Open2C: Open-source Generator for Exploration of Coherent Cache Memory Subsystems

Anastasiia Butko
Lawrence Berkeley National
Laboratory
Berkeley, CA
abutko@lbl.gov

Albert Chen
University of California
Berkeley, CA
albertchen@lbl.gov

David Donofrio
Lawrence Berkeley National
Laboratory
Berkeley, CA
ddonofrio@lbl.gov

Farzad Fatollahi-Fard
Lawrence Berkeley National
Laboratory
Berkeley, CA
ffard@lbl.gov

John Shalf
Lawrence Berkeley National
Laboratory
Berkeley, CA
jshalf@lbl.gov

ABSTRACT

A cache-coherent memory subsystem plays an important role in complex digital computing systems. It maintains memory consistency across on-chip caches that hide the memory latency to improve computational performance. Being managed by hardware, the cache subsystem facilitates multi-core system programming and allows developers to focus on other crucial aspects. However, due to extensive protocol-related traffic and lack of explicit data movement management, cache memory scalability becomes a big concern. Existing evaluation techniques, such as cycle-approximate estimation or cycle-accurate simulation, do not guarantee accurate and fast results in the first case or require tremendous amount of efforts to implement and modify the system in the second. We present Open Cache Coherence (Open2C). The project aims to provide a powerful yet flexible and easy-to-extend tool that enables exploring coherent cache memory subsystem for upcoming large-scale computing systems. Open2C includes a library of basic parameterized components that are required to build a complex coherent cache memory subsystem, such as miss register, TAG array, replacement policy unit, etc. The Open2C generator is written in Chisel language that allows each component to be accessed through provided methods in a functional and modular way. The generated system can be simulated using existing Chisel-based simulation tools or be compiled into the RTL and placed on FPGA for further evaluation. Open2C reduces the amount of effort researchers spend on system implementation – allowing them to focus on the protocol itself or a separate unit optimization.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Computing methodologies** → *Modeling and simulation*; • **Hardware**

→ *Communication hardware, interfaces and storage*; *Best practices for EDA*;

KEYWORDS

Cache coherency, design space evaluation, manycore

ACM Reference Format:

Anastasiia Butko, Albert Chen, David Donofrio, Farzad Fatollahi-Fard, and John Shalf. 2018. Open2C: Open-source Generator for Exploration of Coherent Cache Memory Subsystems. In *The International Symposium on Memory Systems (MEMSYS '18)*, October 1–4, 2018, Old Town Alexandria, VA, USA. ACM, New York, NY, USA, Article 4, 7 pages. <https://doi.org/10.1145/3240302.3270314>

1 INTRODUCTION

A cache-coherent memory subsystem plays an important role in complex multiprocessor computing systems. It ensures transparent data movement to maintain memory consistency to simplify the life of programmers and also manages data movement in caches to hide memory latency – thereby improving computational performance. Being managed by hardware, the cache subsystem facilitates more productive multi-core system programming and allows developers to focus on other crucial aspects of programming. However, with the end of Dennard scaling, the parallelism of chip multiprocessors has been growing exponentially, which leads to serious concerns about the scalability of coherence protocols due to extensive protocol-related traffic. Moreover, the increasing demands in computational performance pushes research towards highly specialized architectures with diverse hardware accelerators that requires memory subsystem to be less generic and could greatly complicate coherence protocols.

These emerging and hybrid solutions require careful and detailed study. The complexity of these protocols already makes it very hard to evaluate, and inhibits our ability to make significant advances in the future. Without an approach to overcome this overwhelming complexity, we are at risk to get stuck with existing approaches.

Existing evaluation techniques are not adequate to the task. Cycle-approximate estimation has an insufficiently detailed model to uncover the key hazards to the scalability of such protocols and

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only.

MEMSYS '18, October 1–4, 2018, Old Town Alexandria, VA, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-6475-1/18/10...\$15.00

<https://doi.org/10.1145/3240302.3270314>

does not guarantee accurate or fast results. Cycle-accurate simulation requires a tremendous amount of effort to implement and modify the detailed system model, which greatly restricts the design space that can be studied.

