Implementation of Memory Subsystem of Cycle-Accurate Application-Level Simulator of the Elbrus Microprocessors

Pavel Poroshin
Department of Verification and Modelling
INEUM, MIPT
Moscow, Russia
poroshin_p@mcst.ru

Dmitriy Znamenskiy
Department of High-Performance Microprocessor Engineering
MCST
Moscow, Russia
znamen_d@mcst.ru

Alexey Meshkov
Department of Verification and Modelling
MCST, INEUM
Moscow, Russia
alex@mcst.ru

***Abstract*—Performance characteristics of any modern microprocessor largely depend on its memory subsystem. Naturally, the memory subsystem software model is an important component of the cycle-accurate simulator, and its validity and quality have high impact on the overall accuracy of the simulation. In this paper the cycle-accurate application-level simulator of the Elbrus microprocessor family is introduced. The structure of the cycle-accurate simulator is briefly explained. After that the software model of memory subsystem and its integration as a part of the cycle-accurate application-level simulator are described. We evaluate accuracy of the application-level cycle-accurate simulator on the SPEC CPU2006 benchmark and analyze the simulation errors. Finally, a brief comparison of different Elbrus architecture simulators is given.**

Keywords—Elbrus Architecture, Memory Subsystem, Cache Memory, Cycle-Accurate Simulator, Microprocessor, Application-Level Simulation, SPEC CPU2006

I. Introduction

As complexity of modern microprocessor and compiler optimizations increases, it becomes almost impossible to predict performance of application without actually running it. This is why cycle-accurate simulators are important, especially during development of hardware.

To be useful, a cycle-accurate simulator should give reasonable approximation of behavior and timings of real hardware, therefore it should be reasonably accurate. And a proper simulation of memory subsystem is a major factor in overall accuracy of simulator.

Due to high complexity of the simulator itself, it is not only important to have a working simulator, it is also important to have means to debug and evaluate it.

In this paper we describe our approach to integration of memory subsystem model into application-level cycle-accurate simulator of Elbrus microprocessors and give an overview of the tools that we developed to help to improve accuracy of this simulator.

The remainder of this paper has the following structure. Section II gives brief overview of Elbrus microprocessor architecture, its memory subsystem characteristics and pipeline specifics. Section III describes synchronous part of pipeline model and its interaction with memory subsystem model. Section IV describes the general design of memory subsystem model. Section V gives an overview of available debugging facilities for the simulator being developed. Section VI is dedicated to the simulator evaluation from the standpoint of its accuracy and performance. Section VII is dedicated to related work about simulator validation. Section VIII gives concluding remarks and briefly describes our plans for further work.

II. Prerequisites

A. Elbrus Architecture

Elbrus microprocessors have ISA (Instruction Set Architecture) of VLIW (Very Long Instruction Word) type. In such architectures, performance is achieved through the use of "Wide Instructions" (WI). Each WI consist of several sub-operations, which are executed by hardware in parallel.

Elbrus ISA implies in-order execution. Extraction of ILP (Instruction Level Parallelism) from a program algorithm is the responsibility of an optimizing compiler. This allows to reduce complexity of the hardware.

Each WI of Elbrus ISA can contain several arithmetic, logical operations or memory access operations, control transfer (CT) operations, operations on predicates and others.

B. Memory subsystem

Since Elbrus is a VLIW architecture, the memory subsystem accordingly has high ILP potential. Up to 4 memory load or 2 memory store operations can be placed in a single WI.

The cache hierarchy of the Elbrus microprocessors memory subsystem includes Instruction Buffer (IB, responsible for code fetching and generally acting as an instruction cache), data caches (L1D, L2, L3) and various address translation structures (ITLB, DTLB).

Additionally, Elbrus microprocessors have Array Access Unit (AAU), which is used for programmable asynchronous data prefetch, and CLW, which is used for automatic cleanup of data on stack. Both of them asynchronously generate requests to memory subsystem. There are also other components (for example TLU) that insert specific memory requests into synchronous memory access operations stream.

C. Pipeline Specifics

Pipeline of Elbrus microprocessors consists of the following stages (from early to late stages from the perspective of individual instruction): L, A, F0, F1, S, D, B, R, E0, E1 etc. First of them (from L to S) are in responsibility of Instruction Buffer. Stage R roughly corresponds to reading of operands, and stages starting from E0 are general execution

stages, including arithmetic, address calculation for loads and stores, etc.

Instruction Buffer incorporates several parallel code fetching pipelines. Special control transfer preparation (CTP) operations can be executed to start fetching code for the upcoming control transfers using dedicated preparation pipelines, and a prepared control transfer operation can switch the main pipeline to the chosen preparation pipeline. This mechanism is used for hiding latency of some control transfers.

Several types of pipeline stalls can be distinguished. Each stall type induces different pipeline reaction.

The first stall type is *regular*. The stalled WI is not progressing on the pipeline until the stall condition is resolved.

The second type of stalls includes *B-Stalls* and *L-Stalls*. The most frequent reason for such stalls are unavailable operands. Pipeline logic detects such situations with some delay, that is why simply stopping pipeline progression for the instruction is not enough. Instead, current results of instruction are discarded and for the next ticks instructions are transferred back to the R stage (from E2 for B-Stall and from E0 for L-Stall) until all affected instructions are placed back into the proper order. There is special pipeline logic for cases when one such stall happens during another one. In most cases, one round of B-Stall effectively adds 4 ticks and L-Stall adds 2 ticks of latency.

Even if only one operation of WI triggers stall condition, the whole WI is affected.

When WI passes stage E2, it can not be stalled anymore and effectively continues its execution without interruptions.

III. Synchronous Pipeline Model

A. Overall Design

The cycle-accurate simulator described in this paper is based on functional application-level simulator of Elbrus microprocessor and shares most of the code base with it.

Both the architecture state and the algorithmic behavior are handled by the functional component of simulator. This includes decoding instructions, maintaining state of memory and registers, simulating effects of executed instructions, emulating system calls, etc.

Also, the cycle-accurate simulator reuses some of the facilities of functional simulator, such as internal cache of decoded instructions [1], logging system, command line options parsing, events system, various configuration mechanisms, etc.

To achieve higher simulation performance and at the same time obtain desirable levels of accuracy and maintainability we implement pipeline model, described in [2] as "Hybrid" pipeline model. General idea of this approach is to simulate timing behavior of instructions by as large continuous chunks of logic as possible, potentially with use of some educated predictions about future behavior. For inevitable situations when predictions are false, simulator maintains additional data for making necessary corrections and does not allow propagation of incorrect speculative behavior to irreversible state.

For example, we assume by default that operations will not be stalled and on this basis we speculatively register availability of the results of the operation according to this assumption. But in case if the operation is actually stalled, the simulator corrects the earlier made assumption and recalculates the moment when results of the operations will be available.

In our case large chunks of timing logic correspond to continuous sequences of pipeline stages that are executed without interruptions.

The algorithm of simulation loop can be summarized by the following steps:

- For the new instructions, do simulation of its algorithmic behavior using functional component of the simulator, saving necessary info for cycle-accurate part of simulation in the process.
- Place this instruction into the pipeline model and pass necessary additional info about its execution.
- Iterate through each pipeline stage and for each stage determine which instruction is at this stage. If it is the first stage of continuous uninterrupted sequence - speculatively simulate all stages of this sequence.
- Update pipeline state (which instructions are at which stage), taking into account possibly occurred stalls.

