This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LCA.2017.2761751, IEEE Computer Architecture Letters



# Multi-Stage CPI Stacks

Stijn Eyerman, Wim Heirman, Kristof Du Bois, Ibrahim Hur
Intel Corporation
{Stijn.Eyerman, Wim.Heirman, Kristof.Du.Bois, Ibrahim.Hur}@intel.com

**Abstract**—CPI stacks are an intuitive way to visualize processor core performance bottlenecks. However, they often do not provide a full view on all bottlenecks, because stall events can occur concurrently. Typically one of the events is selected, which means information about the non-chosen stall events is lost. Furthermore, we show that there is no single correct CPI stack: stall penalties can be hidden, can overlap or can cause second-order effects, making total CPI more complex than just a sum of components. Instead of showing a single CPI stack, we propose to measure multiple CPI stacks during program execution: a CPI stack at each stage of the processor pipeline. This representation reveals all performance bottlenecks and provides a more complete view on the performance of an application. Multi-stage CPI stacks are easy to collect, which means that they can be included in a simulator with negligible slowdown, and that they can be included in the core hardware with limited overhead.

## 1 Introduction

CPI (cycles per instruction) stacks divide the total CPI of an application into components that reflect the time spent in various events. The base component reflects the minimum CPI needed to execute instructions, which equals 1 over the pipeline width. The other components correspond to stall events that make the core perform worse than this ideal performance: instruction and data cache misses, branch mispredictions, dependences between instructions, multi-cycle latency instructions, etc. The size of a component is proportional to its impact on final CPI, such that the sum of all components equals the total CPI. This makes it convenient to visualize a CPI stack as a stacked bar, see Figure 1.

In a superscalar out-of-order processor, CPI stacks are challenging to construct because multiple events can overlap. For example, when an instruction cache (Icache) miss occurs, instructions that are already in the reorder buffer (ROB) can continue executing. Or when an instruction causes a data cache (Dcache) miss, the frontend can still fetch and decode new instructions until the ROB is full.

Previous CPI stack construction proposals solve this by counting the impact of stall events at one stage of the pipeline. For example, the IBM POWER5 CPI stack algorithm [6] considers the commit stage. A stall cycle is defined as a cycle on which no instruction commits. There are two main causes for not committing instructions: (i) the ROB is empty due to a frontend stall (Icache miss or branch misprediction), or (ii) the instruction at the head of the ROB is still executing (due to a long-latency instruction or a Dcache miss).

Similarly, Eyerman et al. [4] propose to measure CPI stacks at the dispatch stage, that is when instructions are entered into the ROB and the reservation stations (RS). Stall cycles are cycles where no (correct-path) instructions are dispatched, either because the frontend does not deliver instructions (due to an Icache or branch miss), or because the ROB or RS is full (due to a long-latency instruction or Dcache miss blocking the head of the ROB).

Yasin [7] proposes to use a hierarchical approach. The top level CPI stack is measured at the dispatch stage, while backend misses (Dcache misses and long-latency instructions) are measured at the issue or commit stage. Because of concurrent miss events, the lower level components do not add to the total CPI. Instead, the performance counter data should be inspected by level: if the top level has a small backend component, one should not consider the lower level backend events, even though they can have a large magnitude. A simple CPI stack representation is therefore not possible.

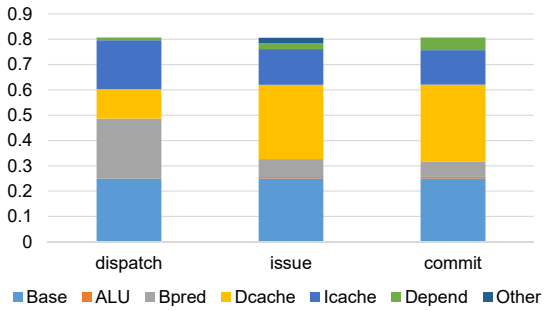


Fig. 1. Example CPI stacks at dispatch [4], issue and commit [6].

