MGSim—A Simulation Environment for Multi-Core Research and Education

Raphael Poss, Mike Lankamp, Qiang Yang, Jian Fu, Irfan Uddin, Chris R. Jesshope
University of Amsterdam
Computer Systems Architecture group
Science Park 904, 1098XH Amsterdam
The Netherlands

***Abstract*—This article presents MGSim[1], an open source discrete event simulator for on-chip hardware components developed at the University of Amsterdam. MGSim is used as research and teaching vehicle to study the fine-grained hardware/software interactions on many-core chips with and without hardware multithreading. MGSim's component library includes support for core models with different instruction sets, a configurable multi-core interconnect, multiple configurable cache and memory models, a dedicated I/O subsystem, and comprehensive monitoring and interaction facilities. The default model configuration shipped with MGSim implements Microgrids, a multi-core architecture with hardware concurrency management. MGSim is furthermore written mostly in C++ and uses object classes to represent chip components. It is optimized for architecture models that can be described as process networks.**

I. Introduction

MGSim is a discrete event simulator for on-chip hardware components, developed at the University of Amsterdam[2] since 2007. MGSim is a research and teaching vehicle to study the fine-grained hardware/software interactions on many-core and hardware multithreaded processors. It includes support for core models with different Instruction Set Architectures (ISAs), a configurable multi-core interconnect, multiple configurable cache and memory models, a dedicated I/O subsystem, and comprehensive monitoring and interaction facilities.

The motivation to develop a new framework instead of reusing existing simulators was twofold.

The first motivation was to focus on research in design of new processor architectures, instead of modeling the behavior of existing processors. MGSim was thus, from the start, optimized for the design space exploration of new components for multi-processor systems-on-chip, i.e. testing different combinations of features and architecture parameters to optimize platforms towards specific applications; it was also optimized towards the design of new techniques in processor micro-architecture, i.e. adding or changing features in individual processor cores and their multi-core interconnect.

The other motivation was to support undergraduate and graduate education activities in computer architecture, parallel programming, compiler construction and operating system design. In particular, MGSim was tailored to three extra usability requirements: provide a human-scale software infrastructure that can be comprehended by standalone students in these fields, integrate the emulation and guest operating software in a software package that can be seamlessly deployed and get ready to run on student computers with minimal effort. MGSim also provides features usually expected from simulation packages, including comprehensive and automatable interfaces to observe and illustrate the internal workings of a system while it is running.

The development of MGSim was originally aimed at exploring the behavior of D-RISC cores [1], [2] when grouped together in a multi-core chip. Because of this historical background, the default model configuration shipped with MGSim simulates the *Microgrid platform*, so that programmers can use MGSim as a full-system emulation of a device or chip containing clusters of D-RISC cores, also known as Microgrids [3]. Since then, MGSim has matured into a versatile framework to simulate many-core architectures.

As a software infrastructure, MGSim's component models and simulation kernel are written in C++; they use object classes to represent chip components. Ancillary tools are written in Python. A characteristic feature of the MGSim framework is that it promotes the definition of architecture models where components across clock domains only synchronize via FIFO buffers, i.e. where models can be described as process networks. MGSim is further available[3] free of charge under an open source license.

This article introduces the MGSim tool box, as of version 3.3. We start in section II by positioning MGSim relative to other simulators. We then present its applications in sections III to V. We review its simulation framework and component model library in section VI. Finally, we outline future developments in section VII and conclude in section VIII.

II. Context and related work

A. Emulation vs. simulation

MGSim is both a simulation framework and a full-system emulator. As a simulator it can be used to predict component behavior, in particular during the architecture design phase. In this role it is useful to the architecture researcher. As an emulator it can be used to reproduce the hardware/software interface of a device. In this role it is useful to operating system and compiler developers.

[1]MGSim was supported by the Dutch government via the project NWO Microgrids, the European Union under grant numbers FP7-215216 (Apple-CORE) and FP7-248828 (ADVANCE), the University of Amsterdam, and grants by the China Scholarship Council.

[2]http://csa.science.uva.nl/

[3]currently hosted at http://svp-dev.github.com/.

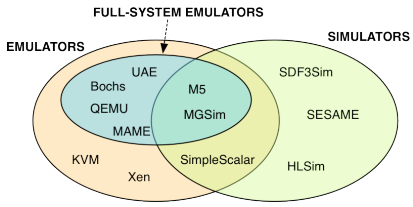


Figure 1. Emulators and simulators: a Venn diagram with examples.

To position MGSim next to related work, we can divide emulators further, between partial and full system emulation. Note that virtualization technologies can be considered as emulations. With partial emulation, only application code runs within the emulation environment, and operating system functions are serviced through a host/guest interface. With a full emulation, the entire software stack runs on the emulated hardware. MGSim can serve both as a hardware simulator and full-system emulator. We illustrate these distinctions in fig. 1.

B. Related work

In the group of software frameworks that are both simulators and emulators, as illustrated above, MGSim most closely relates to SimpleScalar[4] and Gem5[5] [4] (previously called simply "M5"). In contrast to MGSim, SimpleScalar only provide partial emulation: operating system functions are served on the host platform via the `syscall` pseudo-instruction. MGSim was designed as a full-system emulation so as to also study the behavior of operating software when running over the emulated platform. Moreover, SimpleScalar was primarily designed to emulate single-core platforms, where MGSim's focus lies towards multi-core platforms. Its purpose and even its software architecture make MGSim much closer to the Gem5 framework. Gem5, like MGSim, consists of a library of C++ components that can be grouped in configurable topologies to define multi-core platforms. Both frameworks are discrete event, component-based simulations able to emulate full systems. At the time of this writing, Gem5 even offers more monitoring and visualization facilities than MGSim. The differences between Gem5 and MGSim can be found at two levels.

