

UC Davis

UC Davis Previously Published Works

Title

Survey of Memory, Timing, and Power Management Verification Methods for Multi-core Processors

Permalink

<https://escholarship.org/uc/item/0z81c61t>

Journal

2019 IEEE 10TH ANNUAL INFORMATION TECHNOLOGY, ELECTRONICS AND MOBILE COMMUNICATION CONFERENCE (IEMCON), 00

Authors

Patil, Savita
Scholten, Sabrina
Tao, Melody
[et al.](#)

Publication Date

2019-10-19

DOI

10.1109/iemcon.2019.8936198

Copyright Information

This work is made available under the terms of a Creative Commons Attribution-NoDerivatives License, available at <https://creativecommons.org/licenses/by-nd/4.0/>

Peer reviewed

Survey of Memory, Timing, and Power Management Verification Methods for Multi-core Processors

Savita Patil
Electrical and Computer
Engineering Department
University of California,
Davis, CA 95616 USA
syptatil@ucdavis.edu

Sabrina Scholten
Electrical and Computer
Engineering Department
University of California,
Davis, CA 95616 USA
sdscholten@ucdavis.edu

Melody Tao
Electrical and Computer
Engineering Department
University of California,
Davis, CA 95616 USA
mctao@ucdavis.edu

Hussain Al-Asaad
Electrical and Computer
Engineering Department
University of California,
Davis, CA 95616 USA
hsalasaad@ucdavis.edu

Abstract— Currently, the most significant challenge multicore design engineers face is improving the performance of multi-processors while decreasing the power consumption. Multi-core systems provide a good solution to the increasing high computing applications that need performance boost. However, a good multi-core processor requires thorough and efficient verification procedures. We present a survey on verification techniques for multi-core systems specifically targeting memory, timing, and power management schemes. For memory verification we look into structural simulation-based verification for register arrays, co-simulation techniques, threaded core memory model, symbolic trajectory evaluation for memory arrays, hardware logic emulation, graph model of cache hierarchy and assertion-based simulation schemes. Furthermore, we look into post-silicon verification techniques using the inverse of an instruction and a specific method called DACOTA. For timing verification, we cover three categories of verification: full integration, temporal isolation, and interference effects on schedulability. For dynamic power management schemes, we cover statistical model checking, probabilistic model checking, fractal design verification, and verification for schemes that are interwoven in hardware and software. These verification techniques are tested on the following power management schemes: dynamic voltage and frequency scaling, power gating, and fractal design.

Keywords— *Multi-core processors verification, formal verification, memory verification, timing verification, dynamic power management, fractal design, simulation based verification, co-simulation techniques, pre-silicon and post-silicon memory verification, test generation schemes.*

I. INTRODUCTION

In modern multi-core systems, there are two main challenges that need to be addressed - meeting the increased demand of high-computing processors and reducing the battery power of the processor. The areas demanding high-computing power with full accuracy and precision include video editing, medical image processing, mobile applications, etc. However, the energy requirement mandates having a high-power backup battery, affecting the cost and portability of these systems. As a result, there is a tradeoff between power and high computing.

There are various multi-core processors out on the market. For example, the dual-core CPUs (e.g., AMD Phenom II X2 and Intel Core Duo), quad-core CPUs (e.g., AMD Phenom II X4, Intel's i5 and i7 processors), octo-core CPUs (e.g., Intel Xeon E7-2820 and AMD FX-8350), and many more.

There are various ways to classify multi-core systems, as

mentioned in [1]. According to the way cores are used, they are divided into homogenous (identical cores) and heterogeneous (only non-identical cores) multi-core processors. The multi-core processor architecture can also be broadly classified on the basis of common features or characteristics, such as application class, processing elements, power/performance, memory system, and accelerators/integrated peripherals. These lead to several combinations that need to be addressed using complex verification processes.

Since there can be so many varieties of multi-core processors, the work in [2] explains what design types and flows should be avoided with multi-core systems. The paper also describes which scenarios are best suited for verification. In the authors' previous work, they had derived design qualification criteria from the factors for achieving good performance speedups with multi-core simulations and validated with QuestaSim MC2 flow. Their observation was that designs that meet the qualification criteria show more successful results with multi-core simulations. The designs with long simulation runtimes, long debug turn-around times, large designs with huge memory footprint, parallel/concurrent simulation activity, and a balanced partition of load and low inter-partition communication are suitable for multi-core simulations.

This paper focuses on three specific areas that may prove vital in improving the performance of multi-core systems. Firstly, verification of memory is discussed, followed by timing, and lastly, dynamic power management techniques.

II. BACKGROUND

A. Memory

The memory system architecture point of view includes the caches and their levels, consistency model, cache coherence support and the intra-chip interconnect. These determine the way in which the cores will communicate, providing highly efficient programmability and parallelism to the system.

1) Cache levels : The cache configuration deals with the size, associativity, and levels of caches required by the system. The amount of cache required for an architecture depends on the application. The more the data is reused, the more the amount of cache needed. The use of a larger cache may give better performance but it negatively affects die area and power budget. The number of cache levels depends on how far away the main memory is from each processing

element. Hence, we can have distributed or shared or both types of memory machines. This applies to both homogeneous and heterogeneous multi-core processors.

2) **Consistency Model:** The consistency model defines the order in which the instructions are to be executed. Strong consistency models have strict ordering constraints and are difficult to design, whereas weak models are less complex and are easier to design.

3) **Cache Coherence:** The intrachip interconnect is responsible for general communication among processing elements and cache coherence. The type of programming paradigm the architecture supports depends on cache coherence. Cache coherence defines the consistency of data visible to all other processing cores. The cores can hence be classified depending on scheduling of tasks between the cores, partitioned or global.

4) **Memory Management:** Memory management is the process of allocating and sharing available computer memory among various running programs when needed and freeing up the memory when the application process has ended. Efficient memory allocation is important to any system that is required to multitask at any time. As mentioned in [3], memory management is a function of the hardware, operating system, and the applications that are running.

In Section III, the various verification techniques used to verify memories in multi-core processors and their pros and cons are discussed. Section IV contains a brief summary of what type of processors were covered using the discussed verification techniques.

B. Timing

In terms of timing, providing correctness in verification of multi-core processors is essential for real-time systems. For safety-critical applications such as multi-core processors used in embedded systems like health-care applications, automobiles, or airplanes, failure to meet deadlines can become dangerous. Timing verification in multi-core systems, though, is much more complex than it is for single-core processors. In a single-core processor, a two-step approach to timing verification is typically used: a timing analysis step and a schedulability analysis step [4]. Timing analysis refers to the determination of the worst-case execution time (WCET), which, as the name suggests, involves finding an upper bound on the time the processor takes to execute, with no sources of interference on shared hardware, interrupt service routines, etc. In the schedulability analysis, the focus is on establishing a worst-case response time (WCRT), and unlike the timing analysis, this does include the processor's task execution in the context of interference, interrupts, etc.

