A gem5 trace-driven simulator for fast architecture exploration of OpenMP workloads

Alejandro Nocua, Florent Bruguier, Gilles Sassatelli, Abdoulaye Gamatié

▶ To cite this version:

Alejandro Nocua, Florent Bruguier, Gilles Sassatelli, Abdoulaye Gamatié. A gem5 trace-driven simulator for fast architecture exploration of OpenMP workloads. Microprocessors and Microsystems: Embedded Hardware Design , 2019, 67, pp.42-55. 10.1016/j.micpro.2019.01.008 . lirmm-02100235

HAL Id: lirmm-02100235

https://hal-lirmm.ccsd.cnrs.fr/lirmm-02100235v1

Submitted on 22 Oct 2021

A gem5 Trace-Driven Simulator for Fast Architecture Exploration of OpenMP Workloads

Alejandro Nocua[a,*], Florent Bruguier[a], Gilles Sassatelli[a], Abdoulaye Gamatié[a]

[a] *LIRMM - CNRS/UM*
161 Rue Ada, Montpellier (34095)

Abstract

Architecture parameter exploration is one of the main analysis that needs to be performed in order to ensure that a multicore system has an optimal set of parameters. The main drawback of current simulation approaches is the long simulation times in order to extract performance metrics while varying a system parameter. Trace-driven simulation approaches allow to abstract selected components of the system under analysis by creating traces during the execution time of an application. This technique reduces the simulation time while keeping the accuracy levels. Even tough trace-driven techniques have proven to be useful, most of them are focused on mono-core systems, and some does not completely capture the behavior of multi-threaded programs. In this regard, we developed a trace-driven simulation approach based on the gem5 framework. This approach is based on a collection phase of the instructions and dependencies of a given application and two extra traces depending on the selected analysis. It allows weak and strong scaling analysis along with the possibility to perform extensive parameter exploration analysis. For weak scaling analysis simulations, we collect synchronization traces for OpenMP applications, and for strong scaling analysis, we collect task traces for OmpSs applications.

Keywords: Multi-threaded programs, OpenMP, Parameter Exploration, Scalability Exploration, Trace-Driven Simulation.

1. Introduction

One of the main objectives when designing a multicore system is to accurately determine how the different metrics are escalated with the increase in the number of cores. For instance, weak-scaling analysis allows designers to analyze applications that are memory-bonded or that made intensively use of system resources. On the other hand, strong-scaling analysis, allows designers to analyze applications that are mainly computed bounded by determining the

*Corresponding author
Email address: `alejandro.nocua@lirmm.fr` (Alejandro Nocua)

optimal number of core that an application will require to run properly without parallelization overheads. In addition to scalability analyses, another critical point is the need for design tools that allows parameter exploration analysis. It is vital to know how many cores an optimal system has but also how its memory hierarchy is been used, along with which are the optimal set of values.

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 trade-off 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 cost-less manner. In practice, such representations are defined such that they only capture system features that are most 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 an accurate reference architecture, most relevant phenomena are captured such as CPU micro-architecture 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 the CPU is replaced with trace injectors as an abstraction, thereby enabling to refocus the simulation effort on other performance-critical systems as caches, communication architecture and memory sub-system.

Elastic Traces (ET) framework [1] is an extension of the gem5 environment [2] that allows collecting and to 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 by adding probe points in the different 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.

We present the complete framework of the ElasticSimMATE (ESM) tool. A trace-driven approach that allows weak and strong scaling analysis along with a fast parameter exploration capability. In this regard, we presented in [5], the initial framework of ESM and experimental results for weak-scaling analysis along with a parameter exploration analysis. In this case, we extended these two features to perform strong-scaling analysis, the problem size is fixed and the number of cores increased. In addition to being capable of analyzing the impact of changing different architectural parameters. In this regard, ElasticSimMATE enables to conduct explorations belonging to the following categories:

- **Fast System Parameter Exploration**: thanks to trace-driven simulation speed, 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.
- **Weak-Scaling Exploration**: this approach relies on replicating traces for emulating more cores, thereby analyzing how performance scales when increasing the number of cores. This means that the workload is increased with the increase in the number of cores. This approach requires recording and carefully handle the synchronization semantics in the trace-replay phase so as to carefully account for the execution semantics on such an architecture.
- **Strong-Scaling Exploration**: this approach uses the capabilities of task-based programming. Based on the trace collection of a defined problem size application, task traces can be assigned to a different number of cores, allowing the analysis how a fixed size problem scales with the increase of the number of cores.

This paper is organized as follows. Section 2 discusses related work. Section 3 describes the main concepts of the ElasticSimMATE approach. In Section 4 we present the experimental results on selected applications for ESM in synchronization-mode and in Section 5 for ESM in task-mode. Finally, we conclude this paper in Section 6.

2. Related Work

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

2.1. Traditional simulators

Existing techniques can be classified into two fundamental families [6]. The first family focuses on the increase 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 [7, 8]. Distributed simulation is a known difficult technique as the simulation partitioning and event synchronizations among available hosts have to be carefully assigned.

Another popular approach for accelerating simulation is just-in-time (JIT) dynamic binary translation, e.g., OVP [9] and QEMU [10]. 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. CPUś, interconnect infrastructure) following the transaction-level modeling strategy [11] or by using trace-driven simulation [12].

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 of 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, a trace-driven simulation is an alternative approach. It collects reference traces from baseline systems with a low core count. In this way, most relevant phenomena are capture, such as jitter related to Operating System execution. The traces are then reused in a number of target trace-driven simulations in which CPU cores are replaced with trace injectors, thereby enabling to refocus simulation effort on other performance-critical system sub-components (cache, memory sub-system, communication architecture).

Authors in [13] proposed PinPlay a trace-driven technique that captures and replays traces in the form of pinballs (execution log files) for multi-threaded applications. The main drawback of this technique is that it does not take into account timing changes during replay time, leading to not so accurate results on highly non-deterministic applications. Authors in [14, 15] proposed SynchroTrace, a trace-driven technique that address these non-deterministic applications by collecting synchronization and dependency-aware traces. The main idea of this proposal is to record computational, thread synchronization and communication events separately, derived by native runs of the program. And then using this traces on a replay mechanism connected to the gem5 simulator. SynchroTrace is mainly focused on in-order core models, which is a restriction on its usage as most of the current multicore system works with out-of-order (OoO) cores.

Elastic Traces is a gem5 extension that allows collecting and playback micro-architecture dependency and timing annotated traces attached to the 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 to 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].

