Evaluating Out-of-Order Engine Limitations using Uop Flow Simulation

Vincent Palomares[1], David C. Wong[2], David J. Kuck[2], and William Jalby[1]

[1] University of Versailles Saint-Quentin-en-Yvelines, France
{firstname.lastname}@uvsq.fr
[2] Intel Corporation, Champaign, U.S.A.
david.c.wong@intel.com, david.kuck@intel.com

Abstract. Out-of-order mechanisms in recent microarchitectures do a very good job at hiding latencies and improving performance. However, they come with limitations not easily modeled statically, and hard to quantify exactly even dynamically. This paper will present Uop Flow Simulation (UFS), a loop performance prediction technique accounting for such restrictions by combining static analysis and cycle-driven simulation. UFS simulates the behavior of the execution pipeline when executing a loop. It handles instruction latencies, dependencies, out-of-order resource consumption and other low-level details while completely ignoring semantics. We will use a UFS prototype to validate our approach on Sandy Bridge using loops from real-world HPC applications, showing it is both accurate and very fast (reaching simulation speeds of hundreds of thousands of cycles per second).

1 Introduction

Different performance models can be built with very different purposes in mind. For instance, CPU architects may need very low-level cycle-accurate simulators to find and fix bugs in their design, in which case accuracy is so important that practical aspects like processing time and lightweightness become completely secondary. At the other end of the spectrum, models like Intel Architecture Code Analyzer (IACA) [1], Code Quality Analyzer (CQA) [2] and Cape [3] aim to provide good-enough predictions at minimum cost, both in terms of time and space. Many models have been developed as compromises between these two extremes.

In this paper, we will present *Uop Flow Simulation* [4] (UFS), a technique to accurately evaluate a loop's performance for L1 data sets by simulating the behavior of a CPU's out-of-order engine on a cycle-accurate basis, and offering the following advantages:

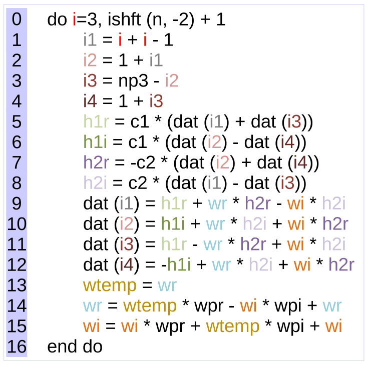
1. Modeling key components of the out-of-order engine (issue, buffers, dispatch...) with a reasonable accuracy.
2. Very fast speed: several hundred thousand simulated cycles per second in typical cases.

3. Low memory consumption: only a few MB of RAM are required.
4. Small input files: only limited, statically extracted information needs to be used for a given loop.

We will present a motivating example to showcase the value of UFS and describe our model in details. Then, we will validate our implementation on real-world loops from the AVBP [5] and YALES2 [6] industrial applications, targeting the Sandy Bridge microarchitecture.

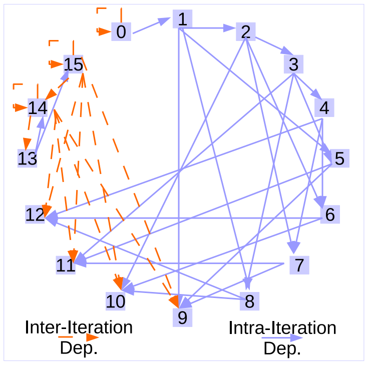
We will use CQA as a reference point to highlight our model's contribution. CQA leverages a performance model close to the one developed in IACA, operating in a mostly bandwidth-centric fashion and overlooking the impact of out-of-order buffers. Both CQA and our UFS prototype operate at the binary loop level and rely on static information extracted using the MAQAO [7] framework to perform their analyses, making them easy to compare.

2 Motivating Example



```
0   do i=3, ishft (n, -2) + 1
1       i1 = i + i - 1
2       i2 = 1 + i1
3       i3 = np3 - i2
4       i4 = 1 + i3
5       h1r = c1 * (dat (i1) + dat (i3))
6       h1i = c1 * (dat (i2) - dat (i4))
7       h2r = -c2 * (dat (i2) + dat (i4))
8       h2i = c2 * (dat (i1) - dat (i3))
9       dat (i1) = h1r + wr * h2r - wi * h2i
10      dat (i2) = h1i + wr * h2i + wi * h2r
11      dat (i3) = h1r - wr * h2r + wi * h2i
12      dat (i4) = -h1i + wr * h2i + wi * h2r
13      wtemp = wr
14      wr = wtemp * wpr - wi * wpi + wr
15      wi = wi * wpr + wtemp * wpi + wi
16  end do
```

(a) Source Code



(b) Source Level DDG

Fig. 1: Realft2_4_de Codelet

Realft2_4_de is a codelet from our Numerical Recipes suite, and is part of an inverse Fourier transform algorithm. The DDG (Data Dependency Graph) was made at source level for clarity purposes.

Realft2_4_de (Figure 1(a)) is a codelet from our Numerical Recipes suite, and is part of an inverse Fourier transform algorithm. It is a particularly interesting codelet as it suffers from an important CQA error even for L1 data sets. Indeed, CQA overestimates the loop's speed by roughly 45% (see Table 1), making it a good case for detailed study.

Detailed investigation using hardware counters revealed that stalls due to out-of-order resource scarcity may have had an impact on the actual performance. While not all stalls necessarily impede the execution, this was still an interesting lead directly incriminating the Reservation Station (RS): as tholds uops until their operands are ready, it gets particularly stressed when there are dependencies between instructions. We gave them a particular scrutiny in Figure 1(b).

Running UFS on the target loop gives encouraging results (see Table 1), completely filling the gap between the CQA cycles estimation and the measurement.

Table 1: Realft2_4_de: Measurements, CQA and UFS Projections

Source	**Cycles**	**Stalls**	**Error (Cycles)**
Measured	23.36	7.85 [RS]	N/A
CQA	16.00	N/A	31.51%
UFS (Normal buffers)	23.01	19.97 [RS]	1.50%
UFS (Large buffers)	19.03	5.65 [ROB]	N/A

The target microarchitecture is Sandy Bridge. Durations and stalls represents values per assembly loop iteration. The error *metric is defined as:*

$$error = \left|\frac{measured\ execution\ time - predicted\ execution\ time}{measured\ execution\ time}\right|$$

While the simulated stalls count the number of cycles where at least 1 uop could not be issued due to a buffer being full, what hardware counters exactly measure is not very clear (and could e.g. be the number of cycles where no uop at all was issued), explaining the importance of the gap between measured and simulated stall counts.