Due to the inherent differences in single-core and multi-core systems, this two-step approach cannot be applied the same way to both types of systems. In multi-core systems, tasks can execute concurrently, and so contention for shared resources is significantly more complex than in single-core contexts. The tasks that execute in parallel across cores are referred to as co-runners in [4], and we will use the same annotation throughout this paper. Calculating WCET and WCRT values on a multi-core platform using the single-core approach will therefore provide very optimistic results, since they can only

consider isolated processors without consideration of co-runners.

Timing verification methods can be divided into subcategories including *full integration*, *temporal isolation*, *integrating interference effects on schedulability*, and *mapping and scheduling*. For this paper, we will focus on the first three, leaving mapping and scheduling for further research. *Full integration*, as the name suggests, includes detailed interference from other co-runners in the analysis. *Analysis using temporal isolation* looks at timing details independent from some tasks, and relies on the assumption that the memory bus contributes most significantly to interference. *Integrating interference effects on schedulability* takes into account effects on schedulability due to shared hardware resources, as well as including delays from the specified arbitration policies into the analysis.

In Section III, within each of these three categories, a discussion of a few verification techniques, with the pros and cons associated with each method is presented. Section IV will contain some comparison of the methods and include possibilities for further research in timing verification of multi-core processors.

C. Dynamic Power Management

Dynamic power management (DPM) techniques are used to efficiently distribute power in multi-core systems. However, with the complexity of modern-day computing systems, DPM schemes have also grown increasingly complex. It is no surprise that this makes verification of DPM schemes a challenging problem to tackle. Various DPM techniques include power gating, dynamic voltage and frequency scaling (DVFS), thread motion, asymmetric cores, and core fusion [5]. Another DPM technique that was proposed within the last five years is a fractal scheme [6] that is scalable and verifiable.

We will focus on verification methods for DVFS, power gating, and the new fractal scheme presented in [6] and [7]. In addition, our focus will be on formally verifiable methods, such as variations of model checking. Before covering the verification methods, we will provide a brief overview of the DPM schemes of interest in the current section. In Section III, we will cover verification methods that have been performed on these techniques. Finally, in Section IV, we will discuss the benefits and disadvantages of the different verification methods. We will end our discussion on DPM verification on future research.

1) **DPM Implementation:** For DPM schemes, most schemes are implemented with a global controller, where the cores are the computing units that send requests to the controller. The global controller monitors the power usage across all cores and keep the power usage to at or below the specified power budget. The power budget, or the maximum amount of power that can be distributed across all cores, is supplied from a higher-level manager compared to the global controller [8].

2) **Power Gating:** Power gating cuts off the power when there are idle periods in RTL blocks that are not currently using up resources. There are two types of power gating: timeout based and adaptive power gating. In timeout-based

power gating, we check if a defined fixed number of clock cycles have passed that are idle. If so, then the blocks are powered off and remain off until after K_{CYCLES} have passed, where K_{CYCLES} is also a defined value. Adaptive power gating is defined as predicting the next idle period length based on the previous idle periods. In power gating, safety and efficiency are crucial measurements in determining whether a method is viable. Safety refers to leaving the power off for too long and overestimating the idle time of the system. Efficiency refers to how much power can be saved by shutting off the power during idle periods correctly [9].

3) DVFS: The dynamic voltage and frequency scaling (DVFS) technique for multi-core processors primarily revolves around adjusting the values of the frequency and the voltage in the following equation:

$$Power = CV^2F$$

where C is capacitance, V is voltage, and F is the working frequency [5]. In DVFS, the V and F are adjusted so that a set of discrete power values are generated. These generated power values are used to determine how much the voltage and frequency need to be scaled after a computing unit sends a request to a global power controller. Fig. 1 shows how a DVFS scheme is structured. The predefined power values can be for one chip or can be per core. For per core DVFS, the global controller receives requests from the cores and each core has its own set of power values. The *safety* measurement in DVFS is related to how much time and power is spent over the power budget [10]. *Efficiency* is a measure of the reduction in the performance based on decreasing the frequency element in cores through DVFS.

4) Fractal Design : The fractal design proposed in [6][7] is similar in structure to a binary tree, where each node in the tree is represented as a DPM controller. Each leaf is considered as a computing unit, requesting power from the controllers. The authors in [6] state that this binary tree design will allow the structure of the DPM to be inductively verifiable. Fig. 2 shows an image of the proposed structure. The advantage of the fractal design scheme is that it can be scaled up or down.

III. VERIFICATION METHODS

A. Memory

The hardware verification of memories can be broadly classified into *pre-silicon* and *post-silicon*. Using *pre-silicon*, or before fabrication verification techniques, we can verify memories using three techniques - *formal verification*, *simulation-based verification*, and *co-simulation verification*.

In the *formal verification technique*, we generate mathematical proofs for behavior of the design. It uses assertion-based methodologies for functional verification of properties of the system. *Equivalence checking* and *property checking* are two of the formal verification methods that are used for verifying memories. The positive aspect of formal verification is that we verify all possible behaviors of the model at all time instances and for all input variations. However, they take a lot of time and consequently, can be used to verify only small to moderately sized designs.

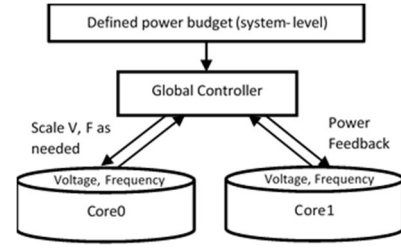


Fig. 1. Structure of a DVFS scheme from [5].

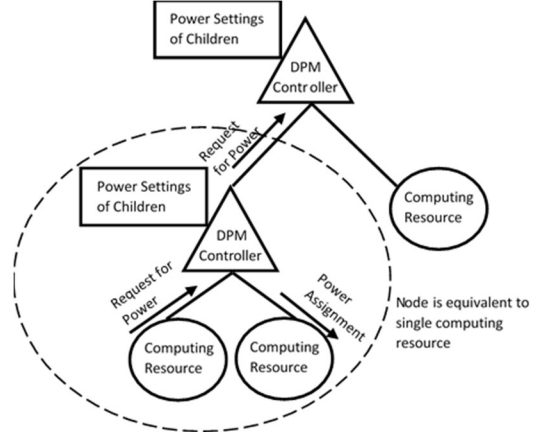


Fig. 2. Proposed fractal design structure from [7].