In Elastic Traces, the replay phase allows playing 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 re-player supports only single-threaded applications which are one main limitation. Elastic Traces demonstrates a speedup of $6-8\times$ compared to a reference Out-of-Order

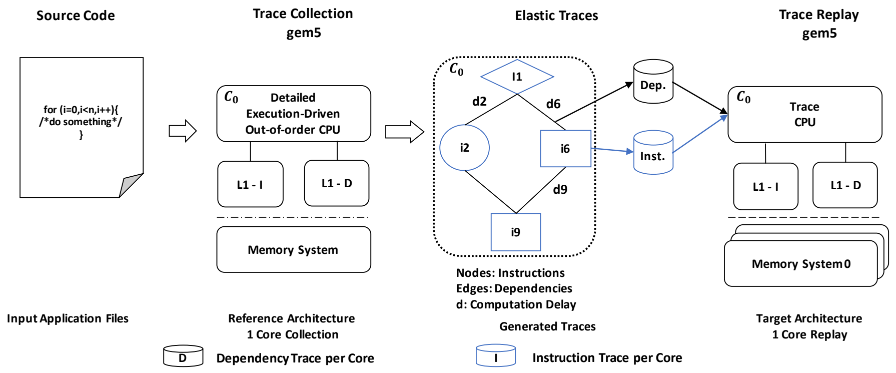


Figure 1: Elastic Traces Capture and Replay Mechanism [1].

core and is accurate, with less than 10% error versus the reference [1].

SimMATE [16] is a trace-driven simulator that operates on top of gem5 and is devoted to the exploration of in-order many-core 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 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.

SimMATE speedup is directly related on the application nature: observed memory-intensive applications result in 7× speedup at worse, whereas compute-intensive kernels with low memory activity result in speedups measured at up to 800×. Accuracy error compared to the reference full-system varies from 0.02% to 6% also depending on the application nature and scope of changed parameters in the targeted architecture configurations [16].

2.2. Trace-Driven simulation of OmpSs programs

OmpSs is a programming language based on StarSs and OpenMP directives. It makes use of task-based programming to enhance the parallelization of the applications. One of its main advantages is that it does not use synchronization points between threads, but allows the application to be executed in an asynchronous way accordingly to the data flow [17]. OmpSs is implemented by using Mercurium and Nanos++ runtime [18]. Task constructs are interpreted by Mercurium compiler and calls are generated by the Nanos++ runtime. The runtime checks if the dependencies of a newly created task have been resolved to properly schedule free-dependency tasks.

A trace-driven approach for OmpSs applications was proposed in [19]. The application is annotated with Nanos++ calls. The trace collection is performed in real hardware. Then, the traces are replayed in a framework called TaskSim. Authors report simulation speedup close to 19× while keeping accurate results with regard to the real execution of the application. One of the main drawbacks on trace-driven simulation approaches is the large size of the generated traces. In order to reduce the trace size problem, the author's proposed in [20] a filtering approach without losing accuracy. It is a sampling methodology that works on top of TaskSim called TaskPoint. TaskPoint identifies the task types and determines how to schedule the tasks. Tasks can be executed in three different modes, a warm-up mode, a detail mode, and a fast-forward mode. In the warm-up mode tasks are executed to avoid misleading results due to cold caches. In detail mode tasks results are gathered into a list that keeps track of the history of executed tasks. This history is then used in the fast-forward mode to fasten the simulation. This approach has been proven in the X86 architecture, however, it cannot yet be applied to ARM architectures.

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. In addition, it makes use of OmpSs programming languages to provide strong-scaling analysis in a fast and accurate manner.

3. ElasticSimMATE Framework

ElasticSimMATE integrates the advantage of Elastic Traces and SimMATE. On the one hand, we modified Elastic Traces to simulate multi-core architectures alongside with synchronization event detection and recording. On the other, based on the synchronization mechanism proposed in SimMATE, we adapted its methodology to be used with out-of-order cores. One of the main differences with regard to SimMATE is the way it is performed the trace collection. In SimMATE, the traces are based only on recording transactions between the CPU and the L1 cache, while ElasticSimMATE uses a complete set of instruction and dependencies.

The main objective of ElasticSimMATE is to allow a fast scalability and parameter exploration. ElasticSimMATE supports the following gem5 features: ARMv7 and ARMv8 ISA, OoO CPU model and SimpleMemory model (required by Elastic Traces). It can be used in two modes, synchronization-mode, and task-mode. ElasticSimMATE is composed of four phases, code annotation, checkpoint creation, trace collection, and trace replay. In this section, we explain in detail these phases for the synchronization-mode, and then the main changes to use it in Task-Mode.

3.1. ElasticSimMATE: Synchronization Mode

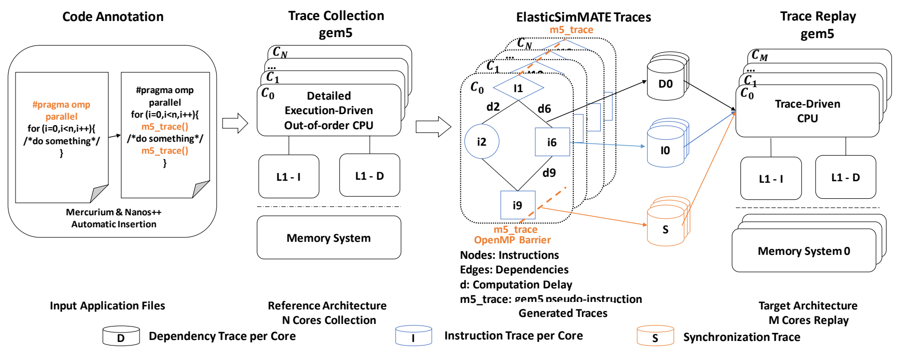


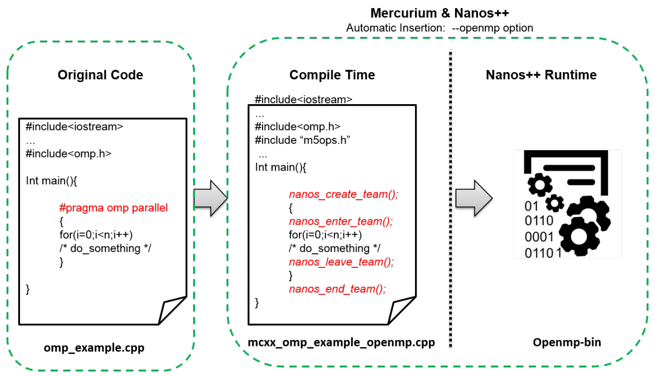
Figure 2: ElasticSimMATE Workflow: Synchronization Mode

Figure 2 conceptually depicts the ElasticSimMATE in synchronization mode (ESM-SM) 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 calling 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 is compatible with both OpenMP3.0 and POSIX thread APIs. The focus is put on OpenMP3.0 in this document. Recording synchronization traces requires using a specific gem5 pseudo-instruction created for this purpose: $m5\_trace()$. This pseudo-instruction requires being inserted either manually or automatically by means of using an instrumented run-time system.

