Hierarchical Cycle Accounting: A New Method for Application Performance Tuning

Andrzej Nowak
CERN openlab and EPFL
Geneva, Switzerland
andrzej.nowak@cern.ch

David Levinthal
Microsoft
Redmond, WA, USA
dlevinth@microsoft.com

Willy Zwaenepoel
EPFL
Lausanne, Switzerland
willy.zwaenepoel@epfl.ch

Abstract

To address the growing difficulty of performance debugging on modern processors with increasingly complex micro-architectures, we present Hierarchical Cycle Accounting (HCA), a structured, hierarchical, architecture-agnostic methodology for the identification of performance issues in workloads running on these modern processors.

HCA reports to the user the cost of a number of execution components, such as load latency, memory bandwidth, instruction starvation, and branch misprediction. A critical novel feature of HCA is that all cost components are presented in the same unit, core pipeline cycles. Their relative importance can therefore be compared directly.

These cost components are furthermore presented in a hierarchical fashion, with architecture-agnostic components at the top levels of the hierarchy and architecture-specific components at the bottom. This hierarchical structure is useful in guiding the performance debugging effort to the places where it can be the most effective.

For a given architecture, the cost components are computed based on the observation of architecture-specific events, typically provided by a performance monitoring unit (PMU), and using a set of formulas to attribute a certain cost in cycles to each event. The selection of what PMU events to use, their validation, and the derivation of the formulas are done offline by an architecture expert, thereby freeing the non-expert from the burdensome and error-prone task of directly interpreting PMU data.

We have implemented the HCA methodology in Gooda, a publicly available tool. We describe the application of Gooda to the analysis of several workloads in wide use, showing how HCA's features facilitated performance debugging for these applications. We also describe the discovery of relevant bugs in Intel hardware and the Linux kernel as a result of using HCA.

1. Introduction

Performance tuning plays a crucial role in the quest for higher efficiency, but the increasing complexity of modern processors makes it more difficult to identify the causes of poor performance. Modern processors include a performance monitoring unit (PMU) that provides statistics on the execution of an application, in the form of counts or frequencies for thousands of micro-architectural events, such as, for instance, frontend stalls or cache misses. Their presence is required to provide coverage for the multitude of complex interactions and issues that can occur. Tools, like perf in Linux, make this information available to the user [1]. Choosing the data to collect and its direct interpretation by end users has proven unreasonably difficult and error-prone. For instance, simple values such as those displayed by perf-stat may or may not highlight a possible performance problem, but do not in any case lead to any specific action. In addition, variations between PMUs on various generations of hardware make it difficult to explain changes in application performance (or the lack thereof) on different platforms, in particular for non-experts. There is no guarantee an event, advertised as "generic" by the software – that is having the same name and purportedly measuring the same effect on any architecture – actually really measures the same phenomena on different architectures.

It is natural to attempt to raise the level of abstraction at which execution statistics are communicated to the end user. Ideally, the mundane quirks of PMU data collection would be obscured (such is Yasin's approach in [2]), and the amount of detailed architecture knowledge would be reduced.

Hierarchical Cycle Accounting (HCA) is a new performance tuning methodology defining a hierarchy of higher-level metrics computed from PMU results, and improving on state of the art in several ways. Most importantly, in HCA all metrics are reported in terms of cycles. Such a uniform presentation of results makes them easier for the user to interpret, allowing direct comparison between different metrics in the tree and between architectures. Metric values can be reported per execution, if sourced from PMU counting data, or per code location at various granularities, if sourced from PMU interrupt based profiling data.

The details of creating an integrated analysis using up to hundreds of carefully selected, validated events is not something that should be enforced on the typical code developer (recent work highlights related difficulties [3][4][5]). HCA has been incorporated in a performance tuning tool, called Gooda, publicly available in open-source. Gooda has been in use since 2012, and we report on some of the experience obtained in using the tool over this time. In particular, we provide results from using Gooda on a large scientific application, focusing on how employing HCA and Gooda allowed non-architecture experts to diagnose and optimize certain performance problems.

The contributions of this paper are:

1) HCA, a new method for application performance tuning. It quantifies in cycles the cost of microarchitectural issues, systematizes them and makes an expert tuning methodology more accessible to non-experts.

2) Gooda, a new profiler implementing the HCA methodology on several variants of x86, supporting power 7/8 and ARM Instruction Set Architectures.

3) A set of observations emerging from practical work with microarchitectural events as reported by PMUs.

The outline of the rest of the paper is as follows. Section 2 describes the HCA methodology and its novel features. Section 3 shows in detail how HCA has been implemented on the Intel Ivy Bridge architecture, and briefly discusses possible implementations on other architectures. Section 4 describes the Gooda tool, which incorporates the HCA methodology. Section 5 reports on the use of HCA on example applications. Section 6 discusses related work and draws conclusions.

2. The HCA Methodology

Hierarchical Cycle Accounting (HCA) is a structured, hierarchical, architecture-agnostic methodology for the identification and quantification of known performance issues in workloads running on modern processors.