B. Support of Memory Subsystem

Memory subsystem plays a major role in overall performance of the system. It is important to accurately simulate its effects.

Memory subsystem of Elbrus microprocessors is rather complex, so instead of implementing its model from the ground up we adopted the model described in [3]. From the perspective of the pipeline model, this memory subsystem model is regarded as black box with clear but limited interface. This helps us to achieve higher levels of modularity and limit influence of design of cycle-accurate simulator on memory subsystem model so it can be used in several projects more easily.

Model of memory subsystem implements IB (Instruction Buffer), all data caches (L1D, L2 and L3) and MC (Memory Controller). Functional component of the simulator does not directly interact with the memory subsystem model, and the cycle-accurate component directly interacts with IB and L1D by regularly (each tick) forming and passing input to them.

Because the described simulator is application-level, there are no OS effects and no proper memory management. The consequence of this is that there is effectively no virtual address translation takes place during simulation, and memory subsystem functions as if all memory accesses are physically addressed.

General architecture of the cycle-accurate simulator is presented on Fig. 1.

1) Memory Access Operations

As rather isolated component of the simulator, memory subsystem model does not implement speculative features of the pipeline model. Moreover, because its high complexity it does not support rollback of the state that happens during

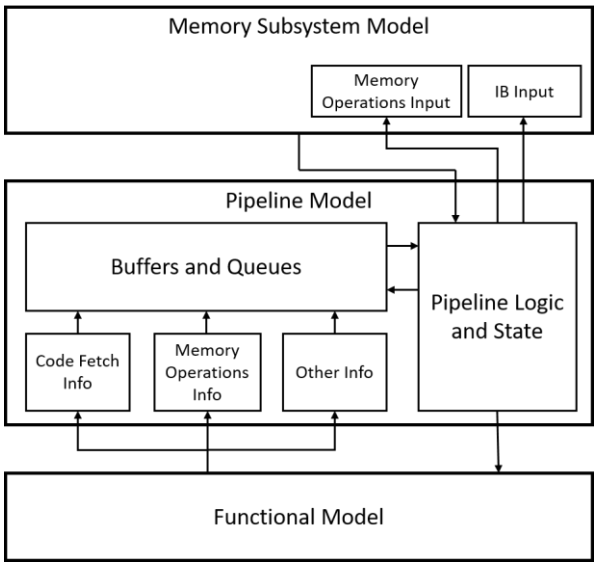

Fig. 1. General architecture of described cycle-accurate simulator

some of the stalls (specifically, L-Stalls that affect stages R and E0, and B-Stalls that affect stages R, E0, E1 and E2). These facts should be taken into consideration during integration of the model into cycle-accurate simulator.

Memory subsystem model is designed as cycle-by-cycle model and expects that its main simulation step should be executed once every simulation tick. In our approach this happens in the main simulation loop body of the pipeline model. From the perspective of the pipeline model, the specific moment when this can happen must satisfy following conditions:

- It must happen before other operations that are currently on the pipeline check availability of operands, otherwise there would be excess stalls. This check happens during simulation of the R stage.
- It must not happen after the most recent instruction with memory access operations (which should be send to memory subsystem model next) irreversibly (non-speculatively) reached earliest stage of possible feedback from memory subsystem that should be immediately acted upon. This corresponds to stage E2 when earliest possible data return from L1D cache can happen, and which should be taken into account during checking availability of operands on the same tick.

This means, at least from perspective of the pipeline model, that it is possible to send new memory access operations and execute memory subsystem simulation step anywhere between simulating (non-speculatively) stages R and E2 of instruction. But from the perspective of memory subsystem model, the earlier it can start processing new operations, the better, as this gives to the model more simulation steps to process each individual operation and probably will require less changes to original memory subsystem model. So in our final approach the information about memory access operations of instruction is sent and simulation step of memory subsystem model is executed just after processing stage E0 of this instruction just before processing stage R of other instructions.

Because memory subsystem model does not support "speculative" features of the pipeline model, and there is no means to make corrections of the model's state, new memory access operations should be sent to model only when it is guarantied that there will be no corrections that can change the data related to these operations. In our current model, the information about memory access operations is mostly determined by the functional component of simulator, which is not affected by "speculative" features of the pipeline model, so this requirement is fulfilled.

Other aspect to consider is the lack of support of rollback during some stalls. This means that in case of rollback of the instruction actions, sending of the memory access operations of this instruction to the memory subsystem model should be postponed until the next opportunity after the rollback. This also means that rollbacks should be known with certainty before (or at least at) stage E0, which is true for our current model.

Information about memory access operations is gathered during functional simulation at the start of processing of instruction. This information includes type of operation, memory address, destination register number (for load operations) and various other attributes. After functional simulation, this data is saved in the pipeline model alongside with other information about instruction and is used to form request to memory subsystem model when the time comes.

There can be several memory operations in-flight at the same time, and to distinguish them each memory access operation is associated with a unique token (internally represented by an integer). Because memory access operations can take many ticks to complete, response from the memory subsystem model can arrive when relevant instruction is no longer present in pipeline model (all instructions that pass stage E2 are deallocated from the pipeline model as there are no more stalls that can affect it and most of the timing behavior is certain and speculatively simulated). To address this, pipeline model implements additional buffers that hold the information necessary for proper action on response from the memory subsystem.

2) Code Fetching

Memory subsystem model also simulates the Instruction Buffer, which handles fetching of instructions from memory.

Process of instruction fetching is divided into several pipeline stages, but for simplicity they are not fully represented in the pipeline model. Pipeline model uses a special *pseudo stage* (internally named F) instead. New instruction is placed on this stage after its functional simulation. At the same time, on the same simulation tick the information about this instruction (mainly its address and size) is passed to the memory subsystem model. Instruction can leave pseudo stage F only after there was a response from the memory subsystem model about successful completion of the code fetching.

Some of the Elbrus control transfer instructions are separated into two parts: "control transfer preparation" (CTP) and "control transfer" (CT) instructions. CTP instructions initiate fetching of the code for the new path (which happens in parallel with regular fetch of the current path), and CT instructions implement control transfer to this code. This functionality allows to reduce latency of branches in some cases. This fetch is also a responsibility of IB and is simulated by the memory subsystem model. Information about CTP

operations is gathered during functional simulation and passed to the memory subsystem model in the beginning of processing of the instruction by the pipeline model (at pseudo stage F). When corresponding CT operation is executed, address of target instruction is passed to the memory subsystem model by standard mechanism (described earlier), and if IB has not fetched target instruction yet, then there will be a pipeline stall (at pseudo stage F).

3) Hardware Generated Operations

In some cases, hardware of Elbrus microprocessor issues special instructions (later referred to as "hardware instructions") that are not directly represented in binary code. Most notable of such hardware instructions are instructions which spill and fill register file contents to/from memory when active register window changes (for example, after procedural control transfers). Most of the hardware instructions perform memory accesses, and therefore play an important role in interaction of synchronous pipeline and the memory subsystem.

The functional simulator already simulates hardware instructions, as the triggering conditions for them are part of instruction set, and it is necessary to account for them to correctly maintain microprocessors visible state. But functional simulation of hardware instructions is significantly simplified, as the whole process takes place during simulation of ordinary instruction with a triggering condition occurring during one of the simulation steps (while one spill of register file can consist of many write accesses to memory).

