Flow-based Simulation Methodology

Morteza Hoseinzadeh

Abstract—This paper presents flow-based simulation, a new methodology for evaluating novel and intricate computer system designs. The main idea of flow-based simulation is to keep the history of every simulated memory element, instead of its latest value, to make it time bonded so that sliding the time forward and backward changes the state of the system accordingly. Having this opportunity, new architectural designs can be evaluated in terms of timing and energy by implementing only a functional simulation. Due to serial execution, the process of the design in a flow-based simulation is traceable and easy to understand. As a result, comparing with cycle-driven and event-driven techniques, complicated algorithms can be evaluated much easier. Flow-based simulation simplifies the burden of the timing simulation, and consequently leads to faster development and simulation time.

Index Terms—Computer architectural simulator, flow-based simulation, simulation methodologies.

1 INTRODUCTION

SOFTWARE simulation has become an indispensable tool for computer architects to investigate new ideas and various designs. Many design principles and methods for software simulation have evolved separately, and a large number of simulation tools have been developed.

A complete system simulator should satisfy four desirable properties: proper accuracy, reasonable speed, ability for running wide range of workloads, and simplicity of use and modification [12]. Current computer system simulators can be categorized into two major classes: timing simulators which can be either cycle-driven or event-driven, and functional simulators. Cycle-driven method is usually used to simulate low-level microarchitectures like a CPU or an electric system board mainly for timing/power measurements. They provide extremely accurate results at the cost of very slow execution (1-1000 KIPS) [3, 9]. MARSS [10] and HORNET [11] are two examples for this type of simulators. Event-driven is usually used for high-level computer systems using which every possible event in a system should be described and precisely handled, but it is much faster than cycle-driven simulators. ZSim [12] and gem5 [2] are examples of simulators benefiting from event-driven techniques. Both of the cycle-driven and event-driven methods can simulate concurrency, but their rigid implementation does not allow users to easily simulate arbitrary computer systems, and as a result, they lack of flexibility which foists long development time. On the other hand, although functional simulators have a short development time, they do not provide any information about physical characteristics since they are merely used for functionality checks.

Many timing simulators use event-driven technique, and they do not provide straightforward facilities for evaluating unconventional designs with complicated functionality. For instance, there are many new technologies that are being investigated such as emerging memory technologies [6, 7], 3D integrated circuit [4], Internet of Things [5], and etc. Motivated by this, we propose flow-based simulation which combines functional and timing simulation with a set of very slight modifications to fundamental concepts: the time can go forward and backward, and a memory element has a history instead of a single value. Flow-based simulation leverages the key insight that a complex algorithm can be better understood when it is serialized. So, in contrast with event-driven method in which the invocation time of events is sophisticated (depends on many factors), in flow-based technique, they are predictable because of serial execution. The goal of flow-based simulation is to let architects measure the performance, energy consumption, and functional verification by simulating only the functionality. In a flow-based simulator, the content of every memory element under simulation is bonded to the time. Conceptually, the time is a global floating point number which can be increased or decreased. Since the state of the system is also time-bonded, it is possible to rollback a sequence of events after a call chain which we call it a flashback.

In the rest of this paper, we first explain the flow-based simulation methodology. Then, we address the challenges arising with the flow-based simulation, and discuss our solutions. After that, we theoretically investigate the accuracy of flow-based modeling hypothesis. Finally, we conclude the paper within a discussion.

2 FLOW-BASED SIMULATION

We define a flow as a sequence of events happening successively which is triggered by invoking a routine and ends when the program counter gets back from that routine. A routine is a piece of code that describes the functionality of an element (we call it a block) in a system. Besides task functions, each block has a latency/power component which is used for timing/energy evaluation. Blocks can take a job and process it; and, they may optionally produce other jobs and send them to the job queue. In the end, the job queue is like a call stack that resembles buffering in hardware, so the programmer does not need to use a job queue very frequently. As shown in Fig. 1, in the event-driven technique, all blocks should dispatch events to invoke each other, while in the flow-based simulation they can call each other directly, although it is still possible to use a job queue to invoke a block which is usually used by a block to call itself (loopback).

Flow-based simulation, additionally, brings the opportunity of fast development by compacting parts of the simulated

1. It does not blockage using multi-threading techniques. In fact, running multiple serialized flows in parallel is also possible.
system into a single block depending on the level of abstraction. For example, we can consider the whole core as a single block which takes CPU-memory transaction trace of Simics [8] as a packet, and works with the memory hierarchy. As a result in this case, it is not required to implement the detailed functionality of the cores while we are investigating caching algorithms.