3.1.1. Code Annotation

In order to collect the traces, gem5 has to be capable of detecting the beginning and end of the recorded events. To do so we annotate the code with a pseudo-instruction that is recognized by gem5 and starts the collection process. We called to this pseudo-instruction $m5\_trace()$ and it is added before compilation either manually or automatically. This pseudo-instruction is only valid int he gem5 framework, and it generates the collection of traces once is detected during the execution of the application. The aim of using this pseudo-instruction is to detect the beginning and the end of a given parallel event. We need to insert it on the source code every time an OpenMP event is called during run-time. This insertion can be done manually or automatically. However,



```
#include<iostream>
...
#include<omp.h>

Int main(){

        #pragma omp parallel
        {
        for(i=0;i<n;i++)
        /* do_something */
        }

}
```

```
#include<iostream>
...
#include<omp.h>
#include "m5ops.h"
 ...
Int main(){

        nanos_create_team();
        {
        nanos_enter_team();
        for(i=0;i<n;i++)
        /* do_something */
        nanos_leave_team();
        }
        nanos_end_team();
}
```

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

manual insertion is rather cumbersome and there we have considered two other automatic options:

- **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 being ported to 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.

We selected the automatic tracing call insertion, which is available for OpenMP using the Nanos++ runtime system and the Mercurium compiler. The advantage of this approach is that we can annotate any code with the $m5\_trace()$. It means that every time a selected OpenMP event is detected, the $m5\_trace()$ pseudo-instruction will be detected by gem5 and activate/stop the recording of the synchronization event. To do so, we modified the source code of Nanos++ that add a macro function that encapsulates the pseudo-instruction, i.e. we included it on the specific functions of Nanos++ that creates the selected OpenMP events. A summary of the currently supported set of constructs is in table 1.

3.1.2. Checkpoint creation

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.

Table 1: 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()

3.1.3. Traces collection

In this phase, ElasticSimMATE restores the system state from the checkpoint and begins trace collection. Three types of events are considered in parallel sections: instruction executed, dependencies (load/store), and synchronization events. Instruction and data dependency traces are captured thanks to an enhanced TraceCPU model. This enhancement is directly related to the incorporation of the synchronization mechanism and arbiter during the replay phase. The following information is 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 between the beginning and the end of a parallel section.
- **Number of data accesses:** the number of data accesses performed between the beginning and the end of a parallel section.

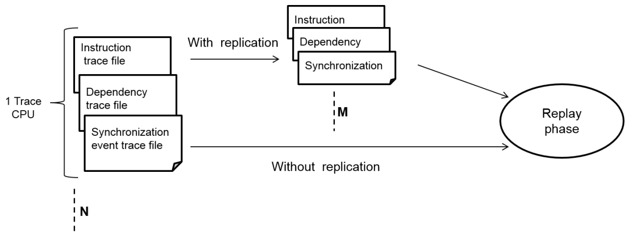
It has to be noted that for each thread under analysis the Tick and PC information will be the same since all threads are created at the same time. It means that the information on the synchronization traces is the same. In the case of the dependency trace, the load and store information is only collected 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.

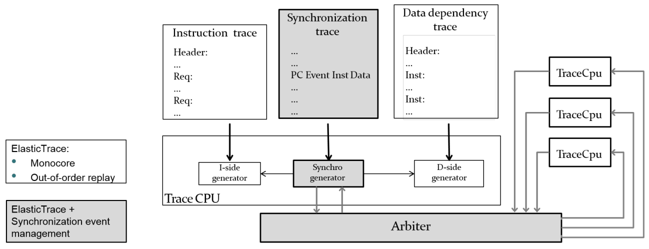
3.1.4. Traces replay

As illustrated in Figure 4(a), collected traces can be replayed in target architecture configurations in different ways. In this case, N represents 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. Note that the replication mechanism allows us to perform weak-scaling analysis as the problem size is increased by the ratio of $\frac{M}{N}$. In addition, the current implementation is performed with no address offsetting mechanism. This means that most of the resources are shared among cores.



(a) The replay phase is based on the previously collected traces on a reference architecture (N cores) and then replicated to M cores



(b) Replay overview including modified TraceCPU and arbiter

Figure 4: Trace replay approach

Figure 4(b) 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 behavior 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.

3.2. ElasticSimMATE: Task Mode

In order to perform strong-scaling analysis, it is necessary to create traces related to the workload execution and not to the cores. For this reason, we decided to use the OmpSs programming language as our approach can detect the beginning and the end of each created task. Figure 5 presents the workflow of ElasticSimMATE in task mode (ESM-TM). In this case, annotations are made in order to track the beginning/end of each task. Then, by doing a Full-System simulation in gem5, traces are collected and split into task traces. Afterward, task-traces can be assigned to be executed by any available core in the replay phase.

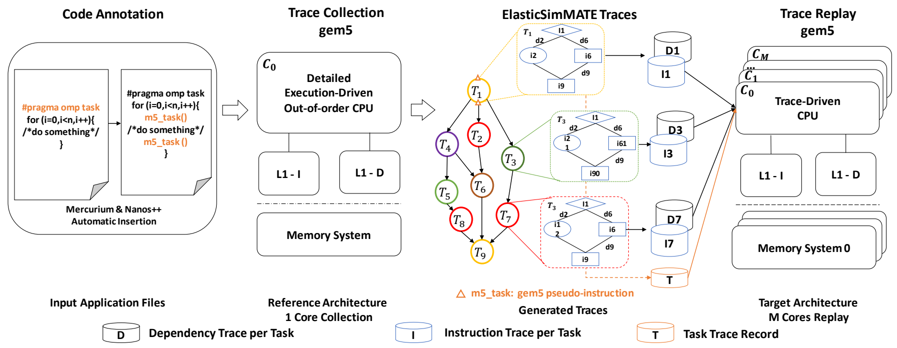


Figure 5: ElasticSimMATE Workflow: Task Mode

3.2.1. Code Annotation

In the task-mode, the application is annotated with calls with a $m5\_task()$ gem5 pseudo-instruction. We modified the API runtime to automatically detect the beginning and the end of a task. In order to properly annotate the application, we need to use Mercurium and the modified version of Nanos++. We determine which functions of the runtime control the creation/exit of a task and added in those function the pseudo-instruction. These functions are the nanosSMPThread $inlineWorkDependent()$ for the task creation and the workDescriptor $finish()$ for the end of the task.