We can notice a 31% gap between the measured time per iteration and the bandwidth-centric CQA projection. Measured resource stall counters indicate the Reservation Station (not modeled by CQA) may have been hindering performance, though they do not guarantee it is the case.

The error for UFS (with regular Sandy Bridge parameters) is of only 1.50%, a significant improvement over CQA's 31.51%. This accuracy gain is due to two main factors:

1. RS size awareness: running UFS again with a virtually infinite RS size shows that 4 cycles could be gained from having larger out-of-order buffers (Table 1).
2. Realistic dispatch: hardware heuristics prioritized uops that are not on the critical path.

Furthermore, running UFS again with all out-of-order resources set to have 1000 entries (*UFS - Large Buffers* row) shows nearly 4 ($23.01 - 19.03 = 3.98$) cycles can be gained by merely increasing buffer sizes, demonstrating that buffers do indeed limit performance here.

The CQA evaluation represents the performance the loop would attain if not for these issues.

3 Model Presentation

The purpose of the model is to account for limitations of the out-of-order engine not taken into account in CQA using a limited cycle-accurate simulation. The semantics of instructions is completely disregarded; only the flow of uops is being computed, estimating the speed at which uops may travel through the pipeline.

Detailed interactions between uops and the pipeline are taken into account for this purpose. For instance, the simulation keeps track of in-flight uops and the number of available resources, constraining the simulated flow of uops as a real system would. Dispatching constraints and heuristics are also implemented, allowing for a realistic estimation of port load in complex loops.

3.1 Limited Input

The simulator uses two types of inputs:

1. Loop information: as the model is only tracking the flow of uops and not their semantic purpose, register and memory values are not needed. It uses only basic information obtainable from static analysis, such as the type, register operands and outputs for each instruction in the studied loop. It also uses Agner's instruction tables [8] as reference for instruction dispatch port(s) and latency. Just as with CQA, loop inputs are generated using the MAQAO [9] framework. An example input is provided in Table 2.
2. Microarchitecture information: simple parameters such as the size of each out-of-order resource or the Front-End and issue uop bandwidth are needed. Default values (see Table 3) can be provided for each target microarchitecture. All the studied microarchitectures have an issue and retire bandwidth of 4 uops per cycle. A few behaviors are also microarchitecture specific, such as the status of microfused uops in the ROB.

The number of loop iterations to simulate can also be specified, with a default value of 1000.

3.2 Model Overview

UFS simulates as many cycles as needed for *the last uop of the Nth iteration to retire*, with N being the number of iterations to simulate. The different simulation steps for a cycle are as follows:

1. *Issue*: inserts new uops in the ROB (and RS if need be) and allocates all needed resources. This step is done in order.
2. *Dispatch*: removes uops from the RS when all the uops they depend on were properly *executed* and a compatible execution port is available. This is done out of order.
3. *Update*: flags uops as being *executed* L cycles after they were dispatched, with L being their attributed latency. Also releases the Load Matrix entries used by executed load uops. This is also done out of order.
4. *Retire*: removes *executed* uops from the pipeline and releases the resources they still held. In a real microarchitecture, this is the step at which *executed* uops' outputs would be committed to the architectural state. Retirement is done in order.

Table 2: Partial UFS Loop Input Example (Realft2_4_de)

#Insn	Nb_FE	Type	Input	Output	Latency	Ports
1	1	compute	XMM1	XMM10	1	P5
2	1	compute	RAX	RDI	1	P1, P5
...						
8	1	load	RSI, RDI	XMM15	4	P2, P3
...						
53	1	store_addr	RSI, R8		1	P2, P3
		store	XMM15		3	P4
54	1	branch	RDX, RAX	test	1	P5

*The specifics of what an instruction exactly does is irrelevant for UFS. Instead, only characteristics such as the number of uops it takes in the Front-End (*Nb_FE*), input/output registers and latency are important.*

The type *category allows us to determine what type of resource the uop will need to get issued (e.g. branch buffer entry for* branch *uops), coupled with the* output *field: for instance, instruction 1 is going to need a vector register as it produces an XMM value, while instruction 2 will need an integer one.*

The ports *column provides a list of ports compatible with the uop.*

In some cases, a single instruction gets split into several uops. As each of them may potentially have different attributes, differences between them need to be described explicitly. This is the case for instruction 53.

Complementary attributes are sometimes needed, e.g. in case of division uops, as they are going to use special resources exclusively for variable amounts of time.

The *current cycle count* (i.e. number of cycles simulated so far) is maintained and updated every cycle.

Figure 2 presents an overview of the model. We will describe details for each simulated component in the coming sections.

3.3 Simplified Front-End

As UFS targets loops, we can safely assume that all the uops sent to the *uop queue* come from the *uop cache*, hence ignoring the legacy decode pipeline and its limitations and providing a constant uop bandwidth of 4 per cycle. While the uop cache has limitations of its own (e.g. it cannot generate more than 32B worth of uops in a cycle), we decided to ignore them as we could not find real world cases where they got in the picture. This is partly due to compilers being smart enough to avoid dangerous situations by using code padding.

We also assume the branch predictor is perfect and never makes mistakes, meaning we do not need to simulate any roll back mechanisms. This is a decently safe assumption for the loops we study due to their high numbers of iterations, but reduces the applicability of UFS for loops with unpredictable branch patterns.

We consequently model Front-End performance in a simplified way:

Table 3: Default Microarchitectural Input Parameters

Microarchitecture	**Sandy Bridge**	**Ivy Bridge**	**Haswell**
Front-End (FE) Bandwidth	4	4	4
Branch Buffer (BB) Entries	48	48	48
Load Buffer (LB) Entries	64	64	72
Load Matrix (LM) Entries	32	32	32
FP Physical Registers (FP - PRF)	112	113	138
Integer Physical Registers (Int - PRF)	128	130	144
Max Number of Allocated Registers (All - PRF)	141	165	177
ReOrder Buffer (ROB) Entries	165	168	192
Reservation Station (RS) Entries	48	51	51
Store Buffer (SB) Entries	36	36	42
ReOrder Buffer Microfusion	No	Yes	Yes
Reservation Station Microfusion	No	No	No

We found the practical sizes of the PRFs, the ROB and the RS to be different from the official ones in all three microarchitectures [10]. Such differences can be explained by e.g. the mode the processor is working in (64-bit mode exposes more named registers than the 32-bit one), resources needed to maintain the architectural state (e.g. 16 physical registers for the 16 named registers), the number of pipeline stages involved in allocating or releasing resources, or by technical limitations unknown to us. In any case, the simulator inputs should be the number of resources available for speculation.

