Fast and Scalable Simulation Framework for Large In-Order Chip Multiprocessors

Yuri Nedbailo
MCST JSC, INEUM im.I.S.Bruka
Moscow, Russia
nonsens@inbox.ru

***Abstract*—As chip technology advances, the number of cores in mainstream chip multiprocessors (CMP) increases, so chips with hundreds of cores may become common within a decade. One of the challenges this trend sets to computer architects is to make the current CMP designs scalable to larger numbers of cores. A tool set that would allow us to predict how various design decisions may affect the performance of larger CMPs is therefore necessary. In this paper, we present a trace-based simulation framework we devised for Elbrus microprocessor family. Its core component, the CMP simulator is scalable to at least one thousand of cores and allows to evaluate the kilo-core CMP performance in just a few days using a mainstream 16-core host computer. It is also highly flexible and architecture-agnostic and, therefore, could be used to simulate other in-order architectures. We validated the framework against a real machine and achieved an average accuracy of 18 percent in single-core tests and 15 percent in four-core, an average error in relative slowdown evaluation of 2.6 percent, and average absolute errors in L2 and L3 cache miss rates within 0.3 bytes per cycle.**

I. Introduction

In the past decade, mainstream chip multiprocessors have increased their core numbers from two or four to tens thanks to advances in chip technology. This trend appears to remain in the near future and after another decade, CMPs with hundreds of cores may become common. For computer architects, this sets a number of challenges. One of such challenges is to make current CMP designs scalable, i.e. reusable in the next generations with higher core numbers. This is vital for coping with growing design complexity within a limited design time budget, especially for small chip-makers.

Some of the design decisions can be made based on theory and common practice. For example, it is already known [1] that a scalable CMP design will need a distributed shared cache, a distributed directory included in it, a two-dimensional (or e.g. optical) on-chip network (Figure 1). Some decisions need evaluation as they depend on the instruction set, core microarchitecture, design constraints, novel schemes used or uncommon features of the particular microprocessor family. For example, the optimal number of memory channels depends on how frequently cores access memory, how often the accesses miss in caches and other factors.

As a result, in addition to accurate simulation of the currently designed processors for verification and fine-tuning of their components, at least rough estimates of larger prospective designs must be performed to ensure design scalability and narrow the spectrum of possible long-term solutions. There are two basic kinds of accurate simulation: field-programmable

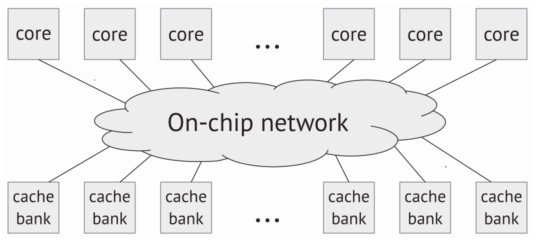

Fig. 1. The idea of a CMP with a distributed shared cache

gate array (FPGA) prototyping and software register-transfer level (RTL) simulation. The former requires expensive hardware, the latter is prohibitively slow to simulate real applications and both require complete and working RTL description of all the design, which is difficult to make in the early design stages. A close alternative is execution-based cycle-accurate simulation, which is essentially a simplified software model of the simulated system. There are good and versatile simulation frameworks of this type that offer ready building blocks (such as cache or memory models), e.g. Gem5 [2]. However, this type of simulators may become too slow and/or memory-consuming to simulate large CMPs, as they accurately model the execution of every instruction of the program with all its data dependencies.

Another approach is trace-based simulation, which is similar to execution-based, except that the execution of a program is decoupled from cycle-accurate simulation of the processor. The program is first executed on a simpler model, usually without emulating most delays occurring in the real system, or even just instrumented, to create a trace, i.e. a log, of the events like memory accesses that drive the work of the microprocessor's components. On the second phase, the CMP simulator is executed with these traces (or some parts of them) as input data. Therefore, no simulation of the program itself and, in the case of memory traces, of the complex logic of the cores is needed in the second phase, which improves the simulation speed and reduces its memory consumption dramatically. A number of trace-based simulation frameworks with different speed, accuracy and capabilities exist.

In this paper, we present a fast and scalable trace-based simulation framework which we devised for the in-order very long instruction word (VLIW) microprocessor family Elbrus [3]. The trace-driven CMP simulator is architecture-independent and, therefore, potentially cabable of simulating any other in-order processor, once a similar trace generator has been made for that processor.

The main contribution of the study can be summarized as follows.

1) We propose a lightweight and fast modification of an architectural simulator allowing to capture a memory trace that reflects dependencies between the accesses.
2) We introduce a fast, scalable, and architecture-agnostic design of a trace-driven CMP simulator, featuring simple and flexible behavioural models of caches and the on-chip network.
3) We evaluate the accuracy and performance of this framework in the SPEC CPU2006 benchmark suite and a synthetic on-chip network test.
4) We demonstrate the use of the framework for design space exploration on the example of a kilo-core simulation.

The rest of the paper is organized as follows. Section II presents an overview of existing execution- and trace-based simulation techniques. Section III describes the proposed solution. Section IV presents an evaluation of its accuracy and performance. Section V demonstrates an example of its use. Finally, Section VI concludes the paper.

II. Background

A. Execution-based simulators

Gem5 [2] is an acknowledged, accurate and flexible full-system simulator, widely used in academic research. It supports most commercial instruction sets and diverse CPU models and features a detailed and flexible memory system, including support for multiple cache coherence protocols and interconnect models. However, the scalability of its detailed mode is limited due to its accuracy and limited multithreading support in modern mainstream operating systems.

A number of other execution-based simulators have been developed to overcome these limitations [4], [5], [6]. For example, ZSim [6] running on a 16-core Xeon E5 host machine can simulate a 1024-core chip at up to 1,500 MIPS speed with simple timings models, and up to 300 MIPS with detailed core and uncore models. However, such speed comes at the cost of various trade-offs: the simulation is based on native execution using binary instrumentation, the simulated cores are not strictly synchronized and uncore models are simplified. The accuracy and design space exploration capabilities of such simulators are, therefore, limited.

B. Trace-based simulators

For uncore design space exploration, which is the crucial part of designing a scalable CMP, detailed modelling of cores at each simulation is unnecessary and is the main simulation speed limiter. On the other hand, data dependencies between instructions must be modelled with cycle accuracy so as to evaluate how uncore latencies affect the overall performance. For this reason, some traces of events, captured during an execution, can be used to drive the uncore model instead of actual events occurring in the execution-based model. Dependencies between the instructions can be represented as those between the events; such traces are sometimes called self-related.

As events, instructions themselves can be used; this is common when out-of-order architectures are simulated. The whole processor can be modelled at this level; the speed gain is accordingly moderate. For example, Elastic Traces [7] allow to achieve a speed-up of 6–8 times compared to detailed Gem5 simulation.

On-chip network simulators can use network injections as events. Examples are [8], [9], [10], [11]. While being fast and accurate, such simulators have limited use as they do not model caches. Design space exploration involving different cache configurations will, therefore, require trace capture for each configuration.

If the task is to model the whole uncore without accurately modelling out-of-order execution (as in the case of in-order processors), memory accesses are the events needed to trace. Sadly, there are few publications on the topic of self-related memory access traces.

A challenge for trace-based simulation is multithreaded programs with frequent communication between threads. A number of solutions have been developed. A hybrid approach combining trace-based simulation of independent parts of the code with an execution-based engine was proposed in [12]. SynchroTrace [13] is claimed to be a scalable, fast and accurate solution that can be integrated into other simulation frameworks.