Although functional simulation of hardware instruction is inaccurate from a timing standpoint, cycle-accurate simulation reuses most of it. During functional simulation information about all execution of hardware instruction is saved and passed to the pipeline model alongside with the instruction which triggered them. When this instruction reaches specific stage the saved information about hardware instructions is used to appropriately insert them into pipeline instead of next ordinary instruction. This continues until all hardware instructions are inserted.

It is possible to save information about hardware instructions in compact way. For example, although each spill (or fill) of register file usually consists of a sequence of many individual instructions, algorithm to generate them is predetermined and defined by several parameters. Also, all of the memory accesses of each individual spill (or fill) sequence share most of the attributes.

While ordinary instructions enter and leave the pipeline model in a FIFO fashion (enter at pseudo stage F and leave after stage E2), this is no longer true in the presence of hardware instructions. They are not fetched from memory (as they are not directly represented in binary code) and enter pipeline at stage R. This was not considered during original design of data structures of pipeline model and necessary changes were implemented.

IV. Memory Subsystem Model

A. Adaptation

For usage inside the application level (AL) cycle-accurate Elbrus architecture simulator, the memory subsystem timing model, which originated from the system-level memory subsystem (SLM) cycle-accurate simulator [3], underwent several changes.

According to the pipelining mechanics described above, code fetching from the IB cache and the instruction-word related memory accesses are processed on different pipeline pseudo stages. The timing model engine code was distributed between those pseudo stages, and separate input request queues for the code fetching stream and the memory access operations, with respective pipeline stalls, were created. It is worth mentioning that both parts of the model continue influencing each other through the “lower” part of the memory subsystem.

The precise WI data dependency checking and conflict resolution logic was implemented as a part of the core pipeline timing model. That is why a simplified scoreboarding mechanism utilized in the SLM cycle-accurate simulator was replaced by a set of fast callbacks, which report the memory operation processing status to the core pipeline timing model (e.g., a cache hit or a cache miss, MU internal queues overflow, etc).

The virtual address translation mechanics is not used for the AL cycle-accurate simulator. All virtual addresses are equal to physical addresses, and all MMU mechanics including page table search, ITLB and DTLB lookup are disabled. In addition, the L1D cache physical tags lookup mechanism, which is used as an anti-aliasing technique for virtual address cache indexing, is turned off. This gives an opportunity to simplify the memory subsystem model code and to increase its performance.

B. Improvements

Besides the adaptation changes described above, we significantly improved the memory subsystem model in terms of accuracy.

Multiple enhancements of IB, L1D, L2 and L3 cache models including a multitude of refinements for caches algorithms, latency calibrations and buffer depth adjustments were made.

For the shared L3 cache of a multicore processor, even for a single-threaded workload attached to a specific core, one should also consider influence of idle processor cores. Each of the idle-cores generates a sparse stream of memory accesses to the L3 cache. The combined idle-core stream can have a sufficient magnitude to interfere with the application core stream or at least to influence L3 cache performance metrics. To emulate the described background idle-core stream for the single-core configuration of the simulator a synthetic adjustable L3 cache access stream was introduced. This mechanism partially compensates idle-core traffic, and its precise adjustment is yet to be done.

Workloads like 410.bwaves from the SPEC CPU2006 benchmark [4] show strong performance dependency on the system memory characteristics like latency or throughput, which in turn can vary across different memory access patterns in different workloads. With a view of the simulator’s accuracy improvement, a model of DDR4 memory controller (MC), as a part of the memory subsystem timing model, was implemented. Current SLM and AL simulator configurations, which correspond to an 8-core Elbrus processor, include four MCs with the original interleaving scheme. The MC model was simplified for the sake of performance increase at the cost of some potential accuracy loss. Even with this tradeoff, adding MC model into the AL cycle-accurate simulator yielded additional 4% overall accuracy improvement on the

SPEC CPU2006 benchmark, with negligible simulator speed degradation.

V. Profiling and Debugging Facilities

It is important to have debugging tools in simulator, both for users of the simulator and for its developers. As more and more features are implemented (most notably accurate model of memory subsystem), complexity of possible mistakes in simulator rises, which demands for more advanced internal debugging capabilities of the simulator.

The simulator described in this paper provides two main debugging capabilities, namely execution log and execution statistics (profiling). Their design is described in this section.

A. Execution Cursor

Both execution log and execution statistics need the way to identify different events in simulator. For this purpose the concept of "execution cursor" is implemented in pipeline model.

Content of the execution cursor should point to the specific moment of simulation process, preferably both in terms of internal phases of simulation and in more simulator independent terms. In our implementation execution cursor specifies execution tick (when event should have happen on real hardware), simulation tick (when event was simulated, with respect to "speculative" nature of pipeline model), pipeline stage, IP of instruction, fetch tick of instruction, specific operation of instruction and internal simulator's description of operation.

Current execution cursor should be accessible from (almost) everywhere in the code. To achieve this "current" execution cursor implemented as a field of pipeline model object, which is accessible almost everywhere.

Logically execution cursor should have different structure for different events and parts of the simulation path. For example, some events (like pipeline stall on code fetching) should be attributed for whole instruction, but specifying one operation is meaningless for them. But for simplification and performance purposes, in our implementation the execution cursor always specifies whole range of attributes, possible with garbage values, but its users (various code pieces of the simulator) know which part of it is valid and which is not. For example, when a pipeline stall happens on fetching and appropriate counter of execution statistics is going to be incremented, it is known that operation related attributes of the execution cursor are undefined.

Also, the execution cursor should function like a stack, since most of the simulation process is stack-like (whole pipeline is at the bottom of the stack and specific instructions and operations are at the top). When simulation moves higher into the stack, relevant attributes of execution cursor should be updated, and when simulation returns back, the execution cursor should be restored as earlier updates are no more relevant. For performance reasons, restoration of execution cursor state is not implemented, as most of the time updated attributes are not valid in lower parts of simulation stack, so no information is lost. In a few cases where information is lost during update, saving and restoration of the execution cursor are implemented manually.

B. Execution Log

The main function of execution log is to present a detailed view of execution process and of microprocessor state evolution with respect to timing. Execution log includes information about ticks when each executed instruction reached certain pipeline stage, whether it was stalled, its stall reasons and etc. This information can be used to investigate performance anomalies and unexpected behavior.

Due to "speculative" features of the pipeline model, the execution log can not simply reuse standard logging facilities of functional simulator. For example, if availability of some register was speculatively but incorrectly updated, it would be wrong to include this update in the log. Therefore logging of any such event should be postponed until its correctness is certain. To support this, any intention to log an event is placed in execution log queue, indexed by tick number when event should actually (non-speculatively) happen. This queue is inspected every simulation step, and every event that is indexed by current non-speculative tick (“tick of no return”) and is still in the queue, is printed. If some speculation was wrong and a correction of speculative state is needed, then the queue of execution log is scanned and all wrong events are erased.

When an event is placed in the queue, current execution cursor is examined and all its information relevant for logging is placed in the queue alongside with the event. For example, when availability of a register changes, the information about current operation (which, in fact, is the reason for this event happening) is saved in the queue for a more informative output.

C. Execution Statistics

Other useful functionality of the simulator is the ability to gather statistics of execution. While the execution log is useful for analyzing specific moment of execution, execution statistics provide more convenient view of integral characteristics of the whole execution process. In other words data of the execution log is spaced in time domain and data of execution statistics are spaced in code domain.