3.2.2. Checkpoint Creation

In a similar way than on the synchronization mode, a checkpoint is created to avoid tracing any non-useful information of the OS boot phase.

3.2.3. Trace Collection

In this case, in addition to the instruction and dependency trace, we collect a task record trace. This trace includes the following information:

- **Task ID:** Task ID number of the executed task.
- **Core ID:** Thread ID assigned by the scheduler.
- **Tick:** Tick count of a CPU at the beginning of the task.
- **Program Counter:** Program counter at the beginning of the task.
- **Number of instructions:** Number of instructions executed inside the task by a CPU.
- **Number of data accesses:** Number of data accesses performed inside the task by a CPU.

In a post-processing phase, we analyze the instruction and dependency traces and based on the information collected on the task trace we generate instruction and dependency traces per tasks. In this way, we can assign different tasks to different cores and perform scalability analysis in the replay phase.

3.2.4. Trace Replay

Figure 6 shows a representation of the trace replay phase in Task-Mode. Once the instructions and dependencies per task are obtained, different tasks can be assigned to different cores. In this version, we use a static scheduler mechanism (task2core file). This file is read by the arbiter and populates a list of task to be executed per core. In this way, we can analyze different types of scheduling mechanisms and how they impact the execution of the application because we can increase the number of cores and assign the different number of tasks to each core.

One of the most important features in this phase is the task2core file. This file works as a fixed schedule and it determines which task will be executed in

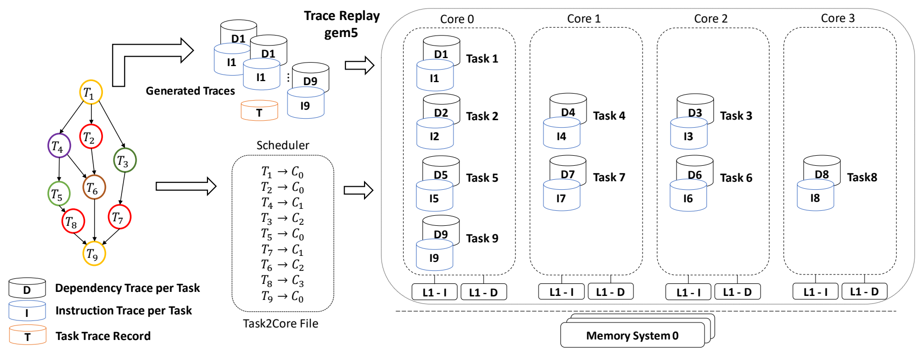


Figure 6: ElasticSimMATE Task-Mode: Trace Replay.

which cores. In this way, it is possible to create different scheduling alternatives and evaluate which one suits better in terms of execution time. In the next version, we plan to include dynamic scheduling of tasks to cores.

4. Experimental Results ElasticSimMATE: Synchronization-Mode

In this section, we evaluate and compare ElasticSimMATE against Elastic Traces and gem5 Full-System simulation (reference), and ElasticSimMATE in synchronization-mode. 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 3.1.4), 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.

4.1. Experimental Setup

As the reference model, we consider an Out-of-Order (or O3) CPU model in gem5 that represents an ARMv7 architecture. Figure 7 depicts a four-core architecture along with the cache hierarchy, an interconnect and the 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 2. 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).

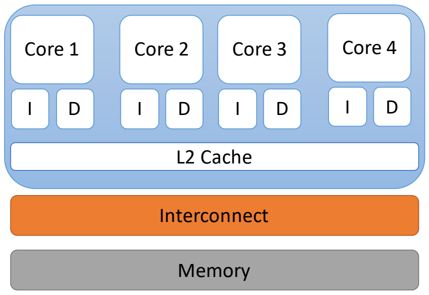


Figure 7: Reference system with 4 cores.

4.2. 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 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

Table 2: Reference baseline system

	Parameter	Value
CPU	Model	O3
I Cache	Size	32kB
	Associativity	2-way
	Cycle Hit Latency	2
D Cache	Size	64kB
	Associativity	2-way
	Cycle Hit Latency	1
L2 Cache	Size	1MB
	Associativity	8-way
	Cycle Hit Latency	12
Main Memory	Model	Simple Memory
	latency	30ns

workload (up to 512x512 matrix sizes) and 3 compute-intensive applications defined in Rodinia [21] and Parsec [22] 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.

4.3. 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 3 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, the error decreases when analyzing multicore systems.

Table 3: 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

4.3.1. ***Speedup Evaluation***

Figure 8 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 a comparatively significant time in simulation initialization versus simulation run.

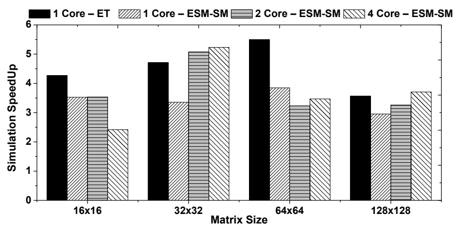
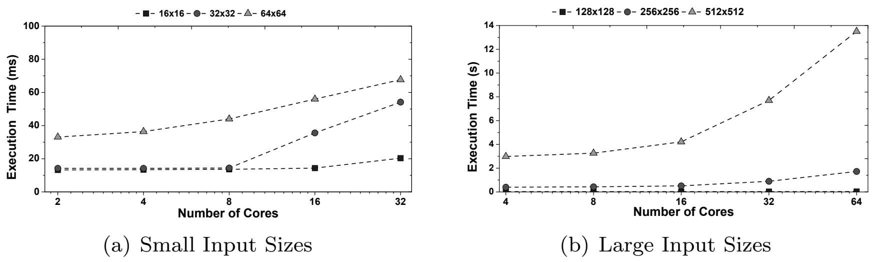


Figure 8: Simulation speedup for matrix multiplication.

4.3.2. ***Trace Replication:***

Results 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.

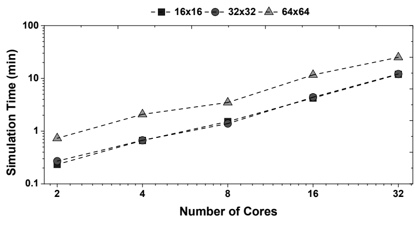


(a) Small Input Sizes (b) Large Input Sizes

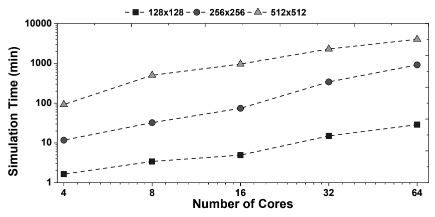
Figure 9: Execution time for matrix multiplication.

Figures 9(a) and 9(b) 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.

Figures 10(a) and 10(b) 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). Simulation times for large input sizes have 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.



