ElasticSimMATE: a Fast and Accurate gem5 Trace-Driven Simulator for Multicore Systems

Alejandro Nocua, Florent Bruguier, Gilles Sassatelli, Abdoulaye Gamatié

▶ To cite this version:

Alejandro Nocua, Florent Bruguier, Gilles Sassatelli, Abdoulaye Gamatié. ElasticSimMATE: a Fast and Accurate gem5 Trace-Driven Simulator for Multicore Systems. ReCoSoC: Reconfigurable Communication-centric Systems-on-Chip, Jul 2017, Madrid, Spain. 10.1109/ReCoSoC.2017.8016146 . hal-01723789

HAL Id: hal-01723789

https://hal.science/hal-01723789

Submitted on 23 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ElasticSimMATE: a Fast and Accurate gem5 Trace-Driven Simulator for Multicore Systems

Alejandro Nocua, Florent Bruguier, Gilles Sassatelli, Abdoulaye Gamatie
LIRMM - CNRS / University of Montpellier
Montpellier-France
Email:<name.lastname@lirmm.fr>

***Abstract*—Multicore system analysis requires efficient solutions for architectural parameter and scalability exploration. Long simulation time is the main drawback of current simulation approaches. In order to reduce the simulation time while keeping the accuracy levels, trace-driven simulation approaches have been developed. However, existing approaches do not allow multicore exploration or do not capture the behavior of multi-threaded programs. Based on the gem5 simulator, we developed a novel synchronization mechanism for multicore analysis based on the trace collection of synchronization events, instruction and dependencies. It allows efficient architectural parameter and scalability exploration with acceptable simulation speed and accuracy.**

***Index Terms*—Multi-threaded programs, Parameter Exploration, Trace-Driven Simulation, Scalability Exploration, Synchronization Mechanism.**

I. INTRODUCTION

Simulation is widely used in system design for evaluating different design options. Depending on the abstraction level considered for simulating a given system configuration, there is a tradeoff between the obtained precision and speed. Generally, simulating a detailed system model provides accurate evaluation results at the price of potentially high simulation time. On the other hand, less detailed or more abstract system representations usually provide less accurate evaluation results, but in a fast and costless manner. In practice, such representations are defined such that they only capture system features that are mostly relevant to the problem addressed by a designer. Trace-driven simulation is a popular technique that enables fast design evaluation by considering system models where inputs are derived from a reference system execution, referred to as traces.

Considering multicore architectures, a typical trace-driven simulation relies on collecting reference traces in a trace-collection phase based on an accurate reference architecture with a low core count. Because traces are collected on such an accurate reference architecture, most relevant phenomena are captured such as CPU microarchitecture events, memory transaction events, event jitter due to the underlying operating system execution, etc. The resulting traces can be then reused in a number of target trace-driven simulations in which CPU cores are replaced with trace injectors as an abstraction, thereby enabling to refocus the simulation effort on other performance-critical system sub-components such as caches, communication architecture and memory sub-system.

Elastic Traces (ET) framework [1] is an extension of the gem5 environment [2] that allows to collect and playback micro-architecture dependency and timing annotated traces attached to the Out-of-Order (OoO) CPU model. The focus of this tool is to achieve memory performance exploration in a fast and accurate way compared to the slow gem5 OoO CPU model. It relies on extensive modifications of the OoO CPU model. Probe points have been added in the pipeline stages. Each instruction is monitored and a data dependency graph is created by recording data Read-After-Write dependencies and order dependencies between loads and stores [3]. Two different traces are produced: one for instruction fetch requests and one for data memory requests. To ease the capture of a large amount of trace data, the Google protobuf format is used [4]. While Elastic Traces simulation provides an attractive design evaluation support, it does not enable to address multicore architecture.

In this paper, we present the ElasticSimMATE (ESM) tool, which extends Elastic Traces with inter-core synchronization features, in order to make possible multicore architectures simulation. ElasticSimMATE enables to conduct explorations belonging to two categories as follows:

- fast system parameter exploration: because trace-driven simulation is fast, the influence of various parameters such as cache sizes, coherency policy, memory speed can be rapidly assessed through replaying the same traces on different system configurations.
- system scalability: this approach relies on replicating traces for emulating more cores, thereby analyzing how performance scales when increasing the number of cores. This approach requires to record and carefully handle the synchronization semantics in the trace-replay phase so as to carefully account for the execution semantics on such an architecture.

This paper is organized as follows. Section II discusses related work. In Section III, describes the main concpets of the ElasticSimMATE approach. In Section IV we present the experimental results on selected applications. Finally, we conclude this paper in Section V.

978-1-5386-3344-1/17/$31.00 ©2017 European Union

II. Related Work

Simulation speed and accuracy are two crucial considerations for architectural and scalability analysis exploration. In the sequel, we review some relevant simulation approaches.

A. Traditional simulators

Existing techniques can be classified into two fundamental families [5].