HCA helps users minimize the number of cycles their programs consume and waste. Its goal is to make a (so far) expert methodology more accessible to a wider audience and to improve the quality of results – in particular in terms of accuracy and the estimated cost of discovered issues. Further, it can enable the comparison of hierarchy components between different architectures. HCA is based on the premise of cycle accounting. In cycle accounting [6], the fundamental metric of merit is a cycle, and cycles consumed by the program can be assigned to specific architectural activities. The approach was popular with IA-64 systems [7] and was implemented in a limited fashion on x86 processors [8]. Our contributions improve on the basic ideas in the following ways.

HCA accounts for cycle usage by means of abstracted architecture-agnostic metrics expressed in core pipeline cycles. Examples include load latency, bandwidth saturation, instruction starvation, branch misprediction – described in section 2.2. For all architectures, at high levels there is a single tree-based decomposition. The architecture-agnostic metrics are constructed as bottom-up sums of event counts times penalties, from carefully selected, validated performance events that have well defined measurable penalties. The events are selected on the basis of positional accuracy of the generated interrupts for profiling, only counting what is desired (so there is a well-defined penalty) and using sufficient numbers of performance events (to ensure the ability to distinguish related problems with distinct optimization approaches). Differences are avoided whenever possible, as the positional accuracy of a difference is unacceptable due to difference in the interrupt skid [9][3] of the events used.

As a bottom-up sum of positionally accurate events is used, the relative importance of issues in different branches and in different regions of code is always comparable even at the finest Instruction Pointer granularities (functions and disassembly) and at the most detailed levels of the tree (e.g., remote socket access to modified cache lines vs. branch mispredictions). This powerful consistency results in never changing the relative size or units of performance data as the largest, easiest to fix issues are evaluated.

The requirement of using a sufficient number of events to identify problems with different solutions defines the quality of the PMU coverage. HCA is architecture-agnostic, but not all problems can be identified on all processors. For example, if the memory access events used to compute load latency due to cacheline movement include anything other than cacheline movements due to retiring loads (e.g., stores, prefetches, cache line movements due to speculative instructions, instruction fetches, and other effects – also discussed in [5]) then that PMU is incapable of producing an accurate answer and does not have coverage for the problem. Similarly, the different treatment of even simple events from this group by different architectures (e.g., Intel and AMD x86) questions the coverage of "generic events". The lack of accurate coverage is one of the dominant problems facing hardware based performance analysis efforts.

2.1 The HCA metric tree and its levels

The core component of HCA is its metric tree, forming a taxonomy of possible issues (Figure 1). The top two levels of the tree are characterized and constructed as follows.

Total cycles are divided into halted and unhalted. The halted state is entered when the OS finds it is blocked from executing anything useful. This is usually an indication that the threads are waiting for interrupts from IO requests. In the halted state the core PMU is frequently powered down (e.g., on Intel processor families SNB, IVB, HSW, or ARM [10]).

The unhalted state can be investigated using the events of the core PMU. Execution cycles can be divided into those with

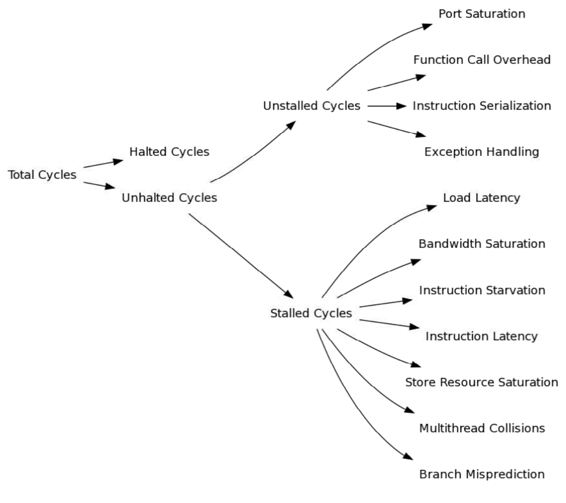


Figure 1: The HCA metric tree

and without pipeline stalls. The exact way these two classes are defined is not of major importance as the objective is to determine the nature and location of the large, easy to fix performance issues, which require a far greater level of detail.

The critical decomposition starts at the third level by defining high level metrics, that might in principle exist on any processor. The categories are shown at the right of Figure 1. These abstracted metrics, measured in cycles, can be directly compared when a given program is run on a range of architectures, as long as each has the event coverage enabling accurate measurement.

Figure 5 in the Appendix shows a block schema for a typical out–of-order processor pipeline. The events measured by the PMU indicate activity at different places in such a diagram. Because an understanding of the measurements ultimately rests on an understanding of the relationship between the abstractions in Figure 1 and the implementation of the measurement in Figure 5, these relationships are highlighted in the discussion of the architecture-agnostic metrics that follows in section 2.2.