In response to these challenges, we present the Open Cache Coherence (Open2C) toolkit, which aims to provide a powerful yet flexible and easy-to-extend tool that enables exploring cache coherency mechanisms for upcoming large-scale computing systems. Open2C includes a modular library of the basic building blocks that are required to build a complex coherent cache memory subsystem, such as miss registers, TAG arrays, replacement policy unit, etc. The generators for these components are written in the Chisel language, which allows each component to be parameterized (so they can be resized or reconfigured) and accessed through provided methods in a functional and modular way. The Open2C generator allows a user to combine library modules to assemble a complete system that follows specific user-defined coherence protocols. The generated system can be simulated using existing Chisel-based simulation tools or be compiled into the RTL and placed on FPGA for much faster cycle-accurate emulation of the model. Open2C reduces the amount of effort researchers are required to spend on system implementation – allowing them to focus on the protocol itself or a separate unit optimization.

The contributions of this work are as follow:

- The first coherent cache memory subsystem hardware generator that enables fast and flexible implementation of functional simulation models as well as Verilog HDL and corresponding synthesizable RTL;
- A MESI protocol case study implementation that evaluates required effort on the part of the user, source code size and complexity, and the performance results of the generated hardware implementation;
- An overview of the validated version of the generator that will be publicly available to the community¹ and description of future work.

The rest of the paper is organized as follows: Section 2 provides the related work on current tools and approaches for coherent cache evaluation. Section 3 describes the Chisel generation methodology that we propose to use for large-scale application-specific cache studies. In Section 4, we provide a detailed description of the Open2C generator, its infrastructure and components, and show a case study implementation of the MESI protocol controllers. Also, we give the current state of the project and describe the future work in Section 5. Section 6 provides concluding remarks.

2 RELATED WORK

In this section, we describe the related work on cache memory subsystem evaluation techniques.

2.1 Separate evaluation

Coherent cache memory subsystem consists of two major components:

- (1) The coherence protocol (e.g. MESI, MOESI, etc.) that enforces a memory consistency model on data shared among multiple local and shared caches.
- (2) Memory storage resources, such as cache data arrays, TAG arrays, miss registers, and also replacement and associativity mechanisms.

Both components act as an integrated system and have significant impact on the subsystem performance. Because of the complexity of such a system, researchers must often chose to evaluate them separately.

Simple cache coherence models target the protocol itself with an abstract notion of the available hardware resources. MultiCacheSim [17] is an emulation tool for experimenting with coherence protocols. It provides MSI and MESI protocols as default options and allows for additional user customization. The tool also allows the user to parametrize a number of key architecture characteristics, such as memory size, cache line size, replacement policy and associativity. MultiCacheSim only provides statistics on cache coherence protocol related events and does not support multi-level cache hierarchy.

From the other side, cache memory resource estimation tools can predict memory latency, power, area by abstracting away dynamic behavior of the coherence control system. For example, CACTI [19] [15] is an analytical model that takes a set of cache parameters as an input and provides cache and memory access time, cycle time, area, leakage, and dynamic power estimation as an output.

Such an approach of separation of concerns (independent evaluation of the cache coherence protocol and the cache physical model) does not guarantee the required realism of the entire system operation – especially in the context of large-scale architecture studies. Moreover, it is crucial to take into account application-specific details, such as amount of shared memory, data granularity and reuse, etc. to fully understand the impact of the design choices in the application context.

2.2 System evaluation

gem5 [3] is a powerful cycle-approximate simulation framework that provides multiple simulation capabilities to address almost the entire stack of the computer architecture research. It contains several CPU models ranging from functional to highly-detailed descriptions, supports most of the existing instruction sets, allows contract complex multi-processor and multi-node systems with multi-level cache hierarchy, interconnect and external memory models. Also, gem5 offers two simulation modes – system emulation and full-system simulation that includes entire software stack (OS, runtime, etc.).

Event-driven simulation technique usually comes at a price of simulation speed and system scalability. Moreover, despite an impressive level of detail and capabilities, gem5 simulation accuracy is still problematic [11] – particularly when used to model novel structures or designs. Therefore, such an approach allows a limited number of large-scale studies with a non-guaranteed level of accuracy.