The first family focuses on the increasing of computational power, e.g., increasing the number of simulated events per second. Usually it is achieved by running the simulation distributed across multiple host machines [6], [7]. Distributed simulation is a known difficult technique as one must carefully deal with simulation partitioning and event synchronizations among available hosts. Another popular approach for accelerating simulation is just-in-time (JIT) dynamic binary translation, e.g., OVP [8] and QEMU [9]. JIT-based simulators are instrumented with timing models so that basic architecture block models and their inter-operations can be driven according to the annotated timing information. The second family of techniques includes approaches reducing the number of simulation events required for accurate results. It concentrates on optimizing component descriptions (e.g. CPUs, interconnect infrastructure) following the transaction-level modeling strategy [10] or by using trace-driven simulation [11]. The above approaches lack expressive modeling supports such as those related to cache hierarchies, coherency protocols and communication architecture which are of bold importance. Such simulators can achieve speeds close to thousands MIPS at the cost of a limited accuracy. They often focus on functional validation rather than architectural exploration.

In order to allow architectural parameter and scalability exploration with acceptable accuracy, trace-driven simulation is an alternative approach. In [1], authors proposed a collection and replay mechanism defined in gem5 simulation framework. However, its application is restricted to mono-core execution and no synchronization mechanism is presented. On the other hand, a synchronization mechanism for multi-threaded application is presented in [12]. In a similar way, authors in [13] proposed a collection mechanism for parallel events and a playback methodology to allow architectural exploration.

B. Trace-driven extensions of gem5

In Elastic Traces [1], the replay phase allows to play back traces for architecture exploration. Instruction traces and data dependency traces are injected on the I-side and D-side generators respectively (see Fig. 1). This trace replayer supports only single-threaded applications which is one main limitation. Elastic Traces demonstrates a speedup of 6-8x compared to a reference Out-of-Order core and is accurate, with less than 10% error versus the reference [1].

SimMATE [12] is a trace-driven simulator that operates on top of gem5 and is devoted to the exploration of in-order manycore architectures. Traces collected on a reference architecture in Full-System mode are made of outgoing memory transactions collected at Level-1 caches, i.e. cache misses. In

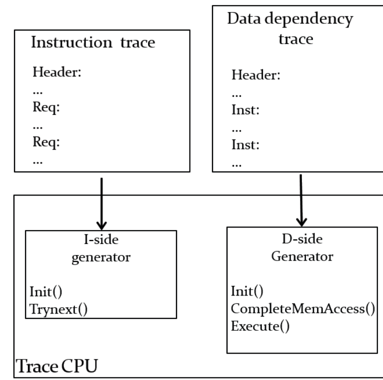


Figure 1: Elastic Traces Replay Mechanism

trace replay phase CPU cores are replaced with Trace Injectors (TIs) that are connected to the interconnect subsystem cache and initiate the transactions recorded in the trace database. The interconnect and memory subsystems remain fully simulated so as to account for the latencies incurred by the traffic in the given simulated architecture configuration. SimMATE takes into account inter-core synchronizations: additional information such as barriers are recorded in the traces through a redefinition of the used shared-memory API functions (e.g., Pthreads) in trace collection. An arbiter takes care of locking/unlocking trace injectors whenever necessary, according to the synchronization constructs recognized in the traces.

ElasticSimMATE leverages the benefits of both Elastic Traces and SimMATE trace-driven approaches in gem5 for multicore architectures: Elastic Traces provides an accurate modeling of CPU core instruction pipeline for Out-of-Order cores whereas SimMATE brings a solution that makes it possible to account for the inter-core execution dependencies. It offers a single simulation solution of great interest for a fast and accurate exploration of next-generation multicore systems.

III. ElasticSimMATE FrameWork

Figure 2 conceptually depicts the ElasticSimMATE workflow, from the OpenMP application source files to the replay on different target architecture configurations. The red-colored $\#pragma\ omp$ statements listed in the source are read by the pre-processor in the usual case and result in the insertion of calls to the OpenMP runtime. In ElasticSimMATE, these calls further require to call a tracing function that will make it possible to record the start and end of a parallel region in the trace. The resulting binaries are then executed in a Full-System (FS) simulation (Trace Collection phase) so as to generate the execution traces. Three traces are created: instruction and data dependencies trace files (as per the Elastic Traces approach) and an additional trace file that embeds synchronization information. These three trace files are used in the trace replay phase devoted to the architecture exploration.

ElasticSimMATE supports the following gem5 features: ARMv7 and ARMv8 ISAs, O3 CPU model and SimpleMemory model (required by Elastic Traces).