Firstly, Gem5 was designed and motivated to emulate *existing* platforms. In particular, one of its design requirements was to be able to run entire existing software stacks unchanged, for example GNU/Linux, FreeBSD, L4K or Solaris. MGSim does not share this requirement, and its implementation is thus much simpler than Gem5's. This makes MGSim more accessible for education activities than Gem5. Secondly, Gem5 started as a single-core system emulator, focusing on the accurate simulation of large, state-of-the-art sequential processors. Multi-core support was only added later, and Gem5 is still optimized for use with few cores sharing a high-level functional emulation of a cache coherency network and inter-processor interrupt network. In contrast, MGSim was designed from the ground-up as a many-core network featuring different detailed memory interconnects and a dedicated point-to-point messaging network between cores. This makes MGSim a potentially more productive tool for research in the design of new operating software for parallel applications.

[4]http://simplescalar.com/
[5]http://m5sim.org

Table I. LOCs COMPARISON WITH GEM5 AND SIMPLESCALAR

Simulator	C/C++ LOCs	Python LOCs	LOCs others
Gem5 9073 (27-06-2012)	298k	77k	800 SWIG, 64k config
MGSim 3.3 (01-01-2013)	42k	800	400 config
SimpleScalar 3.0e	27k	N/A	N/A

Table II. COMPONENT LIBRARY COMPARISON WITH GEM5 AND SIMPLESCALAR

Simulator	Cores (ISAs)	Memories	I/O devices
Gem5 9073 (27-06-2012)	3 (6)	5	10+
MGSim 3.3 (01-01-2013)	1 (3)	7	6
SimpleScalar 3.0e	1 (2)	1	N/A

We summarize the size and scope of MGSim and its closest relatives in tables I and II. In short, MGSim can be considered as the "little brother" of Gem5, oriented towards research in *new core architectures and more diverse memory systems*.

III. ORIGINAL RESEARCH TARGET: MICROGRIDS

The *Microgrid* many-core architecture is a research project at the University of Amsterdam, which investigates whether concurrency management (thread scheduling, synchronization, and inter-thread communication) traditionally under control of software operating systems can be accelerated in hardware to obtain higher efficiency and performance. Microgrids are clusters of a simple RISC core design called D-RISC [1]; each D-RISC core is equipped with a hardware Thread Management Unit (TMU) which can coordinate with neighbouring TMUs for automatic thread and data distribution.

Prior to the use of MGSim, research on D-RISC and the Microgrid was focused on programmability issues and carried out with high-level simulators: both using traditional software multithreading and an API to emulate the TMU services [5], and using a custom functional ISA emulator [6]. As the initial phases of the D-RISC design had built confidence that the design was sound, the EU-funded project Apple-CORE (2008-2011) was started to study its implementability in a system, including a full vertical tooling stack from an FPGA implementation up to benchmarks in higher-level programming languages. To avoid placing the FPGA specification on the critical path of the project, the need arose to develop simultaneously, at lower cost, a cycle-accurate simulator of Microgrids which would support early results with the rest of the tooling.

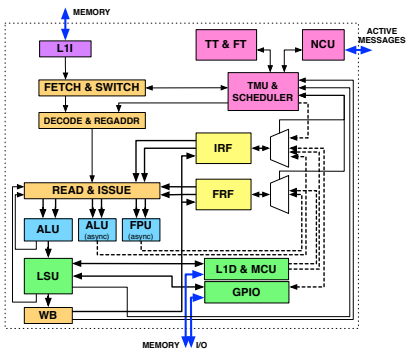


Figure 2. D-RISC core micro-architecture, as simulated in MGSim.

The motivation to implement a new core simulation model from scratch instead of extending an existing RISC model in another simulator was twofold. First, the D-RISC design extends a traditional single issue, 6-stage in-order pipeline with a larger register file equipped with full/empty state bits, and implements thread scheduling based on the availability of input operands to instructions (fig. 2). This causes the pipeline design to diverge significantly from existing models, both structurally and in its timing behavior. The other reason is that the Microgrid architecture uses multiple logical Networks-on-Chip (NoCs) to negotiate coordination and synchronization between cores, and no existing simulator was known at that time with both support for multiple NoCs and customizable core-network interfaces.

The outcome of the Apple-CORE project is summarized in [3], [7]: the D-RISC core was implemented on FPGA as UTLEON3 [8], a model of Microgrids was implemented in MGSim, software tooling was delivered to program Microgrids [9], [10], [11], and D-RISC and Microgrids were confirmed using both UTLEON3 and MGSim to deliver higher performance and efficiency for a class of applications.