2.2.1 Abstraction approaches. In order to speed-up the simulation, several approaches have been developed [6] [18]. Trace-driven

¹<https://github.com/lbnlcomputerarch>

Figure 1: Cache simulation time breakout

simulation proposes to abstract certain system components by emulating their behavior with the pre-collected traces and focus on target component detailed simulation.

Dinero [7] is a trace-driven simulator for uni-processor cache hierarchy studies. It produces a number of useful performance metrics, such as traffic to and from the memory. In [14], Dinero has been used to predict the performance of a 3D processor-memory on-chip stack. However, it does not support multiple cores architectures required for coherence protocol evaluation.

CASPER [13] is another trace-driven simulation framework for multi-core cache performance evaluation studies. The tool was developed to extend existing approaches and support SMP and CMP architectures studies.

However, the provided speed-up in terms of simulation speed does not guarantee simulation scalability. In our prior work [5] in evaluating this methodology, we demonstrated that increasing the number of virtual cores, i.e. trace injectors (TIs) leads to a substantial increase of the coherence traffic. The increased number of events in turn leads to a slow down in simulation performance and poor scalability. Figure 1 shows the cache simulation breakdown for two scenarios:

- *Full-system simulation of a single-core processor architecture.* 69% of the simulation time is spent simulating the core micro-architectural details related to the computation. Almost equal amount of time is spent to simulate cache and interconnect events. Memory simulation takes only 2% for a given evaluation example.
- *Trace-driven simulation of a 256-injector system.* Replacement of the core with a trace injector allows to simulate a 256-injectors system. However, 87% is taken to simulate the cache-related events that does not scale with the size of the system.

Thus, while trace-driven approach can push the simulation time boundary significantly further, the poor scalability of the cache coherency subsystem makes trace-driven approaches not suitable for large-scale studies.

2.2.2 Hardware emulation. Hardware emulation using FPGA is one of the most suitable way to evaluate hardware in terms of accuracy/speed trade-off. In [4], authors implement application-specific cache coherence protocol using VHDL and evaluate it on an FPGA board.

Figure 2: Chisel generating software and hardware from a single codebase.

However, RTL modeling is known to be extremely time-consuming because of the hardware description languages complexity and amount of efforts designer needs to spend for model implementation and debug.

To overcome this complexity, recent trends in hardware generation and evaluation promotes the use of high-level HDLs, such as Chisel [2] or Haskell [12]. These languages offer additional functionalities coming with functional- and object-oriented programming paradigms. Chisel has been successfully used to implement Rocket Chip [1] generator for RISC-V ISA processor tiles and OpenSoC Fabric [9] that allows generating parameterizable on-chip networks. However, no coherent cache memory subsystem generator has been implemented yet.

In this work, we present Open2C generator that allows customizing and implementing complex cache memory subsystem to enable accurate large-scale evaluation studies.

3 CHISEL GENERATION METHODOLOGY

Chisel [2] is a Scala-based HDL. It raises the level of hardware design abstraction by providing concepts including object orientation, inheritance, functional programming, parameterized types, structures of variables, and type inference. Since it is based on Scala, Chisel can use the full power of Scala for non-synthesizable code, such as testbenches. Synthesizable code uses Chisel statements and modules, which includes a powerful library of often-used primitives, such as conversions between different encodings.

Chisel's software model is used for simple network design exploration with fast-functional models, while the hardware model can be used in emulation for cycle-accurate modeling. Using the synthesis path provided by Chisel, hardware generators allow interesting and non-trivial designs to be synthesized for a target ASIC process or FPGA flow, generating accurate power parameters and evaluating cycle time, which software simulators are unable to do. These power measurements can then be back annotated to the software model to drastically improve the software model's power estimates.

Figure 3: An example of the Level 1 cache controller (L1 ctrl).

Alternatively, software models can collect their own statistics for the purposes of power estimation, and can even generate wave forms purely from functional simulation.

Chisel takes care of both the software and hardware models with one single code base as shown in Figure 2. Given that developing Chisel code is much less tedious than RTL and comparable to functional programming, this means that the overall effort to construct both software and hardware models reduces dramatically. In addition, there is very little, if any, effort required to provably match the behavior of the software code to the RTL, which otherwise is a non-trivial and time-consuming process.