As HCA reconstructs its metrics as sums of events from the bottom-up, there is the possibility of temporally overlapping issues causing a cycle overcount. In practice, we observed this is rarely a large issue. When it does occur it shows that more than one issue must be addressed to realistically achieve a net performance gain, so there is actually considerable value in exposing this information. This suggests that additional measurements can be useful and having supplementary informational metrics displayed in an analysis tool can assist in understanding the performance problem rather than obscuring it. These are also critical to discovering problems with the event implementation in the hardware.

2.2 Level 3 – Architecture-Agnostic Metrics

The requirements on the abstracted metrics are that they cover all potential performance bottlenecks and that they are sufficiently fine-grained to convey actionable information, either for performance comparison across architectures or the identification of solvable performance problems. Further, they must be accurately positioned in an interrupt-generated profile [11], otherwise there is no possibility of their being useful for problem identification in an actionable way. Again, this precludes the use of differences or ratios and should only rely on sums if at all possible. For example, a single covering backend metric would not meet such requirements as it leads to no action.

The "stalled cycles" branch covers a range of possible sources of stalls, expressed in a generic fashion and computed as a sum of event counts times penalties (to put all metrics in the units of cycles), so both the count per address and the penalties need to be accurately defined.

2.2.1 Stalled cycles

Load latency estimates the penalty for execution stalls due to long latency data loads. It is used to pinpoint the sources of long loads hitting different levels of the memory hierarchy. This is the most complex metric due to the hierarchy of the cache architecture (Figure 5, items 1, 21, 22) and its many distinct latencies. Thus, to achieve coverage, one PMU event per data source/penalty is required and measured. In addition, they must only count retiring load driven cacheline movement. For CPUs with a DTLB, DTLB translation costs are shown here creating a multi-level structure (see 2.3).

Bandwidth saturation is a condition which occurs when the memory subsystem's data transfer capability is saturated. It is useful for understanding the bottlenecks of memory-intensive code. Measuring the cost of bandwidth saturation is a subtle affair. Measuring the bandwidth is not useful in ascribing a cost in cycles, as bandwidth is a time average and not a cycle-by-cycle measurement. If bandwidth saturation occurs in very short bursts of less than a few milliseconds, it would simply go into a longer time average and will remain undetected. Therefore, to measure the cycles which are bandwidth saturated we detect cycles with large numbers of cacheline retrievals simultaneously in flight. This can be accomplished by monitoring the cycles with high occupancy in the cache miss queues. Bandwidth Saturation conditions can frequently overlap with Load Latency, for example in a gather operation, where Load Latency may account for many times the total cycle count. It is precisely this combination of the two measurements that identifies the nature of the underlying issue. Depending on the location of the queue being monitored, bandwidth saturation can include last-level cache bandwidth saturation. Thus additional cacheline movement count measurements can be used to clarify the issue. Implementation specific details are discussed in section 3.

Instruction starvation occurs when the instruction scheduler, aka reservation station (Figure 5, item 14), has insufficient instructions to schedule effective use of the execution units. This metric, sometimes referred to as simply "Frontend" (FE), estimates the number of idle cycles spent waiting for instructions. It allows to locate sources of code starvation which lowers execution efficiency. In server applications this is dominated by instruction fetch from caches (primarily L2 and above). In client applications substantial contributions can also come from decoding bottlenecks, FE pipeline flow redirection and related effects. Instruction starvation can be measured either by counting cycles where the FE delivers no μops (micro-operations) to the scheduler or by measuring low occupancy in the scheduler. The latter provides higher positional accuracy, being closer to retirement, and avoids counting cycles where a large scheduler already has plenty of μops to keep the execution stages occupied (see sections 3.3 and 5.2). Both techniques suffer from also counting cycles associated with recovering from pipeline flushes due to branch misprediction. Further, store resource saturation can result in the scheduler draining due to blocked μop flow from the FE. Thus overlapping counting is inevitable in all schemes.

Instruction latency estimates cycles of long latency instructions, which can easily create performance limitations. The most common examples are sqrt and divide operations.

This component is frequently only covered and measured by events detecting those two cases.

Store resource saturation occurs upon the exhaustion of the finite resources for allowing store instructions to retire without the data having been committed to cache (store buffers, within item 15 on Figure 5). Detecting this condition helps locate code exerting high pressure on store resources. Latency issues [5] aside, when all such resources are in use, execution can become blocked. Cache coherence requires that stored results become visible in caches in instruction scheduled order. Retrievals of cachelines from long latency sources can in turn result in µop flow being blocked from the FE and the scheduler being drained, causing overlapping counting between the HCA branches. The overlapping counting is identified by seeing both measurements at the same location and the absence of other sources of the scheduler draining.