As a tool, execution statistics collection is implemented as two main parts:

- Facilities (internal for cycle-accurate simulator) for accumulation and counting of interesting events and for forming (raw) output with accumulated data.
- Facilities for processing and transforming (raw) output with statistics into a more human readable format.

1) Accumulation of Statistics

Execution statistics collection in cycle-accurate simulator is represented as an in-memory map indexed by addresses of instructions (IPs). Each entry of this map contains some general counters for this IP (like number of executions, number of stalls etc.) and a map of event counters, indexed by event type and attributes.

All of the events that are currently being collected by simulator are always certain and can not be speculative, unlike events of the execution log. This allows to simplify current implementation of execution statistics, but it is always possible to apply solution from execution log implementation, if necessary.

For each IP, execution statistics accumulate the following information:

- Total number of executions, stalls, NOPs etc.
- Number of stalls of each type (for example, stalls caused by unavailable data in register file and bypasses, stalls caused by fetching of instructions, etc.).
- Number of procedural control transfers.
- Number of non-procedural control transfers.
- Some other statistics.

It is important to properly count pipeline stalls. For example, instruction can be stalled at stage B, but not because it has some reason itself, but because currently B-Stall (or L-Stall) is active, which blocks pipeline progression for instruction on stage B. Counting such stall not only misattributes stall for wrong instruction, but also counts effects of real reason for stall (reason of B-Stall) twice.

It can be useful to separate stalls not only by their general type, but also by some additional factors. For example, for every stall caused by unavailability of data in register file, there is a producer of this data and a consumer of this data, which is actually stalled. For this reason, statistics counters are indexed not only by type of event, but also by additional attributes, such as specific channel of instruction that was stalled, IP and channel of instruction that produces the result that leads to stall etc. For different types of events different types of additional attributes are used.

Counter of (non-)procedural control transfers are specific for each target. These counters are later used for construction of call graph and calculation of approximate inclusive costs of functions. Non-procedural control transfers are also essential for this, as some compiler optimizations (such as tail call optimization) transform procedural transfers into non-procedural ones. Information about non-procedural control transfers helps to detect such transformations.

2) Processing of Statistics

Although raw output of execution statistics is text based and can be analyzed directly without any additional processing, it is meant to be transformed into a more human readable format with some tools external to the simulator.

For this purpose a separate tool was implemented as a script with the following possible output options:

- Disassembly annotated with counters from raw output of execution statistics.
- Summary of counters, aggregated for functions (with non-inclusive aggregation across instructions of function body and with inclusive aggregation across instructions of all callees).
- Summary in the format compatible with the profile visualization tools such as KCachegrind [5].

Unprocessed execution statistics do not contain function names or disassembly. They are supposed to be obtained with objdump utility with support of Elbrus ELF files from binary executable file of the program being investigated.

The first step of execution statistics processing is its parsing and converting into a representation more suitable for further transformations. Then, a call graph is constructed, if necessary for chosen output.

As raw execution statistics for each control transfer counter contain only IPs of source (caller) and target (callee) of the transfer, it is impossible to construct a call graph which considers full call stack at the moment of transfer. To address this, during construction of a call graph a simple heuristic is used, which assumes that for each function all executions of its body are similar for all real call stacks. This assumption allows to calculate inclusive costs of functions by evenly distributing all counters of a callee across all individual calls from callers, starting from leaves of the call graph.

This method of inclusive function costs calculation does not work if a call graph has cycles, which are usually caused by recursive calls. To combat this, before calculation happens all cycles (more specifically, all strongly connected components) are detected and converted into special aggregated nodes of the call graph.

During construction of a call graph, not only procedural control transfers are considered, but also non-procedural ones. But not all non-procedural control transfers should be taken into account, as most of them are truly logically non-procedural. In our implementation, only non-procedural control transfers that cross function boundaries are converted into calls.

For visualization of execution statistics we use Kcachegrind program. It is mainly used for visualization of output from callgrind utility, but the input format is simple and flexible enough for visualization of other call graph like data. Our script supports output in KCachegrind compatible format with only aggregated information about functions and with more granular information about individual instructions. KCachegrind was built in support for viewing annotated disassembly, but not for the Elbrus disassembly. To achieve similar functionality, we utilize view of annotated source code, but in place of source code we use disassembly manually obtained with objdump utility with support of Elbrus binary format.

D. A Validation Case Study

As an illustration of the introduced simulator statistics usage, the authors present a simulator validation case study. During one of the validation iterations, a significant run time mismatch for the 444.namd workload (taken from SPEC CPU2006 benchmark) was found: the AL cycle-accurate simulation duration result was increased by 17%, while the cache hit rate differences stayed tolerable. The tasks' reference system and simulator profiles were collected using perf Linux profiler and the described AL simulator's statistical tools and methodology, respectively. The key profiling parameter was the total number of pipeline stalls. The profiles are shown in listings 1 and 2.

For the AL cycle-accurate simulator the relative weight of the hottest procedures calc_pair_fullelect and calc_pair_energy_fullelect has increased by 9% and 14%, respectively. Taking a closer look at the annotated disassembly of the hottest procedure calc_pair_fullelect revealed a significant stall increase (almost up to 9%) for an instruction at the 0x40020 address, which executed absolutely seamlessly on the reference machine. The perf-generated and

```
12,05%  ComputeNonbondedUtil::calc_pair_fullelect
11,45%  ComputeNonbondedUtil::calc_pair_energy_fullelect
 9,45%  ComputeNonbondedUtil::calc_pair
…
```

Listing 1. 444.namd perf-generated profile, reference machine

```
13,19%  +9,45%   _ZN20ComputeNonbondedUtil19calc_pair_fullelectEP9nonbonded <0x38a30>
13,00% +13,57%   _ZN20ComputeNonbondedUtil26calc_pair_energy_fullelectEP9nonbonded <0x448a0>
 9,53%  +0,79%   _ZN20ComputeNonbondedUtil9calc_pairEP9nonbonded <0x20a98>
…
```

Listing 2. 444.namd simulator-generated profile with relative function weight difference

```
5.02 |3ff78:
     |…
     | faddd,sm %db[94], %db[103], %db[62]
     |…
...
     |40020:
     |…
     | stgdd %r46, 0x0, %db[62]
```

Listing 3. 444.namd perf-annotated disassembly, reference machine

```
0x3ff78                                          # E=20510404 S=10408678 N=0 HW=0 LD=0 ST=0
  …                                              # IB_FETCH_NOT_READY = 178
  faddd,sm %db[94], %db[103], %db[62]            # RF_REG_NOT_READY <source IP 0x3f728> = 88
  …                                              # RF_REG_NOT_READY <source IP 0x3fb00> = 116
                                                 # RF_REG_NOT_READY <source IP 0x40428> = 877024
                                                 # RF_REG_NOT_READY <source IP 0x40450> = 9531272
…
0x40020                                          # E=20510404 S=66987408 N=0 HW=0 LD=0 ST=19440799
  …                                              # RF_REG_NOT_READY <source IP 0x3ff78> = 66987408
  stgdd %r46, 0x0, %db[62]
```

Listing 4. 444.namd simulator-annotated disassembly

the simulator-annotated disassemblies are shown in listings 3 and 4.