All these proposals lead to different CPI stacks, sometimes only slightly different, but sometimes the differences are more expressed, see Figure 1. None of these representations is wrong, nor do they individually reveal all bottlenecks: it is just not possible to summarize all bottlenecks into a single additive stack, because of overlapping events. To show this, we perform the following experiment. We simulate the execution of an application on a core with all stall events modeled and record the CPI. Next, we make one component (e.g., Icache, branch predictor or Dcache) perfect and simulate again. Intuitively, the CPI component of that event equals the difference in CPI between both simulations.

The results for two such experiments are shown in Table 1 (see Section 3 for our setup). Mcf simulated on an Intel Knight's Landing (KNL) core configuration sees a 0.02 CPI reduction if all ALU instructions take 1 cycle. Making the Dcache perfect (but keeping realistic ALU latencies) reduces the CPI by 0.30. However, when assuming a perfect Dcache *and* single-cycle ALU, CPI is reduced by 0.36, which is larger than the sum of the individual improvements. Put differently, if the Dcache is perfect, the ALU component equals $1.11 - 1.05 = 0.06$, while it is only 0.02 initially. The ALU stalls are initially mostly hidden by the Dcache misses, but become apparent when the Dcache

*Manuscript submitted: 10-Jul-2017. Manuscript accepted: 15-Sep-2017. Final manuscript received: 22-Sep-2017.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LCA.2017.2761751, IEEE Computer Architecture Letters



TABLE 1
CPI components by idealizing structures.

| App & core | Config | CPI | Diff. CPI |
|---|---|---|---|
| mcf on KNL | All real | 1.41 | |
| | 1-cycle ALU | 1.38 | 0.02 |
| | perfect Dcache | 1.11 | 0.30 |
| | perf. Dcache & 1-cyc. ALU | 1.05 | 0.36 |
| mcf on BDW | All real | 0.72 | |
| | perfect bpred | 0.39 | 0.33 |
| | perfect Dcache | 0.43 | 0.29 |
| | perfect bpred & Dcache | 0.25 | 0.47 |

misses are removed.

For mcf on an Intel Broadwell (BDW) core configuration, CPI decreases by $0.33$ with a perfect branch predictor and by $0.29$ with a perfect Dcache. Perfect branch prediction *and* a perfect Dcache reduces CPI by $0.47$, which is now *less* than the sum of the individual components. These miss penalties overlap: their combined improvement is smaller than the sum of their individual components. It is impossible to represent both hidden and overlapping stalls in a stacked representation that adds to the total CPI.

We can conclude that a single additive CPI stack is a too simple representation for performance bottlenecks. Instead, we propose to measure multiple CPI stacks at different stages in the pipeline. This reveals all bottlenecks at all stages. The different CPI stacks show the range of the possible CPI reduction if a certain stall event is eliminated. For example, if the Icache component at dispatch is $0.2$, and at commit, it is $0.1$, the potential CPI reduction of eliminating Icache misses is between $0.1$ and $0.2$.

Note that multi-stage CPI stacks cannot *exactly* predict the performance gain by removing a component. An exact prediction requires extensive critical path analysis [5], which considerably slows down simulation or requires complex hardware. Multi-stage CPI stacks give more information than single CPI stacks by providing an upper and lower bound, while at the same time being relatively easy to measure.

We first discuss how the CPI stacks at each of the stages are collected. Next, we discuss our experimental setup and validate our premise that multi-stage CPI stacks are able to estimate the boundaries of the actual CPI improvements, providing more information than single CPI stacks.

## 2 Multi-Stage CPI Stack Construction

The construction of the stacks is similar at each stage. At each cycle, a stall is accounted when no instructions or fewer instructions than the width of the pipeline are processed. On a stall, the ground cause of the stall is inspected. Note that each stage has a separate set of counters, e.g., the Icache miss counter at the dispatch stage is different from the Icache miss counter at the commit stage. We do not discuss TLBs for clarity, but they are included in our measurements: Icache should be interpreted as Icache and ITLB, and similarly for Dcache and DTLB.