1. For Sandy Bridge (SNB) / Ivy Bridge (IVB): 4 uops can be generated every cycle, except a uop queue limitation prevents uops from different iterations from being sent to the Resource Allocation Table (RAT) in the same cycle. For instance, if a loop body contains 10 uops, the uop queue will send 4 uops in the first two cycles, but only 2 in the third.
2. For Haswell (HSW): 4 uops can be generated every cycle: the limit experienced in SNB and IVB was apparently lifted.

In some cases, the uop queue has to unfuse microfused uops (or *unlaminate* uops) before being able to send them to the RAT [11], causing more issue bandwidth to be consumed (and sometimes, also more out-of-order resources). We take common cases into account using the following rules:

1. For SNB / IVB, unlaminate when the number of register inputs for the whole instruction is greater than 2.
2. For HSW, unlaminate when the number of register inputs and outputs *for an AVX instruction* is greater than 4. This rule was obtained empirically.

3.4 Resource Allocation Table (RAT)

The simulated RAT is in charge of *issuing* uops from the Uop Queue to the ROB and the RS, as well as allocating the resources necessary to their proper execution and binding them to specific ports.

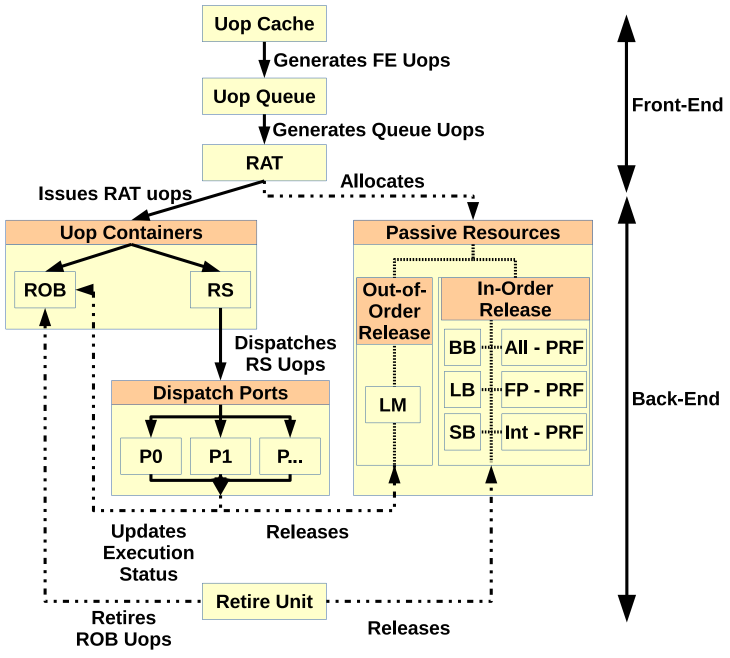


Fig. 2: UFS Uop Flow Chart

Several types of uops are used in the pipeline. For instance, Front-End uops (FE uops) are different from Queue uops or ROB uops. Generally, the closer to the Front-End, the closer to the original instruction the uop is. As different components can split uops when processing them, Back-End uops may contain less of the original information and semantic than their earlier counterparts. In other words, more Back-End uops than FE uops may be needed to describe the same instruction. Such transformations will be detailed in the matching component's modeling description.

The ROB and the RS are both resources and uop containers. Other resources have a more passive role and do not describe uops, acting instead as mere dependencies.

It does not have any bandwidth limit other than the one induced by the Uop Queue's output.

Resource Allocation In regular cases, resource allocation is quite straightforward. For instance, all uops need a spot in the ReOrder Buffer (ROB), loads need Load Buffer (LB) and Load Matrix (LM) entries, etc. However, it gets more complex when an instruction is decomposed into more than a single uop. In our implementation, all resources needed at the instruction level will be allocated when the first uop reaches the Back-End. For instance, stores are decomposed into a *store address* and a *store data* uops: in this case, a Store Buffer (SB) entry will be reserved as soon as the *store address* uop is issued, and the second uop will be assumed to use the same entry. However, individual uop resources (ROB or RS entry) will still be allocated at the uop granularity.

It is important to note that if any resource is missing for the uop being currently considered, the RAT will stall and not issue any other uop until resources for the first one are first made available. This is commonly referred to as a *resource stall.*

Port Binding Available information about dispatch algorithms in recent Intel microprocessors is rare and limited. We decided to bind uops to single ports in the RAT, sparing the RS from having to do a complex cycle-per-cycle evaluation of dispatch opportunities. Smarter strategies could be used, but we preferred to keep our simulation rules as simple as reasonably possible.

The simulated RAT keeps track of the number of in-flight uops assigned to each port, and assigns any queue uop with several port options to the least loaded one. In case of equality, the port with the lowest digit is assigned (this creates a slight bias towards low-digit ports).

This process is repeated on a per-uop basis, i.e. the simulated RAT uses knowledge generated by issuing younger uops in the same cycle, rather than using counts only updated once a cycle, which may in turn be optimistic.

Arbitrary numbers of ports can be activated as their use is regulated by the loop input file anyway (see Table 2). New microarchitectures with more (or fewer) ports could be simulated by tweaking input files' uop port attribution scheme to match the target's.

3.5 Out-of-Order Flow

Reservation Station (Uop Scheduler) When arriving in the Reservation Station, queue uops that are still microfused get split in two, simplifying the dispatch mechanism.

The RS holds uops until a) their operands are ready, b) the needed port is free and c) the needed functional unit is available. When all conditions are met, the RS *dispatches* uops to their assigned port prioritizing older uops, releasing the RS entries they used.

Port and Functional Unit Modeling Ports act as gateways to the functional units they manage. They are modeled as all being completely identical, and being able to process any uop sent to them by the RS. Functional units are not modeled distinctly, and constraints over them are modeled inside their respective port instead. Several rules are applied to match realistic settings:

1. A port can only process a single uop per cycle (enforced by dispatch algorithm).
2. Uops can be flagged as needing exclusive use of certain functional units for several cycles. For instance, division uops will make exclusive use of the divider unit for (potentially) dozens of cycles. A port processing such a uop will flag itself as not being able to handle other uops needing this particular unit for the specified duration. The same mechanism is also used for 256-bit

memory operations on SNB and IVB. A port with busy functional units can still service uops not needing them.
3. While the port itself does not check whether it should legally be able to process a given uop, the RS verifies this a priori, preventing such situations in the first place.