Simulation-based verification involves verification through input space sampling. Functional correctness of the hardware model is checked against either high level references or instruction set simulators. A testbench controller and Design Under Test (DUT) is required in simulation-based verification. The testbench controller feeds input test vectors to DUT. Computed states of the DUT are verified against expected states of the design. It is necessary to implement coverage monitors to generate coverage statistics as, unless all points are sampled, there exists a possibility that an error escapes the verification process. Simulation-based verification techniques can be further divided into *structural* and *functional*. The *structural* simulation targets functional components like ALUs and storage components (*register files*). We achieve high fault coverage for functional components, but low fault coverage for pipelined logic and system co-processors using structural simulation-based verification. The *functional* simulation is best suited for components with unknown implementation details.

The *co-simulation technique*, as proposed by [11], tests a formal model of processor architecture against a behavioral simulator, with the help of a heuristic database. We can also test an RTL model against a golden model in C or an RTL model against a behavior golden reference model in HDL using the co-simulation technique.

One of the *post-silicon* hardware verification techniques that can be used for *pre-silicon* verification of memory is using inverse instructions. The test engineer identifies an inverse instruction or procedure for every instruction in the processor's ISA. We assume the initial random state of the processor is known prior to generating a random program. The instruction and its inverse operations make the system try

to reach to its initial state. Bugs get discovered if they do not match. Only a subset of processor's ISA can be tested, as not all instructions have an inverse. Another *post-silicon* method that is discussed in detail later in the paper is DACOTA (Datacoloring for CONSistency Testing and Analysis).

The test inputs can be generated for verification by *directed test generation* and *random test generation*. The *directed*, or the white box test generation technique, generates deterministic test sequences to test a certain or small group of related instructions. It is a slow and tedious method for exhaustive simulation, but is a quick and easy method for the *very first verification of an instruction*. We also get an early simulation feedback which is helpful for the random test generation method.

The *random test generation* method, also known as the Gorilla or black box test generation method, generates random test programs for testing an instruction or any sequence of instructions. Large number of test programs is generated in short time. Over time, it produces odd and strange programs with unusual instruction sequences that few human programmers will think of and very often, these tests have hit corner and extreme cases and reveal bugs that have not been found by manual or deterministic testing.

Random test generation can be further divided into static and dynamic test generation. In *static random test generation*, the system is initially set to random state to generate random test programs. This method is used by *test bench controllers* or *model simulators*. *Dynamic random test generation*, on the other hand, relies on the simulation feedback to generate random programs on the fly yielding high coverage statistics. The dynamic random test generation is used in *directed acyclic graphs* (DAG) that represent sequential flow in a program. The *Automatic Test Pattern Generator* (ATPG), when using this dynamic random test generation method for functional verification of processor cores, improves its fault coverage. Although the dynamic techniques improve the quality of generated test programs, the achieved throughput is less than static test generators.

We will be discussing methods to verify registers, cache hierarchy, virtual memory, memory arrays and hybrid memory cube of a multi-core system.

1) *Static random generator with co-simulation technique*: A random generator using static techniques is implemented in [11], along with a coverage monitor to evaluate its performance. The design of the random program generator is separated from the testbench controller which makes future upgrades or replacement possible. They used openRISC processor OR1200's RTL model and the instruction level simulation obtained by an openRISC architecture simulator. It generates large test volumes. The testbench controller is a module of the co-simulation testbench that executes the test program simultaneously on the two processor models and compares the state of the processor on a cycle-by-cycle basis.

The sniffer modules are included in the controller to interface the Or1ksim ISS simulator and the Icarus RTL simulator as shown in Fig. 3. After the execution of each instruction, the sniffer modules collect changes in the processor status and register values. The sniffer modules also

keep track of the data memory location being addressed during a store operation. The information from the sniffer is updated to the checker module, which compares and logs the differences in the processor states. If the number of differences exceeds some configured maximum limit, the checker signals the testbench controller to end the simulation.

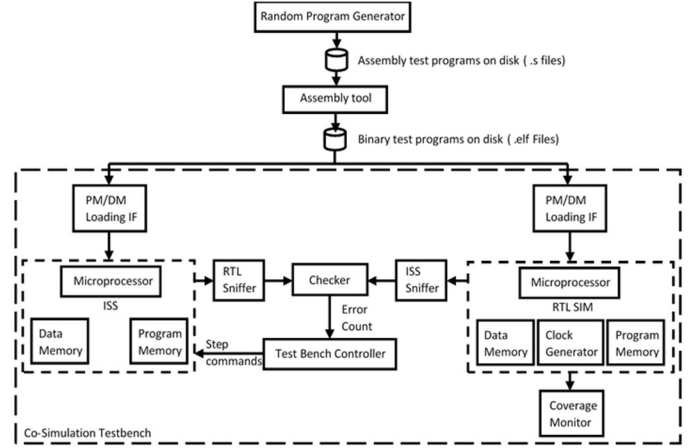


Fig. 3. Design of Testbench Controller [11].

2) *Threaded many core memory core model*: The tasks executed shared by multiple processors is most commonly managed by using a threading model in which tasks can be broken down to separate execution units to run on different processors in parallel. The Threaded Many core Memory core (TMM) model discussed in [12] implements the problem of computing all pairs shortest paths (APSP) on a weighted graph by using four classic algorithms: The Floyd-Warshall algorithm, Johnson's Algorithm (Dijkstra's Algorithm using Binary arrays), Johnson's Algorithm (Dijkstra's Algorithm using Binary Heaps), and n iterations of Bellman-Ford Algorithm. A comparison was done with a parallel random analysis machine (PRAM) model and it was observed that TMM model provided more information for large latency, finite thread cases. The TMM model is useful in multi-core systems that have shared memory machines and SIMD computations with fast context switching. It is assumed that the hardware provides zero-cost and perfect scheduling between threads. It models the machine as having only two levels of memory – one slow global memory and one fast local memory shared in one multiprocessor. In reality, these machines may have many levels of memory. This model was implemented in NVIDIA and AMD's general-purpose GPUs.

It was observed that algorithms that had the same *PRAM* performance had different performance in the *TMM* model, since they are better at hiding memory latency with fewer threads. It was observed that the Floyd-Warshall algorithm performed better for large memory access latency compared to Johnson's algorithm. *TMM* is a model well-suited to compare algorithms and decide which one to implement under particular environments.