Table 2 shows the accounting algorithm at three crucial stages in the core pipeline: dispatch, issue and commit. Similar accounting can be done at other stages (e.g., fetch and decode). In the algorithms, $W$ is the width of the stage (the maximum number of instructions that can be processed per cycle) and $n$ the number of (correct-path) instructions processed in this cycle (which is between 0 and $W$). If the pipeline stages have a different width, we set $W$ to the minimum of all widths, as this will be the bottleneck. FE stands for frontend pipeline (fetch, branch prediction, decode), ROB stands for reorder buffer and RS for reservation stations (or issue queue).

In these algorithms, simplified for clarity, we measure 6 components: the base component (base_comp) for time spent in actually executing instructions; the branch predictor component (bpred_comp) for time spent in resolving branch mispredictions; the instruction cache and data cache component (Icache_comp and Dcache_comp) for time spent in misses in the instruction and data cache; the ALU latency component (ALU_lat_comp) for cycles lost due to multi-cycle latency instructions; and the instruction dependence component (depend_comp) for cycles lost due to limited instruction-level parallelism. An actual implementation could have more components, e.g., differentiating between the different cache levels and TLBs or more structural stalls in the issue stage. All components are initially zero, and keep accumulating until the end of the application.

The algorithms are executed every cycle, and are very similar for all three stages. First, we calculate the fraction $f$ of the width that has been used this cycle and add that to the base component. If $W$ is smaller than the width of the stage, $f$ can be larger than one. In that case, we transfer the part above one to the next cycle. Because we exclude wrong-path instructions (instructions fetched after a branch misprediction), the base component for all stacks is the same, as each correct-path instruction has to traverse all stages. At the end of this section, we discuss how we discern between correct-path and wrong-path instructions.

If the useful fraction $f$ is smaller than 1, we try to find a reason for the stall. Lines 4 to 8 handle frontend stalls: branch mispredictions and Icache misses. Note the different point in time at which frontend miss accounting starts for each stage: when the frontend is empty for the dispatch stage, and when the reservation stations and the ROB is empty for the issue and dispatch stage, respectively. As a result, the frontend miss components at the dispatch stage are always larger than those at the issue stage, which in their turn are larger than those of the commit stage. Conceptually, the frontend stall component at the dispatch stage assumes that instructions that could have been fetched during the frontend stall will have no extra stalls in the backend (through dependences, multi-cycle latency instructions or Dcache misses) and thus can utilize all of the cycles in an ideal execution flow. On the other hand, the frontend stall component at the commit stage assumes that when the frontend stall is removed, the new instructions can only start executing after the ROB is drained, e.g., because they all depend on the last instruction. Clearly, the actual performance gain will be somewhere between these extremes, which is exactly the goal of multi-stage CPI stacks.

Lines 9 through 16 handle backend stalls: Dcache misses, long-latency instructions or dependence chains (which are detected as single-cycle instructions that can only start executing when they are at the head of the ROB because of dependences on older instructions). Here, the commit stage will start accounting sooner than the dispatch stage: when the instruction at the head of the ROB is still executing versus when the ROB is completely full (after the instruction at the head took a long time to execute). The rationale is similar to frontend misses: the dispatch stage assumes that all instructions fetched after the instruction that caused the stall are independent of that instruction and cause no extra stalls. On the other hand, the commit stage assumes that these instructions have to wait until the stalled instruction is finished. Again, the actual penalty is somewhere in between, depending on the characteristics of the application.