Uop Execution Status Modeling ROB uops have a time stamp field used to mark their status, and holding the cycle count at which they will be fully executed. By convention, the default value for newly issued uops is -1: its output is available if $current\ cycle\ count \geq the\ uop's\ execution\ time\ stamp > -1$.

Updating ROB uops' execution time stamp is typically done at dispatch time: as we deal with constant latencies, we can know in advance on what cycle the uop's output is going to be ready ($current\ cycle\ count + uop\ latency$).

In the case of typical *nop*-typed instructions (such as *NOP* and zero-idiom instructions like *XOR %some_reg, %same_reg*), the time stamp is directly populated with a correct value at issue time, reflecting the RAT being able to process them completely in our target microarchitectures. As they also have 0 cycle of latency, their stamp is simply set to $current\ cycle\ count$.

However, we found an extra simulation step to be necessary to handle *zero-latency register moves* (implemented in IVB and HSW), which are *nop*-typed and are entirely handled at issue time too. Contrary to *NOP*s or zero-idioms, register moves have register inputs, the availability of which is not necessarily established yet when the move uop is issued. We tackle this issue by inserting such uops with a negative time stamp *if* their input operand's availability is not known yet, and letting a new "uop status update" simulation step update them when it is.

This new *update* step is also in charge of releasing Load Matrix entries allocated to the load uops whose execution was just completed.

3.6 Retirement

The retirement unit removes uops from the ROB and releases their resources (other than RS and LM entries, which were already freed earlier in the pipeline).

1. Retirement is done in-order: no uop can be retired if an older uop still exists in the ROB. This is necessary to be able to handle precise exceptions and rollback to a legal state.
2. The default retirement bandwidth is the same as the FE's (4 uops per cycle) to prevent retirement from being the bottleneck in terms of throughput. To ensure this, ROB uops that are still microfused in the RAT are only counted as a single uop for retirement purposes.
3. Resources released in a given cycle cannot be reused in the same cycle. Our understanding is that it would be extremely complex to implement a solution allowing this, with very little performance to be gained (potentially increasing each resource's effective size by a maximum of 4). Note: we apply the same reasoning to the RS and the LM, even though their entries are

freed at dispatch time (for the RS) or update/completion time (for the LM) instead of at retirement.

4. Resources allocated at the instruction level at issue time are released when retiring the last uop for this specific instruction. This is consistent with the resource allocation scheme we use at the issue step.

3.7 Overlooked Issues

Many aspects of the target microarchitectures are not simulated. Some of them are inherently so due to our approach and the lack of dynamic information, such as cache and RAM behavior, Read after Write (RAW) memory dependencies and branch mispredictions.

Others are implementation choices and may be subject to change, like the number of pipeline stages (which could have an impact on the resource allocation scheme), the impact of yet-to-be-executed store address uops on later load and store uops, writeback bus conflicts [12] or partial register stalls [13].

Furthermore, while a lot of information is available concerning the way Intel CPUs work, many hardware implementation details are not publicly available. We could fill some of the gaps using reasonable guesses, but they are probably flawed to some degree, restricting the accuracy attainable by our model.

4 Validation

The validation work for UFS is twofold:

1. Accuracy: checking whether the model provides faithful time estimations for loops operating in L1.
2. Speed: making sure simulations are not prohibitively slow for their intended use.

We will focus on Sandy Bridge validation, as identifying performance drops on this microarchitecture was the primary motivation for developing UFS in the first place. Furthermore, its modeling is used as basis for IVB and HSW support, making SNB validation particularly important.

We will use the *fidelity* metric (defined here as $fidelity = 1 - error$) to represent UFS accuracy for each studied loop, and systematically compare UFS results with CQA projections to highlight our model's contributions.

A short study of the time taken by our UFS prototype will be made, and results will be presented in terms of *simulated cycles per second.*

4.1 Fidelity

Experimental Setup The host machine had a two-socket *E5-2670* SNB CPU, with 32 KB of data L1 cache, 256 KB of L2, and 20 MB of L3. It also had 32 GB of DDR3 RAM.

For each tested application, we selected loops that:

1. Are hot spots: the studied loops are relevant to the application's performance.
2. Are innermost, have no conditional code and can therefore be analyzed out of context.
3. Have a measured time greater than 500 cycles per loop call. This is needed to make sure measurements are reliable (small ones can be inconsistent [14]). This may exclude small loops that are called numerous times.

Performance measurements were performed in vivo using the DECAN [15] differential analysis tool.

We use DECAN variant *DL1* to force all memory accesses to hit constant locations, and thereby getting a precise idea of what the original loop's performance would be if its working set fit in L1. This also allows us to make direct comparisons between measured cycles per iteration vs. UFS and CQA projections, as other components of the memory hierarchy are artificially withdrawn from the picture.

AVBP AVBP [5] is a parallel CFD numerical simulator targeting reactive unsteady flows. Its performance scales nearly linearly for up to 4K nodes.

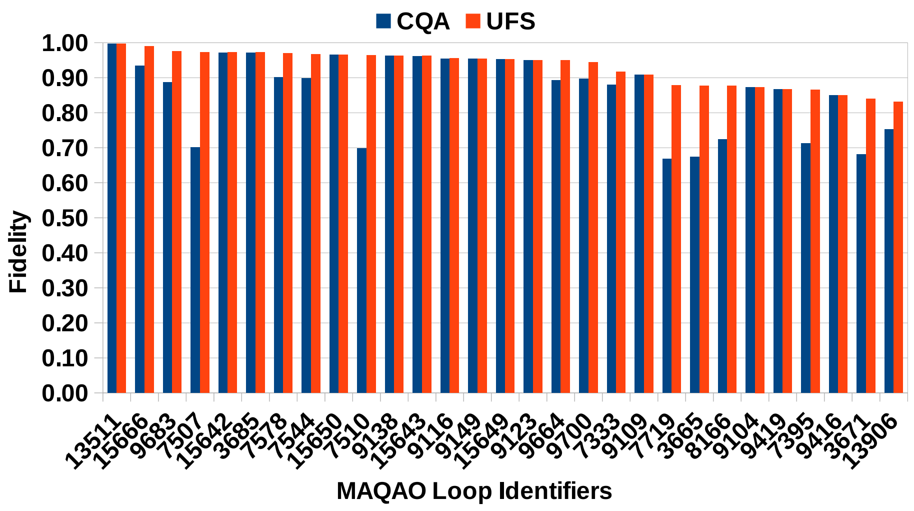


Fig. 3: In Vivo Validation for DL1: AVBP
Results are sorted by descending UFS fidelity.