C. Trace sampling

Apart from reduced model complexity, trace-based simulators naturally offer an opportunity to reduce the simulation time greatly by executing only some small representative parts of the program. Execution-based simulation requires checkpointing [14] or benchmark generation [15], [16], [17] for this, while traces can simply be shortened, or sampled.

The simple method is statistical sampling of intervals of fixed length, or cluster sampling [18]. As long as the interval (cluster) is long enough to compensate cold-start effects, picking a number of intervals at random times of the execution is sufficient for a desirable accuracy in simulation results, e.g. 100 intervals can give an average error of several percent.

The more complex solution is to analyse the program behaviour and select a set of most representative intervals. A method for automatic cluster selection was proposed [19] and allows to achieve the average accuracy of a few percent with only two to ten clusters per test.

III. Fast and scalable simulation framework for large in-order chip multiprocessors

As discussed earlier, a trace-based CMP simulation framework basically consists of an execution engine used for trace capture and a trace-driven CMP simulator. This section describes the proposed implementation and use of them.

A. Trace capture

The trace capture tool was built on top of an existing cycle-accurate user-level execution-driven simulator of VLIW architecture Elbrus. It is described in detail in [20] and, in the basic mode, precisely simulates the execution of a program assuming an ideally fast memory subsystem. Cycle-accurate cache and memory models can also be added to the simulation so as to enable cycle-accurate execution-driven simulation of the whole chip, which is not considered in this work. Only the L1 instruction cache is modelled, as its absence increases the trace size multiple times; an L1 data cache can also be included to reduce the trace size by tens of times [21] and increase the consequent CMP simulation speed with an error in the accuracy of several percent [13].

What is needed during trace capture is to log every memory access (i.e. L1 cache access or miss), including the timestamp of the access and a reference to the first future access dependent on it. The following format for this was chosen:

$$cycle[63:0], addr[31:0], delay[11:0], opc[8:0],$$

where

cycle – is the timestamp of the command issue (reduced to 32 bits during sampling dicsussed further),

addr – is the virtual address of the access,

delay – relative timestamp of a dependent command,

opc – type of access: read/write, inst/data, caching etc.

Cache-unaligned accesses and block prefetches are broken down into separate aligned operations at this level, so all the accesses in the trace are aligned.

All this information about the memory access events except the *delay* comes naturally from the simulation. To track dependencies and calculate *delay*, an additional mechanism was implemented.

Firstly, a "first-in, first-out" buffer (FIFO) was added to store the events a few thousand of simulation cycles before output. In addition to the information listed above, each record in this buffer stores the name of the destination register of the access. Another structure added to the simulator is a look-up table, organized as an array of pointers to this buffer for each possible destination register; the buffer and the table, therefore, store couples of pointers to each other (Fig. 2).

When a memory access is issued, a record is created in the buffer and a pointer to this record is written to the table. When a register is accessed by the program, the table is checked, a corresponding record is found and its *delay* value is calculated as the difference between the current timestamp and the recorded. If a record reaches the end of the FIFO before the register is accessed, the maximum possible value is output in place of *delay* to indicate no dependency found.

To increase the overall simulation speed during trace sampling discussed further, the simulator can dynamically switch this additional logic on and off. Without it, a speed-up by several times is achieved thanks to optimisations described in [22], which is further called fast-forwarding. The speed in this mode could be additionally increased by switching to functional simulation, i.e. disabling the modelling of pipeline delays, but this may lead to significant changes in the sample number and the uniformity of their distribution. Finding a way to avoid these effects could be future optimization.

Although this part of the framework is bound to microarchitecture Elbrus, it appears that it can be easily built on top of other cycle-accurate simulators in a similar way.

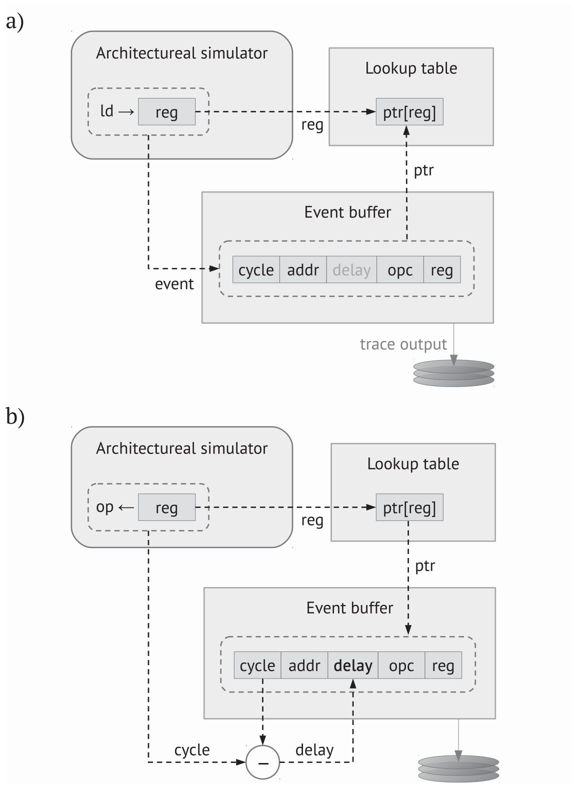

Fig. 2. Trace capture: a) new memory access, b) register use

B. Trace sampling

Although fast-forwarding in the architectural simulator is an order of magnitude faster than trace capture, it is still three orders slower than native execution, as it has to be cycle-accurate. In modern benchmarks, it takes so much time – from days to a few weeks for each test – that single-pass trace sampling is preferable. A simple cluster sampling technique [18] is therefore used to collect a sufficient number of trace intervals of sufficient length in one pass; the traces are then analysed and a representative interval subset is selected.

Cluster sampling means that random intervals of fixed length are traced. The number of cycles a test will run on the trace capture engine can be estimated by running it natively on a machine of the target architecture while counting the cycles the pipeline was busy using hardware monitors. The ratio between the total number of cycles and the number to be traced is then calculated and used as a parameter of trace capture. The simulator randomly switches between fast-forwarding and capture so as to maintain this ratio and, therefore, produce roughly the needed number of traced intervals.

Three choices affect the resulting trace size and accuracy.

1) Cluster size: is chosen so as to limit cold start effects to an acceptable value. In this study, the highest effect comes from cache warm-up similar to cache flushing at the beginning of the simulation. It causes additional processor pipeline stalls due to cache misses, the total number of which has the upper bound of roughly two megabytes per core. One potential worst case is when such misses occur at different times. For a pessimistic estimate of 100 cycles of stall per miss and 64-byte cache lines, this will sum up to roughly three millions of cycles. Another extreme case is when all misses occur simultaneously and slow down the memory controller; in the CMP configurations simulated in this paper, the time needed for the memory to pass through the corresponding amount of data ranged from one to five millions of cycles. For a one-percent level of processor speed measurement accuracy, the cluster size of 300 million cycles is therefore needed.

2) Cluster number: will define how representative of the program they are. As seen in [18], one hundred should be sufficient for an average accuracy of several percent, which is typical for best trace-driven models. In the experiments in this paper, only 20 to 30 intervals per test were sampled, since the model is not as accurate. Capturing one hundred of intervals, 300 million cycles each, on the proposed simulator and a modern host machine takes around a day (using one core) and around 100 GB of disk space (compressed on the fly using *gzip*), which translates to a day or two using 8–16 cores and several terabytes for a whole benchmark suite. As trace generation needs to be run only once, until the core's performance has significantly changed (due to instruction set, microarchitecture or compiler changes), this cost is acceptable and, importantly, independent of the length of the tests.