As we can see, the data for the store operation (%db[62]) in the 0x40020 instruction word is produced by floating-point addition instruction located at 0x3ff78. The consumer (store at 0x40020) is scheduled for execution with a 4-cycle latency after the operand producer (faddd at 0x3ff78), but it gets stalled very frequently because faddd's result is not ready. Notably, on the real hardware the consumer instruction does not stall at all. The authors checked the CPU's scheduling rules, which are a part of the Elbrus ISA. It turned out that the result availability latency of type {faddd → integer / memory} is 6 cycles, but for the special case {faddd → store data}, the latency is 4 cycles. Thus, the complier created an optimally scheduled code. Further examination revealed that this very special case was not taken into account by the simulator's logic. Fixing this inaccuracy shrinked the total run time error of the 444.namd workload to a negligible value (below 1%).

VI. Evaluation

A. Simulator Accuracy

Accuracy is one of the key simulator quality markers in a sense of reference and modelled system characteristics convergence. It can vary significantly across different classes of workloads (for instance, integer or floating-point workloads), and, moreover, across different workloads of the same class. As for physical microprocessor systems, for the cycle-accurate microprocessor simulator characteristics evaluation, a standard benchmark, which includes a variety of workloads from different classes and problem domains, should be used. The authors chose the standard cross-platform SPEC CPU2006 benchmark, which has been used for the Elbrus architecture microprocessors performance evaluation [6][7]. We compiled SPEC CPU2006 workloads with the Elbrus compiler toolchain developed by JSC MCST. It is essential that, for the sake of cycle-accurate simulator compatibility, some optimization features were disabled at compile time. For example, the usage of Array Access Unit was disabled because the latter is not yet appropriately supported in the timing model of the simulator. In other words, all the reference system and the simulator data presented below correspond to some intermediate performance mode sufficient for the simulator validation, but not to the peak performance mode.

The authors used a physical machine based on 8-core Elbrus CPU as a reference system for the AL cycle-accurate simulator accuracy benchmarking. The configuration of the reference system is presented in Table 1.

For most of the SPEC CPU2006 tasks the test input data set was used, and a smaller fraction of workloads was run with the train data set. This choice of input data sets was motivated by a reduction in simulator run time for those workloads, which in turn allowed the authors to speedup the simulator validation workflow loop that requires frequent task re-execution.

Fig. 2 and 3 present the main accuracy evaluation results for the AL cycle-accurate simulator, namely the task run time normalized by the run time on the physical reference system (Fig. 2), and the workload error distribution (Fig. 3). Workloads on Fig. 2 were run with the test input except workloads where the train input data is explicitly specified. The geometric mean (GM) is presented for benchmarks that include multiple runs. These results show that for more than a

TABLE I. REFERENCE SYSTEM CONFIGURATION

CPU	Elbrus, 8 cores, 1 node
Clock frequency	1500 MHz
L1 instruction cache (IB)	128 KBytes, 256-byte cache line, 4-way set-associative, virtually addressed
L1D cache	64 Kbytes, 32-byte cache line, 4-way set-associative, virtually addressed
L2 cache	512 Kbytes, 64-byte cache line, 4-way set-associative, 4-banks interleaved, physically addressed
Shared L3 cache	16 Mbytes, 64-byte cache line, 16-way set-associative, 8-banks interleaved, physically addressed
RAM channels	4 channels DDR4
RAM size	DDR4-2400, 128 GBytes
Operating system	OS Elbrus based on the Linux kernel
Linux kernel version	4.9.0-4.1-e8c2

half of the workloads the error stays inside a 4% limit, where the CINT tasks have, on average, a lower error in comparison to the CFP tasks. The vast majority of the tasks has a 12% upper bound with several outlying cases with 25% error at worst. The geometric mean for the error on the whole SPEC CPU2006 set is below 5%, which is an admissible result [8][9].

Memory subsystem is one of the most contributing sources of pipeline stalls, especially for a VLIW architecture like Elbrus since memory accesses can have sophisticated behavior that can be difficult to be predicted and tackled at compile time. In this context, the hit rate for different levels of the cache hierarchy is an important metric that correlates with the overall system performance, and matching of reference and simulator-originated hit rate values is an important simulator timing validation criterion. Fig. 4, 5 and 6 show hit rates for the L1D, L2 and L3 caches on the reference system and the AL simulator. For private core caches (L1D and L2) there is a high correlation between hit rates on the real and modelled CPU (mean hit rate error is 3% and 2% respectively). The situation is different for the L3 cache, namely the mean L3 hit rate error is noticeably higher (18%).

Modelling errors, which result in performance metrics value mismatches described above, can be arranged in a number of groups as a rule of thumb:

- Functional limitations of the cycle-accurate simulator, like hardwired CPU execution mode, simplified system configuration, etc.
- Logical inaccuracy in the timing model. E.g., redundant or insufficient pipeline stalls, wrong hardware algorithms implementation, etc. This group of errors should be diagnosed and fixed during cycle-accurate simulator validation (see 444.namd validation case study above).
- Different performance counters interpretation. As an example of this error group, one can consider a pipelined cache memory with an number of intermediate structures like Store Miss Buffer, Miss Status Hold Registers, etc. In-flight operations can be stored in these buffers in a multitude of internal states, which in turn can be interpreted in a different way by physical and modelled performance counters.
- Incremental cache hierarchy error propagation. If an error appears at some level of the cache hierarchy model, it tends to propagate further down the lower levels. For example, an illegal L1D cache miss causes a spurious L2 cache request, which alters the hole underlying memory subsystem behavior. That is why lower cache levels in the simulator have higher relative variation in request stream intensity and structure (and, as a result, a greater hit rate error) in comparison with higher level caches like the L1D cache.

When analyzing the error rate for the overall accuracy data on Fig. 2, one can notice several outliers with the highest error: 433.milc (test — 16%, train — 25%), 429.mcf (18%), 445.gobmk (14%) and 450 soplex (train – 13%). These cases are worth discussing while keeping in mind the modelling error classification presented above.

For 433.milc workload (both test and train input data), a relatively high DTLB hit rate is common. On the reference system, among the overall 7017 million memory accesses, DTLB misses comprise around 1%, or 70 million misses in total. This exceeds DTLB miss rate of most tasks considered in this paper by several orders of magnitude. Suppose each DTLB miss takes one memory reference for the page table lookup, and it always hits in the L3 cache (which is an optimistic estimate). The L3 cache hit latency takes a few tens of clock cycles, so the cumulative DTLB miss stall penalty will definitely exceed 1000 million clock cycles. The latter number fully explains the difference in real and modelled execution time. This is a simulator functional limitation case, namely the absence of address translation mechanisms and MMU cache lookups in the AL cycle-accurate simulator. One can say that the AL cycle-accurate simulator has both 100% ITLB and DTLB hit rates, which result in the absence of TLB miss-originated pipeline stalls.