3) *Model-based test generation with heuristic database*: The functional verification of six IBM PowerPC processors using a new methodology and test program generator is

discussed in [13]. The generator contains a formal model of the PowerPC architecture and a heuristic database of testing expertise. The heuristic database allows testing engineers to add knowledge to the tool throughout the verification process. The formal architectural model enables test program generation for different processor architectures using a method called model-based test generation. Separation of the components that were in one software system into architecture, testing knowledge, generation procedures, and behavioral simulator allowed generalization of the generation procedures and reduced its complexity too as shown in Fig 4.

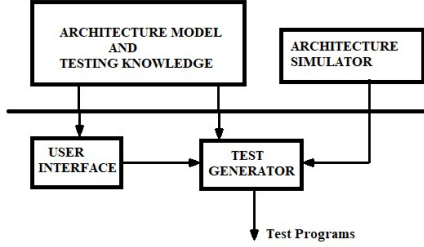


Fig. 4. System components and interrelations [13].

For the architectural model, the instructions are modeled as trees at the semantic level of the processor architecture. The instruction instances are generated by traversing the *instruction tree*. These static relations between operands do not change - they are the same before and after the execution of an instruction. The C language was used for representing generation and validation functions for the heuristic testing knowledge. The functions were invoked while the generator traverses an instruction tree. Traversing a node involves invoking all the generation functions associated with it. The outputs of these functions are used to generate the instances of the current sub-tree. This scheme ensures consistency of the generated instruction instances – namely, the values selected for fields and resources that are shared by several sub-operands are identical. We study the nature of instruction sequences involved in uncovering bugs.

The new test program generator produces only about 10 instructions per second in similar conditions, as opposed to 50 instructions per second produced by previous generator. However, the slow-down factor is compensated by the better quality of the tests. It reduced the functional verification period and time to market as well.

4) *Verification of caches:* Multi-core processors usually have three levels of cache. Level 1 cache is private to each individual core, which is introduced to speed-up the cache look-up. LLC (Last-Level Cache) may be private or shared to the core, where both have their pros and cons. In shared LLC, there may be two cases centralized LLC or distributed shared LLC. Caches can also be classified by the way they are managed by hardware or local memory. For the hardware managed cache, tags are stored on same die area. It reduces space for computation and hence give a non-deterministic performance. If the local memory handles cache, the tag, is not stored and hence has more space for storage in the same area. The tags are then assigned by software stream explicitly.

a) *Using combinatorics:* Verification of cache hierarchy in modern heterogeneous system-on-chips (SOC) can be done with the use of combinatorics, as implemented in [14]. The cache hierarchy of both specialized and general-purpose microprocessors often has a multi-channel or multiport architecture with scalar and vector execution channels. Due to the large combinatorics of the cache memory hierarchy states, its verification requires a huge amount of complex tests, which becomes the main problem for functional and formal verification. Therefore, automatic generation technique for development of system-level test generator was used for multi-core structures verification. Currently, pseudo-random test generators are used for verification of various devices of the processor core: the instruction sampling unit, the memory manager, the control device, the pipeline, and others. A block diagram of the generator is shown in Fig. 5.

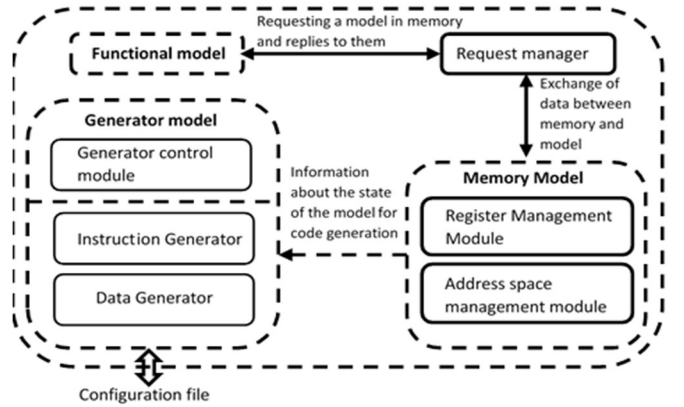


Fig. 5. Block Diagram of the Generator for [14].

b) *Labeled transition system:* A graph model of the cache hierarchy is proposed in [15]. It covers the entire state space of the single cache level. The Labeled Transition System (LTS) as the state/ transition graph provides information for the changes of initial state only. For non-deterministic situations, such as choosing a channel (port) for a multi-channel memory cache, a separate instance of a subgraph $g_i \subseteq G$ was created. If the cache state after the nondeterministic transition (t_i) does not correspond to any state of cache in the model subgraphs, the cache memory does not work correctly. A sign of detecting an error, in this case, is a state other than the state in $\{g_0, g_1, \dots, g_n\}$, where n is the number of cache memory channels (ports). They implement it first on a direct mapped memory. To cover the entire state space of the cache hierarchy, a hierarchical graph model is then constructed.

One test sequence was able to check the interaction of all levels of the cache memory of the processor core. Several critical errors were revealed in verifying the VLIW DSP processor hierarchy cache with the Elcore50 architecture. This approach could be used to verify other processor cores too. The weakness of this method is that the test generators create lot of boundary situations for subsystems. Only a few localized subsets get covered out of the global state space of the entire cache hierarchy, leaving large gaps between these

subsets. Both [14] and [15] have used the same generator model to generate the test vectors. Fig. 6 shows the generator model used by [15], which has updates in their memory model.

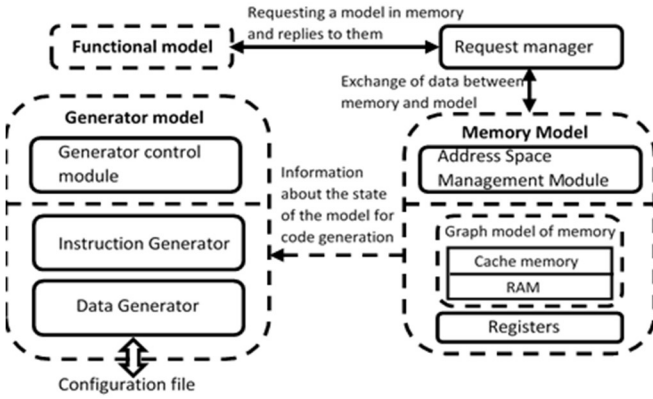


Fig. 6. Block Diagram of the Generator for [15].

c) *Test templates*: For testing corner cases, test programs need to be manually developed. The work in [16] discusses verification with cross compilation. It verifies cache as well as virtual memory using LRU policy. The method is not complex, but it does not guarantee complete coverage. The test data generation involves finding the initial state of processor and then building test programs accordingly with test templates. Generating test programs on the basis of test templates is divided into two stages, which is depicted in Fig. 7. An abstract representation of test programs, the test templates, are prepared in the first stage. There are no values mentioned here - just the constraints on them. The second stage is generating the test programs. The instruction parameters are generated, which involves the initial value of registers, cache memory cells, translation lookaside buffer (TLB) lines, etc. If the individual registers-parameters of instructions were not specified in the test template, they are chosen by the test data generator. The test template has information about the sequence of instructions and parameters of instructions along with the events like overflow, cache misses or hits. The generated test program undergoes equivalence checking for each instruction to match with the behavior declared in the test template.