2.1 Mechanism
Considering the whole simulation as a Turing Machine, we can add the timestep as an extra input and define the state of the system in a time-line which keeps the information of all memory elements at each timestep. The simulation begins with assigning the first job to the first block and invoking it. As blocks get invoked sequentially, they may cause alterations with assigning the first job to the first block and invoking it.

Fig. 1: Job queue in flow-based simulation versus event queue in event-driven technique (numbers beside blocks/events indicate their execution time).

2.2 Concurrency Challenges
In a fully concurrent environment that flows do not influence each other, one flow can be simulated after another, and at the end of each flow, the time flashes back to the beginning of the next flow. However, if there is any interference between flows, we need to manage time movements more carefully. There are two ways that flows can interfere each other: control and data. In this section we discuss on them and explain our solutions for their avoidance.

2.2.1 Control Interference
Control interference means that at a given time an instance of a block runs more than once in different flows, which may be caused by flashbacks. In other words, the control of a block should be handed over to only one flow at a time. To prevent control interference, we move the time before launching the routine to a point that the block is free for a given period (block’s latency component). For this reason, we need to keep track of free time slots for every block. Therefore, each block is equipped with a Busy Time Slot Table (BTST) which maintains all periods of time that the block is not available in a BST data structure. The entering time to a block is calculated according to the calling time and its BTST. It can be whether right on the current time or delayed (stall). If it is right on the current time, it might be sooner than its exiting time in a preceding flow (slip). Slips may cause data inconsistency which we will discuss in Section 2.2.2. Sometimes, there might be an available, but smaller time slot for a block, comparing with its latency. Since in reality, the current flow is concurrent to the others, we split the new busy time slot into multiple parts to fill up the BTST.

Fig. 2 shows an example of time movements. In this example, there are three flows (flow#1, flow#2, flow#3) comprising five blocks (A, B, C, D, E). At flow#1, all blocks are executed immediately one after another since all of them are available. Flow#2 commences after a flashback to \( t = 0 \), but according to the BTST of block A, it is busy during \( t \in [0, 2] \). Therefore, the simulator forwards the time to \( t = 2 \). Likewise, block B is busy during \( t \in [2, 6] \), so a stall should be generated in \( t = [5, 6] \) for subsequent use of B in flow#2. In flow#3, block C is not involved and a new block E is invoked after D. According to the BTST of block D (\( t_1 \) and \( t_2 \)), it is free for \( t < 12 \). Therefore, if block B is engaged for one cycle (Fig. 2a), it is slipped back and takes place in \( t = [10, 12] \), and then, block E starts at \( t = 12 \). But, in case of block B taking more than one cycle (Fig. 2b), the corresponding busy time slot of block D should be split apart into two parts with one part (\( D_1 \)) within \( t = [11, 12] \) and the other one (\( D_2 \)) in \( t > 16 \). This process makes the timing simulation more exact. Notice that, in both cases we consider that there is no dependency between blocks. To further show this mechanism, Fig. 2c depicts the same sequence in reality in which block D is engaged in flow#3 sooner than flow#1. As shown in this figure, there are some intermediate differential stalls in reality shown in hatched orange rectangles (“diff”). In either cases, the total measured timing is identical.

2.2.2 Data Interference
Data interference means that a flow reads/writes contents of a memory element, and another flow, which comes after the first one in the execution sequence, writes/reads the content of that

Each element records its history in a Time-Value Table (TVT) which is stored in a Binary Search Tree (BST) in form of \((t,v)\), where \(v\) is the value of the element at time \(t\). We choose BST because it conducts the search and insertion in \(O(\log n)\) time. The algorithm of getting/setting a value from/to an element’s history is presented in Algorithm 1. To read from an element at a given time \(\tau\), the simulator examines its history to find the latest recorded time before \(\tau\) (not equal) using the getter function. To write to an element at a given time \(\tau\), the simulator simply adds a record to its history with \((\tau, v)\) via the setter method. This process enables atomic operations on simulated memory elements. When the state of the system gets steady at time \(\tau\), the history records before that can be released.