Branch misprediction occurs when branching prediction mechanisms fail. This metric helps assess and localize the true cost of branching issues. Each branch misprediction (or non-prediction) incurs a series of costs. First, the speculative path is followed until the misprediction is detected. This is followed by flushing the wrong path µops from the pipeline and in parallel starting retrieval of the correct path. We choose to group the long latency fetching of mispredicted paths along with the long latency fetching of correctly predicted paths and collecting this cost under instruction starvation. The short fetches from L1I are included in the cost of recovering from the branch mispredict or non-predict. The measurement of the speculative/wrong path execution currently requires a difference of µops being issued by the frontend and the µops retiring. These measurements can have the full size of the scheduler and even some of the reorder buffer between them. With modern object oriented compiled code resulting in instructions retired/call of ~50, these measurements are likely made in different functions. This is a major positional accuracy limitation on all processors we have considered.

As non-predicted branches are the result of overwhelming the branch history tables, and the solution is to reduce the total number of branches in the code (e.g., debug assertions that never execute), base positional accuracy is not required for this event. Only the overall cost to the programs execution is required to detect if this is an issue.

Multi-thread Collisions (blocks) can occur in processors with hardware threading [9]. Detecting this condition helps minimize the hardware side-effects of threading. For example, the FE and retirement phases of the x86 out-of-order pipelines are in-order and thread collisions may or may not occur at those points. Although we are unaware of any processors that measure these phenomena with dedicated events, the effects can to some degree be evaluated in counting mode if an assumption is made about the correlation of execution. It has been done on occasion, assuming completely uncorrelated execution including terms in spreadsheet calculations [12].

2.2.2 Unstalled cycles

The "unstalled cycles" branch covers a range of potential issues that can occur during execution. The penalties established on this level only cover a part of the unstalled cycles, limited to effects that are actually measureable using hardware PMU based methods.

Port saturation occurs when the workload executes dominantly on a single port (Figure 5, items 90-94). Detecting this condition can lead towards a restructuring machine code to reduce port pressure and thus improve efficiency. As a very rough guideline, if 60-80% of cycles are spent on one port, as measured by dedicated events, it is considered to be attention worthy. A recompilation or partial rewrite may alleviate such issues, for example using SIMD instructions to reduce the number of loads. Of course, such saturation may not represent a problem as in the case of a well coded dense matrix multiply, where the vector FP units are utilized 90% of the time.

Function call overhead estimates the penalty for call handling (managing stack frames). It is a particularly useful metric for large, compiled, object oriented code. Examples are C++ based workloads with a considerable code base (e.g., >100 MB), where many relatively small methods are invoked. This results in a low value of instructions retired/call (30-100). In such cases various housekeeping tasks can account for a large fraction of the total instruction count. Since these characteristics are of importance to the programmer, they should be measured and signaled for high usage functions and methods.

Instruction serialization is low instruction throughput due to dependent instruction serialization. It has not been measureable by a PMU since IA-64. In principle it can be constructed by assembly level analysis, as has been shown in tools like Maqao [13].

Microcode and Exception handling focuses on cases where FP exceptions can be handled through microcode (Figure 5, item 12). The cycles in which such µops are issued can be measured to localize offending code. This also identifies very long microcoded instructions like rep mov, sincos and idiv. In both cases these may be suboptimal and avoided by code restructuring.

2.3 Comments on Levels 4 and 5

In most cases, Level 4 items lead directly to raw events on Level 5. However, items included in this intermediate layer are those that we found useful for tuning. Examples include the DTLB substructure mentioned in section 2.2.1, as well as the cost of branching described in section 3.1.

Overall, Level 4 items help draw parallels between architectures, but may or may not be architecture-neutral. Level 5 is composed entirely of raw events.

2.4 The HCA phases

Apart from the analysis tree, HCA presents a suggestion of several steps to follow for an efficient and correct use of the methodology, in what might be seen as a mini-process.

Event selection – in this phase, key architecture-specific events are chosen by an architecture or PMU expert for future collection. At this point their coverage and penalties are not known exactly.

Validation – in this phase, chosen events are validated to measure what is intended (and nothing else), and penalties are established using microkernels. Although a small number of penalties might be workload-dependent, the differences observed in practice are not large enough to shift the bottleneck to a different bin and generate misleading conclusions. Validation should be carried out by the expert as well, who will embed their knowledge in a tool or a specification, although it can also be conducted by experienced users using microkernels.

Collection – in this phase, performance data is collected on the system under test, using a collection tool. The tool must be capable of collecting architecture-specific events for that machine, but the specific choice of the tool is not of great importance.

Analysis – in this phase, the gathered PMU event data is automatically analyzed, organized into the tree, and later presented to the end user for further interpretation. For counting mode data this can be done with a simple spreadsheet template. Interrupt data (e.g., from perf-record) requires that the interrupt location data, disassembly/asm analysis and compiler generated debug information identifying source lines be incorporated into the analysis. While the user is presented with an abstract tree containing useful tuning information, they still must have access to the underlying event data to identify the exact nature of the issues they need to address.

3. HCA implementation on Intel Ivy Bridge

In this section we present an example implementation of selected parts of the HCA tree using the established Intel Ivy Bridge architecture.