Several hardware generators have been successfully implemented in Chisel and are widely used by both academia and industry, such as OpenSoC Fabric [9], Rocket Chip [1], Dynamically Allocated Neural Network (DANA) Accelerator [8].

While the Chisel DSL is still in active development, there have been several examples of processors successfully manufactured that have been designed exclusively in Chisel. This illustrates the faithful representation of the Verilog that Chisel produces from the user's Scala description.

4 OPEN2C GENERATOR

4.1 Structure and Components

The Open2C generator consists of two independent constituents: (i) *hardware resource library* and (ii) *protocol-dependent coherence controllers*. These two constituents interact through the designed access interface. This separation of concerns is inspired by the gem5 RUBY memory generation [10], where the coherence protocol is described with the SLICC domain specific language. The hardware models are implemented as a separate library and the coherence controllers are compiled by combining these two constituents. While the approach is the same, we emphasize that such level of automation and generalization is hard to achieve even using Chisel-like HDL. In this work, we demonstrate a semi-automated approach intending to extend it in the future work. We describe each of the constituent and the interface in the following subsections.

```
class TestConfig extends Config {
  case L1DSize => 32 //KB
  case L1ISize => 32 //KB
  case L2Size => 512 //KB
  case DirectorySize => 512
  case AddressBitLength => 32
  case WordBits => 64
  case PAddrBits => 32
  case BlockBytes => 64
  case NSets => 512
  case Nways => 4
  case TBETableSize => 16
  case BufferDepthSize => 8
  case StateBits => 5
  case ReplPolicy => () => new TreeLRU(site(Nways))
  case StallTableEntries => 8
}
```

Figure 4: An example of the Open2C configuration.

4.1.1 Parameterizable hardware resources library. The parameterizable hardware resource library consists of the basic set of protocol-independent cache modules. It includes the following components:

- **Data array.** The parameters associated with this module are data array size, address bit length, number of block words and the word bits length as well as number of the associativity ways and sets.
- **TAG array.** TAG array keeps track of the addresses present in the cache memory, provide access to the local address of the corresponding cache line and informs about the free space availability. Also, it contains the coherence state per each cache line/set/way and can be parametrize in accordance to the cache data array and other relevant modules, such as coherence state bit length.
- **Replacement policy units.** These units are responsible to keep track of the cache line usage and provide the replacement victim line in case there is no available cache space. The default replacement policy is Least Recently Used (LRU). More policies can be implemented as separate units and changed in the parameter configuration.
- **Miss Status and Handling Register (MSHR) or TBE.** TBE register plays a role of a temporary storage in case the cache miss occurred. Before the data can be stored or written to the cache, a number of transient message exchanges must happen to insure the correctness of the coherence protocol. The TBE register size, data capacity and state width can be parametrize.
- **Stall Table.** The stall table is used to care the pending requests in case the controller is busy or there is no hardware resources available at the moment.
- **Message Buffers.** Message buffers are designed to hold input and output messages of a particular format. Message parameters and buffer depth size can be configured.

4.1.2 Protocol-dependent coherence controllers. Coherence controllers of different cache levels are protocol-dependent. These controllers combine protocol-independent modules, coherence protocol Finite State Machine (FSM) and controller internal resources FSM into a complex interactive system. They orchestrate data movement,


```

// Transitions from IS
}.elsewhen( t == ((L1states.IS << 4) | L1events.Data_all_Acks) ) {
    newState := L1states.S
    setMRU(addr, way, cache) //set most recently used (MRU) for replacement policy
    writeToCache(way, respFromNetBuf.io.deq.bits.respAddr, respFromNetBuf.io.deq.bits.respData, cache)
    generateRespMsg(addr, respFromNetBuf.io.deq.bits.respData, RespTypes.DATA, machineID.U, p_CPU, cache, 0.U, false.B)
    deallocateTBE(respFromNetBuf.io.deq.bits.respAddr)
    updateTagState(addr, newState, way, false.B, cache)
    wakeUpDependents(addr) //wake up dependents: re-analyze stalled msgs
    respFromNetBuf.io.deq.ready := true.B //ready to incoming response buffer
}