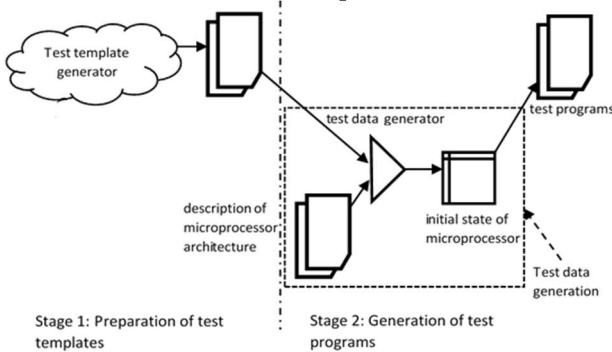


Fig. 7. Generation of test programs on the basis of test templates [16].

The test data generation can be done with some of the prevalent techniques. The combinatorial techniques are good if we can reduce the constraints on variables into simple terms. ATPG is more frequently applied for module testing. The ATPG is helpful in searching the values of the input signals (“vectors”) of the scheme that detect its inadequate behavior. ATPG can be done on a microprocessor if the RTL model of microprocessor is present. The use of this technique for functional verification is limited because the requirement of redundancy of the instruction semantics is violated. It requires additional efforts for the preparation of an RTL model. Promising results were obtained by the test generation tools based on the resolution of constraints.

The test templates make it possible to specify test programs of variable length. For each instruction in the template, one can indicate heuristics for choosing the values of the parameters. Although this approach is scalable for large test templates and complex microprocessor architectures, we need backtracking if we are unable to choose parameters for next instruction. A reduction of the encoding length from $|L| + 2^n$ to $|L| + n$, where L is the memory size and n is the number of changes in the memory was observed.

5) *Symbolic trajectory evaluation*: Successful formal verification of commercial memory arrays has been done using symbolic trajectory evaluation. The work in [17] describes the formal verification tools for the control dominated circuits – memory systems being one of them. The other method that has been successfully used by Motorola, AMD, Intel, and IBM is hardware logic emulation. The RAM programmable FPGAs provide a complete hardware equivalent of a digital system that includes memory such as caches. Performing hardware logic emulation enables software configurable virtual silicon that is functionally equivalent to the chip in design.

6) *Post-silicon verification*: In the post-silicon validation, mentioned in [18], the verification is executed directly on prototype hardware. Testing in post-silicon will be faster, enabling higher coverage only if the internal nodes are observable and there are no limitations in diagnosis. DACOTA, a new high-coverage solution for validating memory operation ordering in multiprocessors, promises to unlock the full performance of post-silicon validation. A significant amount of bugs escape when memory coherence and consistency come into the picture. When the interconnections are done by complex and non-deterministic interconnects rather than the buses, the design becomes more error-prone.

Cache lines are partitioned to include an *access vector* tracking the order of memory accesses as shown in Fig. 8. A portion of each cache is reclaimed and used as *activity log* storage for load/store operations. Finally, the cache controllers are augmented to include supporting hardware. This is enabled only during post-silicon validation. When local cache storage is exhausted, activity logs are aggregated and validated by a software-based algorithm, using the existing processor cores for computation. Validation is performed by building a graph from the activity log, where vertices represent memory accesses and edges indicate the

observed ordering between them. Correct memory ordering is then checked by inspecting the graph for cycles. Since we use existing cache storage and CPU computation resources, we will have an extremely small silicon area overhead with DACOTA. Also, DACOTA can be completely disabled upon product shipment, leading to zero performance impact to the end user.

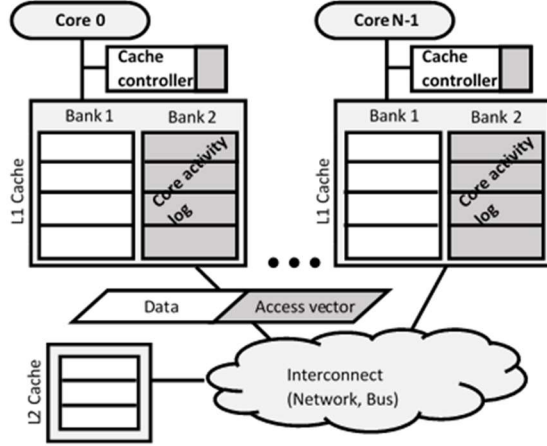


Fig. 8. Multiprocessor reconfiguration for DACOTA validation [18].

7) *SMCSim simulation for hybrid memory cube*: The hybrid memory cube, a new processor in memory architecture, is discussed in [19] to revisit near-memory computation to mitigate the power and performance losses caused by the “memory wall”. An accurate simulation environment called SMCSim has been developed, along with a full featured software stack. In this environment, we need to do a full system analysis of near memory computation, including high-level software to low-level firmware and hardware layers. We also need to consider offloading and dynamic overheads caused by the operating system, cache coherence, and memory management. A zero-copy pointer passing mechanism has been devised to allow low overhead data sharing between the host and the processing in memory. To do this, atomic commands were utilized and offloading overheads was considered.

A performance improvement of 2x was observed in comparison with the host System-on-Chip (SoC), and around 1.5x against a similar host-side accelerator.

B. Timing

The three categories of timing verification for multi-core processors we will cover are: *full integration*, *temporal isolation*, and *integrating interference effects on schedulability*. The following subsections will have detailed descriptions of these methods along with their benefits and drawbacks.

1) *Full integration*: In full integration verification, as much or all information regarding tasks running concurrently are incorporated into the analysis. Because of this, the results of this type of analysis change for every different test case. In a 2010 study by [20], the upper bound for the WCET is found by analyzing interconnection network traffic as a pattern of tasks, and using the number of bus requests for various execution paths. They assume a shared bus with work-

conserving bus arbitration, as well as statically assigned tasks with non-preemptive scheduling. Unfortunately, this analysis looks at each bus transaction independently, and may result in overly pessimistic WCET values.