Figure 3 shows UFS and CQA results for 29 AVBP hot loops on Sandy Bridge. UFS shows fidelity gains of more than 5 percentage points for 9 of them, with a maximum gain of 27 percentage points for loops 7507 and 7510. Other important gains include 20 percentage points for loops 7719 and 3665.

The worst fidelity for UFS is 78.18% for loop 13906 (against 66.76% for CQA on loop 3665).

The average fidelity is of 91.73% for UFS, versus 86.34% for CQA.

YALES2: 3D Cylinder YALES2 [6, 16] is a numerical simulator of turbulent reactive flows using the Large Eddy Simulation method. Its performance scales almost linearly with the number of execution cores even with thousands of cores.

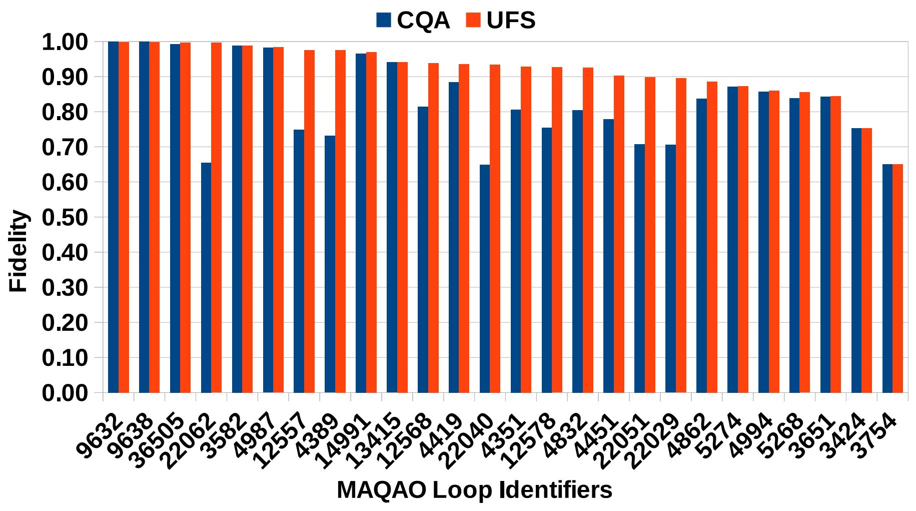


Fig. 4: In Vivo Validation for DL1: YALES2 (3D Cylinder)
Results are sorted by descending UFS fidelity.

Figure 4 shows UFS and CQA results for the *3D cylinder* part of this application. UFS shows fidelity gains of more than 5 percentage points for 12 loops out of 26, with a maximum gain of 35 for loop 22062. Other particularly important gains include 28 and 24 percentage points for respectively loops 22040 and 4389.

Some loops' performance are impacted by factors apparently not modeled by UFS, with disappointing fidelities of respectively 65.01% and 75.32% for loops 3754 and 3424.

The average fidelity is of 91.67% for UFS, versus 82.93% for CQA.

4.2 Simulation Speed

Speed is very important for performance evaluation tools, especially in the context of optimization: various versions of a program can be tested, e.g. trying different compiler flags or hand optimizations.

The quality of a model can be thought of in terms of return on investment: are the model's insights worth their cost?

We will hence study UFS's speed in this section, and evaluate the cost of UFS analyses.

Experimental Setup Simulations were run serially on a desktop machine with an *i7-4770* HSW CPU, running at 3.4 GHz. They were run on a single core, with 32 KB of L1 data cache, 256 KB of L2 cache and 8 MB of L3. It also had 16 GB of DDR3 RAM.

The targeted microarchitecture was SNB, with its default microarchitectural parameters, but simulating different numbers of iterations: 1000 and 100 000. The former is the default one and the most relevant to our analysis, while the latter was run to give an idea of sustained simulation speeds past the initialization phase (slowed down by I/O).

Execution times were measured using the *time* Linux tool, with a resolution time of 10 ms. While other measurement methods would be more precise, we deemed this one to be enough for our intended purposes. Furthermore, the time needed to generate the loop input files with MAQAO is not counted here. Measures were performed with 11 meta-repetitions to stabilize results.

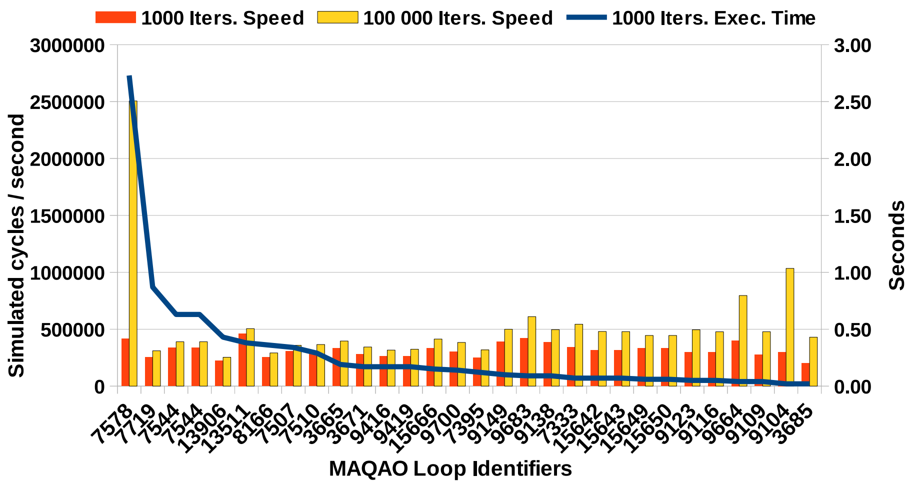


Fig. 5: UFS Speed Validation for AVBP

Results are sorted by descending UFS execution time.

Our UFS prototype simulates an average of 300K cycles per second for the studied loops. This average is 1.6x higher when simulating 100 000 iterations.

AVBP Figure 5 shows simulation speeds for the AVBP loops we studied. Here, the time needed to simulate 1000 iterations has a high variability, and can go from as low as .02 seconds for loop 3685 to as high as 2.73 seconds for loop 7578. The average simulation time is of around .28 seconds for each loop. This is due to the high complexity of some of the loops, which comprise hundreds of instructions (200 assembly statements on average). In the case of loop 7578, there are 1337 instructions (including divisions), making each of the 1000 iterations require many simulated cycles to complete. Hence, each iteration needs more simulated cycles to complete. Furthermore, the number of instructions can impact the locality of our UFS prototype's data structures, with large loops consequently being simulated less quickly.