```

Figure 5: Resource access functions.

exclusive and shared write access, sharers and owner notifications, etc., following specific coherence protocol. The major components of the coherence controllers are:

- **Request and Response messages.** These messages represent the wire bundles of different formats to facilitate the inter-module communications. The format of the message is parameterizable and consists of the following fields: address, type, data, virtual network ID, destination, source and several additional fields. All messages are divided into two major groups: response and request messages. Depending on the message group, they can be of a different type that corresponds to the typical cache communication messages, such as LOAD, STORE, FETCH for communication between the core and its private cache; GETS, PUTX, DATA, etc for inter-controller data communication; UNBLOCK, ACK, EXCLUSIVE, etc. for coherence traffic and MEM_READ, MEM_WRITE, DMA, etc. to memory access.
- **Protocol FSM.** The FSM describes all the transactions between base and transient states. Each transaction requires to make a number of actions, such as update state, allocate or deallocate cache and/or TBE entry, send the data or other coherence message, etc.
- **Controller resources FSM.** This component is responsible to manage the internal resources of the controller. Typically the FSM starts with the idle state, then goes to the lookup in case a request or response arrived and finishes up in protocol transaction, replacement or stall state.

Figure 3 shows an example of the L1 cache controller structure. It includes TAG arrays, data arrays, LRUs per instruction and data caches, i.e. iTagArray, iDataArray, iLRU and dTagArray, dDataArray, dLRU respectively. It has three input ports associated with the request from the CPU, request from the network and response from the network and three output ports associated with the response to the CPU, response to the network and request to the network. Each of the port is connected to the buffer of the corresponding type.

All of the described modules can be configured in a single configuration file. To ensure the consistency of the parameters shared among multiple modules and allow passing of the parameters through the hierarchy, we use Context Dependent Environment (CDE) library [16]. Figure 4 shows an example of the Open2C configurations.

4.1.3 Hardware resources access interface. In order to facilitate the hardware resources access and management from the coherence controller, we provide a set of functions similar to the class methods per each protocol-independent module.

Figure 5 shows an example of the access functions used in the controller FSM. The shown fragment contains source code of a transition from the IS state to S state when Data_all_Acks event occurs. To complete this state transition controller performs a set of operations, i.e. sets the Most Recently Used (MRU) cache line, writes data to cache, generates response message to the CPU, deallocates the TBE buffer space, updates the state in the TAG array, pushes corresponding requests from the Stall Buffer to the queue and removes the network response from the input buffer. All these operations are described in a compact way that takes only seven lines of code.

4.2 Case Study

In this section, we demonstrate the case study implementation of the two-level MESI cache sub-system using Open2C generator. The proposed implementation is based on the MESI Two Level gem5 model [10].

4.2.1 MESI protocol implementation. MESI Two Level is a single-chip 2-level cache system with strictly-inclusive hierarchy [10]. The level 1 caches are private per each core and level 2 cache is shared among all the cores. The cache coherence is maintained by the L2 controller through directory coherence mechanism. Directory controller acts as interface to the memory and is responsible for maintaining coherence across multiple chips and external coherence requests from the DMA controller.

Protocol acronym consists of four basic states:

- **Modified (M):** the cache line is present in only one L1 cache with exclusive permission, it is writable and readable and is dirty.
- **Exclusive (E):** the cache line is also present in only one L1 cache with exclusive permission, but has not been written yet.
- **Shared (S):** the cache line is present in a shared state in one or more L1 caches, as well as in L2 cache, it is only readable and matches the main memory.
- **Invalid (I):** the cache line is invalid or not present.

In addition to the basic set of states, L1 and L2 controllers have separate sets of transient states. L1 controller supports up to 10

Figure 6: Synthesis results of the MESI Cache Coherence Controllers generated by Open2C: area percentage per modules.

transient states including prefetching and 18 events that can trigger state transition. L2 controller additionally supports 8 transient and 5 blocking states, and 17 external and internal events to trigger the transition. Also, the protocol model proposed in [10] has a number of optimizations, such as silent eviction, exclusive permission block for read request, etc. More details about the protocol model, states and optimizations can be found in the project web site [10].