Two example results from Apple-CORE obtained with MGSim are given in figs. 3 and 4. In the first example, a discrete Mandelbrot set approximation of 200×200 points was implemented using one "microthread" (the Microgrid's logical unit of work) per point. The workload is both fine-grained and heterogeneous: each thread executes between 60 and 1000 instructions, but the control flow and number of instructions is different for each point. Two implementations of the benchmark were used. The first was the "even" implementation, where the logical range is divided into P equal segments where P is the number of cores in the cluster, i.e. core p runs microthread indices $\{start_p, start_p + 1, start_p + 2 \ldots\}$. This is the straightforward use of the bulk creation process in D-RISC's TMU. With this implementation, the benchmark's performance had been previously observed to scale from 1 to 32 cores and to benefit from hardware multithreading [3, fig. 4]. However, MGSim's tracing abilities enabled us to plot the diagrams in figs. 3a and 3b, which revealed a shortcoming: because the TMU bulk-synchronizes threads, maximum performance is limited by the longest execution time in a batch of threads. Thanks to this additional insight provided by MGSim, we could design an alternate "round-robin" implementation, where the logical range is distributed in a round-robin fashion over the P cores in the cluster, i.e. core p runs indices $\{p, p + P, p + 2P \ldots\}$. This implementation invokes the D-RISC TMU separately with different logical index ranges on every core of the cluster, which effectively compensates the shortcoming identified above by randomizing the workload across different batches.

In the second example, MGSim's models were used in a configuration as close as possible as one of the Intel IXP chips, so as to compare the Microgrid's performance with the IXP in the NPCryptBench suite [12], [13]. This configuration places the multithreaded cores in a crossbar with two DDR channels, and we also ensured that the core, interconnect and DDR timing parameters were as aligned with the IXP as possible. Full alignment was not possible, because the IXP is clocked at 1.4GHz and the D-RISC's register file access time in this configuration constrain its clock to 1Ghz. However, as the results in fig. 4 show, the Microgrid model provides a throughput advantage for the more complex AES, SEAL and Blowfish ciphers.

From the MGSim perspective, the net outcome of Apple-CORE is a library of components where the more mature models are the D-RISC core and the various memory interconnects relevant to study shared memory performance in a many-core chip. A list of these components with detailed descriptions is provided separately in [14]. Due to Apple-CORE's strategy to explore the spectrum of architecture parameters for Microgrids, all MGSim components and the simulated chip topology are highly parameterizable.

Finally, although the D-RISC model was originally designed for Microgrids, it uses standard ISAs (Alpha, SPARC, MIPS) and its associated software tool chain can be used with standard C code to run common sequential benchmarks. In other words, it is possible for a user of MGSim to ignore the Microgrid ancestry and exploit MGSim to study both computer organization and the behavior of memory systems under multi-core workloads.

IV. Ongoing research projects using MGSim

As of this writing, MGSim continues to support research on the design of Microgrid components, towards a future implementation in silicon. Simultaneously, it also still support undergraduate and graduate education projects.

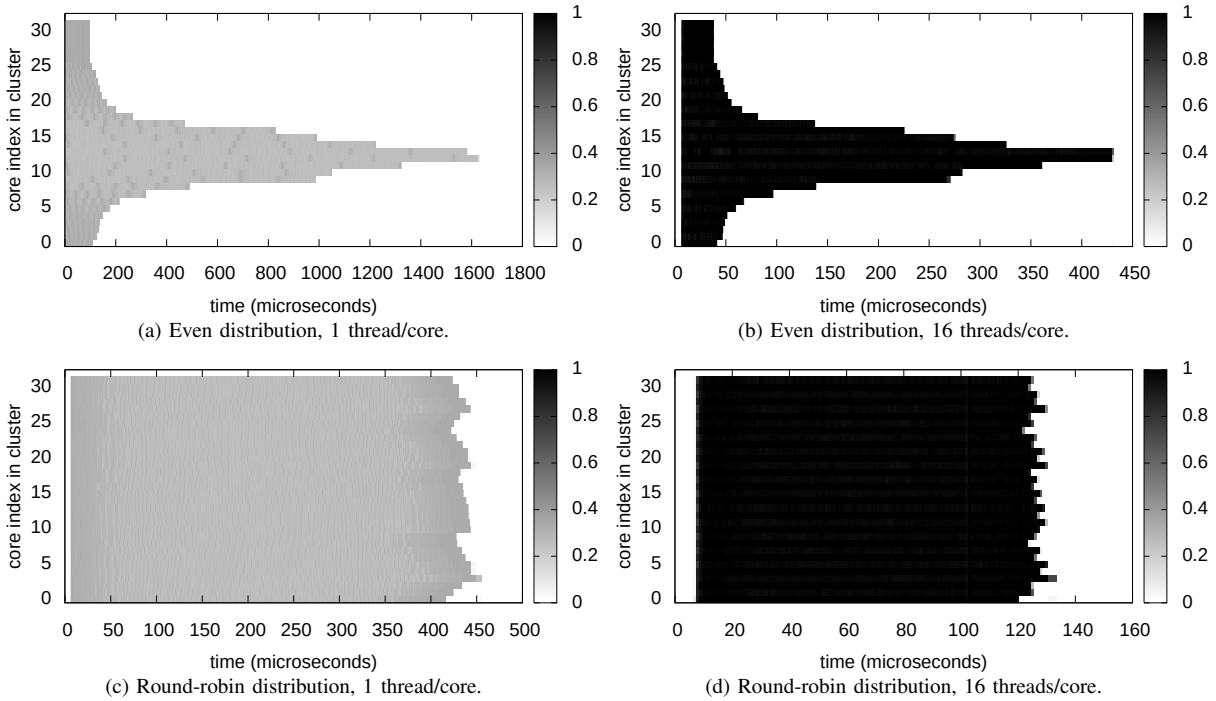
On the research side, an industry-backed project focuses on hardware fault detection and recovery in Microgrids, as well as real-time semantics in D-RISC's thread scheduler. This project uses MGSim to prototype the features and predict their behavior. Graduate and doctoral research projects also include research in three areas. One is the optimization of distributed cache coherency protocols, where MGSim helps validating consistency semantics and deadlock freedom (cf. e.g. [15], [14]). Another is the design and implementation of operating system components in software that account for thread management in hardware, where MGSim provides a full-system emulation platform that enables testing and benchmarking (cf. e.g. [16], [7], [17]). The last is research of high-level models of parallel software behavior when both concurrency granularity and hardware parallelism are model parameters, where MGSim is used to calibrate the high-level models (cf. e.g. [18], [19]).