The backend miss component is handled slightly differently at the issue stage. Instead of looking at the instruction at the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LCA.2017.2761751, IEEE Computer Architecture Letters



TABLE 2
Per cycle CPI accounting algorithm at dispatch, issue and commit (prod = producer).

| | Dispatch | Issue | Commit |
|---|---|---|---|
| 1 | `f = n/W` | `f = n/W` | `f = n/W` |
| 2 | `base_comp += f` | `base_comp += f` | `base_comp += f` |
| 3 | `if f<1:` | `if f<1:` | `if f<1:` |
| 4 | `  if FE empty:` | `  if RS empty:` | `  if ROB empty:` |
| 5 | `  if Icache miss:` | `  if Icache miss:` | `  if Icache miss:` |
| 6 | `  Icache_comp += 1−f` | `  Icache_comp += 1−f` | `  Icache_comp += 1−f` |
| 7 | `  else if bpred miss:` | `  else if bpred miss:` | `  else if bpred miss:` |
| 8 | `  bpred_comp += 1−f` | `  bpred_comp += 1−f` | `  bpred_comp += 1−f` |
| 9 | `  else if ROB or RS full:` | `  else:` | `  else if ROB head not done:` |
| 10 | `  i = ROB head` | `  i=prod(first non−ready instr)` | `  i = ROB head` |
| 11 | `  if i has Dcache miss:` | `  if i has Dcache miss:` | `  if i has Dcache miss:` |
| 12 | `  Dcache_comp += 1−f` | `  Dcache_comp += 1−f` | `  Dcache_comp += 1−f` |
| 13 | `  else if latency[i] > 1 cyc:` | `  else if latency[i] > 1 cyc:` | `  else if latency[i] > 1 cyc:` |
| 14 | `  ALU_lat_comp += 1−f` | `  ALU_lat_comp += 1−f` | `  ALU_lat_comp += 1−f` |
| 15 | `  else:` | `  else:` | `  else:` |
| 16 | `  depend_comp += 1−f` | `  depend_comp += 1−f` | `  depend_comp += 1−f` |

head of the ROB, we look up the instruction that produces data for the first non-ready instruction. By definition, this instruction is still executing, and prevents its consumers from starting their execution. This is a more accurate instruction to blame than the head of the ROB, which could be an older instruction that is almost finished. However, at the dispatch and commit stage, we do not have the dependency information that is available at the issue stage. This ability makes the case for the issue stage CPI stack, which would otherwise seem useless as its frontend and backend components are always in between those of the dispatch and commit stacks. Furthermore, we can also measure other structural stalls at the issue stage, such as the unavailability of functional units or issue ports, (predicted) memory address conflicts between loads and stores, etc.

*Discerning between correct-path and wrong-path instructions*: In the description of the algorithms, we assume that we can discern wrong-path from correct-path instructions in the dispatch and issue stage. However, this is only possible in a functional-first simulator, where the branch target is known before the timing simulation starts. If functional simulation is done at the simulated execute stage (execute-at-execute simulation model) or the accounting mechanism is implemented in hardware, we cannot make this distinction before the branch is actually executed. Note that there is no problem at the commit stage: wrong-path instructions are never committed.

In case wrong-path instruction detection at the dispatch and issue stage is impossible, we propose to initially treat all instructions as correct-path instructions. The resulting CPI stacks will have a larger base component for the dispatch and issue stage than for the commit stage. Because the commit stage has the correct base component and all base components should be equal, we can take the difference between the dispatch/issue base component and the commit base component and add that to the branch miss component. This will account for the part of the branch miss component related to dispatching and issuing wrong-path instructions.

*Multicore and multi-threaded applications*: Our technique produces one set of CPI stacks per core. For multithreaded applications running on a multicore processor, it can be illustrative to add corresponding elements of the per-core CPI stacks to obtain a full application view. Synchronization stalls can be collected by the operating system, and added as an extra component in the visualization. Eyerman and Eeckhout [3] propose per-thread dispatch stage CPI stacks for simultaneous multithreading (SMT) processors. Their approach can be easily extended to SMT CPI stacks at other stages.