(a) Small Input Sizes



(b) Large Input Sizes

Figure 10: Simulation time for matrix multiplication

Experimental results on selected applications: . 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.

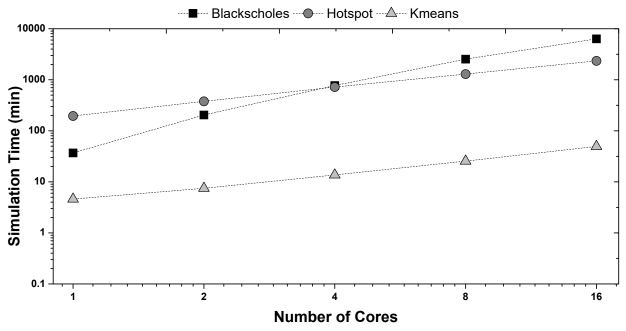
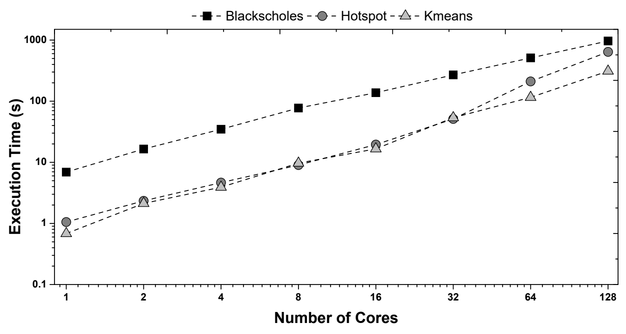


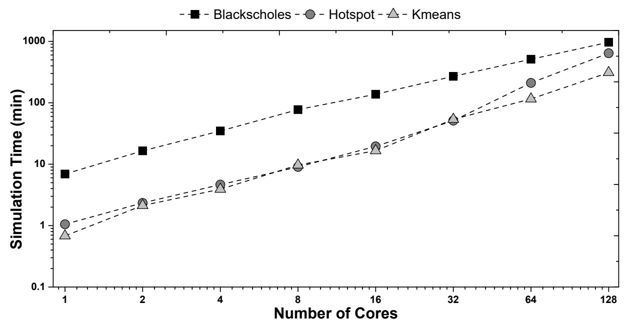
Figure 11: gem5 Full-System Simulation Time for selected applications.

One of the main objectives of ESM is to reduce the overall simulation time. In this regard, we ran some initial gem5 full-system simulations with different core number and a small input set. We then measured the simulation time to have a reference on how much an FS simulation will take. Figure 11 shows the simulation time in log scale for these applications. We can observe that the simulation time increases linearly (hence exponentially due to the log scale) with the number of cores. We can assume that a simulation of 128 cores will take a prohibitive amount of time to finish, even more taking into account if we increase the problem size.

Figures 12(a) and 12(b) give execution times and simulation 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 the hotspot. Simulation times are in the tens of hours for the chosen applications/input set sizes for 128 core systems, which is acceptable.



(a) Execution Time

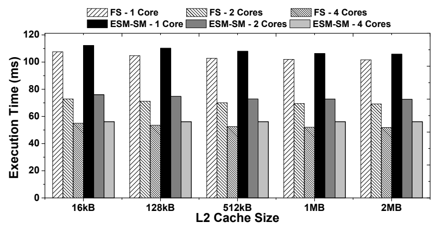


(b) Simulation Time

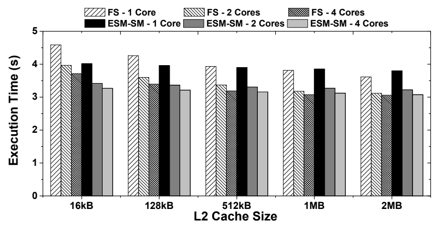
Figure 12: Trace replication analysis for selected applications.

4.3.3. ***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. For this analysis, we chose a compute-intensive application (kmeans) and a memory-intensive application (canneal).



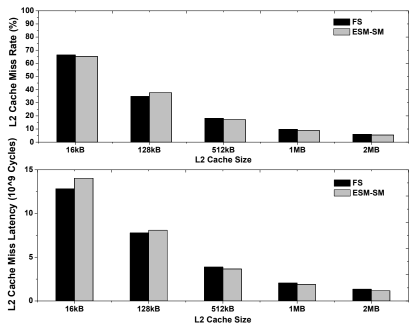
(a) K-means



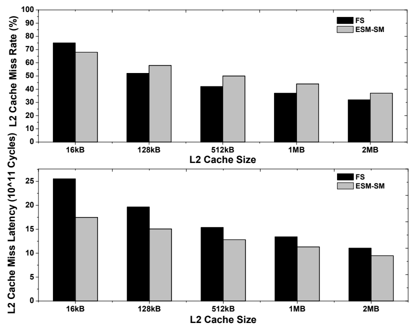
(b) Canneal

Figure 13: Execution time for different core number and L2 cache sizes.

Execution time results are shown in Figure 13. 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 on average 3× faster than FS, a 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 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. Experimental results are presented in Figure 14. In this figure, we can observe that other important parameters for the architectural exploration are well analyzed by ESM for the different type of applications.



(a) K-means



(b) Canneal

Figure 14: Parameter exploration analysis for different L2 cache sizes for selected applications. The upper figure shows how the L2 cache miss rate change when increasing the L2 cache size while the lower the L2 cache miss latency.

4.4. 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 is related to intrinsic Elastic Traces tracing approach.

5. Experimental Results: ElasticSimMATE - Task Mode

ElasticSimMATE in task-mode is validated with regard to gem5 Full-System simulation. In this case, we mainly analyzed the execution time (accuracy analysis) and simulation time (speed-up analysis). In this mode, ElasticSimMATE allows strong scaling analysis as the problem size remains the same while increasing the number of cores. We selected the OmpSs version of the Parsec benchmark suite [23]. In the first analysis, we show the accuracy level of different applications for a system configuration similar to the one used to collect the traces. Then, we selected three applications with different complexities and increased the number of cores (strong scaling analysis).

5.1. Experimental Setup

Simulation results from ElasticSimMATE are compared to Full-System gem5 simulation. The first analysis intends to validate and determine the error percentage of our approach. The second to show how it can be used for strong scaling analysis for applications with different complexity levels. In this case, we performed the analysis for an ARMv8 64b architecture.

The trace collection phase was performed in a system with one core and a simple memory model. We choose the small input set in all cases, as our main objective is to prove the usefulness of the approach. Table 4 shows the application domain, the problem size and the total number of instantiated tasks.

