







par-gem5: Parallelizing gem5’s Atomic Mode

*Niko Zurstraßen , *José Cubero-Cascante , *Jan Moritz Joseph ,

†Li Yichao , †Xie Xinghua , *Rainer Leupers 

*RWTH Aachen University, Institute for Communication Technologies and Embedded Systems

†Huawei Technologies

Abstract—While the complexity of MPSoCs continues to grow exponentially, their often sequential simulations could only benefit from a linear performance gain since the end of Dennard scaling. As a result, each new generation of MPSoCs requires ever longer simulation times. In this paper, we propose a solution to this problem: *par-gem5*—the first universally parallelized version of the Full System Simulator (FSS) gem5. It exploits the host system’s multi-threading capabilities using a modified conservative, quantum-based Parallel Discrete Event Simulation (PDES). Compared to other parallel approaches, *par-gem5* uses relaxed causality constraints, allowing temporal errors to occur. Yet, we show that the system’s functionality is retained, and the inaccuracy of simulation statistics, such as simulation time or cache miss rate, can be kept within a single-digit percentage. Furthermore, we extend *par-gem5* by a temporal error estimation that assesses the accuracy of a simulation without a sequential reference simulation. Our experiments reached speedups of $24.7\times$ when simulating a 128-core ARM-based MPSoC on a 128-core host system.

Index Terms—Parallel Discrete Event Simulation, gem5, Full-System Simulation

I. INTRODUCTION

The general trend in today’s CPU architectures is increased core count and system complexity. This complicates the design process, creating a need for a plethora of design tools to understand the impact of architectural decisions. Full System Simulators (FSSs) [5], [19], [22] are the backbone for computer architects, as they allow evaluating architectural optimization in the CPU’s pipeline, the cache system, and the core interconnects, modeling a real operating system that executes real workloads. Key Performance Indicators (KPIs), such as power consumption or compute performance, can be determined even before physical prototypes are available.

A common problem of FSSs is their inability to simulate massively parallel and complex systems, as their simulation kernels usually only run on a single thread. The most prominent example is *gem5* [5]—the de facto standard open-source FSS for ARM-based systems. It does not support multi-threaded simulation, even though the simulated system can have over 100 cores. More than 1 MIPS accumulated is hardly achievable even on the most modern computers, and the execution of representative benchmarks takes days if not weeks. For example, executing the popular SPEC2017 integer benchmark suite natively on a modern host computer requires approximately 10 minutes. Simulating the same workload in *gem5* modeling a 64-core system requires *more than two years*. This issue is only expected to get worse as the gap between the single-core performance and the transistor count as a representative for the system size increased in the last decades (see Fig. 1) and is likely to continue so. If computer architects do not transition to parallel FSS, the big freeze of simulation will be inevitable.

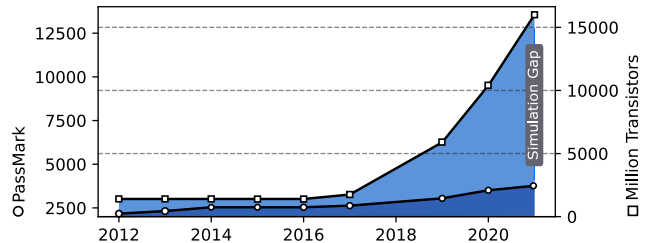


Fig. 1. Development of highest single-threaded *PassMark* results [2] and transistors per CPU. From 2012 (Intel Xeon E3-1290 v2) to 2021 (Apple M1), the single-threaded score increased by 73%, while the number of transistors increased by 1042%.

To tackle this growing gap in simulation tools, multiple solutions have been proposed to increase the simulation speed. For example, trace-driven simulations like *gem5*’s ElasticTraces [12], system emulation with KVM [18], or the Simpoint method [17]. Although considerable speedups can be achieved with these methods, this speedup is always based on an abstraction of partial aspects of an FSS leading to significant inaccuracies. Therefore, these methods do not replace but only approximate FSSs. The only solution to close the gap between limits from stagnating single-core performance in FSSs and the growing complexity of systems is to execute the FSS in parallel. For parallel simulations, the most crucial challenge arises from the synchronization of events between different threads. It is essential to find an efficient solution that does not break the FSS’s functional correctness, but also ensures high performance with a limited number of synchronizations.