V. Application to education

MGSim was used to support lab assignments next to courses in microprocessor architecture, system organization and parallel programming.

A. ISA design and organization

A common activity in architecture education is the implementation of an ISA in a processor simulator. MGSim supports this activity in two ways. Like most other hybrid low-level simulators (incl. Gem5 discussed previously), MGSim separates the component-level simulation, detailed for timing accuracy, from the functional simulation of instruction effects. In particular, a single C++ interface enables plugging multiple ISAs onto the common D-RISC micro-architecture model. Students can thus replace or add an ISA to the simulation



(a) Even distribution, 1 thread/core.
(b) Even distribution, 16 threads/core.
(c) Round-robin distribution, 1 thread/core.
(d) Round-robin distribution, 16 threads/core.

Figure 3. Per-core IPC over time for the Mandelbrot set approximation, running on 32 cores.

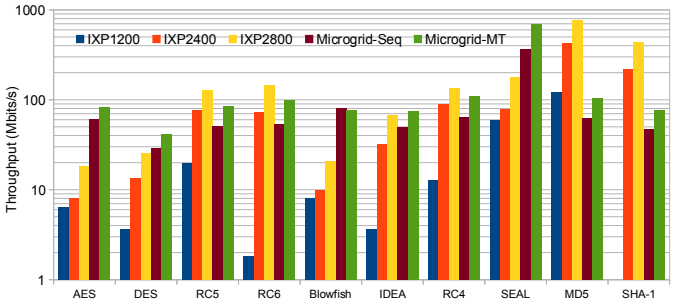


Figure 4. Throughput for one cryptographic stream on one core.

framework and immediately obtain feedback from existing benchmark code simply re-compiled towards the new ISA. This is possible because the hardware/software interfaces stay mostly the same between ISAs for D-RISC: the thin operating system code and benchmark code can be reused as soon as the ISA is implemented, as no other code (e.g. OS drivers, etc) needs to be ported/rewritten. In 2012, this opportunity was used by undergraduate students to introduce support for the MIPS ISA, where MGSim's D-RISC previously only supported the SPARC and Alpha ISAs. The task could be completed by pairs of students within four weeks (10-20 hours/week), without prior exposure to neither MGSim's internals nor other ISAs.

B. Cache behavior analysis

In another activity, undergraduate students were given a couple of memory-intensive sequential benchmarks and a list of architecture configurations. The goal was to investigate the cache behavior of these programs and how the structure of the memory system was impacting their performance.

MGSim supports this activity in three ways. First, its detailed memory architecture models accurately reflect contention and delays in the memory network: the distribution of memory latencies is spread to reflect the diversity found in real chips. Then by running the benchmarks in a full-system emulation environment, the students were able to capture the memory behavior of operating system services as well, which was relevant for some benchmarks. Finally, by providing a fully automatable user interface, MGSim enabled students to run a large number of experiments over many architecture parameters, so as to recognize relevant parameters. For example, through a combinatorial exploration taking less than eight hours of work, and again without prior exposure to cache analysis, students were able to discover empirically that the benefits of increased set associativity become marginal as the cache size grows, and that the benefits of increased cache size depends on the application being executed.

C. Parallel programming

In 2011 and 2012, a parallel programming summer school was organized for graduate students at the University of Amsterdam. The goal was to present the technologies and trade-offs of parallelism over various platforms, including shared memory multi-cores. One of the targets of the practicals was thus the Microgrid platform, and students used MGSim to carry out evaluations on their own computer. Next to the educational value of the Microgrid, which provides a radically different set of trade-offs between concurrency management overheads than other architectures, the particular benefit of MGSim was its determinism: students could re-play the fine-grained interactions between threads and the memory system, investigate race conditions reliably, and observe in detail how their mapping and scheduling decisions impacted the cache behaviour of their code.

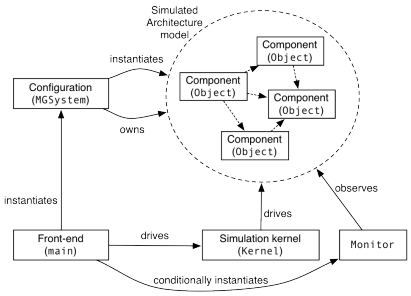


Figure 5. Entity-relationship diagram of an MGSim simulator.

VI. Framework and component models

MGSim's framework is composed of five main parts, illustrated in fig. 5. The *simulation kernel* implements a rather conventional sequential, discrete-event, component-level simulation. It provides base abstractions for processes, registers, buffers, latches, arbitrators and ports, as well as an execution driver that schedules ready component processes at each simulation cycle. The *library of component models* provides object classes for the various component types found on the chip: processors (cores), caches, memory networks, I/O interconnects, etc. The component models are implemented using the base abstractions from the simulation kernel. Typically, a component defines one or more processes, optionally some internal state for its processes, and latches, buffers and/or arbitrators visible from other components. The *system configuration constructor* instantiates the component models and connects them together to form a full architecture model. This part is further distributed between a top-level "system topology" constructor and the individual constructors of component models, which may choose to instantiate subcomponents or dependent components. The *simulation front-end* provides a user interface to MGSim. The interface is composed of command-line parsing, configuration file loader, interactive command interpreter, event trace filtering, asynchronous monitoring, etc. Finally, an optional *asynchronous monitor* runs asynchronously in a separate thread of execution. It periodically samples the state of selected components and writes it to a trace file or FIFO for analysis or visualization by external tools.