For AVBP, we achieve on average:

1. Simulation times (for 1000 iterations) of approximately .28 seconds per loop: we can sequentially simulate around 3.57 loops per second.
2. The simulation of 318K cycles per second for 1000 iterations (and 519K for 100 000 iterations).

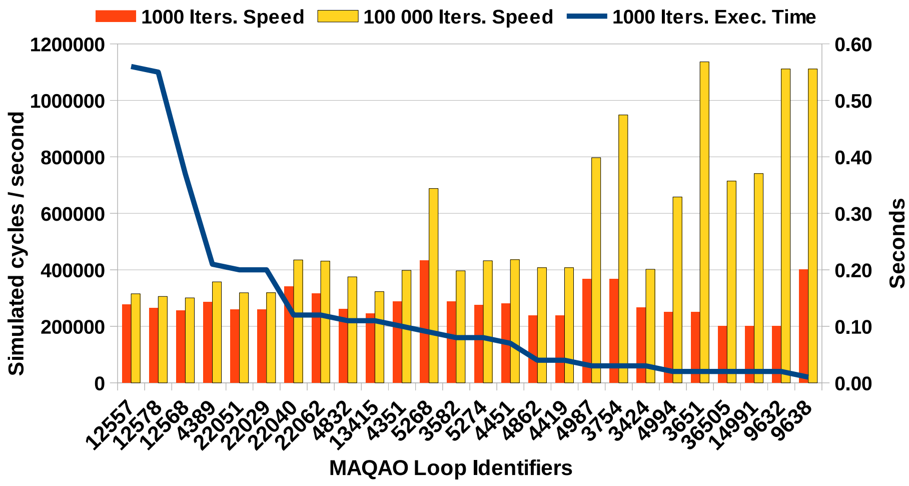


Fig. 6: UFS Speed Validation for YALES2: 3D Cylinder

Results are sorted by descending UFS execution time.

Our UFS prototype typically simulates around 200K cycles per second here. This number doubles when simulating 100 000 iterations.

In practice, simulation times are around .10 seconds for each loop.

YALES2: 3D Cylinder Figure 6 shows simulation speeds for the YALES2 (3D Cylinder) loops we studied.

As with AVBP loops, the simulation time for 1000 iterations is highly variable, going from .02 to .56 seconds. Simulations take .13 seconds on average, which is shorter than for AVBP (.28 seconds). We can hence sequentially simulate an average number of $\sim$ 7.69 YALES2 loops per second. The difference is due to YALES2 loops being relatively less complex, with an average size of 110 assembly statements (against 200 for AVBP).

However, the average number of simulated cycles per second is similar, reaching 281K cycles per second when simulating 1000 iterations –against 318K cycles per second on AVBP– (respectively 549K and 519K when simulating 100 000 iterations).

Comparison with CQA We will quickly assess CQA's speed to compare it to UFS's. To do so, we ran CQA on the AVBP binary for all the loops studied earlier (in a single run).

When removing the overhead due to the MAQAO framework (mostly consisting in disassembling the binary) to make fair comparisons with UFS, we found that CQA could process 30.98 loops per second. We can hence roughly estimate UFS to be 30.98/3.57 $\simeq$ 8.68x for AVBP's hot loops.

Applied to YALES2, the same methodology shows that CQA can process 42.85 loops per second when targeting the hot loops we studied earlier. This brings the overhead for using UFS to 42.85/7.69 $\simeq$ 5.57x for YALES2's loop hotspots.

This difference is larger ($\sim$ 13x) for smaller loops, of which CQA can process around 280 per second, compared to approximately 21 with UFS (the detailed data is not presented in this paper).

Overall, UFS analyses take around 10x more time than CQA's.

5 Sensitivity Analyses

We can use UFS to perform sensitivity analyses and evaluate how loops of interest would behave given different microarchitectural inputs.

5.1 Latency Sensitivity Analysis

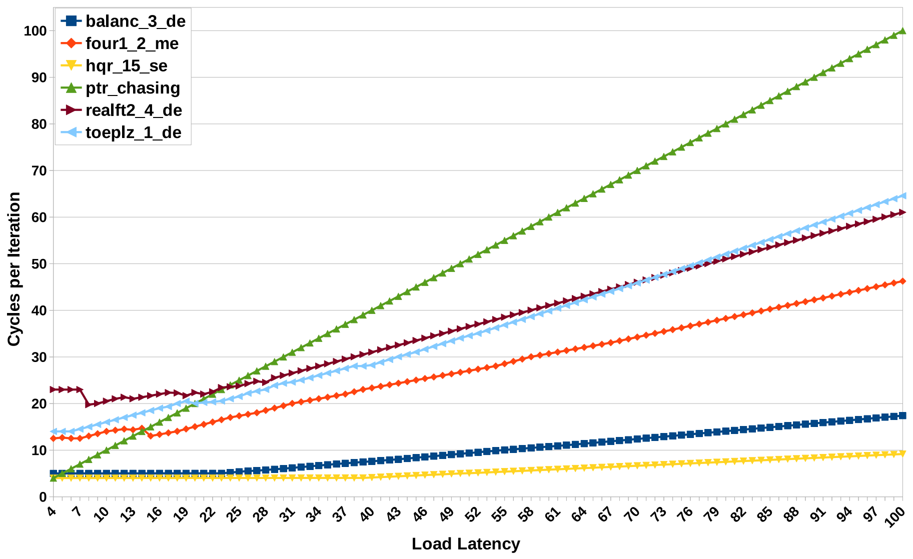


Fig. 7: Sensitivity Analysis: Load Latency
The unit of the Load Latency axis is cycles. The presented loops were extracted from the Numerical Recipes [17, 3], except for ptr_chasing.
We can see loops can react very differently to latency increases, with pointer chasing being most impacted.

We will evaluate the behavior of different loops as the performance of the cache hierarchy varies. While our model does not support detailed cache modeling, we can still change L1 performance by changing the latency of load and/or store uops.