For the AL cycle-accurate simulator, the 429.mcf and 450.soplex (train) tasks have a higher L1D hit rate than expected, which could result in erroneous task speedup. In both cases the reasons for the divergence are likely to be simulator logical inaccuracy, which are yet to be fully investigated. Also, there are some functional limitations of the simulator that tend to distort memory subsystem dynamics and complicate the analysis. Namely, on a real machine, the memory allocation is eventually done by the OS kernel memory management mechanism using processor's MMU hardware. Virtual memory is allocated in 4KB, 2MB or 1GB physical memory pages, which are often grouped into contiguous blocks by the allocation algorithm, where possible. Moreover, some virtual memory pages with distinct virtual addresses can be mapped to the same underlying physical memory page, e.g. for the copy-on-write and zero pages techniques used by the Linux kernel. The AL cycle-accurate simulator does not reproduce those algorithms. The

underlying physical memory is always available, and all physical addresses are identical to virtual, as noted above. This results in effectively different physical addresses footprint for the AL cycle-accurate simulator with respect to the reference system, hence different memory subsystem behavior starting from the L2 cache and below. We found some indirect markers of the limitation's impact. To begin with, we used the page-types Linux utility to get a glance on the virtual-to-physical address mapping of the 426.mcf's task on the reference machine, where only the referenced pages were considered. A physical address heatmap was collected for the cycle-accurate simulator, with physical memory region granularity of 4KBytes. We found that the physical memory footprint on the real machine is far more dense in comparison with the AL simulator, especially when it comes to usage of virtual pages mapped to the same physical page. Also, using aliased virtual pages can cause additional L1D cache misses from the anti-aliasing mechanism, which is not present in the AL version of the cycle-accurate simulator. Secondly, an interesting difference in the L3 statistics for the 429.mcf workload was found. The L3 cache has a special hardware structure for in-flight requests. A hit in this structure occurs when multiple in-flight L3 cache requests access the same cache line. In this situation some additional stalls can occur. The overall L3 request number on the simulator is close to the reference machine (with 20% difference), but the number of hits in the in-flight requests buffer for the reference machine is by three orders of magnitude greater. Thus, frequent physical address collisions on the reference system are not reproduced by the AL cycle-accurate simulator, which indirectly points to L2 cache model inaccuracy or to the address mapping limitation described above, or both.

The 445.gobmk workload with test input consists of several subtasks, which have very diverse numbers of instructions, from tens of millions to billions. "Short" subtasks tend to have high inaccuracy due to the reference machine run time variation and some simulator limitation effects, which are negligible for "long" workloads. Those "short" subtasks have high negative impact on the overall 445.gobmk workload accuracy. For the same reason, the 462.libquantum task was run only with the train input data.

The cache hit rate error data on Fig. 4, 5 and 6 can be also interpreted with the help of the introduced simulator error classification. Generally speaking, the private core caches (L1D and L2) hit rate errors are quite low except a couple of outliers, but the mean shared L3 cache hit rate error is greater. For private core caches (Fig. 4, 5) most high error cases are likely to be performance counter interpretation errors of the memory subsystem timing model (like 462.libquantum for L1D and 482.sphinx3 for L2). The L1D and L2 errors are almost uniformly distributed across workloads, and it is difficult to pick an evident error group for most of them. For the L3 cache the situation is different. Here errors of all four groups severely distort modelling results. Multicore reference configuration has idle core L3 traffic. Moreover, cache coherency mechanisms impact the L3 cache hit rate and alter L3 performance metrics meaning on the reference system. Incremental error propagation uniformly augments error margins across all workloads. Finally, some specific L3 cache model logical inaccuracies exist. For the 410.bwaves task, the hit rate is too high, since this workload is known as a streaming one with low L3 hit rate [6]. Nevertheless, high L3 hit rate error does not always severely influence overall results, since for most tests the memory accesses stream intensity falls when moving to the lower levels of memory hierarchy, especially for workloads that fit in higher level caches.

B. Simulator Performance

Running speed is an important cycle-accurate simulator characteristic. It directly affects the simulator validation rate and also the simulator's usability: the speed should be sufficient to run the standard benchmarks common for the modelled architecture. Here we present a brief speed comparison of the pure functional Elbrus architecture simulator (FUNC), the considered AL cycle-accurate simulator (AL) and the system-level memory subsystem simulator described in [3] (SLM). We conducted comparison on several benchmarks from the CINT and the CFP packages. Those benchmarks are chosen in such a way that their clocks-per-instruction (CPI) parameter varies significantly. We chose the clock-per-wide instruction (CPWI) characteristic measured on the reference system as a simple approximation of the CPI. Modelling results are shown in Table II.

Functional simulator speed is often expressed in terms of ticks per second, where a tick basically corresponds to an executed instruction. In case of comparison with the cycle-accurate simulators, choosing this performance measurement unit would create some confusion. Instead we expressed the functional simulator's speed in the same performance units as both cycle-accurate simulators' speed, notably in cycles per second, where the number of cycles was obtained from the AL simulator run for a particular benchmark. Table II shows that functional simulator's speed is between 5,081 MHz and 10,565 MHz, which is comparable to FPGA processor prototypes. The AL simulator speed varies between 462 KHz and 817 KHz, which is an order of magnitude slower than pure functional instruction set modelling. This is typical since detailed timing models overhead is usually quite severe.For the SLM simulator, the selected benchmarks were run under OS "Elbrus" based no the Linux kernel version 4.19.72. The speed is within the limits of 577 KHz and 1066 KHz. For both considered cycle-accurate simulators the speed tends to increase with the CPWI parameter. This can be explained by simulators' design: during the stall cycles processing, only a smaller fraction of pipeline and memory subsystem model mechanisms are active, and such kind of cycles usually run faster. On average, the SLM simulator is by 13% faster than the AL simulator. For the considered CFP workloads the geometric mean speed difference is 4%, and for the CINT workloads is 21%. The gap gets narrower for lower CPWI tasks (453.povray and 471.omnetpp), and wider for higher CPWI tasks (410.bwaves and 429.mcf). The latter means that the full accurate Elbrus core pipeline timing model overhead of the AL cycle-accurate simulator is greater than the SLM simulator's overhead imposed by the MMU model, lightweight memory scoreboarding and the OS code simulation time combined together. Yet there is some room for optimization (see "VIII. Conclusions and Future work"). To summarize, simulators' running speeds are quite comparable to each other and acceptable for running the SPEC benchmark package with test and train input data sets: running a 40-billion cycle benchmark, takes roughly a 24-hour period.