A. Simulation overview

Upon initialization of an MGSim instance, the front-end parses the command-line parameters and configuration file(s). It then creates a *configuration object* that holds a database of configuration variables. The front-end then instantiates the *configuration constructor*, which in turn populates the architecture model by instantiating components according to the configuration. After this point, the model is ready: no further objects are constructed, and the simulation can start.

If invoked to run interactively, the front-end displays an interactive prompt and accepts user commands. For example, invoking the `run` command starts the simulation by triggering the *step* method of the simulation kernel. This method advances the simulation by one or more *master cycles*, from a master clock running at the lowest common multiple of the frequencies of all clocks in the model. At every cycle, the following happens.

First, any pending updates to stateful structures shared by components (e.g. FIFO buffers) are committed, to become visible during the new cycle. Then the *acquire* phase of the cycle is run for all active component processes. During the acquire phase, process handlers declare their intent to use shared structures and *request arbitration*. Also during acquire, processes may not update internal state. After the acquire phase completes, all involved arbitration requests are resolved by the kernel. Once arbitration has been resolved, all active processes run the *check* phase of the cycle. During this phase, the *results of arbitration is reported to each process*, which determines *which control path to use* (e.g. stall, access another shared storage, etc.). Again, during this phase, processes may not update internal state. Once the check phase has completed, all remaining non-blocked processes run the *commit* phase of the cycle. During this phase, processes use the control path chosen during the check phase, may update their internal state, and declare updates to shared storage to be effected at the start of the next cycle. They may also emit informational messages to be logged to a synchronous event trace by the kernel.

At the end of each cycle, active processes are then rescheduled to run at the next cycle or some cycles later, according to their simulated clock frequency.

Note that during the check phase, processes may become blocked because of denied arbitration, but also when attempting to read from empty FIFOs. When a process becomes idle on an empty input FIFO, it will thus only be reactivated after a subsequent cycle produces data into the FIFO. This is the mechanism by which MGSim models the behavior of asynchronous networks of components.

B. Anatomy of a component

Components in the simulation framework correspond to components on chip, i.e. to an area of hardware. They are organized in a tree, where each child node represents a subparts of its parent component. For example, the `DCache` (L1 data cache) and `Pipeline` components are child nodes of the processor component (`Processor`) which encompasses them. Each component is related to its *parent* component and *children* components, if any, a *clock domain* (either its own or shared with its parent), and the specific simulation kernel that drives the entire component tree.

Additionally, each component may define one or more of the following. *Processes* represent state machines or functional circuits. *Shared storages and arbitrators*, e.g. FIFO buffers, registers or single-bit latches, may be used by two or more processes including processes from other components, and may cause processes to block upon access. *Internal state* is used by only one process, or represents state shared by processes of the same component that does not require arbitration nor decide process scheduling. *Services* provide part of the logic of processes from other components. *Inspection handlers* are invoked from MGSim's interactive command prompt upon user commands. Finally, *administrative data* may be implemented as well for meta-information that does not represent hardware components (e.g. counters for statistics).

Processes in the simulation framework represent the activities of data transformation and communication in the system. They are triggered by the availability of data in a specific

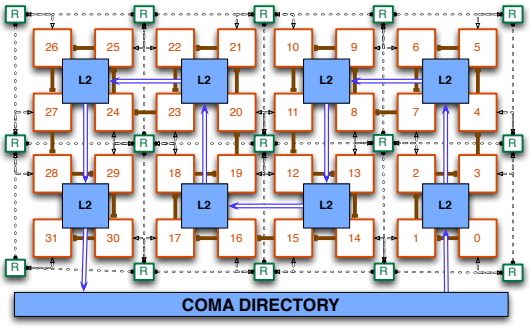


Figure 6. Example MGSim configuration: a 32-core Microgrid.

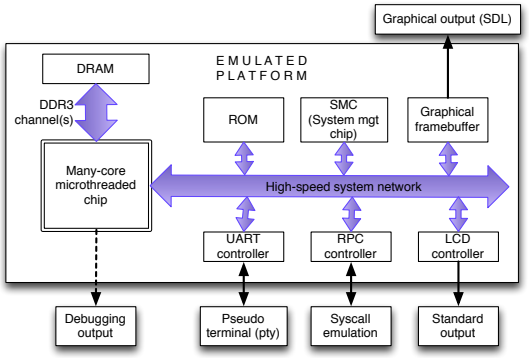


Figure 7. Example MGSim configuration: a full-system Microgrid platform.

shared storage, which is called its *source storage*. When triggered, a process becomes active and its *cycle handler* is called by the kernel at every cycle of the corresponding clock domain. The process' cycle handler may then in turn attempt to acquire more storage or arbitrators, fail while doing so and thus stall. When stalled, the cycle handler re-tries the same behavior in subsequent cycles until the behavior succeeds. Upon successful completion, a process may either consume data from its source storage or stay ready to be invoked again for another behavior in the next cycle (e.g. in state machines). A process becomes idle when its source storage becomes empty.

C. MGSim's component library

The standard platform configuration shipped with MGSim is depicted in figs. 2, 6 and 7. The origin of this specific configuration and its relationship with the FPGA UTLEON3 implementation was described above in section III and previously published in [3], [20].