Figure 7 shows how different loops react to latency variation. We can notice a wide range of behaviors, with interesting outliers:

– *ptr_chasing*: a loop chasing dependent pointers (1 pointer per iteration), and where only one load uop can consequently be executed in parallel. Load latency entirely governs its performance: its *Cycles per Iteration* metric scales perfectly with the latency of loads on the studied range of latencies (slope = 1x). This represents the worst case scenario for latency scaling.
– *hqr_15_se*: a loop with very few arithmetic operations per load, allowing it to execute many load uops in parallel. It can absorb important amounts of latency wihout getting degraded performance (up to 39 cycles), and then its *Cycles per Iteration* value scales only weakly with latency (slope $\simeq$ 0.09x).

Furthermore, some loops surprisingly get better performance for higher latencies (which is most noticeable on *realft2_4_de*, where the number of *Cycles per Iteration* drops on point 8): the change in latency causes uops from the Reservation Station to become ready at different times, changing the order in which they get dispatched (and coincidentally reaching a better dispatch scheme than with a lower load latency). This only happens locally, though, and the regular behavior (of performance dropping as latency increases) gets back in the picture on later data points.

5.2 Resource Size Sensitivity Analysis

We can also easily quantify how sensitive a loop is to the size of out-of-order buffers, i.e. see the impact of buffer sizes on instruction level parallelism (ILP) for the studied loop.

In Figure 8, we evaluate how a loop's performance varies depending on the sizes of the RS and other buffers (and particularly the ROB). We can see that no speedup can be achieved from merely increasing the size of the RS (coordinates (2, 1)). However, increasing the size of other buffers (and particularly that of the ROB, in this case) by 25% can provide a speedup of 1.22x (coordinates (1, 1.25)). We can observe diminishing returns, though, as higher speedups are very expensive to get. For instance, reaching e.g. 1.32x requires increasing the size of the RS by 25% and those of other buffers by 75%.

Interestingly, we can see that *reducing* the size of the RS can provide a speedup of 1.06x (coordinates (0.8, 1)). Similarly to the odd cases presented above for latency changes, this is due to how the dispatch order of uops can be changed in a coincidentally better way when degrading buffer sizes. However, such counter-intuitive cases are uncommon.

Furthermore, we can observe that decreasing the size of all buffers by 60% (coordinates (0.4, 0.4)) causes a negative speedup of only 0.69x (i.e. a 31% performance penalty).

We can hence easily determine the sweet spot for *performance per buffer entry* with UFS, as well as any degrees of compromise between small buffer sizes and best achievable ILP. However, other models and tools are needed to evaluate

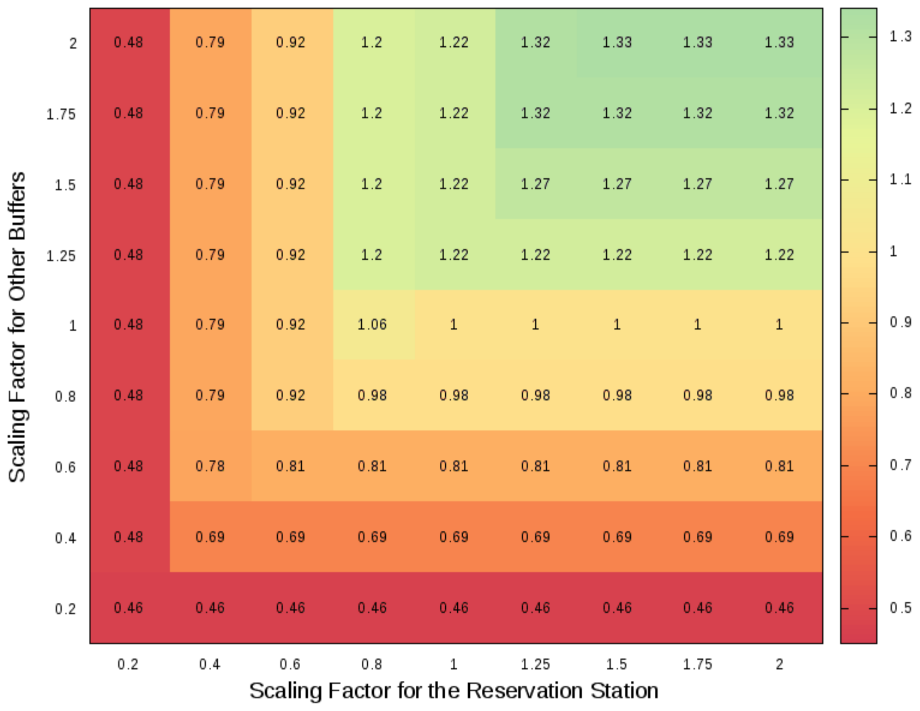


Fig. 8: Sensitivity Analysis: Resource Scaling Speedup (YALES2: Loop 4389)
This heatmap represents the speedup obtainable when scaling the size of the Reservation Station or/and other out-of-order buffers, with regular Sandy Bridge parameters being used as reference (on coordinates (1, 1)).

the consequences of such buffer size changes in terms of hardware complexity and power consumption.

6 Related Work

Code Quality Analyzer (CQA) [2], to which we compared UFS throughout this chapter, is the tool the closest to UFS that we know of: both analyze loops at a binary / assembly level, rely on purely static inputs and have a special emphasis on L1 performance. They actually both use the MAQAO framework to generate their inputs. CQA works in terms of bandwidth, which it assumes to be unimpeded by execution hazards. As its name suggests, it assesses the quality of targeted loops, for which it provides a detailed bottleneck decomposition as well as optimization suggestions and projections. UFS differs by focusing solely on time estimations, accounting for dispatch inefficiencies and limited buffer sizes. It does so by simulating the pipeline's behavior on a cycle-accurate basis, adding accuracy at the cost of speed. Finally, CQA supports more microarchitectures than UFS.

IACA [1] works similarly to CQA, and estimates the throughput of a target code based on uop port binding and latency in ideal conditions. It can target arbitrary code sections using delimiting markers, while both CQA and UFS only operate at the loop level. It does not account for the hazards UFS was tailored to detect, and we consequently expect it to be faster but less accurate. Like CQA, IACA also supports more microarchitectures than UFS.

Zesto [18, 19] is an x86 cycle-accurate simulator built on top of SimpleScalar [20] and implements a very detailed simulation of the out-of-order engine similar to that of UFS. However, as with other detailed simulators like [21], the approaches are very different: it works as a regular CPU simulator and handles the semantics of the simulated program. Its simulation scope is also much wider, with a detailed simulation of branch prediction, caches and RAM. UFS focuses solely on the execution pipeline, and particularly the out-of-order engine. It disregards the semantics, and targets loops directly with no need for contextual information (such as register values, memory state, etc.), making it considerably faster due to both not having to simulate regions of little interest and simulating significantly fewer things. Furthermore, UFS targets Sandy Bridge, Ivy Bridge and Haswell, while to the best of our knowledge Zesto only supports older microarchitectures.