Algorithm 1 Getting and setting values from the history (TVT)

1: function LOCATER\((t)\)
2: \(n \leftarrow \text{ROOT(TVT)}\)
3: while \(n \neq \text{Null} \) do
4: \(\quad\) if \(t < n.t\) then
5: \(\quad\) if \(n.left = \text{Null} \) then
6: \(\quad\) \return\( n.parent\)
7: \(\quad\) else
8: \(\quad\) \(n \leftarrow n.left\)
9: \(\) else
10: \(\quad\) if \(n.right = \text{Null}\) then
11: \(\quad\) return \(n\)
12: \(\quad\) else
13: \(\quad\) return \(n.right\)
14: \endfunction

15: function GETTER
16: \(n \leftarrow \text{LOCATER}(\text{time})\)
17: return \(n.v\)
18: procedure SETTER\((v)\)
19: \(n \leftarrow \text{LOCATER}(\text{time})\)
20: if \(n.t = \text{time}\) then
21: \(n.v = v\)
22: else if \(t < n.t\) then
23: \(n.left \leftarrow \text{NEWNODE}(\text{time}, v)\)
24: else
25: \(n.right \leftarrow \text{NEWNODE}(\text{time}, v)\)
element in an earlier simulation timestep. Apparently, the data read in the first flow was outdated and inconsistent. This might also happen in an even-driven simulation when some events for updating a memory element take a long time. Notice that, in the flow-based simulation, all flows are executing sequentially in a single thread, and there would be no data inconsistency if there was no flashbacks.

There are two possible approaches to impede this problem: Stall Injection (SI) and Provisional Execution (PE). In both, each element has two distinct history buffers for read and write which indicates which block accessed the element and when. In SI, which is easier to implement, we simply consider an Available Time Table (ATT) for each memory element which is similar to BTST. While accessing the element for writing, if data has been read by another block in a future simulation timestep, the simulator inserts stalls until its last reading time. The same process is taken for reads, when there is a write in the future. Obviously, this mechanism prevents data hazard conditions and consequently causality errors. In Fig. 2-a, consider an element d being updated in flow#2 block C and is going to be read in flow#3 block D. Although there is an available time slot in \( t = [10,12] \), the ATT of the shared element indicates that the data is not available in \( t < 14 \), where block C finishes writing on it. As a result, the simulator detects a Read-after-Write (RaW) hazard and inserts stalls up to \( t = 14 \). Then, block D will have a free time slot in \( t \geq 16 \) (red blocks in Fig. 2-a).

In PE, all records of the history buffers are considered as provisional. Once a write happens to an element, the element is taken out from the provisional state, and every flow which accessed the element may either rerun, or commit. A rerun happens when reads are after the write, and it is done by rolling back to the flow with the earliest read after this write. Committing the modification occurs when reads are realized earlier to the write. This process reorders the execution of flows such that the flow which is writing to the element executes earlier to the write. This process reorders the execution of flows in a single thread, and there would be no data inconsistency if there was no flashbacks.

2.3 Coding Example

For better understanding, we provide an example of real implementation in this section. Listing 1 presents an example of flow-based simulation for reading a cache line from a cache with Non-Uniform Cache Access (NUCA) architecture with 16 banks.

Listing 1: NUCA read

```c
u64 nuca_read(u64 addr) {
    calc(this, phys,(.lat=1, .apwr=12, .ppwr=8));
    u64 cline=INVALID;
    int ln;
    concurrent {
        for(int i=0; i<16; i++) {
            if(bank[i].lookup(addr, &ln)) {
                cline = bank[i].readline(ln).getter();
            }
        }
    }
    return cline;
}
```

In line 2, function `calc` is called to calculate the physical metrics such as latency (.lat), active power (.apwr), and passive power (.ppwr). In line 6, the keyword `concurrent` makes everything inside it to be run concurrently. So, after each iteration of the `for` loop, the time will be back to its original point, and because the `concurrent` code block is serial to the rest, after exiting from the loop, the time will be added with the longest iteration’s latency. Inside each iteration, in line 8, the code block `serial` makes all inner blocks it to be run serially. As a result, `lookup` and `readline` are serial to each other. We also use a `getter` function to read data from the selected line.
This function also may cause the time to be moved. Also, it may cause the whole flow to be void, and rerun in case of enabling PE.

