T. Ohtsuki. Exact and fast l1 cache simulation for embedded systems. In *ASP-DAC*, 2009.

[26] W.-H. Wang and J.-L. Baer. Efficient trace-driven simulation method for cache performance analysis. In *SIGMETRICS*, 1990.

[27] Z. Wu and W. Wolf. Iterative cache simulation of embedded cpus with trace stripping. In *CODES*, pages 95–99, New York, NY, USA, 1999. ACM.

[28] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 84 – 95, june 2003.

[29] Xtensa. Xtensa lx2 product brief. www.tensilica.com.

[30] XTMP. The xtensa modeling protocol and xtensa systemc modeling for fast system modeling and simulation. www.tensilica.com.

[31] W. Zang and A. Gordon-Ross. T-spacs: a two-level single-pass cache simulation methodology. In *ASPDAC*, 2011.