This platform is composed of reusable components from MGSim's library. We provide an overview of this library in table III. When MGSim starts, the system configuration constructors aggregates high-level configuration requests from the users, for example "desired type of memory interconnect," "desired number of cores" and "desired number of cores per L2 cache," then derives automatically a system topology and instantiates the components. The user thus does not need to explicitly list the configuration of each component individually. However, if so desired the user can optionally override the configuration of some components to produce heterogeneous

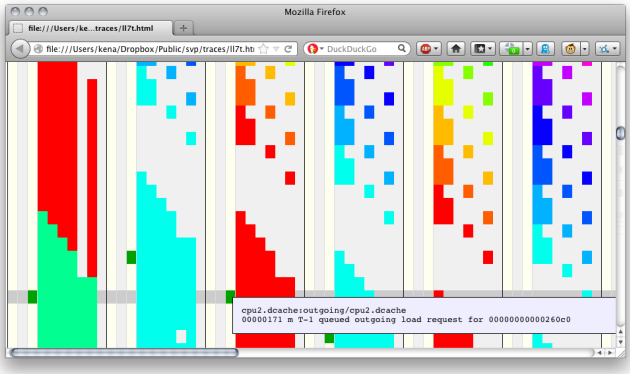


Figure 8. Example visualization of synchronous event traces.

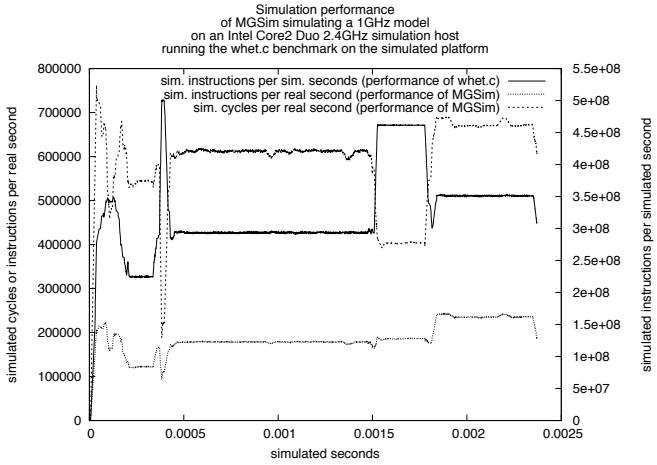


Figure 9. Example application of asynchronous monitoring.

models; for example a configuration can specify "16 cores, but so that cores 4-15 are not connected to the I/O subsystem," or "32 cores, but so that core 0 has larger L1 and L2 caches."

D. Trace visualization and simulation speed

Like most cycle-accurate simulators, MGSim can produce detailed event traces of a simulation, reporting all component-to-component interactions. MGSim's event trace format is homogeneous and can thus be processed automatically to produce interactive visualization. An example is given in fig. 8: a 8-core model is running a parallelized implementation of the equation of state fragment found as loop 7 of the Livermore benchmark suite [21]. The visualization uses one column per component and one row per cycle. Within one column, different colors are used for different hardware threads in the D-RISC core. The browser window is centered in the start of the benchmark's data-parallel operation; the cursor is hovering at the intersection between cycle 171 and the L1 D-Cache of core 2, and a pop-up label shows a memory event occurring at that location.

MGSim also provides asynchronous monitoring, to capture the evolution over time of semi-continuous variables in the simulation model. It is implemented using a monitor thread running concurrently with the simulation thread, which re-

Table III. OVERVIEW OF THE MAIN COMPONENTS IN MGSIM'S LIBRARY.

Top-level simulation component	Hardware component modeled	Simulation detail	Main sub-components
`Processor`	D-RISC core and TMU-to-TMU NoC protocol	circuit-level interactions	`Pipeline, DCache, ICache, RegisterFile, Allocator, Network, MMUInterface, IOInterface`
`FPU`	Asynchronous FPU pipeline	intruction latencies and structural hazards	
`DDRChannel`	DDR3 controller	DDR latencies and D-RAM bank contention	
`SerialMemory`	Single D-RAM bank with bus interconnect	request latencies and contention	
`ParallelMemory`	Single D-RAM bank with crossbar	request latencies and contention	`Port`
`BankedMemory`	Multiple D-RAM banks with crossbar	request latencies and contention	`Bank`
`DDRMemory`	Multiple DDR channels with crossbar	request latencies and contention	`DDRChannel`
`ESAMemory`	Multiple DDR channels with L2 cache and crossbar	request latencies and contention	`Cache, DDRChannel`
`COMA`	Ring-based L2 cache diffusion network with write-update coherency protocol	request latencies, network/cache interactions, contentions	`Cache, Directory, DDRChannel`
`ZLCOMA`	Ring-based L2 cache diffusion network with write-invalidate coherency protocol	request latencies, network/cache interactions, contentions	`Cache, Directory, DDRChannel`
`ActiveROM`	Passive ROM and DMA controller	request bandwidth and latency	
`UART`	NS/PC16650 UART	request bandwidth and latency	
`Display`	Graphical output interface	request bandwidth and latency	`FrameBufferInterface, ControlInterface`
`RTC`	Programmable real-time clock	functional only	
`LCD`	Character matrix display	functional only	
`RPC`	Pseudo-device: exposes the host's filesystem	functional only	
`SMC`	Pseudo-device: enumeration and initialization	functional only	

Table IV. MGSIM PERFORMANCE COMPARED TO RELATED TOOLS.