In this work, we solve this issue by proposing a parallel execution kernel for *gem5*. We use a modified synchronous, conservative approach in the Parallel Discrete Event Simulation (PDES) kernel with relaxed causality constraints. The simulation time is divided into slices, called quanta, in which event queues, including cores and caches, run independently in parallel. Threads only synchronize their events along with the barriers of the quanta at fixed times during the simulation. Compared to other works, our approach accepts errors in causality to increase simulation performance. In other words, we trade a speedup from parallelism with simulation accuracy. As the most important finding of this work, we show that this approach retains the target system’s functional correctness, and the errors in simulation statistics are small.

II. BACKGROUND

A. *gem5*’s Discrete Event Simulation

The gist of *gem5* is a Discrete Event Simulation (DES) engine. Objects in *gem5*, such as the CPU model, can schedule events at a certain simulation time, or remove them if the

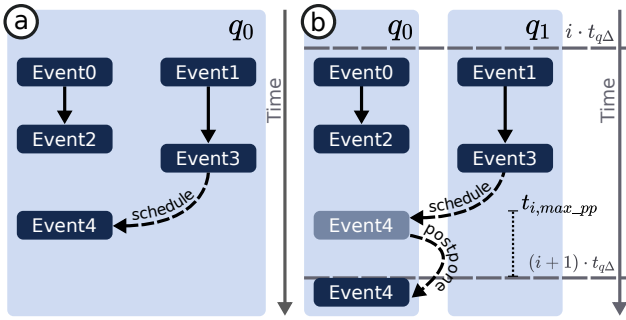


Fig. 2. Example of scheduling events in gem5.

event has not yet been processed. A scheduled event is initially located in an event queue, which is processed step by step and in chronological order by the simulation engine. When the event is processed, a function assigned to the event executed, fulfilling implementation-specific tasks. For example, the *tickEvent* of gem5’s atomic CPU executes the next instruction. In most cases, events also schedule or deschedule other events, resulting in a dependency graph, as depicted in Fig. 2 a. The dependency graph $G = (V, E)$ of a DES can be described by the following mathematical model:

$$V \subseteq \{(v, t, q) \mid v \in Events, t \in \mathbb{Z}^+, q \in EventQ\} \quad (1)$$

$$E \subseteq \{((v_0, t_0, q_0), (v_1, t_1, q_1)) \mid (v_i, t_i, q_i) \in V \wedge t_0 \leq t_1\} \quad (2)$$

The graph comprises a set of scheduled events/vertices V , whereby each event v has an associated timestamp t and an event queue q . In the default sequential gem5 simulation, a single event queue handles all events. Schedule or deschedule dependencies refer to the edges E . Since the system is causal, edges can only advance in time and never reach back to past events. Due to dynamic dependencies, the exact shape of the graph can only be determined to a limited extent during the simulation. When all events have been processed or the simulation stop event is reached, the simulation terminates.

B. Parallel Simulation in FSSs

Fundamental work on PDES was first published by Chandy et al. [7] in 1979. The work shows how independent processes are simulated in a distributed manner without focusing on any specific use case. One of the most difficult challenges of implementing any Parallel Discrete Event Simulator (PDES) is the synchronization of the individual threads. Depending on how synchronization is achieved, simulators can be either classified as *synchronous* or *asynchronous* [20].

In asynchronous simulations, as used in [22], [23], the individual threads communicate their local time with each other. Each thread keeps track of the times of all other threads, to determine when it is safe to execute a given event. This, however, poses the risk of situations where no event is deemed safe. Consequently, asynchronous simulations require a sophisticated deadlock prevention/recovery [11].

A synchronous simulator uses global synchronization events to advance the simulation in a lockstep fashion. In its simplest form, a synchronous simulation only parallelizes the execution of all events scheduled for the same simulation

timestamp, also called *delta cycle*. Such an approach is used in [8], [9], [20]. Within a delta cycle, a simulator can exploit parallelism, as there is no partial order for the given events. However, the restriction to delta cycles limits the potential performance gain [23]. To achieve greater speedups, distance-based approaches, as used in [3], [16], only synchronize every $t_{q\Delta}$. This significantly increases the exploitable parallelism, but poses the risk of causality errors as there is no guarantee of a chronological execution order for the events. To prevent these errors, most PDES implementations fall back to either *conservative* or *optimistic* methods [4].