3) Cluster selection: Each interval is then evaluated for L3 cache miss rate using a simple cache model and, based on read and write miss rates, one most representative interval per test is selected. Choosing multiple clusters could be more accurate: in [19], by using two to ten clusters per test, the average error was reduced from 17 to 3 percent. As cache and memory access delays are the only delays added at the stage of trace-driven simulation, cache miss rate appears to be the most adequate criterion for interval choice. Single-core trace-driven uncore simulation could be used to compare the intervals in more detail; the time it will take will be still negligible. For tests that experience dramatic slowdown on a real machine or the CMP simulator due to uncore delays, the respective beginning parts of the clusters should be analysed during selection; otherwise, much longer CMP simulation will be needed to maintain accuracy.

As an alternative to "blind" cluster sampling, a sophisticated technique like SimPoints [19] could be used. First, a fast-forwarded run of a test could gather some execution statistics for each 300M-cycle interval, then, after comparing the statistics and selecting the most representative intervals, a second run would generate the traces. This would reduce the time and the storage space needed for the trace capture yet require an additional fast-forwarded run and, therefore, increase the overall run time for long tests nearly twice.

C. Multithreading support

Trace capture and sampling discussed above imply a single-threaded program. Multithreaded programs could be traced and simulated in a similar way. However, this is not an accurate solution [23] and if a full-system simulator is used for trace capture, this will probably limit the number of threads.

More accurate and unrestrained simulation of multi-threaded programs requires user-level trace capture, allowing to intercept system calls so as to control thread scheduling and OS-level synchronization during replay. Unfortunately, the architectural simulator available for the target architecture of

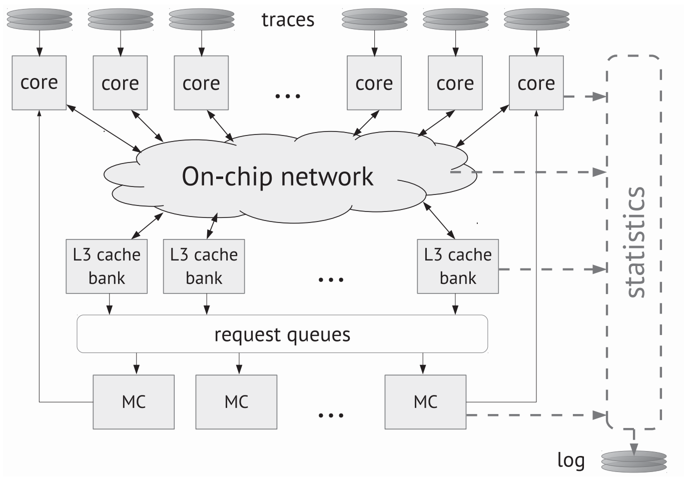

Fig. 3. The CMP simulator structure

this study currently supports multithreading only in full-system mode. When it is implemented in user mode, there is a way the trace-based framework described here can be updated.

SynchroTrace methodology [13] basically means capturing synchronization and communication events and adding a thread scheduler to the trace-driven simulator. Since the architectural simulator in this study is lower-level compared to the binary instrumentation-based used in the original work on Synchro-Trace and already uses a shadow memory, no significant performance drop in trace generation is expected.

For now, the CMP simulator is made with such future update in mind, while this paper basically considers single-threaded programs assigned to different cores. The extreme cases of well-scalable applications which use every core almost independently, and single-threaded programs using only one can thereby be examined. Section V shows how this approach may help in making design decisions.

D. CMP simulator

After trace samples are captured and representative subsets are chosen, they can be used for trace-based simulation of various configurations of the CMP. The CMP simulator includes trace replay engines at every core, and cache, on-chip network and memory controller models. It is a single-threaded cross-platform console application written in C++; the configuration of the simulation is defined by command-line parameters. Since it is architecture-agnostic, it could be used to simulate any in-order target architecture.

The overall structure of the simulator with main event processing paths is shown in Fig. 3.

E. Trace replay

Since traces are large, they are compressed during capture using *gzip*, which reduces the size each event takes on the storage drive from 12 bytes (in binary format) to approximately four bytes. To decompress traces, *zlib* library [24] is used; this reduces the simulation speed by roughly ten percent, which is acceptable. So as to reduce randomness in storage accesses, which might become a bottleneck when multiple kilo-core simulations are run in parallel on the same host machine, each trace replay engine has event queues (one for regular memory accesses and four for asynchronous prefetches used in the target architecture), each of them accessing the trace file assigned to the core only when it becomes empty so as to fetch thousands of events from the same file at once.

Every simulation cycle, if the core is active, the heads of its event queues are checked, and if the timestamp of an event matches current simulation time with some offset, it is executed as a memory request.

Based on in-order execution model, each core model contains variables indicating whether it is currently active (*no_stall*), when its pipeline will begin to stall due to dependencies if no current memory request completes (*t_stall*), when it will run again if it is inactive (*t_resume*), and the current timestamp offset, which is the sum of all previous stall cycles. These variables are updated in every simulation cycle based on memory request execution states. Private cache hits are executed instantly by the models described further, updating the three stall-related variables according to the expected cache hit latency (including the accumulated delays in the L2 cache bank pipelines due to conflicts).

The states of private cache misses are tracked by a model of miss status holding registers, which is basically a buffer for 64 requests. When a request is added into (on L2 cache miss) or deleted from it (when the request is complete), *t_stall* variable is updated accordingly. Additionally, to prevent dependency loss when a regular request is preceded by a prefetch (which updates the cache state instantaneously and may cause false hit), all private cache accesses check the buffer and update *t_stall* and the *delay* field of the prefetch request, if it is found. When the buffer is full, the core is considered inactive.

F. Cache models

As cache hit and miss latencies are modelled as described previously, it is possible to use the simplest behavioural models of the caches themselves. All private caches of a core (L1i, L1d, L2) are combined in one C++ class and use one method to access all of them simultaneously; similarly, a shared cache bank is combined with a directory slice in another class. A cache (bank) or a directory slice is modelled as three arrays: addresses and coherence protocol states for cache lines and replacement mechanism states for sets.

The access method returns hit or miss result, the requests addresses and opcodes in the case of cache miss and/or write-back needed; the same method call in the shared cache class also calls the invalidation method of the corresponding private caches on back-invalidation.

The coherence protocol used is MOSI with three versions of messaging for inclusive, non-inclusive or exclusive schemes of shared and private cache communication. Various replacement algorithms are implemented as separate functions, accessed through a function pointer of the class object, and include LRU, Pseudo-LRU, Random, Global LRU and others.

The L3 cache can be configured as private, distributed shared (S-NUCA) or hybrid, which is the simplest access time optimization [25]. Other algorithms will be added in the future.

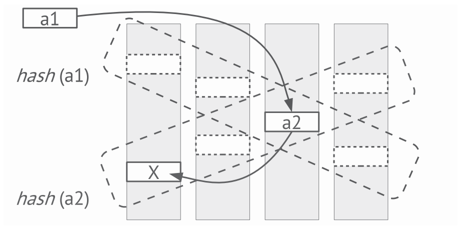

Fig. 4. Two-step replacement in ZCache

Being extremely simple and flexible, these models are easy to modify for design space exploration. For example, different partitioning schemes were implemented, including way-partitioning, Partitioning First and Futility Scaling [26] and skewed organization with ZCache replacement mechanism [27] (Figure 4). As shown in Section IV, even the most complex scheme is simulated at the same order of speed.