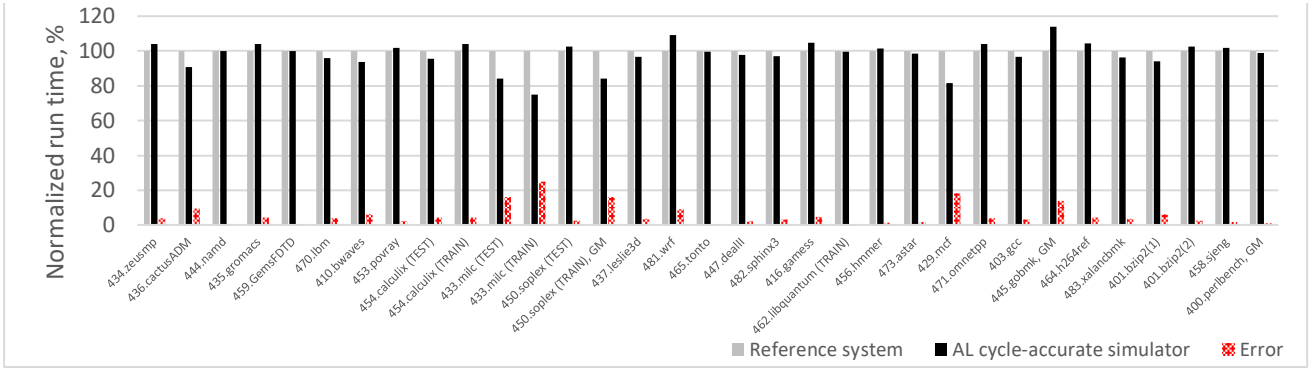

Fig. 2. SPEC CPU2006, AL cycle-accurate simulator accuracy

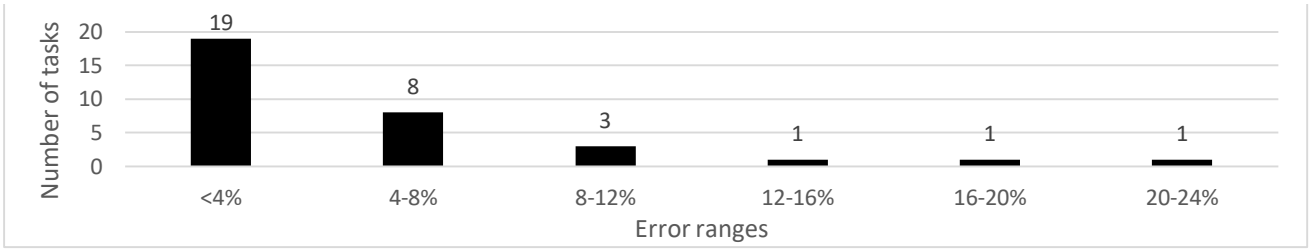

Fig. 3. SPEC CPU2006 workload error distribution

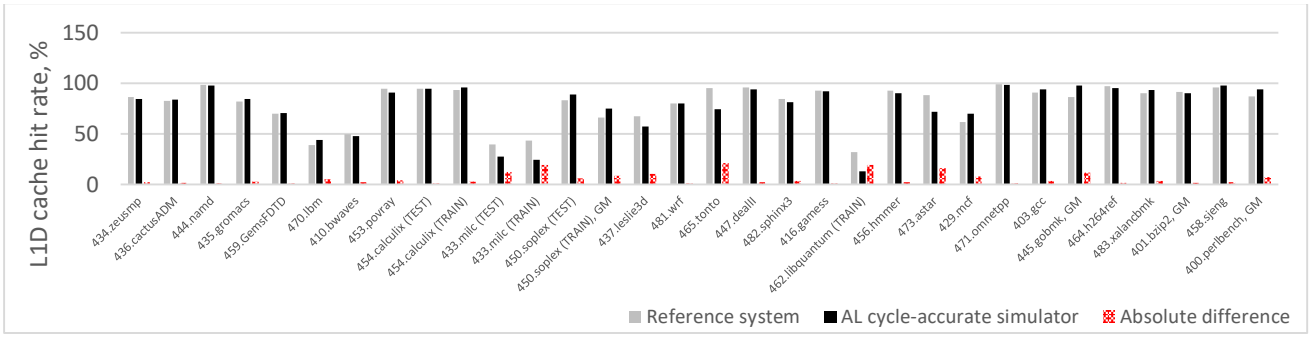

Fig. 4. SPEC CPU2006 L1D cache hit rate

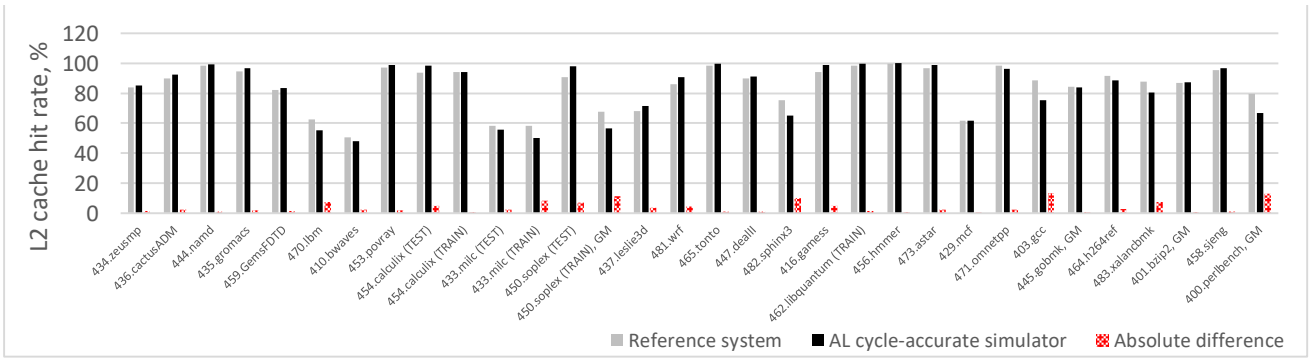

Fig. 5. SPEC CPU2006 L2 cache hit rate

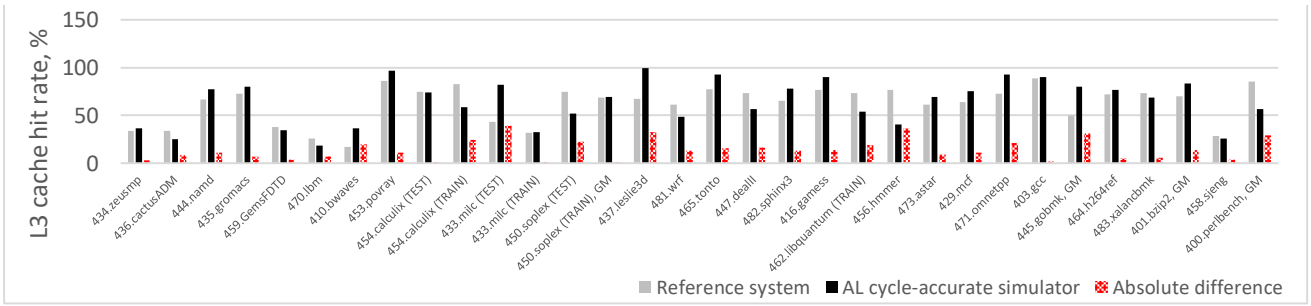

Fig. 6. SPEC CPU2006 L3 cache hit rate

Table II. The AL cycle-accurate simulator and the SL memory subsystem simulator speed comparison

Workload		CFP			CINT		
		453.povray	**434.zeusmp**	**410.bwaves**	**471.omnetpp**	**462.libquantum**	**429.mcf**
CPWI		1.6	3.1	6.6	1.7	3.5	4.6
Machine		Intel Core i7-2600 CPU @ 3.40GHz, 16GB RAM		Intel Xeon E3-12xx v2 (Ivy Bridge), 32GB RAM			
Simulator speed, MHz	FUNC	5.081	4.152	9.442	3.579	3.361	10.565
	AL	0.796	0.670	0.817	0.462	0.609	0.795
	SLM	0.734	0.699	0.965	0.577	0.733	1.066
Speed (FUNC) / speed (AL)		8.37	8.43	12.67	6.30	5.43	13.77
Speed (AL) / speed (SLM)		1.09	0.96	0.85	0.80	0.83	0.75

VII. Related Work

It was demonstrated by [10] that proper validation and calibration of simulator is very important for its accuracy. Accuracy evaluation made during this study showed that the least error in accuracy is achieved by simulators Sniper and ZSim, which have been validated against real hardware. It is also worth mentioning that these simulators are most accurate despite being of application level type.