With optimistic methods, causality errors are not prevented in the first place. Instead, the simulation detects and corrects them. The correction mechanism can be implemented by rolling back the system to a known correct state and reiterating the error-causing time slot with tighter synchronization. Examples of the optimistic approach can be found in [6], [13], [14].

In contrast to optimistic approaches, conservative methods prevent causality errors in the first place. This can be achieved by using design-specific timing information, which is either derived from the models or set manually. The multiple simulation threads synchronize every $t_{q\Delta}$, whereby a lookahead of t_{la} is provided for each thread. Causality can be retained if $t_{la} \geq t_{q\Delta}$. Conservative approaches are used in [16], [23].

Although gem5 is one of the most popular FSSs, almost all the aforementioned publications refer to the parallelization of SystemC [8]–[10], [14], [20], [23] or other frameworks. The only parallel extension of gem5 to date is *dist-gem5* [16], a work resulting from the consolidation of *pd-gem5* and *multi-gem5*. It can be classified as a synchronous, conservative PDES. The focus of *dist-gem5* is on simulating distributed systems connected via a Network Interface Controller (NIC). Each simulated node runs on its own gem5 instance and synchronizes with the other nodes at simulation time intervals of $t_{q\Delta}$, also called the *quantum*. Causality errors are prevented by equating $t_{q\Delta}$ with the simulated latency of the NIC t_{NIC} . This lookahead perfectly retains the causality, but limits the applicability of *dist-gem5* to the simulation of distributed systems connected via NICs.

According to [5], parallelizing gem5 is one of the community’s proclaimed future goals. But 10 years after the paper’s publication, parallel gem5 did not progress beyond a few lines of code primarily intended to support the Kernel Virtual Mode (KVM) simulation mode. To the best of our knowledge, a generalized parallel gem5 is not existent. Based on this fundament, we implemented *par-gem5*; a parallelized version of gem5 for the atomic mode, as described in the following sections.

C. Timing Mode vs. Atomic Mode

Depending on accuracy and performance constraints, gem5 can be operated either in the so-called *timing mode* or *atomic mode*. Similar concepts can be found in other frameworks, for example, SystemC, where the modes are referred to as *approximately timed* and *loosely timed*, respectively.

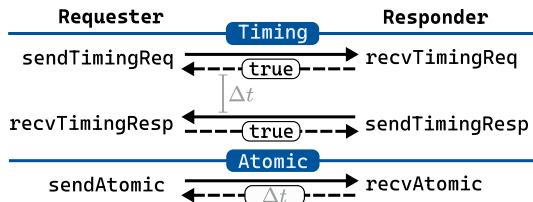


Fig. 3. Functional call diagram of gem5's timing mode and atomic mode.

The timing mode allows for accurate modeling of complex communication channels, where exchanging data between two modules comprises multiple phases. In the simplest case, there is only a request phase and response phase (see Fig. 3). Between requesting and receiving data, a simulation time of t_{Δ} passes and there is also a possibility of rejecting requests/responses by returning false instead of true. In contrast to the timing mode, the atomic mode implements the same functionality in one phase. With only one phase, dynamic effects like outstanding requests or bus contention cannot be modeled, reducing the accuracy of the simulation. However, the lower number of function calls is beneficial for the simulation performance and code complexity. Furthermore, each transaction is finished in zero simulation time, which aids the design of a thread-safe memory system as shown in Section III-C. Note that a latency t_{Δ} is returned by the function call (see Fig. 3). This latency is not part of the request/receive process, and it is up to the requester to annotate it correctly in its following processing.

III. PARALLELIZING GEM5

A. Implementation of the PDES

The principle of the *par-gem5* PDES is to increase the number of events queues and simulation threads to N , and to execute them in parallel instead of one simulation thread and one event queue. Each simulation thread has its own local time, which is synchronized globally every $t_{q\Delta}$ seconds. Hence, our implementation can be classified as a *synchronous* simulation.

As mentioned in Section II, a synchronous approach faces challenges in retaining causality. Usually, conservative or optimistic algorithms are used to circumvent that, but neither of these can be considered for our implementation. Optimistic algorithms would require an intrusive checkpoint system, which must be included from the beginning of the software project. With more than 500 kilo Software Lines Of Code (kSLOC), gem5 is already too far advanced in development. For conservative algorithms, the event latency between the individual threads must be known. This results in a rigid simulation structure with clearly defined thread boundaries, as for example seen in *dist-gem5* [16]. Since our system is supposed to be flexible, this approach is not an option either.