Simulator	Performance
UTLEON3 (FPGA)	20MIPS, 1 core
MGSim 3.3	100-1000KIPS shared by all active simulated cores
MGSim 3.3 (traces disabled)	0.5-2 MIPS shared by all active simulated cores
HLSim (higher-level, software)	100MIPS-1GIPS, scalable to multiple host cores

peatedly samples a set of selected variables to an output trace. An example use case is illustrated in fig. 9. In this example, a 16-core simulation model was configured to run the classical Whetstone benchmark[6] (a sequential program) using an Apple MacBook Pro as simulation host. The monitor thread was configured to sample the simulation cycle counter and the counter for the number of instructions executed in the cores' pipeline. The sampling rate was configured to 1000 samples/s. The execution of `whet.c` lasted for approximately 4.22 real seconds, little over 2.3ms of simulated time; it ran about 3.1M instructions. During this time, the monitor thread produced 3705 samples at an approximated effective rate of 876 samples/s.

As the figure shows, for this model and this host MGSim runs approximately 200K instructions per real second (KIPS) in general, or 150-200x slower than an equivalent hardware implementation. This perspective gives a better view of the simulation speed than the naive estimation based on the final counts, which indicate $3.1 \times 10^6/4.22$ or about 740KIPS for this particular program.

We have measured that enabling traces (both synchronous and asynchronous) slows down the simulation by a factor 3x-6x on average. We indicate the position of MGSim's performance compared to related tools in table IV.

VII. SHORTCOMINGS AND POSSIBLE FUTURE WORK

Our experience using MGSim for architecture research and education has revealed a few shortcomings, which we briefly review here.

[6]http://www.netlib.org/benchmark/whetstone.c

The first is the common occurrence of implementation or design errors when implementing a new model in MGSim. The most common error is the definition of deadlocking circuits due to circular dependencies. Although the component model exposes all dependencies between buffers and processes, the MGSim framework is not yet able to analyze and detect circular dependencies automatically. The implementation of such a detection mechanism would significantly reduce the time required to troubleshoot modeling errors.

The second shortcoming is the lack of a facility to checkpoint/restore the entire simulation state. When a failure occurs, the only mechanism available to-date to reproduce the issue is to re-play the entire execution scenario since the start of the simulation. If the program further uses I/O, an exact re-execution is nearly impossible. This becomes an issue particularly when troubleshooting long-running software within the simulated platform. Mechanisms to serialize and de-serialize the simulation state, similarly to the "freeze" feature of virtual machines, would greatly increase the suitability of MGSim as a sandbox environment to troubleshoot simulated software.

The third shortcoming is visible in the light of the previous two: were MGSim extended to address the issues already mentioned using the same C++ infrastructure, the complexity of the source base would gradually increase and may put it out of the intellectual reach of students or newcomers to the field. Moreover, increasing the amount of code without automated functional validation would increase the rate of specification errors and decrease the overall quality of the project. To ensure the continued relevance of MGSim, a shift to higher-level specifications is required. We can for example envision using languages like CλaSH [22] or BlueSpec [23] and generate both MGSim components and RTL-level models from the same input specification.

Next to these shortcomings, the question arose of what to do about the similarities between MGSim and Gem5, discussed previously in section II-B. Despite the different project goals, the overlap between the technical approaches is striking; in particular, the inter-component interfaces, component granularity and configuration facilities are intriguingly similar between

the two projects. This opens two opportunities. The first is to investigate whether MGSim's library of memory models could be reused with Gem5, which is somewhat still lacking in this regard. The other is to determine whether Gem5's core models could be reused with MGSim, to provide increased platform compatibility to programs running on the simulated platform.

VIII. Summary and conclusions

We have presented MGSim, an open source framework and component library to simulate many-core processors. MGSim's framework is written in C++ and implements a highly configurable, discrete-event, multi-clock simulation engine. Its library of components provides a versatile hardware multithreaded in-order RISC core supporting multiple ISAs, multiple memory interconnects, and an I/O subsystem which enables full-system emulations. Its comprehensive inspection and monitoring facilities make it suitable for both architecture research and education.

MGSim is currently used at the University of Amsterdam and its partners. Its applications include scientific research on the Microgrid architecture [3] and general graduate-level education on processor, cache and memory architectures. Performance-wise, MGSim is known to run models containing thousands of components at 100-1000KIPS on conventional desktop-grade hardware.

MGSim is similar to Gem5 [4], another C++-based framework for discrete-event, component-based multi-core simulations. The two frameworks run with comparable performance. Where Gem5 focuses on compatibility with real hardware and intra-core accuracy on models with few cores, MGSim focuses on implementation simplicity and accuracy with large many-core models.

References

[1] A. Bolychevsky, C. Jesshope, and V. Muchnick, "Dynamic scheduling in RISC architectures," *IEE Proceedings - Computers and Digital Techniques*, vol. 143, no. 5, pp. 309–317, September 1996.

[2] T. Bernard, K. Bousias, L. Guang, C. R. Jesshope, M. Lankamp, M. W. van Tol, and L. Zhang, "A general model of concurrency and its implementation as many-core dynamic RISC processors," in *Proc. Intl. Conf. on Embedded Computer Systems: Architecture, Modeling and Simulation (IC-SAMOS 2008)*, W. Najjar and H. Blume, Eds. Samos, Greece: IEEE, July 2008, pp. 1–9.