G. Network model

In a large shared-cache CMP, the on-chip interconnection network is responsible for communication between cores and shared cache banks and may, therefore, contribute to shared cache access time greatly. Therefore, network latencies must be modelled with good accuracy. Network throughput and fairness must be modelled for the same reason.

However, accurate simulation of every packet may become slow as the network size grows. In a 32x32 mesh network, which is modelled in the kilo-core simulations in this paper, a packet makes 21 hops on average, many of which involve arbitration among conflicting packets when traffic is high.

In the proposed CMP simulator, the on-chip network model was simplified thanks to the observations that the main traffic in a distributed shared-cache CMP is between cores and shared cache banks and such cache normally involves bank interleaving. Additionally, to stress the network, scenarios where all cores execute the same task can be simulated and if the network does not become a bottleneck, it can be expected to cope with smaller numbers of active cores as well.

Therefore, the traffic can be modelled as uniform random, with some realtime checks of the adequacy of such a model. The network model correspondingly needs to reproduce packet latency distribution as a function of packet injection rate (which will also determine throughput), maintain a desired degree of fairness and evaluate network load so as to detect and assess any inaccuracy.

1) Latency model: The proposed network model is based on input queues at each node, storing the packets, output queues storing the pointers to input packets and their estimated arrival time (based on the distance between the input and the output in the modelled topology), and state variables of every output (Fig. 5). Every cycle, unless the network throughput is exceeded, the simulator checks the state of every output and if there are ready packets, they are output and the corresponding queues and the output state are updated.

The initial intuitive idea was to use only one queue at each output to achieve the highest possible simulation speed. However, in this case, it would be difficult to either input or

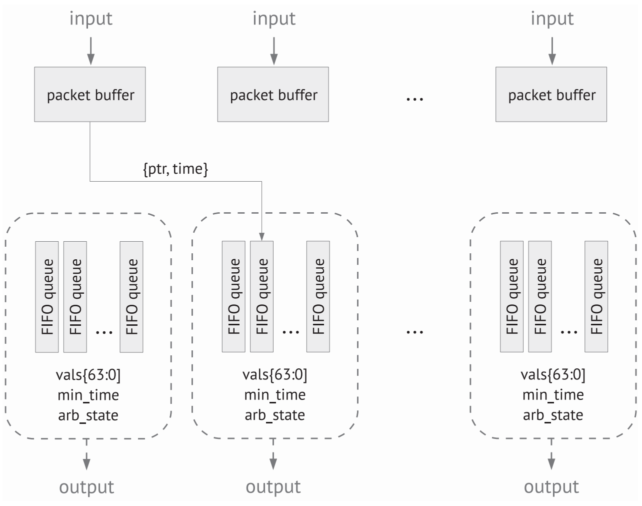

Fig. 5. The on-chip network model structure

output the packets so as to maintain their nominal arrival time, as the input and output times of packets hardly correlate. A less obvious problem would be the difficulty in modelling network unfairness like the "parking lot" effect common in mesh networks when the closer sources of packets are somewhat prioritized [28].

As a result, the decision was to use multiple queues at each output according to input-output distances of the packets. For a 32x32 mesh network, a packet traverses 0 to 62 channels (using any minimal routing), so 63 queues are needed. With this approach and even the simplest FIFO queues, the packets are readily sorted by their arrival time, so they can be simply added to the tail and checked and fetched at the head of the queues. Distance-related network unfairness can be modelled by adding an appropriate arbiter at the queue outputs; a simple round-robin was used in this study, assuming some fairness mechanism implemented in the network routers [29]. To speed up the check of 63 queues, their common state is maintained during every input and output operation:

- The array of their non-emptiness packed as a 64-bit integer, comparing which to a zero readily tells if there are packets in any queue;
- The minimal arrival time of the head packets, maintained not strictly (which would be slow);
- The arbiter state. In the case of round-robin arbiter, it keeps the number of the last granted queue.

Only when the first two 64-bit variables show the possible presence of ready packets at this output, the queue heads are checked; this check is performed usually quickly, as the first variable also tells which queues to check (and the arbiter state tells the order). If after all checks a ready packet is still not found, the second state variable is updated to the precise value (as all non-empty queues are checked in this case) so the next queue check will be successful.

Virtual channels can be modelled with separate network models using common thoughput limiters.

To simulate latency increase with traffic increase, adjustments can be made to packet arrival times. It will be explained and tested in Section IV.

2) Fairness: Network throughput distribution among its agents can be controlled not only by arbiters at the network outputs but also by a source throttling mechanism [30].

A simple mechanism was implemented. Packet injections are counted at every input. Every 1,000 cycles, the average number is calculated, and throttling coefficients for every input are computed as a function of this number and the number at the input. In the next 1,000 cycles, the injection at the input is blocked with the probability based on this coefficient.

This scheme is fast, independent of traffic pattern, network topology or other details. Like quality of service (QoS) mechanisms in real networks [29], it is able to reduce the throughput deviation under uniform loads to several percent.

3) Adequacy check: As discussed earlier, this approach must be adequate for uniform traffic typical for rarely interacting threads in a distributed-shared-cache CMP. It can be extended to non-uniform traffic without dramatic speed loss based on two facts for the network theory [28]:

- With deterministic routing, the network throughput under any specific traffic pattern is determined by one mostly used network channel;
- With adequate adaptive routing, network throughput is normally higher.

The solution, therefore, is to compute channel usage statistics assuming deterministic routing (XY in the case of mesh topology), which is trivial, and assess the possible slowdown the program could have experienced based on it.

The channel use statistics are checked and reset every 1,000 cycles. The number of possible additional CPU stall cycles is computed as the maximum number of excessive flits in a channel. Based on this, every 1,000,000 cycles, along with other simulation statistics, the accumulated slowdown and the maximum and average maximum channel use among 1,000-cycle intervals are written into the simulation log. If this estimated overall slowdown is significant (which is supposed not to be a very common case even in parallel applications), the program could be re-run using an appropriate network model of sufficient precision. Alternatively, wider channels can be tried for simulation to prevent channel overload.

In this study, such statistics are counted for the request subnetwork, it costs only a few percent of the speed and rarely showed overload in the experiments.

H. Memory controller model

There exists an accurate and extensible DRAM simulator that shows acceptable speeds, called Ramulator [31]. However, as the proposed CMP simulator was mainly created to evaluate future CMPs, a simpler model was implemented in it. Firstly, the exact specifications (even frequencies) of future memories are not known, so the simulation cannot represent future systems very accurately anyway. Secondly, as the number of threads increases, memory performance tends to that under random traffic [32], which is simple to model.

The model is based on DDR4 controller design described in [33] and emulates bank interleaving, page switching delays and the main timings. Assuming the memory traffic to be relatively low, the model is not connected to the on-chip network but uses simple FIFO queues to emulate delays.

IV. Evaluation

A. Accuracy

First, the simulator was validated against a real eight-core machine of the target VLIW architecture. Table I shows its CPU configuration for *slow* and *fast* scenarios. On this machine, test execution time was measured using the Linux *time* command (*user* field) while other statistics were obtained from hardware monitors.