Contrary to conservative and optimistic approaches, our chosen method allows causality errors. However, with sophisticated event queue partitioning, their occurrence can be reduced to a minimum, achieving a high speedup with low inaccuracy. With our approach, causality errors can occur

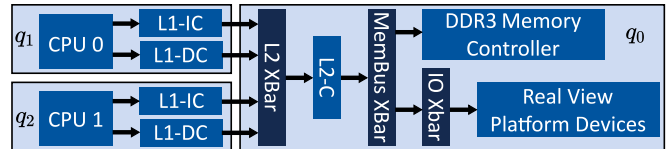


Fig. 4. Mapping of objects to event queues for a simplified dual core system.

when an event (de)schedules another event outside the event queue within a quantum (see Fig. 2 b):

$$((v_0, t_0, q_0), (v_1, t_1, q_1)) \in E \wedge q_0 \neq q_1 \quad (3)$$

$$i \cdot t_{q\Delta} < t_0, t_1 < (i + 1) \cdot t_{q\Delta} \quad (4)$$

For example, it is possible that event queue q_0 is at the beginning of the quantum ($i \cdot t_{q\Delta}$), while event queue q_1 is already at the end of the quantum ($(i + 1) \cdot t_{q\Delta}$) waiting for synchronization. If an event in q_1 is now scheduled with $t_1 < (i + 1) \cdot t_{q\Delta}$, this event will no longer be executed, because the simulation time has already exceeded the point t_1 . This scenario is depicted in Fig. 2 where Event 3 initially attempts to cross-schedule an Event 4 into the event queue q_0 within the same quantum.

One of the most important findings of our work is that in the context of FSS the "if" has a greater impact on the simulation than the "when" for most cross-scheduled events. A timer interrupt, for instance, being issued a few microseconds too late affects the simulation accuracy only slightly, while a non-issued timer interrupt might break the functionality of the whole system. Consequently, we implemented a mechanism that postpones causality breaking events to the next quantum border, as shown in Fig. 2 b.

Even though this mechanism deeply intervenes in the simulation, it retains the system's functionality. Most benchmarks and the boot sequence can be executed correctly (see Section IV). Our approach unlocks parallel simulation for gem5, showing significant speedups closing the simulation gap, while retaining the functional correctness and being flexible enough to model any kind of system.

B. Assignment of Objects to Event Queues

In gem5, the individual simulated hardware modules are referred to as simulation objects. The model for a CPU core, a memory controller, or an attached UART device are examples of simulation objects. A system is built by instantiating objects hierarchically, replicating the structure of the target hardware platform. Since simulation objects always refer to an event queue, finding an assignment that minimizes causality errors is one of the key challenges. We empirically observed that exploiting the inherent concurrency of multi-core architectures yields the best results. To achieve this, we map simulation objects to a specific event queue according to the following rules:

- 1) Each CPU subsystem has a dedicated event queue.
- 2) All other objects are assigned to the default q_0 .

The event queue assignment occurs during the initialization phase. One simulation thread is created for each event queue. Consequently, the total number of simulation threads is the

```

1 BaseCache::recvAtomic(...) {
2   port.lock();
3   if (xBarNeeded()) {
4     port.unlock(); // Prevent deadlocks from snoops.
5     port.lock_peer(); // Take ownership of XBar.
6   }
7   ... // Untouched gem5 cache code.
8   if (xBarNeeded()) {
9     port.unlock_peer();
10  } else {
11    port.unlock();
12  }
13 }
14
15 CoherentXBar::forwardAtomic(...) {
16   ...
17   p.lock_peer(); // Locks snoop target caches.
18   Tick latency = p->sendAtomicSnoop(pkt);
19   p.unlock_peer();
20   ...
21 }

```

Listing 1. Simplified C++ code of the thread-safe cache and crossbar model.

number of cores plus one. Fig. 4 shows a simplified block diagram of a 2-core platform and the resulting event queue assignment. The block *Real View Platform Devices* comprises all peripheral devices, such as the generic interrupt controller, system timers, real-time clock, and IO interfaces. It can be seen that all objects of the CPU subsystem (core, TLBs, and private L1 caches) are assigned to the same queue. This assignment allows each CPU core to advance in the simulation using a local time within the quantum borders. Memory accesses that can be satisfied by the private L1 caches are handled by a single simulation thread. All other modules, including the shared L2 cache, DDR memory controller, and peripherals, are put into a separate event queue.