Very fast simulators exist, but typically focus on different problematics. For instance, Sniper [22, 23] uses both interval simulation (an approach focusing on miss events) and parallelism to simulate multicore CPUs efficiently. As said events (cache misses and branch mispredictions) are irrelevant in the cases targeted by UFS (memory accesses always hit L1, loops have no if statements and have large numbers of iterations), the use cases are completely disjoint.

UFS is to our knowledge the only model targeting binary / assembly loops that both disregards the execution context and accounts for dispatch hazards and limited out-of-order resources.

7 Future Work

Evaluating the impact of unmodeled hardware constraints would be interesting to determine whether or not implementing them in UFS could be profitable. Such constraints include writeback bus conflicts and partial register stalls.

The impact of simulating fewer loop iterations should also be studied, as our current default value of 1000 may be unnecessarily high and time consuming.

As our base UFS model is aimed at Sandy Bridge, we could easily construct models for incremental improvements such as Ivy Bridge and Haswell on top of it. However, a validation work is necessary to evaluate their respective fidelities, and see if more microarchitecture-specific rules have to be implemented. Expanding the model to support further "Big Core" microarchitectures (e.g. Broadwell, Skylake...) would also be of interest.

The idea of Uop Flow Simulation can be applied to vastly different microarchitectures (such as the one used in Silvermont cores, or even ARM CPUs), and could have interesting applications beyond performance evaluation tools. For instance, its working out of context means it could easily be used by compilers to better evaluate and improve a generated code's quality.

In terms of codesign, UFS models could be used to quickly estimate the impact of a microarchitectural change on thousands of loops in a few minutes. Coupling this modeling technique with a bandwidth-centric fast-simulation model such as Cape [3] would allow for non-L1 cases to be handled efficiently as well.

8 Conclusion

We demonstrated UFS, a cycle-accurate loop performance model allowing for the static, out-of-context analysis of assembly loops. It takes into account many of the low-level details used by tools like CQA or IACA, and goes further by estimating the impact of out-of-order resource sizes and various pipeline hazards. It can also be used to evaluate how a loop would behave given different microarchitectural parameters (such as different out-of-order buffer sizes or load latencies).

Our Sandy Bridge UFS prototype shows that UFS is very accurate and exposes formerly unexplained performance drops in loops from industrial applications and in vitro codelets alike. Furthermore, it offers very high simulation speeds and can serially process dozens of loops per second, making it very cost effective.

9 Acknowledgements

We would like to thank Gabriel Staffelbach (CERFACS) for having provided our laboratory with the AVBP application, as well as Ghislain Lartigue and Vincent Moureau (CORIA) for providing us with YALES2.

We would also like to thank Mathieu Tribalat (UVSQ) and Emmanuel Oseret (Exascale Computing Research) for performing and providing the in vivo measurements we used to validate UFS on the aforementioned applications.

This work has been carried out partly at Exascale Computing Research laboratory, thanks to the support of CEA, Intel, UVSQ, and by the PRiSM laboratory, thanks to the support of the French Ministry for Economy, Industry, and Employment through the COLOC project. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the CEA, Intel, or UVSQ.

References

[1] Intel: Intel architecture code analyzer (IACA) (Jun 2012), `https://software.intel.com/en-us/articles/intel-architecture-code-analyzer`

[2] Oseret, E., et al.: CQA: A code quality analyzer tool at binary level. HiPC '14

[3] Noudohouenou, J., et al.: Simsys: A performance simulation framework. RAPIDO '13, ACM (2013)

[4] Palomares, V.: Combining Static and Dynamic Approaches to Model Loop Performance in HPC. Ph.D. thesis, UVSQ (2015), Chapter 7. Uop Flow Simulation.
[5] The AVBP code `http://www.cerfacs.fr/4-26334-The-AVBP-code.php`
[6] YALES2 public page `http://www.coria-cfd.fr/index.php/YALES2`
[7] MAQAO: Maqao project. http://www.maqao.org (2013)
[8] Fog, A.: Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus (Mar 2015), `http://www.agner.org/optimize/instruction_tables.pdf`
[9] Djoudi, L., et al.: The design and architecture of maqao profile: an instrumentation maqao module. In: EPIC-6. p. 13. IEEE (2007)
[10] Palomares, V.: Combining Static and Dynamic Approaches to Model Loop Performance in HPC. Ph.D. thesis, UVSQ (2015), Appendix A: Quantifying Effective Out-of-Order Resource Sizes, Appendix B: Note on the Load Matrix.
[11] Intel: 2.2.2.4: Micro-op queue and the loop stream detector (LSD). Intel 64 and IA-32 Architectures Optimization Reference Manual (Sep 2014)
[12] Intel: 2.2.4: The execution core. Intel 64 and IA-32 Architectures Optimization Reference Manual (Sep 2014)
[13] Intel: 3.5.2.4: Partial register stalls. Intel 64 and IA-32 Architectures Optimization Reference Manual (Sep 2014)
[14] Paoloni, G.: How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. Intel Corporation, September (2010)
[15] Koliaï, S., all: Quantifying performance bottleneck cost through differential analysis. In: Proceedings of the 27th international ACM conference on supercomputing. pp. 263–272. ACM (2013)
[16] Moureau, V., et al.: From large-eddy simulation to direct numerical simulation of a lean premixed swirl flame... Combustion and Flame (2011)
[17] Press, W.H., et al.: Numerical recipes: The art of scientific computing (1992)
[18] Loh, G.H., et al.: Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. In: Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on. pp. 53–64. IEEE (2009)
[19] Loh, G.H., Subramaniam, S., Xie, Y.: Zesto (Jan 2009), `http://zesto.cc.gatech.edu`
[20] Burger, D., Austin, T.M.: The simplescalar tool set, version 2.0. ACM SIGARCH Computer Architecture News 25(3), 13–25 (1997)
[21] Binkert, N., et al.: The gem5 simulator. ACM SIGARCH Computer Architecture News 39(2), 1–7 (2011)
[22] Carlson, T.E., et al.: Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In: SC. p. 52. ACM (2011)
[23] Heirman, W., et al.: Sniper: scalable and accurate parallel multi-core simulation. In: ACACES-2012. pp. 91–94. HiPEAC (2012)