As described in section 2, the HCA tree is composed of three architecture-agnostic levels (Figure 1) and of two virtual levels that are architecture-specific. In this implementation Level 4 mostly contains architecture-specific metrics that support the 11 HCA branches. Level 5 contains the PMU events used as subcomponents for Level 4 metrics.

In the Ivy Bridge case, for a *counting* mode (perf-stat) analysis we make use of nearly 240 PMU events on Level 5, to distill roughly 70 events and metrics on Level 4 [12]. The metrics which do not directly enter Level 3 formulas are displayed for informational purposes, and some of these are devoted to cross checks of measurement accuracy. Although many of the Level 3 metrics could be constructed out of simpler formulas involving fewer events, we attempt to take full advantage of the events already existing in the robust Intel PMU in order to increase accuracy and completeness of interpretation. In a general case, a future PMU could add support for the events defined in levels 1-3, and therefore fill all the branches of the tree directly, without the need for partial metrics. However, doing that would leave the user without a detailed breakdown of the actual performance problems as even the HCA metrics are on the border of being too coarse to be directly actionable.

In the case of a tree built for *profiling* where such a large number of basic events would be prohibitive to data collection limitations (perf-record, vtune [14] and similar tools), the platform-specific low-level event list can be paired down to the minimum required to accurately cover considered performance problems. Whenever possible, the 11 branches are computed directly from the low-level events without intermediate metrics. On levels 1 and 2 (Table 7 in the Appendix), most formulas map directly onto existing events.

When in the halted state, the CPU may enter a low power state and reduce its frequency, therefore to calculate the halted cycle metric, reference cycles coming from the TSC must be used. When measuring stalled and unstalled cycles, we use event masks which count the number of cycles where the measured condition reached at least the specified numerical value ("cmasks"), or which invert the condition to a "less than" ("invmasks"). Therefore here, we measure cycles where at least one μop was retired, or cycles where no μops were retired.

It is worth noting that already a metric as simple as cycles can be tricky to interpret. For instance, depending on the processor family, a generic event of "unhalted cycles" could measure bus cycles (early Intel Core), reference clock cycles, base frequency cycles (Intel Nehalem to Haswell), turbo boosted core pipeline cycles, uncore cycles or some other frequency used in the system. In HCA, most penalties are for on-socket issues, measured in core cycles. The only exception of DRAM access uses a constant cycle penalty.

Using simple metrics such as UNHALTED_CORE_CYCLES on the architected counter and CPU_CLK_UNHALTED:THREAD_P on one of the 4 (8 in HT off mode) programmable counters, it is possible to verify multiplexing quality. This number can be further referenced to a sum of cycles where μops were issued or were not issued.

The metrics for Level 3 are shown in Table 8 in the Appendix. Due to the lack of space, we are unable to discuss every metric in detail, and we present Branch Misprediction in section 3.1 as well as the most complex metric, Load Latency, in section 3.2 and Table 1. We also make a comment on our choice of the Instruction Starvation ("Front End") metric.

3.1 The Branch Misprediction metric

In HCA on Ivy Bridge, costs of mispredicted branches are summed by accumulating the cost of three conditions:

- Branch misprediction events
- Branch Address Calculator clears (BACLEAR)
- Cycles spent on the wrong path.

Splitting branch prediction issues into those components is of high importance: Each of these effects generates a penalty of its own and the first two are solved by different approaches. Furthermore, it is possible to localize with a good degree of accuracy the occurrences of branch misprediction events, but not BACLEARs or cycles spent on the wrong path. This is because BACLEARs are detected at the front end of the pipeline and have very large skid [3], that is the sampling interrupt doesn't occur on the instruction which caused the event. Fortunately, the solution is to reduce total branch count in the program, so the location is unimportant. The last term can only be computed as a difference of a frontend and backend events currently – the distance between the two can be dozens of µops, the size of a typical function - and thus the two measurements could be located in entirely different functions. This would benefit from improvements in future hardware - a µops canceled backend event would help localize the sources of wasted cycles.

3.2 The Load Latency metric

The most complex metric in HCA is Load Latency, and for this reason we present it in detail. Table 1 presents the dominant components on Ivy Bridge and some of the range of problems they can identify. This illustrates the point that while a high-level metric leads to the problem, it is the lowest-level platform-specific events which enable precise identification of the nature of the performance bottleneck, either alone or in many cases in conjunction with other events. The events needed to evaluate the load latency cost also distinguish NUMA, data layout and even thread synchronization problems.

These events only count line movements due to loads and most of them are PEBS events for retiring loads (PEBS, or Precise Event Based Sampling, improves the accuracy of sampling on Intel hardware). Thus they are the most precise measurements available.

The variety of problems identified by the different events and the large variation in the penalties illustrate why this level of detail is required. Although HCA summarizes the penalty in a single metric, it also gives the developer the opportunity to drill deeper so that they can establish a precise fix corresponding to the real reported problem. For example, in this Ivy Bridge case, remote access to modified lines ("remote_hitm") is a completely different problem from local DRAM latency issues, which has different solutions from L3 non snoop access due to the difference in the latency that must be hidden. If these issues were grouped together, the information in most cases would be insufficiently detailed to help the developer.