C. Thread-Safe Memory System

So far, this work has focused on the theoretical aspects of PDES. However, a significant part of the work on *par-gem5* was spent on the practical problem of thread safety. Since *gem5*'s software stack was designed for sequential execution, function calls or data accesses between objects assigned to different event queues pose the risk of race conditions.

In our chosen assignment of event queues, this particularly affects the communication from L1 caches to the crossbar (see Fig. 4). We performed a flow analysis of transactions, paying special attention to those moving between two or more event queue domains. Based on this, we extended the *gem5* port class with a new locking functionality. Ports, which *gem5* uses to transfer transactions between modules, now additionally include the functions `lock()`, `unlock()` and `unlock_peer()`, `lock_peer()`. These can be used to unlock/lock the own or the peer's recursive mutex, respectively.

As presented in Listing 1 the new port class is used to guarantee that only one cache at a time is accessing the crossbar. Before a cache starts to process a CPU's request, it has to contend its port mutex (line 2). Once the mutex is held, it is determined whether the request requires access to the crossbar (line 3). Access is required for any transaction that cannot be served locally and requires communication to other caches or memories. For instance, read misses, write

misses, cache maintenance operations, or load-linked/store-conditionals. If access is required, the cache mutex is released to prevent deadlocks from incoming snoops, and the crossbar mutex is contended using the `port.lock_peer()` function (lines 4-5). Next, the default *gem5* cache mechanisms are executed (line 7), which might include calls to the crossbar. Since the crossbar communicates with other caches via snoop request, an additional lock (lines 17-19) ensures that the access does not lead to race conditions. Note that a cache snoop can lead to writebacks that need access to the crossbar in return. To avoid deadlocking by multiple crossbar accesses, the ports use recursive mutexes, allowing multiple accesses by the same thread. Finally, any held mutex is released (lines 8-12).

To mitigate deadlocks with multi-hop interconnects, all crossbars share the same static mutex. This approach still leaves room for optimization since the locality of the data in downstream caches (L2, L3, etc.) is not exploited optimally. Leveraging this potential requires extensive changes to the memory system that are left for future work.

D. Temporal Error Estimation

One of the biggest challenges of the proposed approach is determining a quantum that yields an acceptable compromise between accuracy and speedup. From a speedup perspective, the largest possible quantum should be selected, since this reduces the number of synchronization points and enables better load balancing. For high accuracy, however, the quantum should be as small as possible due to the lower probability of temporal errors. In addition, determining the influence of the quantum in the first place is not trivial. While the speedup can be roughly estimated from the required wall-clock time, the impact on accuracy requires a sequential simulation as a reference point. Executing a sequential simulation annihilates the speed-up benefits of *par-gem5*.

For this reason, we have extended *par-gem5* with a novel temporal error estimation. At the end of each parallel simulation, an analytical worst-case estimation of the temporal error is output as a simulation statistic. This estimation helps the user to assess the accuracy of the simulation without the need for a reference simulation.

The idea of our error estimation is to accumulate timing errors of postponed events. For this purpose, *par-gem5* records for each quantum i , which postponed event had the largest shift t_{i,max_pp} in time caused by the postpone-mechanism (see Fig. 2 b). It is assumed the postponed event was on the program's critical path, prolonging the simulation by t_{i,max_pp} . By accumulating the prolongation times t_{i,max_pp} over all quanta, the following formula describes a worst-case temporal error estimation:

$$e_{rel,t} = \frac{t_{sim,meas}}{t_{sim,meas} - \sum_{i=0}^Q t_{i,max_pp}} - 1 = \frac{t_{sim,meas}}{t_{sim,est}} - 1 \quad (5)$$

With Q being the number of simulated quanta and $t_{sim,meas}$ being the measured simulation time. We validate the formula by precisely detecting inaccurate simulations as shown in the next section.