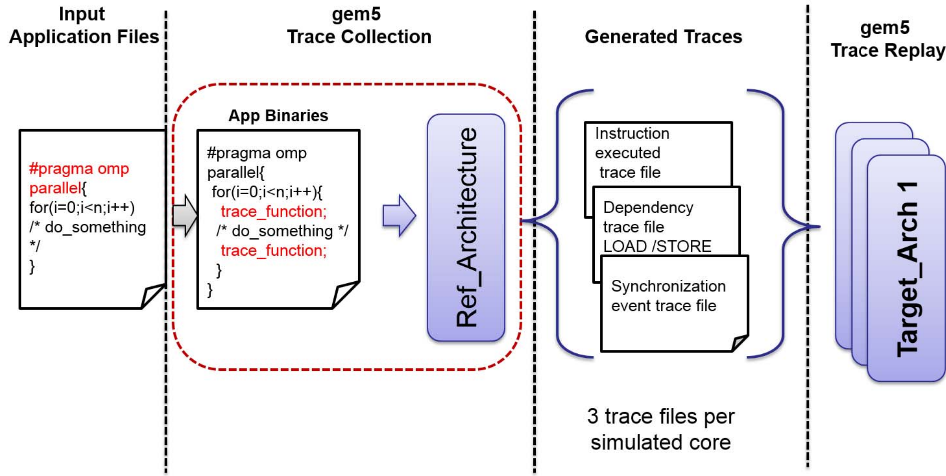


Figure 2: ElasticSimMATE workflow

ElasticSimMATE is compatible with both OpenMP 3.0 and POSIX thread APIs. Focus is put on OpenMP 3 in this document. Recording synchronization traces requires using a specific gem5 pseudo-instruction created for this purpose: $m5_trace()$. This pseudo-instruction requires to be inserted either manually or automatically by means of using an instrumented runtime system.

A. *ElasticSimMATE workflow*

ElasticSimMATE simulation is composed of four steps: code annotation, checkpoint creation, trace collection and replay, introduced in the following subsections.

1) Code Annotation: First of all, it is necessary to annotate the code with $m5_trace()$ pseudo-instruction before compilation either manually or automatically. This pseudo-instruction takes care of recording synchronization information in the trace, such as the program counter and the number of instructions and dependencies. It must be inserted at the beginning and at the end of each event.

While manual insertion of the tracing calls are possible at source code level this is rather cumbersome and therefore two other options have been considered:

- **Source to source approach.** This approach relies on parsing application input files and automatically inserting the $m5_trace$ calls wherever needed. We have verified that the proposed approach works for C/C++ but would require to be ported for other languages.
- **Automatic tracing call insertion.** This solution relies on modifying the API runtime so that whenever a parallel code region is detected a tracing call gets automatically inserted right at the precise instant where parallel execution starts. It is regarded as the most suitable solution as it is accurate and only requires to work with a specific version of the runtime system.

Automatic tracing call insertion is the option selected at this stage and is available for OpenMP using the Nanos++ runtime system / Mercurium compiler [14]. Mercurium source to source compiler automatically inserts Nanos++ function calls whenever a specific runtime handling is required. Figure 3 illustrates the process and the function calls inserted before being passed on to gcc: $nanos_create_team()$ and $nanos_end_team()$ alongside the nested $nanos_enter_team()$ and $nanos_leave_team()$ are displayed at respectively the beginning and end of the parallel region. The approach here relies on modifying those functions so as these carry out the required work for tracing. Table I describes the currently supported set of OpenMP constructs.

Table I: OpenMP constructs supported in Nanos++ tracing tool

OpenMP Event	Position	Nanos++ Call
Parallel / Parallel for	Beginning	nanos_enter_team()
	End	nanos_leave_team()
Critical	Beginning	nanos_set_lock()
	End	nanos_unset_lock()
Barrier	Beginning	nanos_omp_barrier()

2) Checkpoint creation:

a) Principle: once the desired architecture parameters are decided for the trace capture, the simulation is launched in order to create a checkpoint after system boot and before application execution. It makes possible to obtain clean traces without OS boot phase information. This checkpoint resets all statistics in gem5 and allows to resume simulation from that point.

Checkpoints are acquired by setting cpu-type to $arm_detailed$. While this model is slow because of its accuracy, this only applies to the trace capture which is to be performed only once per application.

3) Traces collection:

a) Principle: In this phase, ElasticSimMATE restores the system state from the checkpoint and begins trace collection. Three types of events are considered during parallel sections: instruction executed, dependencies (load/store), and synchronization events. Instruction and data dependency traces