[3] R. Poss, M. Lankamp, Q. Yang, J. Fu, M. W. van Tol, and C. Jesshope, "Apple-CORE: Microgrids of SVP cores (invited paper)," in *Proc. 15th Euromicro Conference on Digital System Design (DSD 2012)*, S. Niar, Ed. IEEE Computer Society, September 2012. [Online]. Available: pub/poss.12.dsd.pdf

[4] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, pp. 52–60, 2006.

[5] M. W. van Tol, C. R. Jesshope, M. Lankamp, and S. Polstra, "An implementation of the SANE Virtual Processor using POSIX threads," *J. Syst. Archit.*, vol. 55, no. 3, pp. 162–169, 2009.

[6] K. Bousias, N. Hasasneh, and C. Jesshope, "Instruction level parallelism through microthreading – a scalable approach to chip multiprocessors," *The Computer Journal*, vol. 49, no. 2, pp. 211–233, March 2006. [Online]. Available: http://comjnl.oxfordjournals.org/content/49/2/211.abstract

[7] R. 'kena' Poss, "On the realizability of hardware microthreading—revisting the general-purpose processor interface: consequences and challenges," Ph.D. dissertation, University of Amsterdam, 2012. [Online]. Available: http://www.raphael.poss.name/on-the-realizability-of-hardware-microthreading/

[8] M. Daněk, L. Kafka, L. Kohout, J. Sýkora, and R. Bartosinski, *UTLEON3: Exploring Fine-Grain Multi-Threading in FPGAs*, ser. Circuits and Systems. Springer, November 2012. [Online]. Available: http://www.springer.com/engineering/circuits+%26+systems/book/978-1-4614-2409-3

[9] D. Saougkos and G. Manis, "Run-time scheduling with the C2uTC parallelizing compiler," in *2nd Workshop on Parallel Programming and Run-Time Management Techniques for Many–Core Architectures, in Workshop Proceedings of the 24th Conference on Computing Systems (ARCS 2011)*, ser. Lecture Notes in Computer Science. Springer, 2011, pp. 151–157.

[10] R. k. Poss, "SL—a "quick and dirty" but working intermediate language for SVP systems," University of Amsterdam, Tech. Rep. arXiv:1208.4572v1 [cs.PL], August 2012. [Online]. Available: http://arxiv.org/abs/1208.4572

[11] C. Grelck, S. Herhut, C. Jesshope, C. Joslin, M. Lankamp, S.-B. Scholz, and A. Shafarenko, "Compiling the Functional Data-Parallel Language SaC for Microgrids of Self-Adaptive Virtual Processors," in *14th Workshop on Compilers for Parallel Computing (CPC'09), IBM Research Center, Zurich, Switzerland*, 2009.

[12] Z. Tan, C. Lin, H. Yin, and B. Li, "Optimization and benchmark of cryptographic algorithms on network processors," *IEEE Micro*, vol. 24, no. 5, pp. 55–69, September/October 2004.

[13] Y. Yue, C. Lin, and Z. Tan, "NPCryptBench: a cryptographic benchmark suite for network processors," *SIGARCH Comput. Archit. News*, vol. 34, no. 1, pp. 49–56, September 2005.

[14] M. Lankamp, R. Poss, Q. Yang, J. Fu, I. Uddin, and C. R. Jesshope, "MGSim—simulation tools for multi-core processor architectures," University of Amsterdam, Tech. Rep. arXiv:1302.1390v1 [cs.AR], February 2013. [Online]. Available: http://arxiv.org/abs/1302.1390

[15] C. Jesshope, M. Lankamp, and L. Zhang, "Evaluating CMPs and their memory architecture," in *Architecture of Computing Systems – ARCS 2009*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, vol. 5455/2009, pp. 246–257.

[16] L. van Duijn, "Dynamic program loading in a shared address space," BSc Thesis, University of Amsterdam, Institute for Informatics, June 2012. [Online]. Available: http://dist.svp-home.org/doc/leendert-van-duijn-program-loading.pdf

[17] M. W. van Tol, "On the construction of operating systems for the Microgrid many-core architecture," Ph.D. dissertation, University of Amsterdam, 2013. [Online]. Available: http://dare.uva.nl/record/436834

[18] M. I. Uddin, M. W. van Tol, and C. R. Jesshope, "High level simulation of SVP many-core systems," *Parallel Processing Letters*, vol. 21, no. 4, pp. 413–438, December 2011.

[19] M. I. Uddin, C. R. Jesshope, M. W. van Tol, and R. Poss, "Collecting signatures to model latency tolerance in high-level simulations of microthreaded cores," in *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, ser. RAPIDO '12. New York, NY, USA: ACM, 2012, pp. 1–8. [Online]. Available: pub/mirfan.12.pdf

[20] R. Poss, M. Lankamp, M. I. Uddin, J. Sýkora, and L. Kafka, "Heterogeneous integration to simplify many-core architecture simulations," in *Proc. 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, ser. RAPIDO '12. ACM, 2012, pp. 17–24. [Online]. Available: pub/poss.12.rapido.pdf

[21] F. McMahon, "The livermore FORTRAN kernels: A computer test of the numerical performance range," Lawrence Livermore National Lab., CA (USA), Tech. Rep. UCRL-53745, Dec 1986.

[22] R. Wester, C. P. R. Baaij, and J. Kuper, "A two step hardware design method using CλaSH," in *22nd International Conference on Field Programmable Logic and Applications*, ser. FPL'12. Oslo, Norway: IEEE Computer Society, August 2012, pp. 181–188.

[23] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *Proc. 2nd ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, ser. MEMOCODE '04, june 2004, pp. 69–70.