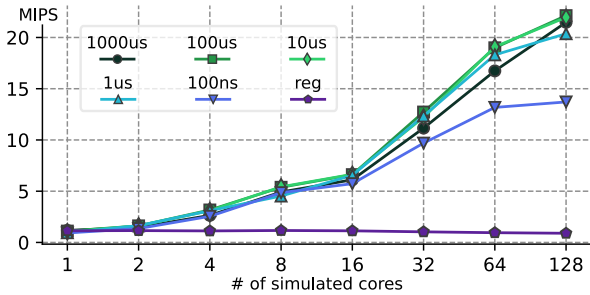


Fig. 5. Accumulated MIPS when executing bare-metal bubble sort with *par-gem5* using various quantum sizes.

IV. RESULTS AND DISCUSSION

To evaluate the accuracy and performance of *par-gem5*, we ran a set of benchmarks using the multicore system configuration shown in Table I. The assignment of simulation objects to event queues was done following the strategy explained in Section III-B. As a host system, we used an *AMD Ryzen 3990x* (64 cores, 128 threads) with 128GiB of 3200MHz DDR4-DRAM running Ubuntu Linux 20.04.

A. Speedup and Accuracy of Bare-Metal Bubble Sort

To perform an initial examination of our parallel simulation approach, we created a simple multi-core bare-metal bubble sort benchmark. It comprises N threads, with the number of available CPU cores N , whereby each thread independently sorts its own array of integers. We designed this benchmark to attain a near best-case simulation throughput, meaning that thread synchronizations and accesses to shared memory are reduced to a minimum.

This simple program was run multiple times, changing the number of simulated cores and the simulation quantum. The throughput in MIPS achieved in this experiment is shown in Fig. 5. As it can be seen, the accumulated throughput grows with the number of simulated cores in parallel mode, while it stagnates in the regular single-threaded simulation. A maximum throughput of 22.1 MIPS is attained with the 128-cores configuration, representing a speedup factor of $24.7\times$. The achievable speedup increases with the size of the quantum and saturates at a certain point between $10\mu\text{s}$ and $1000\mu\text{s}$. This observation is consistent with other publications [15], [16] and can be attributed to the lower number of synchronizations and better load balancing.

Due to the possible causality errors of our approach, the accuracy of the simulation must be evaluated as well. Fig. 6 shows a comparison of the simulated time for the individual settings. It can be seen that the inaccuracy grows with increasing quantum and number of cores. Also shown in Fig. 6 are the results of the temporal error estimation. A large discrepancy between measured value and estimated value $t_{sim,est}$ indicates a larger inaccuracy of the simulation. Any greater inaccuracy is precisely detected by the temporal estimation (see 64 cores with $1000\mu\text{s}$ quantum).

B. Speedup and Accuracy of NAS Parallel Benchmark

To evaluate *par-gem5* with industry-standard multi-core software workloads, we ran the NAS Parallel Benchmarks

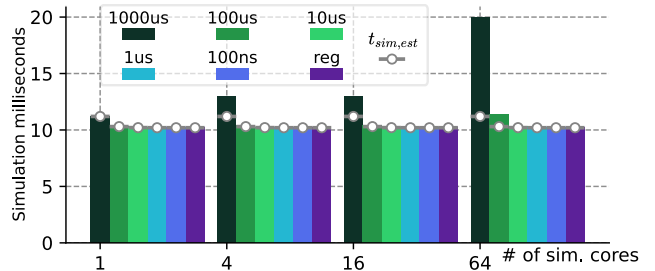


Fig. 6. Accuracy of simulated seconds when executing bare-metal bubble sort with *par-gem5* using various quantum sizes.

(NPB) [1] on Ubuntu Linux 14.04. Specifically, we ran the original eight benchmarks specified in NPB 1 (IS, EP, CG, MG, FT, BT, SP, LU). We used the OpenMP implementation from NPB version 3.4.2 and the Wide (W) data size configuration. While the different test programs of the NPB are all designed for multi-core platforms, they do not scale homogeneously, as they vary greatly in terms of memory access patterns, type of operation, and frequency of synchronization points. A detailed analysis of the scalability of each benchmark is presented in [21].