Table 1: Dominant components of Load Latency

Event	Penalty (cycles)	Identified problem
Mem_load_µops_llc_hit_retired		
Xsnp_none	52	Prefetch/layout
Xsnp_hit	75	threading
Xsnp_miss	75	threading
Xsnp_hitm	85	Thread synch
Mem_load_µops_llc_miss_retired		
Local_dram	200	Prefetch/layout
Remote_dram	400	Numa/pf/layout
Remote_hit_fwd	325	Threading/numa
Remote_hitm	400	Thread synch
Dtlb_load_misses		
Walk_completed	7	
Walk_duration	cycles	Layout/large pages

L2 cache hits and second level TLB hits are not used normally as most or all of the small penalties (6-8 cycles) can easily be hidden through compiler and/or OOO execution load hoisting. Loads blocked from store forwarding are also not used, as the penalties can vary by almost two orders of magnitude and their location cannot be accurately determined with the current event spectrum.

These metrics can be supplemented with additional measurements to add further insight. The 'cycle_activity' event can be very useful, but it is subject to cacheline movements due to speculative loads and some of its components can have intermittent errors in counting. Further, it is not a precisely located event as are the ones HCA depends on.

3.3 Instruction Starvation on Ivy Bridge

The final component explained in detail is Instruction Starvation. It is a simple metric, which on Ivy Bridge is reduced to a measurement of the number of cycles during which the Reservation Station is empty. A commonly used alternative [2][14][1] are cycles where no µops were delivered – but this count is less precise, since it does not address the case in which the reservation station might in fact still have a large pool of instructions to work on already. Figure 2 illustrates the absolute difference between the two measurement techniques, expressed as a percentage of cycles of each hotspot, evaluated on a per function basis on a large C++ application discussed in section 5.2.

3.4 Notes on HCA on other architectures

HCA was originally developed on the Intel Westmere architecture, which was the first Intel processor with sufficient problem coverage by hardware events to attempt it with a real chance of success. In section 0 we describe an optimization case with that implementation.

Furthermore, the IBM Power 7 and 8 families have reasonable problem coverage [15], although determining how good it is

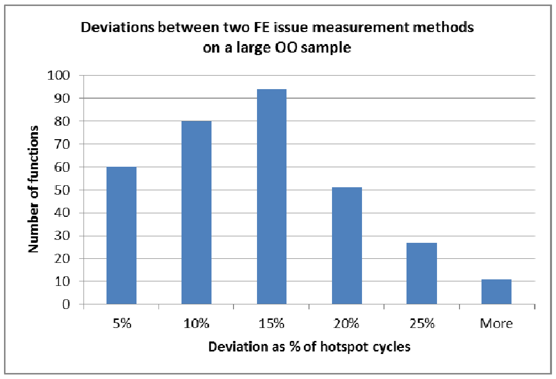


Figure 2: Differences between two Instruction Starvation measurement methods expressed as a percentage of hotspot cycles

requires detailed event validation as described in section 2.3. The Gooda distribution has a collection of micro kernels for event validation for x86 under Linux. Most of these can be ported to the Power instruction set with relative ease.

Other architectures, such as Intel Knights* (Xeon Phi) or Intel Avoton, can be added relatively easily assuming the event coverage is sufficient to make the effort worthwhile.

In particular, HCA is not tied to the architecture itself as much as it is tied to the problem coverage a PMU provides on that architecture. Implementing HCA on an architecture with inadequate coverage is unlikely to give good results.

4. Implementation of the Gooda tool

In this section we present an architecture-independent production profiling tool we implemented to support the HCA methodology. The open-source software, called Gooda, has been published online [12] and was used in successful tuning scenarios on large-scale production code, as we describe in section 5. Architecture-specific examples in this section refer to the Intel Ivy Bridge architecture, and largely to its predecessor, Westmere.

Gooda, which runs on Linux, is a result of a collaboration between Google, LBNL, the ATLAS experiment at CERN, and the CERN openlab. It consists of three main components: linux perf, the analyzer and the visualizer. In addition, scripts for the perf tool are supplied in the distribution for a range of architectures.

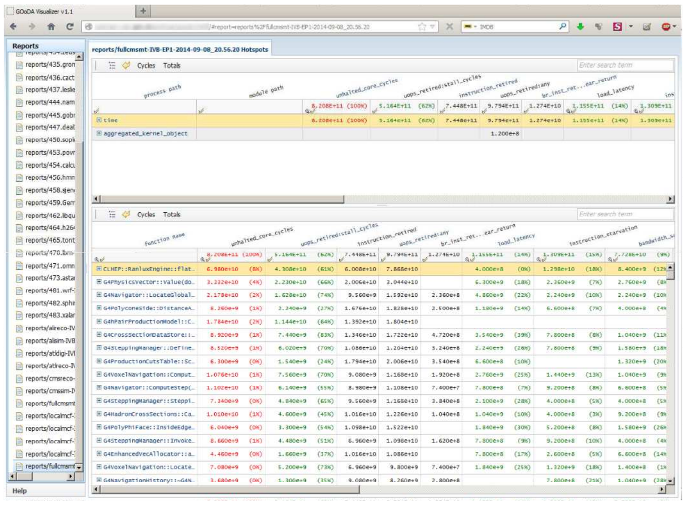
4.1 Performance data collection

The collection of data is performed with the help of perf, in one of two standard modes, with identical overheads as perf itself:

- In counting mode, summary data about an application run is collected in counting mode using “perf-stat”. The result is a single value for each event, without the possibility to drill down into modules, functions, source or assembly lines. This approach is useful for exploration, general workload characterization, platform tuning and comparison.
- In sampling mode (“profiling”, started with “perf-record”), the sampling infrastructure of perf collects data for the events specified for a given architecture. It does so by periodically sampling the location of the Instruction Pointer every time the count of an event reaches a predefined value, and writing a binary data file (perf.data) with records for each interrupt captured. Practical experience, which cannot be widely discussed here due to space constraints, suggests that in order to obtain reliable results from multiplexing, there should be at least one second of steady state workload runtime per each collected PMU event.

4.2 Performance data analysis

The Gooda analyzer, written in C, parses the collected perf.data files. For every supported architecture, an analysis



process path	module path	unhalted_core_cycles		uops_retired:stall_cycles		instruction_retired	uops_retired:any	br_inst_ret...ear_return	load_latency		ins
		8.208E+11	(100%)	5.164E+11	(62%)	7.448E+11	9.794E+11	1.274E+10	1.155E+11	(14%)	1.309E+11
time		8.208e+11	(100%)	5.164e+11	(62%)	7.448e+11	9.794e+11	1.274e+10	1.155e+11	(14%)	1.309e+11
aggregated_kernel_object							1.200e+8				

function name	unhalted_core_cycles		uops_retired:stall_cycles		instruction_retired	uops_retired:any	br_inst_ret...ear_return	load_latency		instruction_starvation		bandwidth_s
	8.208E+11	(100%)	5.164E+11	(62%)	7.448E+11	9.794E+11	1.274E+10	1.155E+11	(14%)	1.309E+11	(15%)	7.728E+10 (9%)
CLHEP::RanluxEngine::flat…	6.980e+10	(8%)	4.308e+10	(61%)	6.008e+10	7.868e+10		4.000e+8	(0%)	1.298e+10	(18%)	8.400e+9 (12%
G4PhysicsVector::Value(do…	3.332e+10	(4%)	2.230e+10	(66%)	2.006e+10	3.044e+10		6.300e+9	(18%)	2.360e+9	(7%)	2.760e+9 (8%
G4Navigator::LocateGlobal…	2.178e+10	(2%)	1.628e+10	(74%)	9.560e+9	1.592e+10	2.360e+8	4.860e+9	(22%)	2.240e+9	(10%)	2.240e+9 (10%
G4PolyconeSide::DistanceA…	8.260e+9	(1%)	2.240e+9	(27%)	1.676e+10	1.828e+10	2.500e+8	1.180e+9	(14%)	6.600e+8	(7%)	4.000e+8 (4%
G4hPairProductionModel::C…	1.784e+10	(2%)	1.144e+10	(64%)	1.392e+10	1.804e+10						
G4CrossSectionDataStore::…	8.920e+9	(1%)	7.440e+9	(83%)	1.346e+10	1.722e+10	4.720e+8	3.540e+9	(39%)	7.800e+8	(8%)	1.040e+9 (11%
G4SteppingManager::Define…	8.520e+9	(1%)	6.020e+9	(70%)	1.086e+10	1.204e+10	3.240e+8	2.240e+9	(26%)	7.800e+8	(9%)	1.580e+9 (18%
G4ProductionCutsTable::Sc…	6.300e+9	(0%)	1.540e+9	(24%)	1.794e+10	2.006e+10	3.540e+8	6.600e+8	(10%)			1.320e+9 (20%
G4VoxelNavigation::Comput…	1.076e+10	(1%)	7.560e+9	(70%)	9.080e+9	1.168e+10	1.920e+8	2.760e+9	(25%)	1.440e+9	(13%)	1.040e+9 (9%
G4Navigator::ComputeStep(…	1.102e+10	(1%)	6.140e+9	(55%)	8.980e+9	1.108e+10	7.400e+7	7.800e+8	(7%)	9.200e+8	(8%)	6.600e+8 (5%
G4SteppingManager::Steppi…	7.340e+9	(0%)	4.840e+9	(65%)	9.560e+9	1.168e+10	3.840e+8	2.100e+9	(28%)	4.000e+8	(5%)	4.000e+8 (5%
G4HadronCrossSections::Ca…	1.010e+10	(1%)	4.600e+9	(45%)	1.016e+10	1.226e+10	1.040e+8	1.040e+9	(10%)	4.000e+8	(3%)	9.200e+8 (9%
G4PolyPhiFace::InsideEdge…	6.040e+9	(0%)	3.300e+9	(54%)	1.098e+10	1.522e+10		1.840e+9	(30%)	5.200e+8	(8%)	1.580e+9 (26%
G4SteppingManager::Invoke…	8.660e+9	(1%)	4.480e+9	(51%)	6.960e+9	1.098e+10	1.620e+8	7.800e+8	(9%)	9.200e+8	(10%)	4.000e+8 (4%
G4EnhancedVecAllocator::a…	4.460e+9	(0%)	1.660e+9	(37%)	1.016e+10	1.086e+10		7.800e+8	(17%)	2.600e+8	(5%)	6.600e+8 (14%
G4VoxelNavigation::Locate…	7.080e+9	(0%)	5.200e+9	(73%)	6.960e+9	9.800e+9	7.400e+7	1.840e+9	(25%)	1.320e+9	(18%)	1.400e+8 (1%
G4NavigationHistory::~G4N…	3.680e+9	(0%)	1.300e+9	(35%)	9.080e+9	8.260e+9	2.800e+8			7.800e+8	(21%)	1.040e+9 (28%