In 2015, the authors in [21], improved on some of this pessimism by assuming a composable system where local and global worst-cases always occur in the same situation. They did this using satisfiability modulo theory (SMT) to determine execution time while taking into consideration shared resources and a shared bus with Time Division Multiple Access (TDMA) arbitration. Using a TDMA arbitration policy, bus accesses are split between cores into time slots with potentially differing priority. This is a commonly used policy across timing verification techniques. To model the SMT expressions, they use a PAGAI tool to generate the expressions, which encode execution times for the processor that are checked against predetermined upper bounds. The general workflow of their method is shown in Fig. 9.

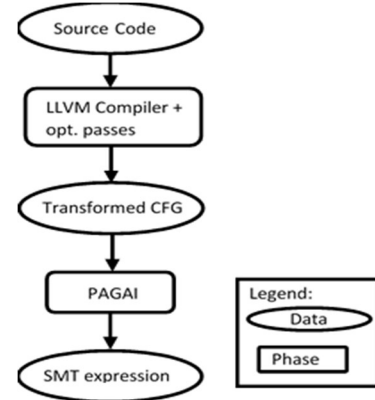


Fig. 9. Workflow in order to generate SMT expressions [21].

In [22], WCET analysis has high reliance on tight upper bounds, and therefore requires many approximations due to the increasing processor complexity and shared-resource interference. To mitigate this somewhat, a framework for a reliable WCET alternative that utilizes “iterative over-approximation” of abstract traces on each core was proposed. In this way, every core is considered in isolation while still in the context of indirect shared-resource interference using WCET bounds. This framework makes approximations using fewer execution behaviors than most real-world programs. It also relaxes and avoids restrictions on standard WCET analysis.

2) *Temporal isolation*: Temporal isolation methods assume that contention on the memory bus among co-runners is the most significant form of interference on the processor’s execution time. In [23], they propose an execution model that is designed by the programmer and implemented at run-time. This model is enacted in three parts: a development phase, an integration/allocation phase, and the run-time/hypervisor application. Their implementation is target-dependent, so they chose to apply it to the Kalray MPPA-256 because its cores have good timing compositionality and are not overly heavy power consumers, in order to maintain uniformity and potentially be used in avionics applications.

In [24], a second paper using the Kalray MPPA-256, provided an optimized framework using a method of mixed-criticality scheduling for cluster-based multicores, followed by a corresponding response time analysis. They focus specifically on real-time, safety-critical applications in embedded systems, which have varying failure protections, and so they define parameters for ensuring a correct scheduling strategy, composable and incremental certifiability, and measures of interference. They perform resource utilization optimization using their mixed-criticality scheduling policy, FTTS.

In [25], WCET values are again used to verify multicore processors, this time on an ARM7TDMI processor. They propose using an instruction-level parallelism (ILP) based bus-aware instruction set using scratchpad memory and a TDMA arbitration schedule to optimize for multi-core designs. Their approach used dynamic bus offset prediction, and while this method greatly increases complexity over standard WCET analysis, on benchmark tests it proved to significantly decrease WCET values.

3) *Integrating interference effects on schedulability*: The verification method of integrating interference effects on schedulability uses an analysis that includes delays from chosen arbitration policies into their schedulability analysis. The focus here is on the demand for resources rather than the typical WCET values.

In [26], the authors propose another verification method using a TDMA arbiter for shared resource access, and this one focused on analyzing the WCRT for a superblock. Superblocks are defined in this context as execution blocks with most memory accesses occurring at the beginning or end of the block. Following a comparison of schedulability models for scheduling tasks in a superblock, they propose a sequential model as superior, where each superblock executes serially.

In [27], the authors designed a framework focusing on the demands of shared hardware resources to determine the response time of the system under scope. Their framework is very general and can be scaled to many different architectures. This framework was chosen because typical interplay between scheduling and timing in many standard timing verification techniques is overly pessimistic, overestimating contention for shared resources in multicore applications. Therefore, their framework models processor behavior through execution traces rather than WCET values, with the evaluation based on both processor and memory demand.

C. Dynamic Power Management

1) *Statistical model checking*: Verification using statistical model checking is used on power gating schemes. In [9], statistical model checking on adaptive power gating and timeout-based power gating schemes are used. Sample input paths are used with a model checking tool to keep track of outputs. Specification of the RTL block must be provided and using a Monte Carlo simulation, the input paths to the power gating scheme are generated. The data is collected at each clock cycle, where the results depict the frequency that a block remained in idle periods as well as the number of paths

needed to guarantee a specified percentage of safety and efficiency. The efficiency and safety of both power gating schemes were tested using the model checking tool, Ymer, on an Open SPARC multi-core processor [9]. The percentage is supplied to the model checker, where percentages 80% and 90% were supplied for measuring the efficiency, while 85% and 95% were supplied to measure the safety. Results showed that for both methods, efficiency and safety were guaranteed for 80% and 85%, respectively.

2) *Probabilistic model checking*: Verification of DPM schemes using probabilistic model checking is discussed in [10]. They propose making designers take into account verification early on so that designs that are difficult to verify are discouraged. Their verification method targets use of DVFS, as shown in Fig. 10. This DPM scheme aims to keep the power usage below the user specified amount. Their paper investigates the efficiency and safety of their design, along with the amount of effort required to verify the design.

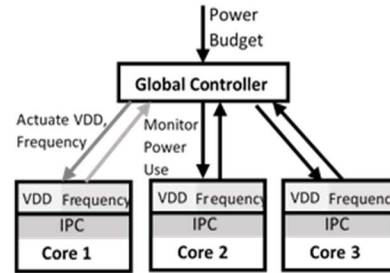


Fig. 10. DPM schemes verifiable using probabilistic model checking [10].

The probabilistic model checking tool used is called PRISM, where the inputs to the checker are: *state elements* of the system, the *probabilistic transition rules*, and the *correctness properties*. *Transition rules* are defined as a description of how the behavior of the system can vary state to state. *Correctness properties* are defined as the requirements needed to guarantee correctness of the system. They also use *rewards*, which are similar to *tokens*. States that satisfy certain conditions are considered *tokens*. The model checking tool can traverse the entire state space and verify if the correctness properties are correct throughout. When rewards are specified, the values are calculated over some number of system transitions.

In [10], heterogeneous and homogeneous DPM schemes are analyzed, where the controller uses a priority-based algorithm to allocate power to the different units. The first core gets the largest voltage; the second core gets the second largest, and so forth. For homogeneous schemes, the controller distributes the greatest voltage possible to each core that, when summed together, keeps the system below the user's specified max voltage, which is similar to the Chip-Wide DVFS method. For probabilistic model checking, [10] found that for cores per chip (CPC) > 1, there were significant improvements in performance. However, a linear increase in cores per chip leads to an exponential increase in state space. They determined that performance did not improve much for CPC > 2 and in fact, found that it became more frequent for the DPM scheme to remain above the specified power budget. The work in [8] also reaffirms this research. The work in [10]

also found that varying the frequency levels significantly impacts the verification efforts negatively.