Using Open2C capabilities, we have implemented the described MESI protocol system that includes one L1 cache controller per each core, one shared L2 controller and one Directory controller. Hardware parameters of the generated system are similar to the configuration shown in Figure 4.

4.2.2 Results. Using Chisel tool flow shown in Figure 2, we generate the described components for software simulation and hardware emulation. Here, we demonstrate the hardware flow studying Verilog HDL implementation of controllers.

To analyze generated hardware in terms of chip area, we use Yosys Open SYnthesis Suite [20]. Yosys framework provides a basic set of synthesis algorithms, converting Verilog RTL to BLIF/EDIF/BTOR/SMT-LIB/etc. and reports area, number of cells, number of wires and other results based on a provided technology library. For our experiments, we use 14nm technology.

Figure 6 shows synthesis results per L1 cache, L2 cache and Directory controllers. Each sub-figure includes a list of modules as well as the number of each module per controller. For example, L1 controller shown in Figure 6 (a) contains two data arrays, TAGs and replacement units: first set is for instruction cache and second set is for data cache. 62% of the chip area is occupied by the data arrays, 13% and 10% are taken by input and output request and response buffers respectively, the rest of the area is distributed among TAGs, stall table, TBE, LRUs and controller logic itself.

L2 controller shown in Figure 6 (b), has only one set of data array, TAG and LRU, but the size of the data array is significantly larger (512KB versus 32KB per data and 32KB per instruction in L1). 56% of the area is occupied by the data array. TAG area percentage is larger than in L1 controller. Input and output buffers occupy similar percentage of the controller area, i.e. 12% and 13%.

Directory controller shown in Figure 6 (c), does not have typical cache modules. Most of the area is taken by request and response

buffers, i.e. 32% and 33%, TBE table and stall table, i.e. 21% and 12% respectively.

This case study demonstrates an example of the coherent cache memory sub-system evaluation flow in terms of the area usage. Same experiments can be easily conducted with different set of system parameters (e.g. data array size, buffer depth, register number, etc.) as well as different technology library.

5 CURRENT STATE AND FUTURE WORK

Open2C currently provides a basic set of protocol-independent modules and MESI protocol-dependent set of controllers, i.e. L1 controller, L2 controller and Directory controller.

Our current plans for future work are as follows:

- *Protocol correctness validation:* To ensure the correctness of the protocol implementation, we plan to design a testbench that will generate multi-core traffic emulating different application patterns.
- *Hardware resources library extension:* We plan to extend the existing hardware resources library to support different replacement policies, optimized TAGs and TBE register management.
- *Automatic protocol-dependent generation:* Current protocol-dependent modules generation is based on the hand-written Chisel code. We plan to extend our generation methodology to support fully automatic construction of the controllers based on the protocol description file. Such enhancement, will significantly facilitate generation of the protocol-dependent modules.
- *System integration:* We plan to integrate our generator in a single multi-functional framework that includes not only cache memory sub-system, but cores, interconnect and other components. The desired system generated by such integrated framework is shown in Figure 7. Cores generated with Rocket Chip [1] for example and memory are connected to the Open2C coherent cache controllers. Memory controllers are connected using OpenSoC Fabric [9] generated network composing a complete system on chip.

Figure 7: System on Chip implemented using multiple open-source generators as a single integrated environment.

6 CONCLUSIONS

In this paper we described Open2C - an open-source coherent cache memory subsystem generator. Open2C is a powerful tool written in Chisel. Being based on Scala, Chisel allows implementing hardware in a modular, flexible and efficient way. Open2C allows to construct and customize coherent cache using the provided set of basic protocol-independent modules, such as Data Array, TAG, replacement policy units, etc.. Module access interfaces are designed to facilitate inter-module communication in a same way it's done in C++ models. Open2C allows generating high level C++ models for functional simulation as well as Verilog HDL code that can be placed on FPGA for hardware emulation or be used for ASIC fabrication.