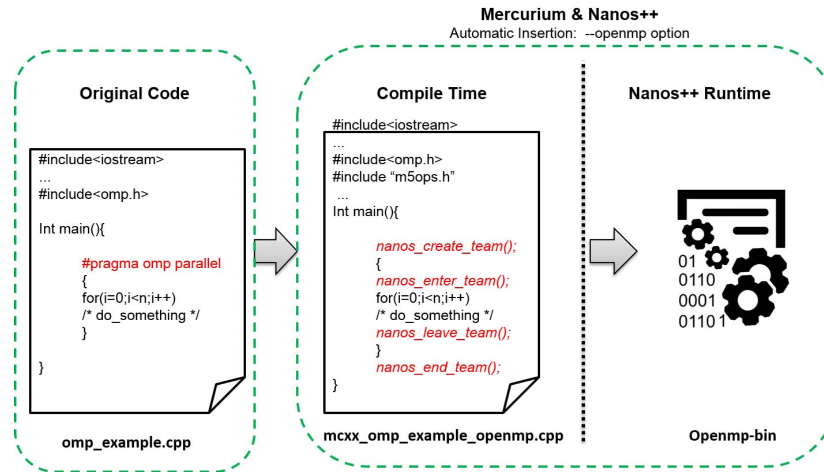


Figure 3: Function insertion while using Nanos++ and mercurium

are captured thanks to an augmented TraceCPU model. The following information are captured into the synchronization trace for each CPU and each event:

- **Tick:** the tick count of a CPU at the entrance in the parallel region.
- **Program Counter:** the program counter at the beginning of a parallel region ; it will be used during the replay phase for identifying parallel sections.
- **Thread ID:** the thread ID assigned by the scheduler.
- **Event Type:** an enumerate type that encodes events corresponding to *parallel for*, *critical* and *barrier*.
- **Number of instructions:** the number of instructions executed by a CPU during the recorded event.
- **Number of data accesses:** the number of data accesses performed during the recorded event.

It has to be noted that for each thread under analysis the Tick and PC information will be the same since all threads are created at the same time. It means that the information on the synchronization traces is the same. In the case of the dependency trace, the load and store information are only collected between the CPU and the L1 caches. At the end of the trace collection phase, three Google Protobuf files are obtained per simulated CPU core with the required data for the replay phase:

- Instruction Executed Trace File.
- Dependency Trace File (LOAD/STORE).
- Synchronization Event Trace File.

4) Traces replay:

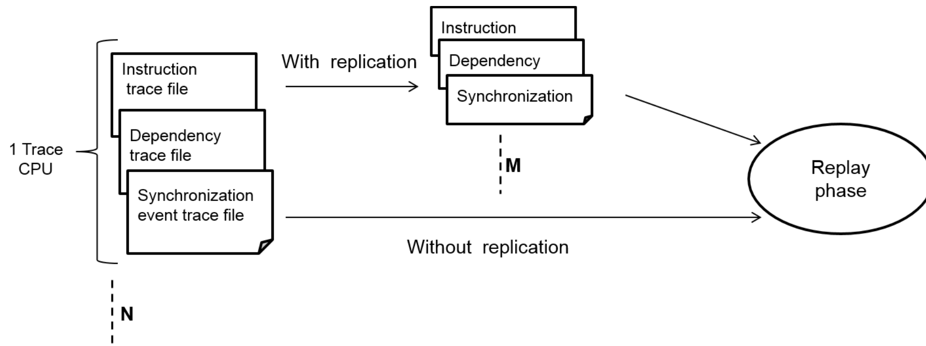
a) Principle: As illustrated in Figure 4a, collected traces can be replayed in target architecture configurations in different ways. Letting N be the number of cores used in trace collection and M in trace replay, two main purposes are considered as follows:

- **Parameters exploration:** we perform an architectural exploration in which we replay the exact number of simulated cores, i.e., N = M. The objective is to analyze the influence of a number of architectural parameters such as cache sizes, interconnect bandwidth or memory latency.
- **Replication:** we perform a scalability analysis, by targeting a higher core count compared to that of the initial system from which given traces are captured, i.e., $M > N$. This is achieved by simulating more trace injectors. Noted that the replication mechanism allows us to perform weak-scaling analysis as the problem size is increased by the ration of $\frac{M}{N}$. In addition, current implementation is performed with no address offsetting mechanism. This means that most of the resources are shared among cores.

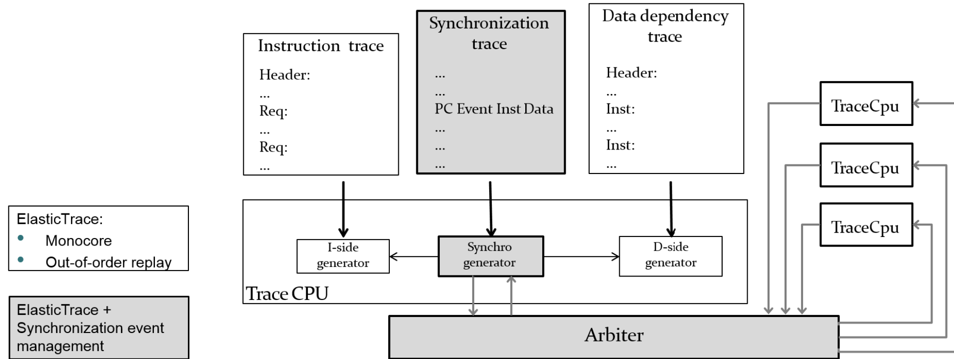
b) Implementation: Figure 4b shows the interplay of the principal objects involved during the replay phase in ElasticSimMATE. A number of TraceCPU objects operate and check if LOAD/STORE dependencies are met on the basis of the traces they access, as per the Elastic Traces base model. These further read out the synchronization trace and keep track of the parallel regions.