TABLE I. Target CPU configuration used for validation

Component	Configuration
Core	8 cores @ 1372 (slow) / 1000 (fast) MHz
L1i cache	128 KB, 4-way, 128 B line
L1d cache	64 KB, 4-way, 32 B line
L2 cache	512 KB, 4-way, 64 B line
L3 cache	8 banks x 2 MB, 16-way, 64 B line
Network	4-node ring, 1 request / 32 B data per cycle
RAM	4 ch x 2 GB DDR4 @ 1375 (slow) / 2000 (fast) MT/s

The CMP simulator had the same configuration in this section with the exception of the network topology of 4x4 mesh instead of a four-node concentrated ring, which causes negligible differences in the results, and 16 shared cache banks one megabyte each instead of eight banks of two megabytes, which is practically the same.

Relative execution speed was measured instead of instructions per cycle (IPC) and miss rates are in bytes per cycle instead of misses per kilo instructions (MPKI) as traces do not show instructions. Average values were calculated as geometric mean. Relative errors were calculated as:

$$err(a, b) = max(a/b; b/a) - 1.$$

SPEC CPU2006 benchmark suite was used with *train* input data, as *ref* versions show similar cache miss statistics yet run roughly ten times longer on the native machine, which had limited availability for experiments. One test (*462.libquantum*) was excluded as too short for good precision. One most representative trace interval of 300 million wide commands was executed for each test in the CMP simulator. For multi-threaded scenarios, single-threaded instances of each test were executed simultaneously on different cores.

The resulting single-core performance in *fast* configuration of the real machine and the CMP simulator relative to *ideal* architecture simulator is shown in Figure 6a.

In single-core scenarios with fast memory, the average error was 18 percent. This order of precision was expected since only one small interval of each test was used for simulation; by properly choosing several intervals, the error could be reduced significantly [19]. Additionally, the simulator tends to overestimate the performance as it doesn't model various minor delays occurring in real cores. On the one hand, most of them could be added in the model and it will not cause a major simulation speed loss. On the other hand, some part of this overestimation represents how the real microarchitecture could be improved by reducing these delays. Therefore, both kinds of improvements are better to be made together.

Then the CPU and memory frequencies were switched to *slow* mode, four tests were run simultaneously and the increase in execution time was measured, caused by the reduction of memory bandwidth available per core by eight times, increased memory access time and reduced L3 cache capacity per core. The resulting IPC reduction relative to the previous scenario is shown in Figure 6b. While single-core performance was overall overestimated by the simulator, its reduction was predicted with the average error of only 2.6 percent and the average accuracy improved to 15 percent.

Miss rates measured for L2 and L3 caches are shown in Fig. 7. They include all types of requests and were calculated in bytes per *ideal* execution cycle, assuming every request to be 64-byte, to represent the test's memory throughput requirements. The average relative errors in L2 and L3 miss rates are mostly dictated by tests with low miss rates and are, therefore, meaningless. The average absolute errors were 0.30 B/c and 0.25 B/c, respectively, which is within one percent of the corresponding maximum miss rates. Therefore, it can be concluded that caches are modelled adequately.

Overall, the measured errors correspond to the precision of the model and are sufficient for early design space exploration of future CMPs.

The four-core simulation stressed all the components but the on-chip network, which was small and had a good margin of throughput. It was therefore decided to compare the network model to an accurate RTL model of 16x16 mesh network under random traffic, which is the main pattern of traffic in the simulations. Since the CMP simulator has full and direct control over the network throughput and fairness based on packet counters, only packet latency needs to be examined.

As a harder challenge and more likely candidate for the future CMPs, an EVC-based design of the real network [34] was considered. The results of the comparison are shown in Fig. 8. *Sim* denotes the default model with a constant delay per hop of two cycles and a trivial implementation of throughput restriction by ejecting packets once in 4 cycles. The graph resembles that of a classical network, just as expected. Then the simple throughput limiter was relaxed twice and a counter-based one was added so as to achieve the necessary slope of the graph (*Sim-T*). Finally, an offset was added to the packet delays at network outputs as a function of current network throughput so as to eliminate the error on the rest of the graph *Sim-L*. As a result, the error of the model below the throughput limit was within one cycle.

B. Performance

Simulation performance was measured on the following host machine configuration used for all the experiments:

CPU: Xeon E5 2698 v3 (16 cores, 32 threads) @ 3.2 GHz

RAM: 4x16 GB DDR4-2133 CL12

SSD: 1024 GB Samsung PM981

HDD: 6000 GB Western Digital WD60EFRX

Traces were captured to the HDD, then their selected parts were stored on the SSD for CMP simulation. Judging by CPU load during simulation, the SSD offered sufficient speed.

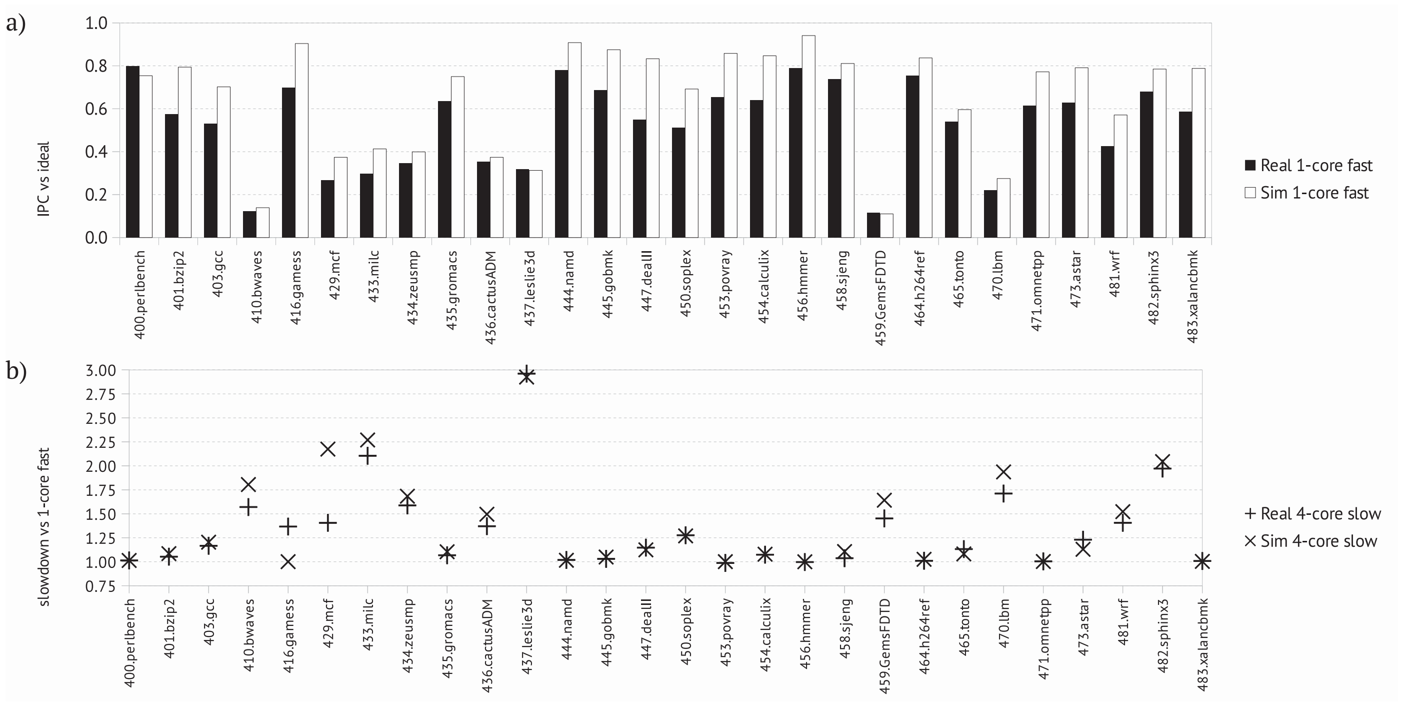

Fig. 6. Execution speed on the real machine and the simulator relative to the architectural simulator and its relative slowdown