Table 4: Selected Benchmark Characteristics

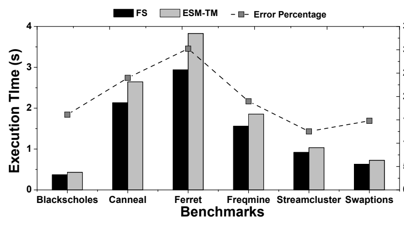
Benchmark	**Application Domain**	**Input Set**	**#Tasks**
blackscholes	Financial Analysis	16K	400
canneal	Engineering	10000 swaps per temperature step	32
ferret	Similarity Search	16 image queries, database with 3544 images	384
freqmine	Data Mining	250000 anonymized click streams	1737
streamcluster	Data Mining	4096 input points, block size 4096, 32 point dimmensions	1319
swaptions	Financial Analysis	16 swaptiosn, 10000 simulations	16

In order to do a fair comparison, the task2core file is created based on the way Full-System simulation executes the tasks. This means that we emulated the FS-scheduler to avoid any bias in our analysis.

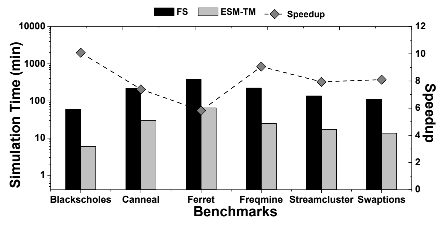
subsectionAccuracy and Speedup Evaluation

As an initial validation test, we collected the traces for the different application of the PARSEC benchmark suite. Then we replay all the tasks using ElasticSimMATE in Task-Mode. Figure 15(a) shows the execution time for selected PARSEC applications. The average error percentage is 19%, with a minimum value of 12.5 or streamcluster and a maximum of 30.2 for ferret. These errors are directly related to the way ElasticTraces handles the dependency graph and then it impacts the way dependencies are executed inside a task.

Figure 15(b) shows the simulation time for the selected applications. In average, ESM-TM is 8× faster than Full-System simulation. The speed-up depends on the application type and according to with our findings, it can be between 5× and 10×.



(a) Execution Time



(b) Simulation Time

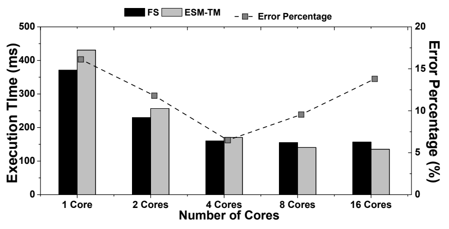
Figure 15: Simulation results of selected PARSEC applications

5.2. Strong Scaling Analysis

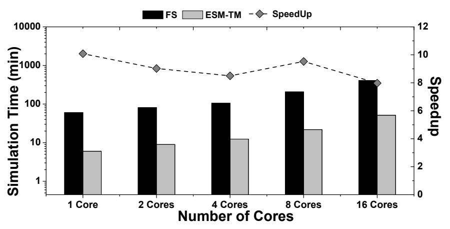
Strong scaling analysis targets the analysis of how the execution time of a solution of a problem varies with the increase in the number of cores for a fixed size problem. In this case, we choose three different applications, with different complexity levels and increased the number of cores from 1 to 16. Then, we compared our results to FS simulation.

5.2.1. Blackscholes

Figure 16(a) shows the execution time of Blackscholes when the number of cores is increased. We can observe that after 4 cores the execution time is no longer improved. This can be explained based on the fact that the application can execute only four tasks in parallel and then a taskwait synchronizes the result of each of these tasks. We observe in Figure 16(b) that the simulation time in Full-System simulation increases exponentially with the number of cores, whereas in ESM-TM, it increases linearly, been 9× faster than FS gem5.



(a) Execution Time

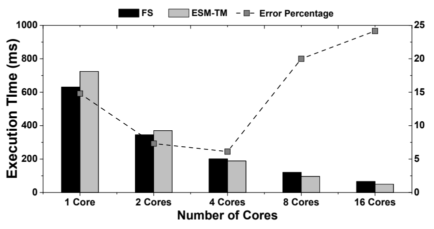


(b) Simulation Time

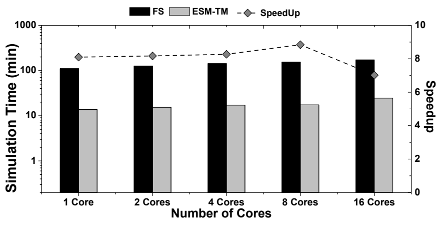
Figure 16: Blackscholes Simulation Results

5.2.2. Swaptions

Figure 17 shows the execution and simulation times for swaptions application for different core number. The particularity of this application is that it only presents 16 tasks. The execution time is reduced whit the increase in the number of cores. In this case, the smaller execution time will be accomplished with a system with 16 cores, as the 16 tasks will be executed in parallel. We can observe the execution time tendency in Figure 17(a). Regarding the simulation time, ESM-TM is 8× faster than FS simulation.



(a) Execution Time

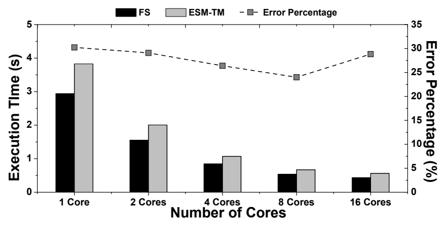


(b) Simulation Time

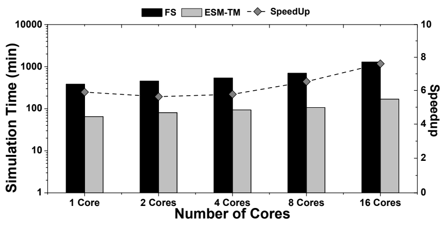
Figure 17: Swaptions Simulation Results

5.2.3. Ferret

Finally, the third application is Ferret. It has a total of 384 tasks and it does not present any taskwait condition in its code. With the increase in the number of cores, we can observe a clear reduction on the execution time (Figure 18(a)). Even though the absolute error is in average 26%, we can observe that the overall tendency is well tracked by ESM-TM. This means that ESM-TM presents an excellent relative accuracy and with a 5× speed-up over gem5 FS simulation (Figure 18(b)).



(a) Execution Time



(b) Simulation Time

Figure 18: Ferret Simulation Results

5.3. Comparison against real-hardware simulation

This section aims at analyzing the accuracy of the predicted results when targeting a physical platform. The de-facto platform is here the Mont-Blanc

3 Dibona prototype [24]. This platform is based on the Cavium ThunderX2 chip, which includes full custom out-of-order 64 bits cores and features a 32KB instruction and data L1 caches, a 256KB L2 shared cache and a 32MB L3 distributed cache. It uses an ARM CCI interconnect and presents 8 DDR4 memory controllers for a total size of 256GB [25].