The actual behaviour when entering a parallel region is as follows:

- **Init**: whenever one such region is detected on a TraceCPU, a notification is sent to the arbiter so as to properly handle the synchronizations.
- **Processing**: TraceCPU model continues the execution. The length of a region is encoded in the trace in form of a number of instructions to be executed alongside a number of data dependencies to be met. Local counters keep track of both instruction and dependency counts.
- **Stall**: when counters reach the two values (number of executed instructions and number of executed dependencies) listed in the synchronization trace record TraceCPU stalls (locked state) and simultaneously notifies the arbiter it has reached a barrier.
- **End**: when the arbiter has received lock notifications from all TraceCPU objects it unlocks them all and execution is resumed.



(a) Trace replay: N represents the number of cores for the collection and M the number of cores for the replay



(b) Replay overview including modified TraceCPU and arbiter

Figure 4: Trace replay approach

IV. Experimental Results

ElasticSimMATE is evaluated and compared against both Elastic Traces and gem5 Full-System simulation, the latter being the reference. As figures of merits we analyze execution time, simulation time and the simulation accuracy with respect to both the reference gem5 Full-System simulation and Elastic Traces.

Further results are reported concerning scalability analysis. They rely on the "trace replication" approach (see Section III-A4), which is based on trace reuse for emulating the presence of more cores in the considered targeted system. As traces are replicated on a per-core basis, these results account for *weak-scaling* analysis. Finally, parameters explorations are performed considering different L2 cache sizes.

A. Experimental Setup

1) Baseline system: As reference model we consider an Out-of-Order (or O3) CPU model in gem5 that represents an ARMv7 architecture. Figure 5 depicts a four-core architecture along with the cache hierarchy, an interconnect and a main memory. For the trace collection, we set up the same configuration from 1 to 4 cores while omitting the L2 cache in a similar way as Elastic Traces approach.

Unless otherwise stated, all experiments are done using the parameters shown in Table II. Each core has its own L1 Data and Instruction caches. The unique L2 cache is shared between all cores through a bus. We run FS simulation which is instrumented for capturing traces. All experiments are conducted on a 56-core server (Xeon E5 clocked at 2.6GHz).

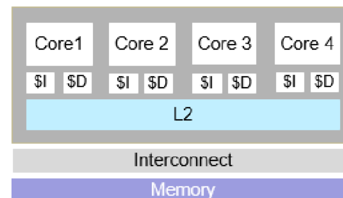


Figure 5: Reference system with 4 cores

2) Benchmarking: We perform benchmarking of ElasticSimMATE in three different modes so as to analyze both intrinsic accuracy / simulation speed and usability in scalability analysis. The following sections therefore display results that correspond to three modes:

- **Base replay**: we use a matrix multiplication workload with matrix input sizes ranging from 16x16 to 128x128 such that simulation complexity can be easily scaled. For each input size experiments are performed on 1, 2 and 4 cores. Full System (FS) simulation in gem5 is performed as a reference scenario for both accuracy and simulation speedup evaluation.
- **Trace replication for scalability analysis**: these results are gathered on the basis of a 1-core trace that is reused for every TraceCPU of the target simulation. Per-core

Table II: Reference baseline system

	Parameter	Value
CPU	Model	O3
I Cache	Size	32kB
	Associativity	2-way
	Cycle Hit Latency	1
D Cache	Size	32kB
	Associativity	2-way
	Cycle Hit Latency	1
L2 Cache	Size	1MB
	Associativity	16-way
	Cycle Hit Latency	12
Main Memory	Model	SimpleMemory
	latency	30ns

workload therefore remains unchanged, as well as synchronization semantics: a synthetic synchronization barrier is emulated by the arbiter that ensures all TraceCPU objects reach the end of any parallel region before resuming the execution of the subsequent statements in a code. These experiments are conducted on the matrix multiplication workload (up to 512x512 matrix sizes) and 3 compute-intensive applications defined in Rodinia [15] and Parsec [16] benchmark suites respectively: *Hotspot, K-means* and *Blackscholes*.

- **Architectural parameter exploration**: we use K-means application with 1, 2 and 4 cores for collection. Then, we replay varying the L2 cache size. FS simulations are also run to serve as references.

B. Accuracy and Speedup Evaluation

In this section we evaluate how correlated are the results obtained with ESM in relation to ET and FS. In all cases, the deviation percentage is calculated based on FS results ($\frac{V_{FS}-V_{ET,ESM}}{V_{FS}}$). Table III shows the execution times reported by the three tools. We observe that ElasticSimMATE preserves Elastic Traces accuracy with negligible deviation in predicted execution time for single core experiments. On the other hand, error tends to decrease on multicore experiments.