In our case study, we demonstrated the MESI protocol implementation using Open2C. We generated Verilog HDL code for L1 cache controller, L2 cache controller and Directory controller. We used Yosys open synthesis system to evaluate the generated implementation in terms of the chip area with the 14nm technology library.

As a future work, we listed four major directions: design of the protocol correctness validation testbench, extension of the hardware resources library, automatic protocol-dependent modules generation and integration of the Open2C with other generators in to a single system. We plan to make Open2C publicly available for the community after the MESI protocol validation.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the the U.S. Department of Energy, grant agreement n° DE-AC02-05CH11231.

REFERENCES

[1] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report

UCB/EECS-2016-17. EECS Department, University of California, Berkeley. <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html>

[2] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzyniec, and K. Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. 1212–1221. <https://doi.org/10.1145/2228360.2228584>

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. [n. d.]. The Gem5 Simulator. *SIGARCH Comput. Archit. News* ([n. d.]).

[4] David Brooks and Margaret Martonosi. 1999. Implementing Application-Specific Cache-Coherence Protocols in Configurable Hardware.

[5] Anastasiia Butko. 2015. *Fast Cycle-approximate Simulation Techniques for Many-core Architecture Exploration*. PhD dissertation. University of Montpellier, France.

[6] A. Butko, R. Garibotti, L. Ost, V. Lapotre, A. Gamatie, G. Sassatelli, and C. Adeniyi-Jones. 2015. A trace-driven approach for fast and accurate simulation of manycore architectures. In *The 20th Asia and South Pacific Design Automation Conference*. 707–712. <https://doi.org/10.1109/ASPDAC.2015.7059093>

[7] J. Edler and M. D. Hill. [n. d.]. Dinero IV Trace-Driven Uniprocessor Cache Simulator. <http://pages.cs.wisc.edu/markhill/DineroIV/>.

[8] Schuyler Eldridge. 2016. Neural network computing using on-chip accelerators. <https://open.bu.edu/handle/2144/19511>

[9] F. Fatollahi-Fard, D. Donofrio, G. Micheliogiannakis, and J. Shalf. 2016. OpenSoC Fabric: On-chip network generator. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 194–203. <https://doi.org/10.1109/ISPASS.2016.7482094>

[10] gem5. 2016. Ruby Memory Subsystem. <http://www.gem5.org/Ruby>

[11] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver. 2014. Sources of error in full-system simulation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 13–22. <https://doi.org/10.1109/ISPASS.2014.6844457>

[12] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, Maria M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. 1992. Report on the Programming Language Haskell: A Non-strict, Purely Functional Language Version 1.2. *SIGPLAN Not.* 27, 5 (May 1992), 1–164. <https://doi.org/10.1145/130697.130699>

[13] R. Iyer. 2003. On modeling and analyzing cache hierarchies using CASPER. In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003*. 182–187. <https://doi.org/10.1109/MASCOT.2003.1240655>

[14] P. Jacob, O. Erdogan, A. Zia, P. M. Belemjian, R. P. Kraft, and J. F. McDonald. 2005. Predicting the performance of a 3D processor-memory chip stack. *IEEE Design Test of Computers* 22, 6 (Nov 2005), 540–547. <https://doi.org/10.1109/MDT.2005.151>

[15] H. P. Labs. [n. d.]. *CACTI 6.0: A Tool to Model Large Caches*. Technical Report. <http://www.hpl.hp.com/techreports/2009/HPL-2009-85.html>

[16] Jim Lawson. 2016. Context Dependent Environments. <https://github.com/ucb-bar/context-dependent-environments>

[17] Brandon Lucia. [n. d.]. MultiCacheSim: A coherent multiprocessor cache simulator. <https://courses.cs.washington.edu/courses/cse471/11sp/sim.html>

[18] Ali Saidi and Andreas Sandberg. 2012. gem5 Virtual Machine Acceleration. <http://www.gem5.org/wiki/>

[19] S. J. E. Wilton and N. P. Jouppi. 1996. CACTI: an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits* 31, 5 (May 1996), 677–688. <https://doi.org/10.1109/4.509850>

[20] Clifford Wolf. 2017. Yosys Open Synthesis Suite. <http://www.clifford.at/yosys/>