Power management using DVFS and power gating have certain challenges unique to them. The main difficulty is that design description languages at the gate level do not have ways to represent voltage as a variable. The other DPM schemes such as adaptive voltage scaling (AVS), multi-threshold CMOS (MTCMOS), active body bias (ABB) and voltage scaling (VS) also face this issue.

3) *Fractal verification*: Fractal verification requires an entirely different DPM structure compared to those previously discussed. The proposed model aims to be verifiable using existing model checkers. The verification method requires that there be a fractal invariant at every scale i.e. a property must be prevalent and unchangeable regardless of the system's size. The invariant they use requires that both children of a DPM controller cannot be in the same high-power state at the same time. In their method, there are five possible power allocations: high, med-high, med, med-low, and low. The invariant must be mapped from fractal invariants to global invariants. If one resource is high, the other computing units, at max, can be med-high.

Murphi, an enumerative model checker and protocol description language/verifier is used. Murphi searches through all the reachable state space and checks that the specified invariant is true for each state. An aggregate checking method is used, based on the method performed by Park et al. to do equivalence verification with Murphi [6]. To verify the design, induction is used, where the base case is defined as the minimum system that satisfies the power constraints specified in the checker. The base case is one root, one internal DPM controller, and two computer resources. The inductive step verifies that larger systems are equivalent to smaller systems and uses the looking up and down method. Their results were compared with an oracle DPM, which is an unimplementable DPM that exhaustively searches for the optimal power allocations. The fractal DPM fell short of some of the optimal power allocations made by the oracle [7]. However, for the most part, the fractal DPM behaved similarly.

4) *Software/hardware DPM scheme verification*: This method focuses on DPM schemes that have global controllers in software/firmware and local controllers in hardware. In [28], the global controller issues directives to the local controllers and the local controllers are the ones to actually change the power states. Two methods to verify such schemes are proposed. In the first method, the formal model of the DPM scheme is translated into a Verilog state-machine, which is then co-verified with local power controllers. The second approach requires formal assumed properties to be extracted from the formal model. Then, the architecture of the local power controllers are proven based on the assumptions.

Before verification, a formal model of the global power management (GPM) must be created. To extract the formal model, all locations where a state variable is outputted are marked. Then, all possible inputs and all possible present state conditions are generated and passed through these locations. This is to determine the next program location, next state, and

the output. The generated formal model is then used in both verification methods. In Method 1, their tool translates the model into a Verilog state machine and models the entire GPM with the state machine. In Method 2, their tool extracts assumed properties in System Verilog Assertion (SVA) from the model. This allows the user to choose which properties to verify. Both methods can then be passed down to a model checker to verify the design. The model checker takes in global assertions and the local power manager hardware, then outputs true or false.

5) *Runtime verification*: The verification during runtime method as in [29], although not tested specifically on DPM schemes, can be potential research avenue. This method requires the processor to be equipped with the capability of synchronizing with an external emulator that is used to emulate the behavior of one of the cores, where it checks whether or not the system violates or satisfies a provided correctness property. This method needs to be used in conjunction with other verification methods, like model checking.

IV. CONCLUSIONS

Performance in multi-core architectures is inherently meant to be greater than that of single-core processors due to the ability to exploit parallelism of the multiple cores. However, due to the complexity and interplay between processors analysis on real-time performance of multi-core processors must be done thoroughly to ensure maximum performance. We found that only hardware emulation and assertion based verification techniques can be used for verification of both memory and DPM.

In this paper, memory verification of open RISC processor OR1200 [11], NVIDIA and AMD's general purpose GPUs [12], IBM PowerPC series [13], MIPS64 architecture [16], VLIW DSP processor hierarchy cache with the Elcore50 architecture [15], Open SPARC architecture [18] and Hybrid Memory Cube [19] was studied. Verification of intrachip interconnections, Crossbar, Link, mesh or network on chip was not surveyed. They are generally used between core-to-cache bank networks and this is a topic of further research.

Three categories of timing verification on multi-core processors were covered in this paper. For full integration, standard WCET values were improved upon to find reasonable upper bounds on execution time. Temporal isolation methods moved a bit away from the standard WCET methods, and focused on arbitration policies to reduce the effect of memory-bus contention from the analysis. As far as the integration of interference effects on schedulability, a total change in direction from typical WCET values, with a focus on demand for resources from both the processor and memory was seen. A fourth category of timing verification for multi-core processors for researching further consists of mapping and scheduling. This category, according to [4], focuses on mapping tasks to cores, and determining execution time while looking at timing behavior as well as memory requirements. Interference delays are integrated into the analysis as well.

For DPM schemes, there are various formal verification methods. However, based on research for this paper, there seems to be no single method that suits all DPM methods.

Using [8] and [10]'s results, small clusters of cores would make this verification method plausible, if DVFS schemes were used. While this would require more controllers and thus, increase the cost and area of the design, their results are promising in terms of coverage. For power gating, statistical model checking has decent coverage, where [9] demonstrate at least 80% efficiency and safety. The fractal DPM scheme is scalable, but like the clustering method, it would require a lot of controllers to maintain the structure. Lastly, for [28], both methods had similar performance, but are only expected to supplement other verification methods. Hence, each verification method has its pros and cons and is based on the DPM scheme in question and circumstances. Future work can be done on investigating both hybrid verification methods and DPM architectures.

In our survey till now, we have not been able to find verification methods that can be used for a particular multicore processor that verifies its memory, timing and power management as a whole. The simulation schemes individually are unable to give full coverage in optimum time. A combination of several simulation schemes need to be implemented. Some verification schemes stood out because of their performance improvement or coverage. DACOTA for example offers post-silicon memory verification that can be disabled so that there is no impact on the customer.