## 3 Experimental Setup

We implement the dispatch, issue and commit stage CPI stacks in the Sniper [2] multi-core simulator. The simulation time increases by 2% compared to the original version of Sniper (which already includes measuring dispatch CPI stacks). We simulate all SPEC CPU 2017 [1] single-threaded benchmarks with the reference input sets (36 benchmark-input combinations). To limit simulation time, we fast-forward 10 billion instructions and simulate 1 billion instructions into detail. Each benchmark is simulated on an Intel Broadwell (BDW; 4-wide out-of-order pipeline) and an Intel Knights Landing (KNL; 2-wide out-of-order) inspired core configuration. All uncore components are scaled down by the socket core count, in order to mimic a fully loaded processor. For example, shared cache size and memory bandwidth is divided by 18 for the BDW configuration, as our BDW socket configuration contains 18 cores.

In order to show the validity of the measured CPI stacks, we also perform simulations where certain components are idealized. In particular we simulate a perfect L1 Icache (each access hits in L1), a perfect L1 Dcache, perfect branch prediction (including perfect target prediction), and single-latency instructions (all arithmetic and logic instructions are finished in 1 cycle). We collect the CPI of each simulation, and deduct it from the realistic simulation CPI to quantify the performance improvement.

## 4 Validation

To validate that multi-stage CPI stacks provide more information than individual stacks, we perform the following study. For the Icache, Dcache, bpred and ALU component, we select the benchmarks for which the component is at least 10% of the total CPI (in any of the stacks). For these benchmarks, we simulate a configuration with a perfect Icache, perfect Dcache, perfect bpred, or single-cycle ALU operations. Next, we calculate the 'error' on the component: the difference between the predicted CPI component and the actual CPI reduction. We calculate the error for each of the three stacks (dispatch, issue and commit), and the multi-stage stack representation, where we assume a zero error if the actual CPI reduction is within the minimum and maximum component. If it is not within the boundaries, the error equals the error of the closest component (of the three stacks).

Figure 2 shows the results for BDW and KNL (the ALU component on BDW was larger than 10% for only one benchmark, so we do not show results for this one). Clearly, the multi-stage

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LCA.2017.2761751, IEEE Computer Architecture Letters



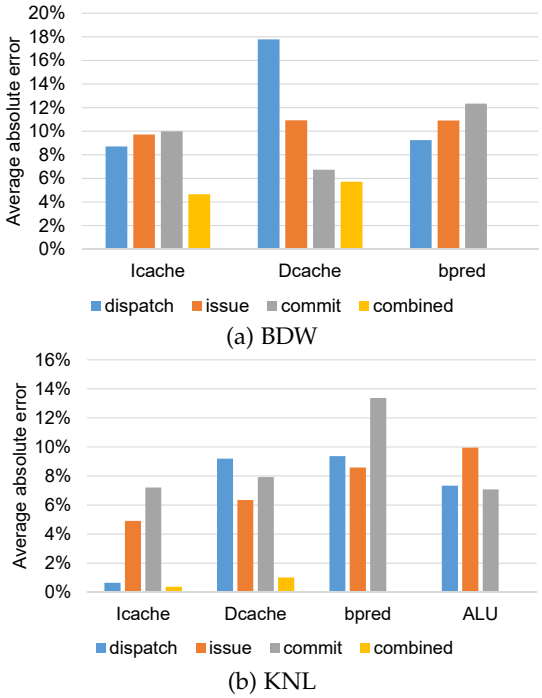


Fig. 2. Error on the components for the individual CPI stacks and the combined multi-stage CPI stacks, on (a) BDW and (b) KNL.

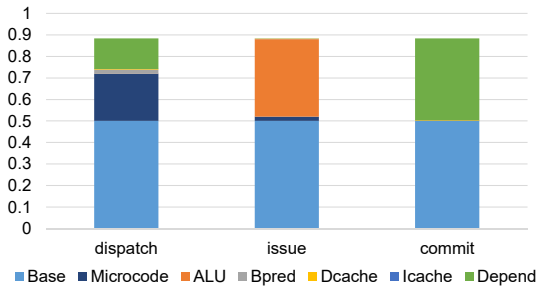


Fig. 3. Multi-stage CPI stacks for imagick on KNL.