Table III: Simulation accuracy for the matrix multiplication: Execution time comparison

	#Core	FS [ms]	ET [ms]	ESM [ms]	FS vs ET [%]	FS vs ESM [%]
16x16	1	115.61	98.55	98.72	14.76	14.61
	2	99.36		102.15		-2.80
	4	105.75		106.79		-0.99
32x32	1	116.83	99.77	99.77	14.60	14.60
	2	100.19		102.82		-2.63
	4	106.25		107.46		-1.14
64x64	1	126.34	109.30	109.31	13.48	13.48
	2	106.84		106.58		0.25
	4	110.43		109.42		0.91
128x128	1	225.39	183.92	183.92	18.40	18.40
	2	159.96		142.41		10.97
	4	132.67		127.47		3.92

2) Speedup Evaluation: Figure 6 shows the simulation speedups achieved by respectively Elastic Traces and ElasticSimMATE compared to gem5 FS simulation. Speedups are in the same order of magnitude for both solutions. Modest speedups of around 3x for small input set sizes find root in the short application execution time for which gem5 spends

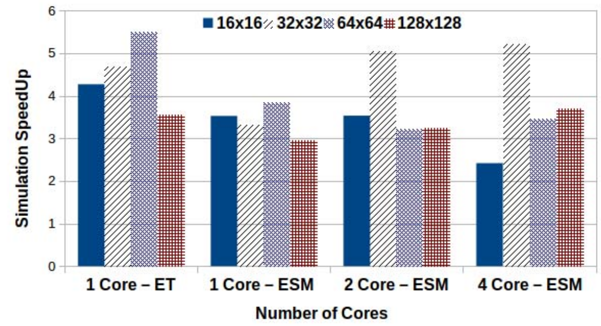
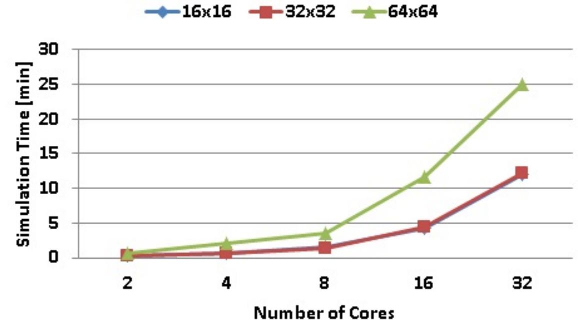


Figure 6: Simulation speedup for matrix multiplication

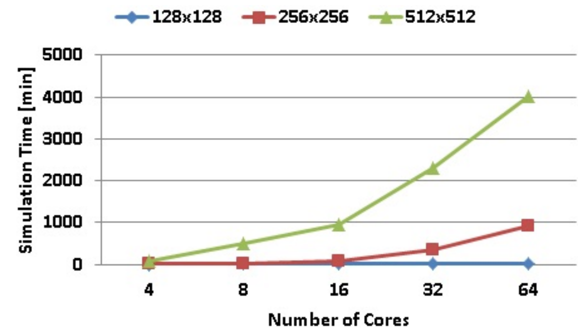
a comparatively significant time in simulation initialization versus simulation run.

C. Trace Replication

Results shown in this section use trace replication only. Though traces can be collected on an arbitrary number of cores (up to 4 in our setup), all figures reported here are made on the basis of 1 core trace collection that is replicated according to the target core count. Similar results were obtained when using two and four cores count. All of the experiments in this section are carried out using only ESM, since FS simulation up to 128 cores would take a prohibitive amount of time.



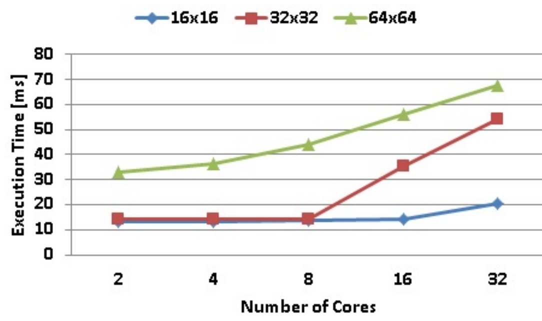
(a) Small Input Sizes



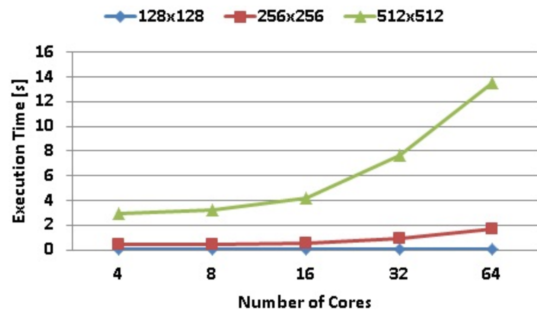
(b) Large Input Sizes

Figure 7: Simulation time for matrix multiplication

Figures 7a and 7b show the simulation time versus core count for the matrix multiplication for 2 sets of input sizes, small (16x16 to 64x64) and large (128x128 to 512x512).