REFERENCES

- [1] Shruti Jadon, Rama Shankar Yadav, "Multicore processor: Internal structure, architecture, issues, challenges, scheduling strategies and performance", 11th International conference on Industrial and information systems (ICIIS), 2016, Page(s): 381 – 386.
- [2] Rohit K Jain, Shobhana Sudhakar, "Verifications with multi-core parallel simulations: Have you found your sweet spot yet?" DVCON US 2016.
- [3] "The memory management reference", Ravenbrook Limited, 2016. [E-book] <https://www.memorymanagement.org/mmref/begin.html>
- [4] Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, Robert I. Davis, "A survey of timing verification techniques for multi-core real-time systems," in Verimag Research Report no. TR-2018-9, 2018
- [5] K. M. Attia, M. A. El-Hosseini, H. A. Ali, "Dynamic power management techniques in multi-core architectures: A survey study," in Ain Shams Engineering Journal, Volume 8, Issue 3, September 2017, Pages 445-456.
- [6] D. J. Sorin, O. Matthews, and M. Zhang, "Scalably verifiable dynamic power management," in 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), 2014, Page(s): 579 – 590.
- [7] D. J. Sorin, O. Matthews, and M. Zhang, "Architecting dynamic power management to be formally verifiable," in Proceedings of the 51st Annual Design Automation Conference, 2014, Page(s): 1 – 3.
- [8] P. Bose, A. Buyuktosunoglu, J. A. Darringer, M. S. Gupta, M. B. Healy, H. Jacobson, I. Nair, J. A. Rivers, J. Shin, A. Vega et al., "Power management of multi-core chips: Challenges and pitfalls," in DATE, 2012, Page(s): 977 - 982
- [9] J. A. Kumar and S. Vasudevan, "Verifying dynamic power management schemes using statistical model checking," in Asia and South Pacific Design Automation Conference, IEEE, 2012, Page(s): 579-584.
- [10] A. Lungu, P. Bose, D. J. Sorin, S. German, and G. Janssen, "Multicore power management: Ensuring robustness via early-stage formal verification," in Proc. of MEMOCODE'2009, Page(s): 78–87.
- [11] Anantha Ganesh Karikar Kamath, "Automatic verification of microprocessor designs using random simulation", Master Thesis, Department of Information Technology, Uppsala University Uppsala, Sweden, July 2012.
- [12] Lin Ma, Kunal Agrawal, and Roger D. Chamberlain, "A memory access model for highly-threaded many-core architectures", 2012 IEEE 18th International Conference on Parallel and Distributed Systems, Page(s): 339 – 347.
- [13] Aharon Aharon, Dave Goodman, Moshe Levinger, Yossi Lichtenstein, Yossi Malka, Charlotte Metzger, Moshe Molcho, Gil Shurek "Test program generation for functional verification of PowerPC processors in IBM", Available at <https://www.research.ibm.com/haifa/projects/verification/papers/dac95.pdf>.
- [14] Larisa G. Gagarina, Anton V. Garashchenko, Alexey P. Shiryaev, Alexey R. Fedorov, Ekaterina G. Dorogova, "An approach to automatic test generation for verification of microprocessor cores", 2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), Page(s): 1490 – 1491.
- [15] Anton V. Garashchenko, Fedor M. Putrya, Larisa Gagarina, Alena V. Garashchenko, Albert A. Dzhurakulov, "Automatic test generation methodology for verification of a cache memory based on the graph model of cache hierarchy" 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), Page(s): 1876 – 1879.
- [16] E. V. Kornykhin, "Generation of test data for verification of caching mechanisms and address translation in microprocessors", ISSN 0361 7688, Programming and Computer Software, 2010, Vol. 36, No. 1, Page(s): 28–35. © Pleiades Publishing, Ltd., 2010. Original Russian Text © E.V. Kornykhin, 2010, published in Programmirovaniye, 2010, Vol. 36, No. 1.
- [17] Carl Pixley, Noel R. Strader, W.C. Bruce, Jaehong Park, Matt Kaufmann, Kurt Shultz, Michael Burns, Jai Kumar, Jun Yuan and Janet Nguyen, "Commercial design verification: Methodology and tools", IEEE International Test Conference, 1996, Page(s): 839 – 848.
- [18] Andrew DeOrio, Ilya Wagner and Valeria Bertacco, "DACOTA: Post-silicon validation of the memory subsystem in multi-core designs", 2009 IEEE 15th International Symposium on High Performance Computer Architecture, Page(s): 405 – 416.
- [19] Erfan Azarkhish, Davide Rossi, Ignore Loi, Luca Benini, "Design and evaluation of a processing-in-memory architecture for the smart memory cube", proceedings of the 29th International Conference on Architecture of Computing Systems- ARCS 2016, Volume 9637, Page(s): 19-31.
- [20] B. Anderson, A. Easwaran, J. Lee, "Finding an upper bound on the increase in execution time due to contention on the memory bus in cots-based multicore systems," ACM SIGBED Review - Special Issue on the Work-in-Progress (WIP) Session at the 2009 IEEE Real-Time Systems Symposium (RTSS), Volume 7 Issue 1, January 2010 Article No. 4.
- [21] H. Rihani, M. Moy, C. Maiza, S. Altmeyer, "WCET analysis in shared resources real-time systems with TDMA buses," in 23rd International Conference on Real-Time Networks and Systems, 2015 Page(s): 183-192.
- [22] M. Jacobs, S. Hahn, S. Hack, "A framework for the derivation of WCET analyses for multi-core processors," in Proceedings of the Euromicro Conference on Real-Time Systems, 2016, Page(s): 141-151.
- [23] Q. Perret, P. Maurere, E. Noulard, C. Pagetti, P. Sainrat, et al, "Temporal isolation of hard real-time applications on many-core processors," in IEEE Real-Time Embedded Technology Applications Symposium, 2016, Page(s): 1-11.
- [24] G. Giannopoulou, N. Stoimenov, P. Huang, L. Thiele, B. de Dinechin, "Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources," in Real-Time Systems 52, 2016, Page(s): 399-449.
- [25] D. Oehlert, A. Luppold, H. Falk, "Bus-aware static instruction SPM allocation for multicore hard real-time systems," in 29th Euromicro Conference on Real-Time Systems, 2017, Page(s): 1:1–1:22.
- [26] A. Schranzhofer, R. Pellizzoni, J. J. Chen, L. Thiele, M. Caccamo, "Worst-case response time analysis of resource access models in multi-core systems," in Proceedings of the Design Automation Conference, 2010, Page(s): 332-337.
- [27] R. I. Davis, S. Altmeyer, L. S. Indrusiak, C. Maiza, V. Nelis, J. Reineke, "An extensible framework for multicore response time analysis," in Real-Time Systems 54, 2018, Page(s): 607-661.
- [28] R. Mukherjee, P. Dasgupta, A. Pal, S. Mukherjee, "Formal verification of hardware/software power management strategies," in 2013 26th International Conference on VLSI Design, 2013, Page(s): 326 – 331.
- [29] R. Backasch, C. Hochberger, A. Weiss, R. Lasslop, "Runtime verification for multicore SoC with high-quality trace data," in ACM Transactions on Design Automation of Electronic Systems (TODAES): Volume 18 Issue 2, March 2013.