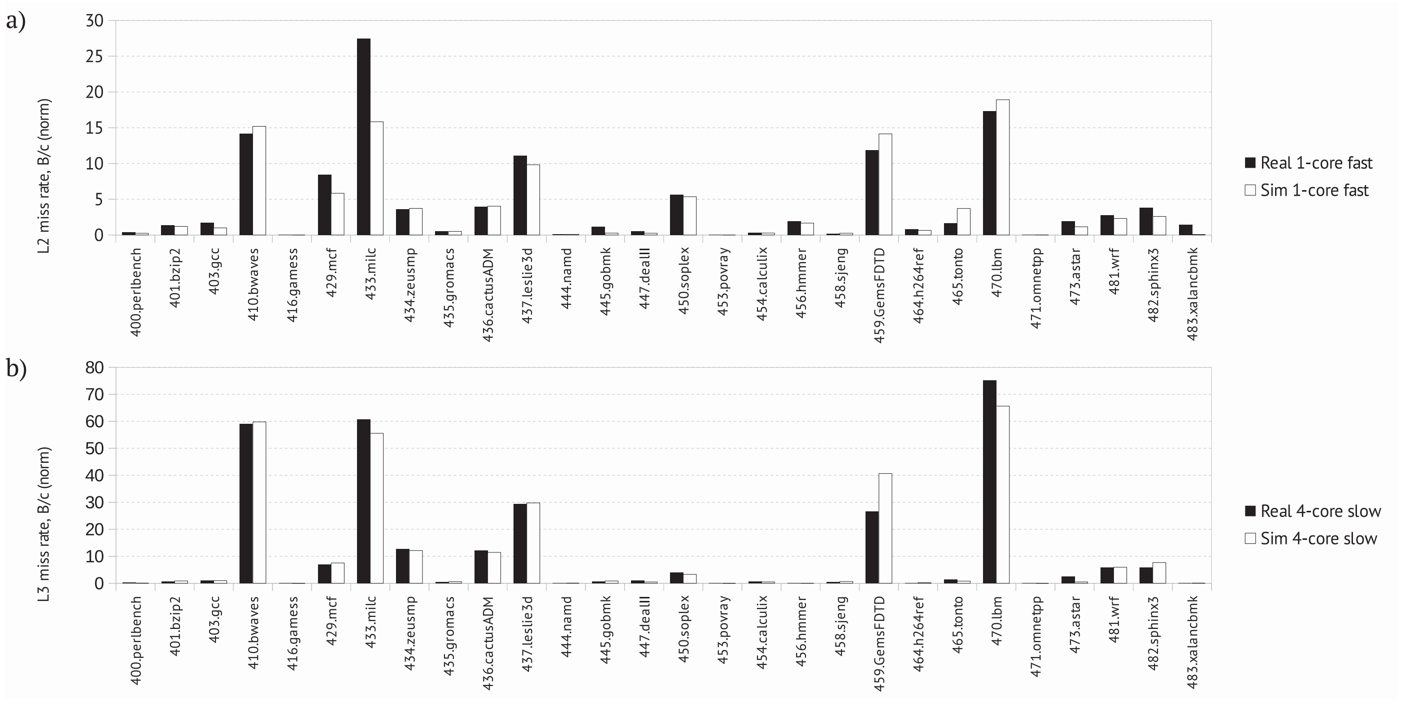

Fig. 7. Cache miss rate measured on the real machine and the simulator, normalized to cycles of the architectural simulator

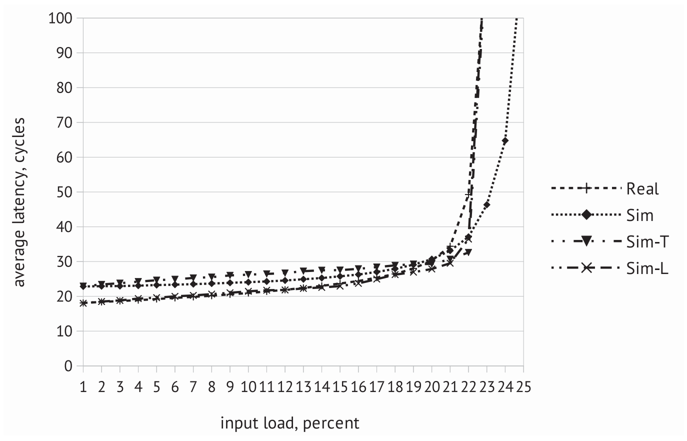

Fig. 8. Packet latency vs network throughput under random traffic

All speed measurements were performed by running 30 instances in parallel. Each instance consumed up to 2 GB of memory, so 64 GB of RAM were sufficient. With fewer instances, speeds are significantly higher due to Intel's hyper-threading, Turbo Boost and Smart Cache technologies.

Trace capture speed was difficult to measure precisely for this reason and due to very different test execution time. Overall, fast-forwarding runs with the speeds between 1 and 3 MHz and trace capturing has speeds of around 300 KHz, which is sufficient to complete the capture of most tests in a few days, in parallel. More of interest is the CMP simulator performance, which has to be run as many times per test as many CMP configurations are in the design space.

The simulation speed of a 1024-core CMP in two configurations is shown in Fig. 9. They included 1024 cores with 1 MB L2 cache, eight memory channels with 8:1 RAM:CPU frequency ratio, 32x32 mesh network, 1024 2 MB 32-column non-inclusive S-NUCA shared cache banks with either Partitioning First mechanism or ZCache and Futility Scaling with the associativity of 1024; the rest was the same as used previously. The directory was not modelled assuming its good implementation and therefore negligible effect [1]. The simulator was compiled with *g++* 4.6.2 after profiling on a 1024-core *410.bwaves* simulation for 20 million cycles.

Smaller CMPs are simulated accordingly faster, e.g. single-core configurations run at several MHz.

Using memory traces, the speed in instructions per second cannot be measured. However, assuming that *ideal* model executes one command per cycle (which is not far from true) and using the measured relative slowdown, the 1024-core model, on average, simulates 0.91 or 0.56 million commands per second depending on configuration. Since every wide command consists of four instructions, these speeds correspond to up to 3.6 MIPS. Running 30 simulators in parallel yields, therefore, up to around 100 MIPS in total.

V. Example of use: a kilo-core simulation

One of the major decisions needed to make while considering a CMP scalability is what measures must be taken to mitigate the increasing cache and memory latencies. A 32x32 CMP described previously was therefore simulated with one single-threaded program running and the shared cache configured as a private or hybrid (Fig. 10a). As the shared cache had 2 GB of total capacity which could not be warmed up during a short simulation (limited by 500 million cycles in this study), the hybrid scheme with always forced home-bank hit was also simulated, as if the whole program data was fully cached (which is likely). Lower and upper bounds on the hybrid scheme performance were thereby obtained.