For each of the benchmarks, we conduct the same experiment as described in the previous section, incrementing the number of simulated cores. In this case, we fix the quantum to $1\mu\text{s}$ due to the length of the simulations. With the default single-thread mode, a throughput between 0.57 and 0.96 MIPS is recorded, while in parallel mode, this value reaches up to 7.37 MIPS. We compute the speedup factor for each test and present the results in Fig. 7. A maximum speedup of $12.13\times$ is observed for the EP benchmark in the 64-core configuration. In addition to possible constraints from the host platform hardware, the potential speedup of our parallelization approach is also affected by the simulated application. Software workloads with a high parallelization potential and a low core-to-core communication achieve the best throughput in our simulator. This is the case of EP and BT, which show significant speedups when running both natively [21] and on *par-gem5*. Nevertheless, it can be concluded that all benchmarks in all configurations benefited from parallelization, consistently achieving speedups greater than 1. Each NPB test program prints a report, including several values, such as the total execution time and the measured performance of the system. It can be regarded as a macro statistic that sums up all performance influencing micro parameters, like cache hit rate or memory bandwidth. The deviation of the benchmark report between sequential and parallel simulation is always below 1.53%.

Besides the aforementioned macro statistics, other statistics, e.g., on the performance of the storage system, may also be of

TABLE I
 CONFIGURATION OF THE TARGET SYSTEM.

CPU	ARM64, AtomicSimpleCPU @ 2GHz
# cores	{1, 2, 4, 8, 16, 32, 64, 128}
Caches	64kiB L1-D, 32kiB L1-I, 2MiB L2 shared
Main Memory	DDR3 RAM @ 1600MHz
Periph. Sub-system	Real View Virtual Express V1
OS	{Bare-metal, Ubuntu 14.04}

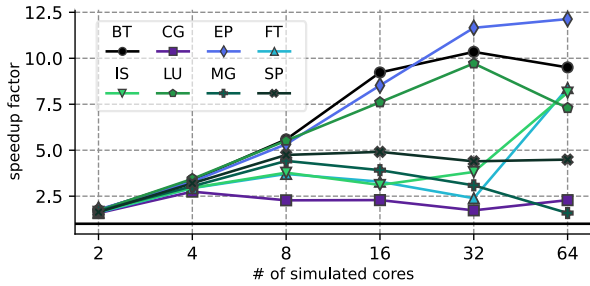


Fig. 7. Speedup attained with *par-gem5* for each NPB test program using a $1\mu\text{s}$ quantum. The black baseline represents a speedup of 1.

interest. Table II compares the L1 cache miss rates for regular *gem5* and *par-gem5* when simulating a 4-core system. Here, the inaccuracy of the parallel simulation never exceeds 5.7%. Also other parameters, such as the main memory bandwidth or L2-cache hit rate, exhibit accuracies of $> 99\%$. The accuracy of the temporal error estimation is exemplarily shown for the IS benchmark in Fig. 8. Here, larger inaccuracies could reliably be detected (see 4 and 16 cores for $100\mu\text{s}$ quantum) as well.

In addition to pure performance measurement, the NPB suite checks the results for correctness. There were no irregularities in any of the benchmarks, which underpins the functional correctness of *par-gem5*.

C. Other Benchmarks

In addition to the aforementioned benchmarks, experiments with other benchmarks like PARSEC, SPEC2017, and STREAM were conducted. Again, a similar picture emerges: arithmetic-intensive applications achieve high speedups ($18\times$, PARSEC blackscholes @ 128 cores), while benchmarks with a high load on shared resources, such as STREAM, show low to no speedups.

V. CONCLUSION & OUTLOOK

In this paper, we showed how *gem5* can be parallelized using an synchronous, conservative PDES. Our work achieves a $24.7\times$ speedup for a best-case synthetic benchmark, while retaining a $> 99\%$ accuracy for most simulation statistics. For the widely used NAS Parallel Benchmark collection, a maximum speedup of $12\times$ has been observed when simulating a 64-core ARM MPSoC on a 64-core/128-threads host machine.

In our implementation, events will be postponed up to a quantum length in the worst case if they impair causality. However, the functionality of the ARM MPSoC and the OS are not affected for relevant quanta. Yet an influence on the accuracy of various simulation parameters can be observed.

Finding a quantum that allows high accuracy with high speedup is one of the main challenges. While a large quantum allows high speedups, a small quantum yields better

TABLE II
 4-CORE $1\mu\text{s}$ QUANTUM SIMULATION OF NPB IS.