Simulator Sniper was validated [11] against Intel Nehalem architecture and was shown to achieve average error of 11.1% on the SPLASH-2 benchmarks [12]. This high accuracy preserved (with reasonable variation) during later evaluation conducted in [10] on SPEC-CPU2006 [4] and MiBench [13] benchmarks targeting Intel Haswell architecture.

Simulator ZSim was validated [14] against Intel Westmere architecture. It achieved absolute IPC error of 8.5% on SPEC-CPU2006 [4] benchmark (single-threaded benchmarks) and .11.2% on combination of PARSEC [14], SPLASH-2 [12] and SPEC-OMP2001 [15] benchmarks (multi-threaded benchmarks). Absolute error for Intel Silvermont configuration of simulator is 20.9%, which is roughly twice as large comparing to Intel Westmere configuration. This emphasizes importance of validation and calibration against specific target hardware.

Simulator SiNUCA was validated [9] against Intel Core 2 Duo and Intel Sandy Bridge microprocessors. Validation was conducted on two sets of benchmarks: specially constructed micro-benchmarks and SPEC-CPU2006 suit [4]. Micro-benchmarks were constructed to evaluate specific features of microarchitecture and interactions of different components of microprocessor. On these micro-benchmarks simulator achieves average error of 10% for Intel Core 2 Duo and 6% for Intel Sandy Bridge, while on SPEC-CPU2006 it achieves average error of 19% for both hardware configurations, which is significantly larger than for microbenchmarks. This error increase reaffirms importance of selection of representative workloads during simulator accuracy validation.

VIII. Conclusion and Future Work

Cycle-accurate simulators are very useful and important class of tools for investigating performance of applications and for exploring and evaluating design space of processor microarchitectures. And memory subsystem simulation is major part of any cycle-accurate simulator.

But, as microprocessors become more and more complex, simulator complexity grows accordingly. That is why it is important not only to implement working software model of microprocessor, but also to have means to debug it and make it more accurate.

In this paper we described our approach to integration of memory subsystem model into application-level cycle-accurate simulator of Elbrus microprocessors and the tools we implemented and used to debug this simulator.

However, while the simulator described in this paper is reasonably performant and accurate (with mean running time error below 5%), there is always more work to be done.

Firstly, there are still some features and components of microprocessor that are not yet simulated with sufficient accuracy. Most notably, simulation of Array Access Unit (AAU) is still very simplified. Part of our future work is to properly implement more components. Also, one can always find bugs and inaccuracies that should be fixed.

For the memory subsystem timing model specifically, the authors plan to further validate the model and to reduce the error rate via fixing of simulator inaccuracies and adjustment of performance counters.

It is known that behavior of OS and effects of I/O can have significant effect on performance [16][17]. Such details are not properly captured by application-level simulators, unlike system-level simulators. Concrete examples of this can be seen in our evaluation. It is interesting to explore ways to take into account such details (for example, stalls caused by DTLB, TLU and other components), while staying within AL simulation.

Secondly, while performance of simulator is reasonable enough and stays within our original requirements, complete simulation of many applications can take days of real time. It is important to improve this situation.

One obvious way to address this is to reduce inefficiencies of current implementation. The main advantage of this approach is that it does not negatively affect accuracy of simulation. The work on several optimizations of this sort is currently in progress.

Another way to speed up the simulation process is to identify and simplify the least impactful parts of the simulation. It is less obvious how to do it properly, but as accuracy and reliability of current simulator increases, it becomes easier to investigate and evaluate such options.

Thirdly, as it became evident during implementation of cycle-accurate simulator, proper tooling has significant impact on productivity. And it is quite reasonable to improve existing tools to further improve efficiency of debugging and developing of simulator. For example, it could be useful to add more types of events to execution statistics and to implement more verbose version of execution log.

Finally, the support of the upcoming version of the Elbrus ISA is yet to be done.

REFERENCES

[1] Kutsevol V.N., Meshkov A.N., Chernykh S.V. The approaches to the performance optimization of multi-core «Elbrus» processors program models. *Voprosy radioelektroniki*, 2017, no. 3, pp. 57–61.

[2] Poroshin P.A., Meshkov A.N. An exploration of approaches to instruction pipeline implementation for cycle-accurate simulators of «Elbrus» Microprocessors. *Proc. ISP RAS*, 2019, vol. 31, no. 3, pp. 47-58.

[3] Znamenskiy D.V., Kutsevol V.N. Development of a cycle-accurate simulator of the Elbrus processor core memory subsystem. *Radio industry (Russia)*, 2019, vol. 29, no. 2, pp. 17-27, DOI: 10.21778/2413-9599-2019-29-2-17-27. (In Russian)

[4] Henning J.L. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 2006, vol. 34, no. 4, pp. 1-17.

[5] Weidendorfer J. KCachegrind: Call graph viewer. *Official Website http://kcachegrind.github.io/html/Home.html*, Accessed April 10, 2020.

[6] Kozhin A.S., Neiman-zade M.I., Tikhorskiy V.V. Memory subsystem impact on the 8-core «Elbrus-8C» processor performance. *Issues of radio electronics*, 2017, no. 3, pp. 13–21. (In Russian)

[7] Ermolitckii A.V., Neiman-Zade M.I., Chetverina O.A., Markin A.L., Volkonskii V.Y. Aggressive Inlining for VLIW. *Proc. ISP RAS*, 2015, vol. 27, no. 6, pp. 189-198, DOI: 10.15514/ISPRAS-2015-27(6)-13. (In Russian)

[8] Yourst M.T. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. *2007 IEEE International Symposium on Performance Analysis of Systems and Software*, 2007, pp. 23-34, DOI: 10.1109/ISPASS.2007.363733.

[9] Alves M.A.Z., Villavieja C., Diener M., Moreira F.B., Navaux P.O.A.. SiNUCA: A Validated Micro-Architecture Simulator. *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, 2015, pp. 605-610

[10] Akram A., Sawalha L. A survey of computer architecture simulation techniques and tools. *Ieee Access*, 2019, vol. 7, pp. 78120-78145.

[11] Carlson T.E., Heirman W., Eyerman S., Hur I., Eeckhout L. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014, vol. 11, no. 3, pp. 1-25.

[12] Woo S.C., Ohara M., Torrie E., Singh J.P., Gupta A. The SPLASH-2 programs: Characterization and methodological considerations *ACM SIGARCH computer architecture news*, 1995, vol. 23, no. 2, pp. 24-36.

[13] Guthaus M.R, Ringenberg J.S., Ernst D., Austin T.M., Mudge T., Brown R.B. MiBench: A free, commercially representative embedded benchmark suite. *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538), IEEE*, 2001, pp. 3-14.

[14] Bienia C., Kumar S., Singh J.P., Li K. The PARSEC benchmark suite: Characterization and architectural implications. *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72-81.

[15] Aslot V., Eigenmann R. Performance characteristics of the SPEC OMP2001 benchmarks. *ACM SIGARCH Computer Architecture News*, 2001, vol. 29, no. 5, pp. 31-40.

[16] Vandierendonck H., De Bosschere K. On the Impact of OS and Linker Effects on Level-2 Cache Performance. *14th IEEE International Symposium on Modeling, Analysis, and Simulation, IEEE*, 2006, pp. 87-95.

[17] Cain H.W., Lepak K.M., Schwartz B.A., Lipasti M.H. Precise and accurate processor simulation. *Workshop on Computer Architecture Evaluation using Commercial Workloads, HPCA*, 2002. vol. 8.