The worst relative speed increased from 0.472 to 0.549...0.628. This result indicates that the hybrid scheme is not a silver bullet, the main priority in future CMP design must be to reduce network latencies. On the other hand, such slowdown is acceptable, so the idea of a single distributed shared cache for all cores is still viable. Basic S-NUCA scheme was also simulated and showed that roughly one percent gain comes from the hybrid optimization, so more effective cache latency optimizations need considering.

Another question that required experiments was shared cache associativity. It is known from the literature that a partitioning mechanism, necessary to implement cache QoS policies, may reduce the performance severely if the associativity in insufficient [27]. A simple Partitioning First mechanism was simulated with cache associativity of only 32, which was expected to perform poorly, and a Futility Scaling mechanism on top of the sophisticated ZCache scheme with the effective associativity of 1024. Surprisingly, the results showed practically no difference (Figure 10b). This could be caused by the non-inclusive L2-L3 communication scheme used. It needs further validation and analysis, but at least it is known that both schemes must be considered.

VI. Conclusions and future work

In this paper, we described a cycle-accurate trace-based simulator fast enough even to simulate kilo-core CMPs. Due to highly-accurate architectural model used for trace generation, it may take weeks for very large tests, but as it needs to be performed once for each program to simulate, this speed is just acceptable. More importantly, every single-threaded instance of the CMP simulator runs with speeds of several kilohertz, which, used with adequately selected trace intervals of a few hundred millions of cycles, allows to simulate a whole benchmark suite on a mainstream 16-core computer in just a few days. Comparing a number of different potential configurations of a large CMP thereby becomes possible.

The average accuracy of the simulation results is 15–18 percent. It is not comparable with the most precise techniques of this kind, yet is sufficient to get the overall picture of how the system will perform. Since our approach has no principal tradeoffs in simulation precision of the considered scenarios compared to other trace-based techniques, our future work will be to approach the precision of the most precise methods.

Although we have not implemented multithreaded program support by now due to the limitations of the architectural simulator, the SunchroTrace method, known to combine good speed and accuracy, appears to be compatible with our method. This is another avenue of our future work.