CPU	dcache miss rate			icache miss rate		
	$1\mu\text{s}$	reg	error	$1\mu\text{s}$	reg	error
0	0.040828	0.042419	3.8%	0.000506	0.000497	1.7%
1	0.040065	0.040325	0.6%	0.001457	0.001445	0.8%
2	0.040093	0.042527	5.7%	0.000288	0.000278	3.6%
3	0.040960	0.040621	0.8%	0.000460	0.000446	3.1%

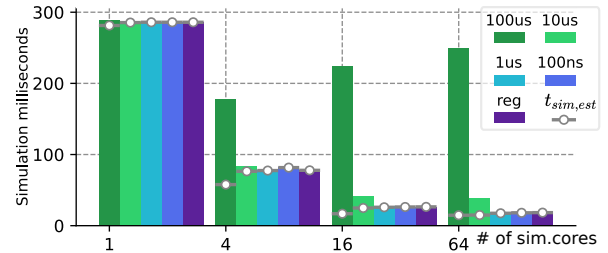


Fig. 8. Accuracy of simulated seconds when executing NPB IS with *par-gem5* using various quantum size.

accuracy in terms of simulation statistics. To find an optimal quantum without time-consuming reference simulations, we developed a temporal error estimation, which reliably detects inaccurate simulations in executed benchmarks.

One of our future goals is to extend our work to *gem5*'s timing mode. Contrary to the atomic, the timing mode uses multiple events to model a transaction, complicating the process of making *gem5*-internal models thread-safe. Nevertheless, the same PDES as presented here can be used for the timing mode.

REFERENCES

- [1] "NAS Parallel Benchmarks," <https://www.nas.nasa.gov/software/npb.html>, accessed: 2023-01-23.
- [2] "Passmark Single Thread Performance," <https://www.cpubenchmark.net/singleThread.html>, accessed: 2023-01-23.
- [3] M. Alian *et al.*, "pd-gem5: Simulation Infrastructure for Parallel/Distributed Computer Systems," *IEEE Comput. Archit. Letters*, 2016.
- [4] R. Bagrodia, "Language support for parallel discrete-event simulations," in *Proceedings of Winter Simulation Conference*, 1994.
- [5] N. Binkert *et al.*, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, 2011.
- [6] G. Busnot *et al.*, "Standard-compliant Parallel SystemC simulation of Loosely-Timed Transaction Level Models," in *ASP-DAC*, 2020.
- [7] K. Chandy *et al.*, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, 1979.
- [8] B. Chopard *et al.*, "A Conservative Approach to SystemC Parallelization," in *Computational Science - ICCS*, 2006.
- [9] M.-K. Chung *et al.*, "SimParallel: A high performance parallel SystemC simulator using hierarchical multi-threading," in *ISCAS*, 2014.
- [10] R. Dömer *et al.*, "Parallel discrete event simulation of Transaction Level Models," in *ASP-DAC*, 2012.
- [11] R. Fujimoto, "Parallel Discrete Event Simulation," *Commun. ACM*, 1990.
- [12] R. Jagtap *et al.*, "Exploring system performance using elastic traces: Fast, accurate and portable," in *SAMOS*, 2016.
- [13] D. R. Jefferson, "Virtual Time," *ACM Trans. Program. Lang. Syst.*, 1985.
- [14] M. Jung *et al.*, "Speculative Temporal Decoupling Using fork(), in *DATE*, 2019.
- [15] L. Jünger *et al.*, "Optimizing Temporal Decoupling using Event Relevance," in *ASP-DAC*, 2021.
- [16] A. Mohammad *et al.*, "dist-gem5: Distributed simulation of computer clusters," in *IEEE ISPASS*, 2017.
- [17] E. Perelman *et al.*, "Using SimPoint for Accurate and Efficient Simulation," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, 2003.
- [18] A. Qumranet *et al.*, "KVM: The Linux virtual machine monitor," *Proceedings Linux Symposium*, vol. 15, 01 2007.
- [19] D. Sanchez *et al.*, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," *SIGARCH Comput. Archit. News*, 2013.
- [20] C. Schumacher *et al.*, "parSC: Synchronous parallel SystemC simulation on multi-core host architectures," in *CODES+ISSS*, 2010.
- [21] S. Seo *et al.*, "Performance characterization of the NAS Parallel Benchmarks in OpenCL," in *IISWC*, 2011.
- [22] J. Wang *et al.*, "Manifold: A parallel simulation framework for multicore systems," in *IEEE ISPASS*, 2014.
- [23] J. H. Weinstock *et al.*, "SystemC-link: Parallel SystemC simulation using time-decoupled segments," in *DATE*, 2016.