In order to compare ESM-TM with regard to Dibona, we devised a dibona-like model. The main idea was to ensure that most of the available gem5 parameters match with the values of this platform, these values are presented in Table 5.

Table 5: Dibona gem5 model (Latencies are reported in cycles)

	Parameter	Value
CPU	Model	O3
I Cache	Size	32kB
	Associativity	8-way
	Cycle Hit Latency	4
D Cache	Size	32kB
	Associativity	8-way
	Cycle Hit Latency	4
L2 Cache	Size	256kB per core
	Associativity	8-way
	Cycle Hit Latency	15
L3 Cache	Size	32MB (1MB per core)
	Associativity	16-way
	Cycle Hit Latency	68
Main Memory	Model	DDR4
	Size	8GB
	Frequency	2.4GHz
	Channels	8
	Ranks	2
Interconnect	Model	XBar
	Width	128
	Frequency	2GHz

5.3.1. Accuracy analysis

As the gem5 dibona-like model is a first approximation to the real hardware configuration the absolute values of ESM-TM are higher than the once computed on the Dibona platform. In this case, we decided to analyze the relative accuracy of the strong scaling experiments by taking into account the speedup reported by both approaches. Figures 19(a), 19(b) and 19(c), show the speedup figures obtained from 2 to 16 cores, with both ESM-TM replay simulations and averaged runs on the Dibona prototype. For all three applications, ESM-TM replays accurately capture reference scaling properties, with a slight tendency towards overestimation. Average tracking error in speedup is below 5%, with a maximum 15% error on Blackscholes on 4 and 8 cores experiments.

5.3.2. Experimental Results on Real-World Applications

We perform experimental results on a real-world application. We selected LULESH application, as is one of the most representatives among the HPC study problems [26]. We choose a medium problem size (120) with dynamic scheduling. We use the OmpSs run-time, which seamlessly treats each of the loop chunks as a task. We perform native runs of the application in Dibona with 1 to 16 cores, which are the comparison baseline for our analysis.

Figure 20 shows the different speedup curves obtained on for Dibona native executions and the different simulation infrastructures. As can be seen for up to 8 cores all speedups of ESM-TM are well correlated with Dibona platform. However, for 16 core we can observe that ESM-TM speedup is higher than the one reported on Dibona. We assume this is due to the fact that there is no gem5 model for the ARM CCI interconnect and that we are using a generic crossbar working at the same frequency. Further work will focus on the modeling on this kind of interconnect to enhance our results. Another source of error is the values on the created gem5 dibona-like model, as some of the parameters are initial estimations. This is due to the fact that true values are not publicly available.

5.4. Summary

ElasticSimMATE in task-mode uses the task construct capabilities of OmpSs programming languages. By collecting traces in one defined reference system we can perform fast strong-scaling analysis with a high relative accuracy. This means that ESM-TM is able to track correctly how an application scales with the increase in the number of cores. Experimental results show that our approach is in average 8× faster than Full-System gem5 simulation. The average absolute error is close to 18%. In this approach, we replay individual tasks based on a task2core file that works as a static scheduler, which is not the case in FS simulation. In comparison with real-hardware ESM-TM shows great accuracy for different academical benchmarks.

6. 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.

Our approach can be used to perform scalability and parameter exploration analysis. The synchronization-mode uses an adequate trace replication mechanism to allow weak-scaling analysis. 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-SM is at least 3× faster than FS simulation.

The task-mode uses a task-based mechanism to assign tasks to available cores, allowing the strong-scaling analysis. It can be used to analyze how a given workload can be divided and allocated into the available cores and how it can be adapted to the number of cores. Experimental results show that, in comparison with FS simulation, ElasticSimMATE is 8$\times$ faster with an average error of 18%. Despite this error, our approach can accurately track the overall tendency of scaling experiments for different architectures in a fast way. In comparison with real-hardware execution, ESM-TM shows a good correlation up to 16 cores for academical benchmarks and up to 8 cores with real-world applications. Further work will be focused on improving our gem5 models to enhance the overall accuracy of ESM.

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. We also plan to enhance our synchronization-mechanism to take into account trace offsetting. For the task-based methodology, we plan to implement a dynamic scheduler that checks on-line data dependencies. Finally, our tool will be freely-available once we have made the proposed improvements.

7. Acknowledgment

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 number 671697.

References

[1] R. Jagtap, S. Diestelhorst, A. Hansson, M. Jung, Exploring system performance using elastic traces: Fast, accurate and portable, in: 2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), 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, D. A. Wood, The Gem5 Simulator, ACM SIGARCH Computer Architecture News 39 (2) (2011) 1–7. `doi:10.1145/2024716.2024718`.
URL `http://doi.acm.org/10.1145/2024716.2024718`

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

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

[5] A. Nocua, F. Bruguier, G. Sassatelli, A. Gamatié, ElasticSimMATE: A fast and accurate gem5 trace-driven simulator for multicore systems, in: 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip, ReCoSoC 2017, Madrid, Spain, July 12-14, 2017, 2017, pp. 1–8. `doi:10.1109/ReCoSoC.2017.8016146`.
URL `https://doi.org/10.1109/ReCoSoC.2017.8016146`

[6] P. E. Heegaard, Speed-up techniques for simulation, TELEKTRONIKK 91 (1995) 85–7130.

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

[8] M. Alian, D. Kim, N. S. Kim, pd-gem5: Simulation infrastructure for parallel/distributed computer systems, IEEE Computer Architecture Letters 15 (1) (2016) 41–44. `doi:10.1109/LCA.2015.2438295`.

[9] OVP. Open Virtual Platforms, http://www.ovpworld.org/ (2017).

[10] QEMU. Open source processor emulator, http://wiki.qemu-project.org/Main_Page (2017).

[11] L. Cai, 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.

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

[13] H. Patil, C. Pereira, M. Stallcup, G. Lueck, J. Cownie, Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs, in: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '10, ACM, New York, NY, USA, 2010, pp. 2–11. doi:10.1145/1772954.1772958.
URL http://doi.acm.org/10.1145/1772954.1772958

[14] S. Nilakantan, K. Sangaiah, A. More, G. Salvadory, B. Taskin, 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), 2015, pp. 278–287. doi:10.1109/ISPASS.2015.7095813.

[15] K. Sangaiah, M. Lui, R. Jagtap, S. Diestelhorst, S. Nilakantan, A. More, B. Taskin, M. Hempstead, Synchrotrace: Synchronization-aware architecture-agnostic traces for lightweight multicore simulation of cmp and hpc workloads, ACM Transactions on Architecture and Code Optimization 15 (2018) 1–26.