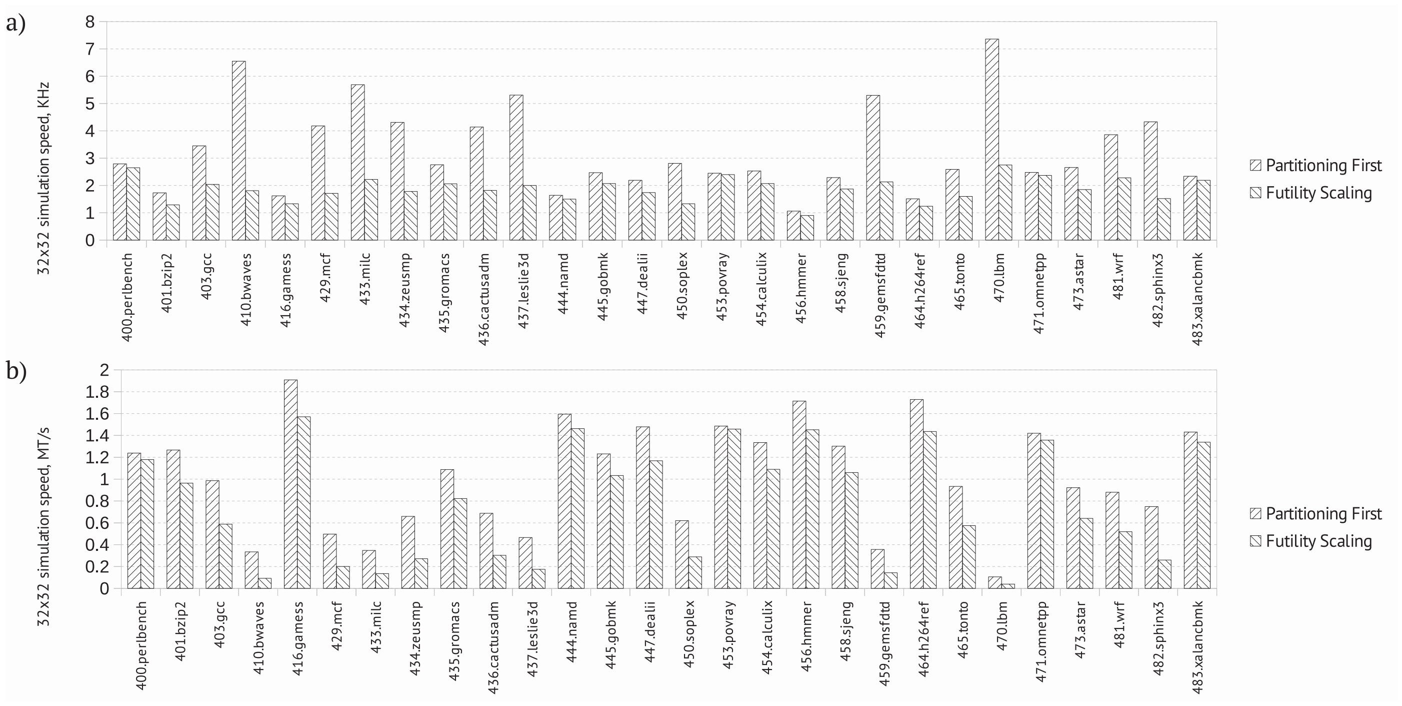

Fig. 9. 1024-core CMP simulation speed in: a) clocks b) memory accesses

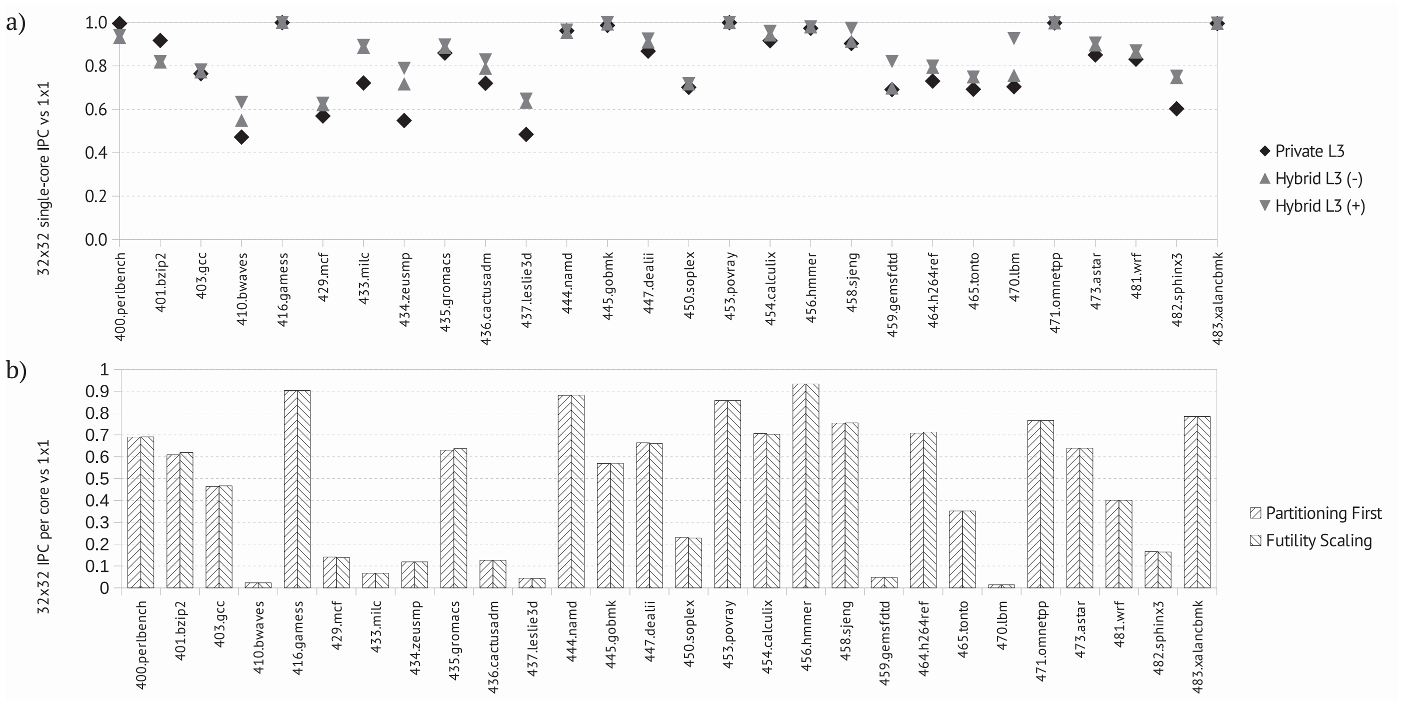

Fig. 10. Mean execution speed per core in a kilo-core simulation relative to single-core

References

[1] Yu.A. Nedbailo. Avoiding common scalability pitfalls in shared-cache chip multiprocessor design. In *2019 International Conference on Engineering and Telecommunication (EnT)*, Nov 2019.

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, and et al. The Gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):17, August 2011.

[3] A. S. Kozhin, N. Y. Polyakov, D. M. Alfonso, R. V. Demenko, P. A. Klishin, E. S. Kozhin, M. V. Slesarev, E. V. Smirnova, D. A. Smirnov, P. A. Smolyanov, V. O. Kostenko, F. A. Gruzdov, V. V. Tikhorskiy, and Y. K. Sakhin. The 5th generation 28nm 8-core VLIW Elbrus-8C processor architecture. In *2016 International Conference on Engineering and Telecommunication (EnT)*, pages 8690, Nov 2016.

[4] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 112, Jan 2010.

[5] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *SC 11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1 12, Nov 2011.

[6] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA 13, page 475486, New York, NY, USA, 2013. Association for Computing Machinery.

[7] R. Jagtap, S. Diestelhorst, A. Hansson, M. Jung, and N. When. Exploring system performance using elastic traces: Fast, accurate and portable. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 96 105, July 2016.

[8] Joel Hestness, Boris Grot, and Stephen W. Keckler. Netrace: Dependency-driven trace-based network-on-chip simulation. In *Proceedings of the Third International Workshop on Network on Chip Architectures*, NoCArc 10, pages 3136, New York, NY, USA, 2010. ACM.

[9] Y. S. Huang, Y. Chang, T. Tsai, Y. Chang, and C. King. Attackboard: A novel dependency-aware traffic generator for exploring noc design space. In *DAC Design Automation Conference 2012*, pages 376381, June 2012.

[10] C. Nitta, K. Macdonald, M. Farrens, and V. Akella. Inferring packet dependencies to improve trace based simulation of on-chip networks. In *Proceedings of the Fifth ACM/IEEE International Symposium*, pages 153160, May 2011.

[11] F. Trivino, F. J. Andujar, F. J. Alfaro, J. L. Sanchez, and A. Ros. Self- related traces: An alternative to full-system simulation for NoCs. In *2011 International Conference on High Performance Computing Simulation*, pages 819824, July 2011.

[12] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero. Trace-driven simulation of multithreaded applications. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pages 8796, April 2011.

[13] Karthik Sangaiah, Michael Lui, Radhika Jagtap, Stephan Diestelhorst, Siddharth Nilakantan, Ankit More, Baris Taskin, and Mark Hempstead. SynchroTrace: Synchronization-aware architecture-agnostic traces for lightweight multicore simulation of CMP and HPC workloads. *ACM Trans. Archit. Code Optim.*, 15(1), March 2018.

[14] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA 03, page 8497, New York, NY, USA, 2003. Association for Computing Machinery.

[15] E. Deniz, A. Sen, B. Kahne, and J. Holt. Minime: Pattern-aware multicore benchmark synthesizer. *IEEE Transactions on Computers*, 64(8):22392252, Aug 2015.

[16] K. Ganesan, J. Jo, and L. K. John. Synthesizing memory-level parallelism aware miniature clones for SPEC CPU2006 and implantbench workloads. In 2010 *IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 3344, March 2010.

[17] K. Ganesan and L. K. John. Automatic generation of miniaturized synthetic proxies for target applications to efficiently design multicore processors. *IEEE Transactions on Computers*, 63(4):833846, April 2014.

[18] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, pages 468477, Oct 1996.

[19] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 4557, New York, NY, USA, 2002. Association for Computing Machinery.

[20] P.A. Poroshin and A.N. Meshkov. An exploration of approaches to instruction pipeline implementation for cycle-accurate simulators of Elbrus microprocessors. In *Proceedings of the Institute for System Programming*, pages 4758, 2019.

[21] A. Limaye and T. Adegbija. A workload characterization of the SPEC CPU2017 benchmark suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 149 158, April 2018.

[22] P. A. Poroshin, A. N. Meshkov, and S. V. Chernyh. Development of simulator with support of cycle-accurate simulation mode on base of the existing instruction set simulator of the Elbrus architecture. *Voprosy radioelektroniki*, 2:6975, 2018.

[23] C. B. Stunkel, B. Janssens, and W. K. Fuchs. Collecting address traces from parallel computers. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, volume i, pages 373383 vol.1, Jan 1991.

[24] http://zlib.net/.

[25] A.S. Kozhin and Yu.A. Nedbailo. Methods of shared cache access latency optimization in chip multiprocessors. *Voprosy radioelektroniki*, 3:2732, 2017.

[26] Ruisheng Wang and Lizhong Chen. Futility scaling: High-associativity cache partitioning. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 356 367, Washington, DC, USA, 2014. IEEE Computer Society.

[27] Daniel Sanchez and Christos Kozyrakis. The ZCache: Decoupling ways and associativity. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 43, pages 187198, Washington, DC, USA, 2010. IEEE Computer Society.

[28] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[29] Yu.A. Nedbailo. On-chip network design for prospective chip multiprocessors. *Trudy MFTI*, 9 (2):151163, 2017.

[30] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via source throttling: A configurable and high-performance fairness substrate for multicore memory systems. *ACM Trans. Comput. Syst.*, 30(2):7:17:35, April 2012.

[31] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A fast and extensible DRAM simulator. *IEEE Computer Architecture Letters*, 15(1):4549, Jan 2016.

[32] Yu.A. Nedbailo and Igor A. Petrov. Increasing DDR4 SDRAM throughput in parallel workloads. In *2020 Moscow Workshop on Electronic and Networking Technologies (MWENT)*, Mar 2020.

[33] I. Petrov. Development of memory controller for todays Elbrus microprocessors. *Radio industry (Russia)*, 29:4147, 08 2019.

[34] Amit Kumar, Li-Shiuan Peh, Partha Kundu, and Niraj K. Jha. Express virtual channels: Towards the ideal interconnection fabric. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA 07, pages 150161, New York, NY, USA, 2007. ACM.