CPI stack representation has the lowest error, most significantly for the bpred and ALU components, where the error reduces to 0 (all perfect bpred or 1-cycle ALU CPI reductions are within the boundaries).

The Icache and Dcache components still have a non-zero error. Icache and Dcache misses are highly intertwined, because all caches from level two and higher contain both instructions and data. When the L1 Icache is made perfect, no accesses to the L2 cache are made, and no data elements are evicted by instructions. So making the Icache perfect reduces the L2 miss rate for data, and therefore the Dcache miss component also reduces. And vice-versa, the Icache component reduces when the L1 Dcache is made perfect. These are second-order effects, which are impossible to predict with a simple accounting mechanism. Furthermore, they all lead to a larger CPI reduction than predicted, which is less harmful than having a smaller improvement than predicted.

The results also show that the dispatch stack is on average more accurate for frontend misses (Icache and bpred), while the commit stack is more accurate for backend misses (Dcache). Because the frontend miss components are larger for the dispatch stack than for the commit stack, and vice-versa for the backend misses, frontend and backend misses are often overlapping: their individual performance gains are larger than their combined gain.

The errors on the components of the issue stack is in between that of the dispatch and commit stacks, or even lower than both for KNL. This is because the components are usually in between that of the two other stacks. Therefore, the issue stage CPI stacks may seem superfluous. However, as discussed in Section 2, the issue stage has unique knowledge about the dependences between instructions. As an example, Figure 3 shows the CPI stacks for the imagick benchmark on the KNL core. There is an extra component, namely 'Microcode', which indicates the time spent in dispatching microcoded instructions, which are instructions that consist of multiple micro-operations, and therefore take multiple cycles to dispatch. The dispatch and commit stacks show that the largest stall component is dependences between single-cycle latency instructions. On the other hand, the issue stack shows that instructions mainly wait on multi-cycle ALU instructions. Setting the latency of these instructions to 1 indeed reduces CPI by 0.14, which is about the maximum possible performance improvement, as the microcode component cannot be removed.

## 5 Conclusions

CPI stacks are an intuitive way to visualize performance bottlenecks in a processor core. However, because of superscalar and out-of-order execution, single CPI stacks are an overly simple representation and hide much stall information. We propose to measure multiple CPI stacks at different stages in the pipeline. This representation shows all stall events, indicating the range of the performance improvement that is expected when a stall event is eliminated.

Our experiments show that in most of the cases, the actual performance improvement is within the boundaries predicted by the dispatch, issue and commit CPI stacks. The cases where the performance improvement is not within these boundaries can be attributed to second-order effects that cannot be predicted with a simple accounting mechanism. Furthermore, none of the three stacks is consistently more accurate than the others. Adding the infrastructure to measure the stacks has a negligible impact on simulator performance. We can conclude that multi-stage CPI stacks are a valuable addition to the analysis toolbox of a simulator or to the performance counter infrastructure on a core.

## References

[1] "SPEC CPU2017 benchmark suite," https://www.spec.org/cpu2017/.
[2] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 28:1–28:25, Aug. 2014.
[3] S. Eyerman and L. Eeckhout, "Per-thread cycle accounting in smt processors," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV, 2009, pp. 133–144.
[4] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate cpi components," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII, 2006, pp. 175–184.
[5] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn, "Interaction cost and shotgun profiling," *ACM Trans. Archit. Code Optim.*, vol. 1, no. 3, pp. 272–304, Sep. 2004.
[6] A. Mericas, "Performance monitoring on the power5 microprocessor," *Performance Evaluation and Benchmarking*, pp. 247–266, 2006.
[7] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014, pp. 35–44.