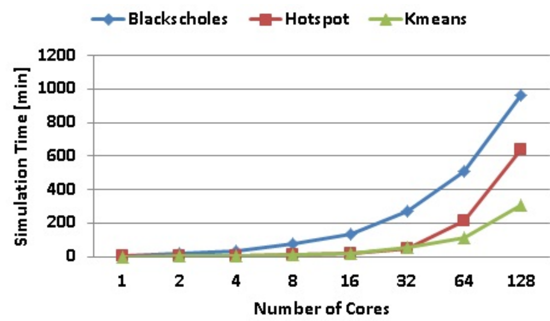


(a) Small Input Sizes

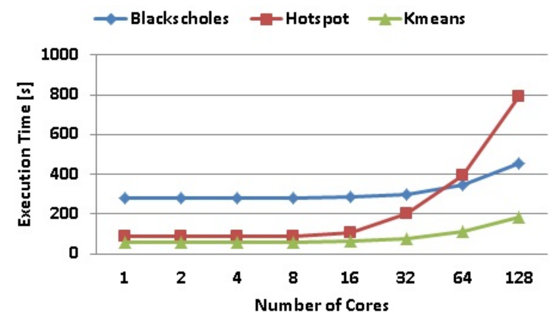


(b) Large Input Sizes

Figure 8: Execution time for matrix multiplication



(a) Simulation Time



(b) Execution Time

Figure 9: Trace replication analysis for selected applications

Simulation times for large input sizes have been experimented for systems comprising up to 64 cores. In the worst case (512x512 matrix sizes, 64 cores) simulation time was about 65 hours which remains tractable for scalability evaluation.

Figures 8a and 8b show the corresponding execution times accounting for weak scaling. The rather early increase in execution time obviously relates to contention in the interconnect / memory subsystem (shared bus in these experiments). Note that trace replication is in the current version made without any address offsetting i.e. all cores issue requests to the same addresses (encoded in the trace) which results in unrealistic data sharing during replay. This is confirmed after analyzing gem5 execution statistics which report well above 80% data sharing in most experiments.

Similar experiments have been carried out on sample applications extracted from Rodinia and Parsec benchmark suites. Blackscholes, Hotspot and K-means have been selected for their different memory access patterns. Figures 9a and 9b give simulation times and execution times for systems comprising up to 128 cores. Better weak-scaling is observed compared to the matrix multiplication. Interconnect saturation occurs from 32 cores for hotspot. Simulation times are in the tens of hours for the chosen applications / input set sizes for 128 core systems which is acceptable.

D. Architectural Parameter Exploration

By using ElasticSimMATE in an architectural parameter exploration mode we vary the L2 cache size and measure the execution time. We compared our results with regard to gem5 Full-System simulation for one, two and four cores. Here we focus on relative accuracy between FS and ESM instead of absolute accuracy. Execution time results are shown in Figure 10. While the observed execution times for ESM and FS differ, they globally follow the same tendency. For instance, given any pair of configurations (i.e., L2 cache size) the relative comparison of their associated execution times is similar for both simulation approaches. In addition, for a given cache configuration the relative comparison of the execution times obtained with different core counts is similar for both simulation approaches. The above observations suggest the soundness of ESM with regard to FS. Since ESM is in average 3x faster than FS, designer can therefore exploit the capabilities of our approach to perform detailed and complex architecture parameter exploration in a fast way. To illustrate this opportunity, let us consider a simple exploration

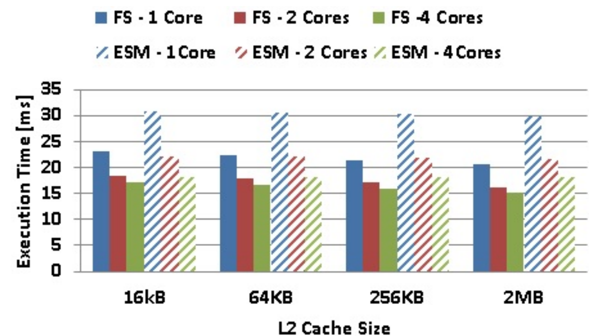


Figure 10: K-means execution time for different L2 cache size

of typical design decisions that can have an impact on system performance. Here, we vary the size of the L2 cache in the memory hierarchy and we analyze the resulting effect on the related performance metrics such as the total cache miss rate and the total cache miss latency. Table IV reports the experimental results for K-means in FS and ESM simulations. In FS changing the L2 cache from 16kB to 1MB represents a reduction in the cache miss rate of 57.7% while ESM shows 60.3% reduction. Furthermore, increasing the L2 cache from 16kB to 2MB reduces the cache misses for about 65.6% with FS and 67.7% with ESM. In this case, a designer would consider 1MB L2 cache as a preferable choice instead of 2MB cache size, as the improvement in the performance is marginal in the latter case. This in turn would reduce the cost in area and power consumption. A similar analysis can be made based on the total cache miss latency.