3 Analysis
Flow-based simulation brings simplicity by allowing the designer to perform timing evaluations through developing functional simulators. The timing/energy measurements can be handled automatically by a transpiler. The developing time depends on the desired abstraction level. The lower level, the more detail, the more precise. At the lowest level, flow-based simulation turns into a cycle-driven simulation in which for every single cycle, the whole system is simulated, and afterwards, a tiny flash back (1 cycle) occurs. At the highest level, the functionality of the whole system is implemented in a single block with an average latency/power component. The probability of accurate simulation can be defined as follows.

\[
P(Acc) = \prod_{i=1}^{N} (1 - P(E_i)),
\]

\[
P(E_i) = P(\{L_{real} - L_{sim} > \delta \mid B_i\})
\]

where \(N\) is the number of blocks representing the granularity, \(P(E_i)\) is the likelihood of timing measurement error which is defined in Eq. (2), \(L_{real}\) is the measured input-to-output latency of the circuit implementing the block, \(L_{sim}\) is the simulated value of that, \(B_i\) is an event in which the controller reaches to block \(i\), and \(\delta\) indicates the tolerable error. Generally, we can see that 
\[
\forall \delta \in \mathbb{R} \lim_{N \to \infty} \prod_{i=1}^{N} (1 - P(E_i)) = 1
\]

which shows an extremely exact simulation of all parts of the system which is not limited to the electrical components only. Notice that, energy evaluation has the same probability of accurate estimation, but it has a simpler total accuracy modeling (i.e., sum up all consumed energy).

On the other hand, the memory usage, the code complexity, and the simulation runtime are inversely proportional to the granularity. Considering that the longest flow consists of \(n\) blocks, and the total number of simulated blocks and flows are \(N\) and \(f\), respectively, and BTST has a maximum of \(m\) entries, the simulation runtime would be \(O(f \cdot n \cdot \log m)\) and the memory growth would be \(O(N \cdot m)\). This shows that too detailed implementation adversely impacts the runtime, memory usage, and the code complexity.

Nevertheless, it is not practically possible to describe everything. So, we model the system such that it can tolerate a small error range of \(\delta > 0\). In this case, the likelihood of accurate simulation can be also the maximum while ignoring negligible differences. When the level of abstraction reaches to a point that \(L_{real} = L_{sim}\) (i.e., \(P(E_i) = 0\)), finer granularities do not help for achieving a better accuracy. Thus, \(\exists k \in \mathbb{N}, \delta \in \mathbb{R}\) such that \(k \ll \infty\) and \(\lim_{N \to \infty} P(Acc) = 1\), which is called the optimum level of abstraction. We desire to maximize \(P(Acc)\) subject to an acceptable \(\delta\) with the smallest \(k \in \mathbb{N}\). This problem is NP-hard, but the designer can experimentally find reasonable \(k\) and \(\delta\).

4 Discussion and Evaluation
Flow-based simulation is a new simulation methodology which can be used in many different areas albeit focused on architectural simulations. The convenience of investigating new algorithms through implementing concurrent flows makes it possible to instantly convert a functional simulator to a timing one. For example, to implement an out-of-order execution, the designer calls a function to store the instruction in the core's buffer. Then s/he calls another function to pull out a ready instruction from the buffer without reasoning about the complex interaction between components that event-based simulation requires. The implementation is just like the description without any difference. Based on the needs, we can enhance the accuracy of the results at a cost of complicating the implementation.

Flow-based simulation requires large memory space in the host machine to keep the history of each memory element before reaching to a steady state. To mitigate this problem, many approaches can be employed. One of them is to cut off the corresponding tables and guarantee up to a fixed number of records. We have observed that there is less than 5% misses in ATT, TVT, and BTST while simulating a simple machine with 4 cores running concurrently. However, there may exist some extreme cases in which the lag time between one part and others is too large, and simulating that part requires having long history. To cover those cases, we engage another solution which is to consider infinite buffers, and use a simple tiering algorithm to archive dated information to the disk. This may cause performance degradation while simulating some parts in the far past, but we know that such cases are very rare.

The implementation of a flow-based simulator is significantly easier than that of other techniques, mostly because of its straightforward definitions which let us to develop a transpiler to automatically convert a functional implementation to a timing simulator, and thus dramatically alleviate the complexities. Our experiment of simulating a 4-core UltraSPARC III v9 with 32 KB instruction cache, 64 KB data cache, an 8 MB L2 shared cache, and a 4 GB memory, and comparing it to Ruby. Running PARSEC 2.1 shows around 6x faster in simulation runtime, with a considered level of abstraction which provides almost the same results, but with 5x smaller number of code lines.

References