[16] A. Butko, R. Garibotti, L. Ost, V. Lapotre, A. Gamatie, G. Sassatelli, 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, 2015, pp. 707–712. doi:10.1109/ASPDAC.2015.7059093.

[17] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas, Ompss: A proposal for programming heterogeneous multi-core architectures, Parallel Processing Letters 21 (02) (2011) 173–193. arXiv:https://www.worldscientific.com/doi/pdf/10.1142/S0129626411000151, doi:10.1142/S0129626411000151.
URL https://www.worldscientific.com/doi/abs/10.1142/S0129626411000151

[18] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, J. Labarta, Nanos mercurium: a research compiler for OpenMP, in: Proceedings of the European Workshop on OpenMP, Vol. 8, 2004, p. 56.

[19] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, M. Valero, Trace-driven simulation of multithreaded applications, in: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2011, pp. 87–96. doi:10.1109/ISPASS.2011.5762718.

[20] T. Grass, A. Rico, M. Casas, M. Moreto, E. Ayguadé, Taskpoint: Sampled simulation of task-based programs, in: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2016, pp. 296–306. doi:10.1109/ISPASS.2016.7482104.

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in:

IEEE International Symposium on Workload Characterization (IISWC), 2009, Ieee, 2009, pp. 44–54.

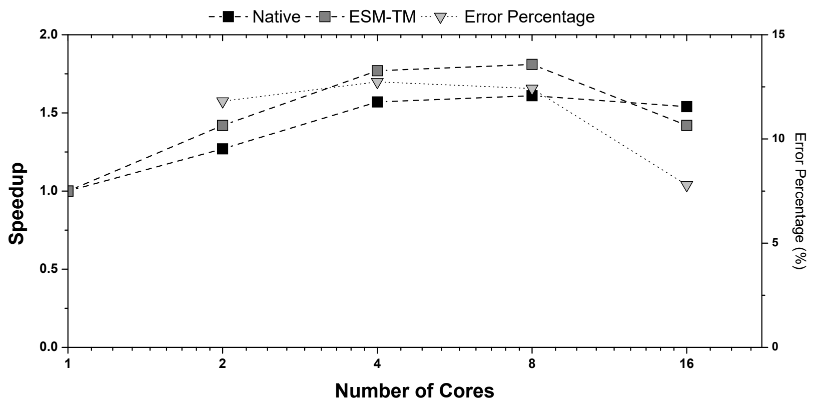
[22] C. Bienia, S. Kumar, J. P. Singh, K. Li, The PARSEC benchmark suite: Characterization and Architectural Implications, in: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, ACM, 2008, pp. 72–81.

[23] D. Chasapis, M. Casas, M. Moretó, R. Vidal, E. Ayguadé, J. Labarta, M. Valero, PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite, ACM Transactions on Architecture Code Optimization 12 (4) (2015) 41:1–41:22. doi:10.1145/2829952.
URL http://doi.acm.org/10.1145/2829952

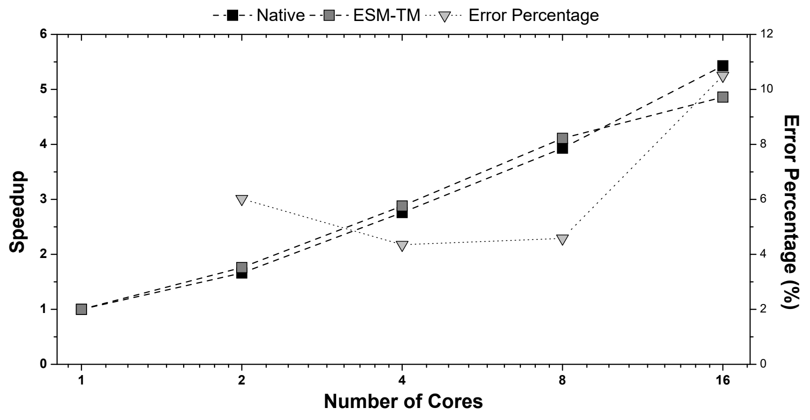
[24] MONT-BLANC 3 PROTOTYPE: DIBONA, http://montblanc-project.eu/ (2018).

[25] ThunderX2 ARM Processors, https://www.cavium.com/product-thunderx2-arm-processors.html (2018).

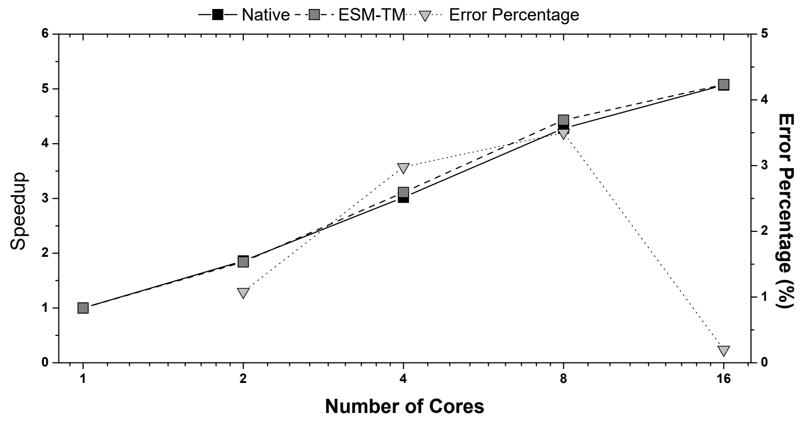
[26] Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH), https://computation.llnl.gov/projects/co-design/lulesh (2018).



(a) Blackscholes



(b) Swaptions



(c) Ferret

Figure 19: Speedup comparison against Dibona

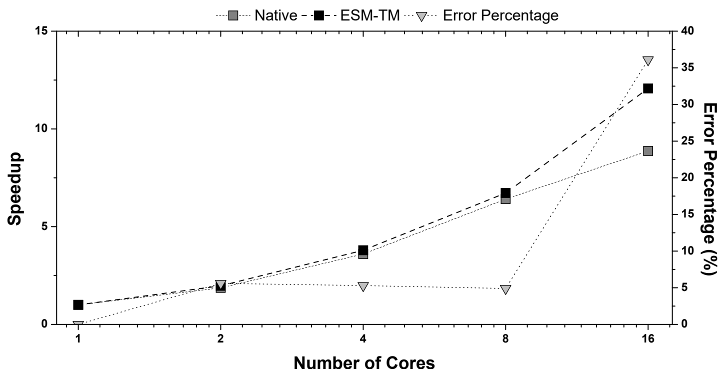


Figure 20: LULESH Speedup using Dibona (Native) and ElasticSimMATE-Task Mode (ESM-TM).