Table IV: Parameter exploration analysis for k-means.

L2 Cache Size	16kB		1MB		2MB	
Simulation Approach	FS	ESM	FS	ESM	FS	ESM
Cache Miss Rate (%)	75.9	76.7	18.3	16.4	10.3	8.9
Cache Miss Latency (ms)	59.6	59.1	13.6	12.4	7.5	6.8

E. Summary

The displayed results show that overall simulation accuracy remains in the same range compared to that of Elastic Traces for low core counts while a slight error is observed towards higher numbers of cores. This finds roots in the lack of address offsetting when emulating more cores in the target simulation, as well as a coarse grained handling of instructions and data synchronizations. Simulation time scales satisfactorily and most simulations completed in usually hours, occasionally days when selecting large input sets and core counts. Trace collection, even though done once for each application is time-consuming and produces large trace files, in the order of tens of gigabytes for the applications used in these experiments. Synchronization trace account for well below 1% of overall trace files, the rest being related to intrinsic Elastic Traces tracing approach.

V. Conclusion and Future Work

This paper describes a gem5 trace-driven simulation solution. It relies on two former contributions, Elastic Traces and SimMATE. The resulting tool, ElasticSimMATE, preserves the accuracy at the heart of Elastic Traces and makes it possible to conduct a fast architectural parameter exploration. We illustrated the opportunity offered by ESM for fast and sound architecture exploration through the impact analysis of L2 cache size on system performance. In addition, we showed the scalability of ESM based simulations, thanks to an adequate trace replication mechanism. This mechanism relies on reusing traces collected on a reference architecture onto more cores thereby enabling to perform weak scaling experiments (workload/problem size remains same per core). Experimental results confirmed that ESM can simulate up to 128 cores. Furthermore, based on the application complexity, ESM is at least 3x faster than FS simulation.

One important extension to the current work is to enhance the considered trace format in order to enable strong-scaling experiments. Indeed, it will make possible complementary evaluations of multicore architectures such that a given workload could be divided and allocated to available cores, i.e., workload/problem size could be adapted to the number of cores. Beyond the simple architectural exploration reported in this work, we plan to address further design issues, e.g., interconnect topologies and protocols, memory hierarchy, etc. Finally, our tool will be freely-available once we have make the proposed improvements.

VI. Acknowledgement

The research leading to these results has received funding from the H2020 RIA Framework under the Mont-Blanc 3 project: http://www.montblanc-project.eu, grant agreement no 671697.

References

[1] R. Jagtap, S. Diestelhorst, A. Hansson, and M. Jung, “Exploring system performance using elastic traces: Fast, accurate and portable,” in *Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), 2016 International Conference on*. IEEE, 2016, pp. 96–105.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[3] (2017) ElasticTraces. http://gem5.org/TraceCPU.

[4] (2017) Protocol Buffers. https://developers.google.com/protocol-buffers/.

[5] P. E. Heegaard, “Speed-up techniques for simulation,” *Telektronikk*, vol. 91, no. 2, pp. 1–29, 1995.

[6] M. Lis, P. Ren, M. H. Cho, K. S. Shim, C. W. Fletcher, O. Khan, and S. Devadas, “Scalable, accurate multicore simulation in the 1000-core era,” in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 175–185.

[7] M. Alian, D. Kim, and N. S. Kim, “pd-gem5: Simulation infrastructure for parallel/distributed computer systems,” *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 41–44, Jan 2016.

[8] (2017) OVP. open virtual platforms. http://www.ovpworld.org/.

[9] (2017) QEMU. qemu open source processor emulator. http://wiki.qemu-project.org/Main_Page.

[10] L. Cai and D. Gajski, “Transaction level modeling: an overview,” in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2003, pp. 19–24.

[11] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero, “Trace-driven simulation of multithreaded applications,” in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 87–96.

[12] A. Butko, R. Garibotti, L. Ost, V. Lapotre, A. Gamatie, G. Sassatelli, and C. Adeniyi-Jones, “A trace-driven approach for fast and accurate simulation of manycore architectures,” in *The 20th Asia and South Pacific Design Automation Conference*, Jan 2015, pp. 707–712.

[13] S. Nilakantan, K. Sangaiah, A. More, G. Salvadory, B. Taskin, and M. Hempstead, “Synchrotrace: synchronization-aware architecture-agnostic traces for light-weight multicore simulation,” in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2015, pp. 278–287.

[14] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, “Nanos mercurium: a research compiler for openmp,” in *Proceedings of the European Workshop on OpenMP*, vol. 8, 2004, p. 56.

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 2009, pp. 44–54.

[16] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The parsec benchmark suite: Characterization and architectural implications,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.