Figure 3: The Gooda tool, main view

template spreadsheet is prepared by experts and included in the distribution. It contains key information about the events and metrics that form the HCA tree on a given architecture. The columns specified in the template mostly belong to specific branches of the HCA tree (Figure 6 in the Appendix). The branch metrics can be expanded in the visualizer to display the contributing components and additional informational metrics or events that are deemed useful (e.g., cycle_activity under Load Latency, offcore_response in bandwidth saturation). The analyzer will also display any events not specified in the template that the user chooses to collect in addition.

Adding analysis for an Instruction Set Architecture (ISA) that is already supported (x86, Power and arm32) requires only making a new template and a few lines of code to identify when the new template should be used.

The output of the analyzer is a directory tree of precomputed spreadsheets of annotated asm and source files, as well as basic block control flow graphs (CFGs). The source, asm and CFG files are generated for either the top 95% functions across all processes and modules monitored (limited to 200). With this data made available, it is highly unlikely the binary data file ever needs to be re-analyzed.

The analyzer can also construct a call count graph of the binary using the Last Branch Records on Intel processors, without any instrumentation in the code or at runtime (Figure 4 in the Appendix).

In most of our experiments on x86, Gooda’s analyzer was approximately two orders of magnitude faster than Intel’s VTune 2013, and delivered results even on complex code.

4.3 Performance data visualization

The Gooda visualizer is a javascript application which makes the interactive visualization, analysis and sharing of performance data possible through a standard web browser, as a regular link (profiles can also be distributed as small single file archives). The visualizer organizes data as precomputed, server-side report, each of which is the result of the analysis of a single perf.data file.

The top part of the screen (Figure 3) accounts performance penalties to binaries (processes and load modules), while the bottom part accounts performance penalties to functions and methods within the available binaries or to selected modules from the upper spreadsheet. Clicking on a function leads to a combined source assembly view and a panning, zoomable basic block control flow diagram (CFG). Hot basic blocks are highlighted in red and orange, and can be navigated dynamically as an interactive map. Clicking a basic block on the map takes the user directly to the related location in the code. The graphical manipulations possible with the visualizer are too extensive to describe here (e.g., call count graph, basic block execution counts, sorting, search).

5. Results

We start our report on the results with a discussion of a kernel that highlights several key properties of HCA. We continue by showing results from a SPEC 2006 benchmark and scientific codes, running in production on hundreds of thousands of cores. The section concludes with a discussion of an Intel microprocessor bug, which was discovered thanks to HCA's cross-architectural portability and internal consistency metrics.

Except for Case 3, the system employed for performance studies was a dual socket server with Intel Xeon E5-2695 v2 CPUs. A total of 24 cores (24 threads) could access 96 GB of DDR3 memory. The frequency of the CPUs was constant, jobs were pinned to their cores and prefetching was turned on. The operating system used was a Red Hat Linux 6.5 clone, on triple Intel 520 series SSDs.

5.1 Case 1: Walker kernel

This synthetic kernel demonstrates how HCA's accurate cycle accounting produces actionable information in its branches, in this case Bandwidth Saturation and Load Latency and the subcomponents of the latter. The kernel initializes a memory structure on core 0 of socket 0 and subsequently walks through it as a randomized linked list, reaching the shared L3 cache, the local DRAM and remote DRAM depending on the size of the buffer created and core on which the walk is initiated – simulating potential real issues with data access.

The Load Latency branch highlights heavy usage, which is quantified in cycles and directly attributed to a particular level of the memory hierarchy. Branches not shown in Table 2 consume between 0 and 1% each. Each cycle consumption value calculated by HCA through memory penalties is proportional to the separately measured actual wall clock time spent in the walking routine for a particular memory level – demonstrating full consistency. The data provided clearly indicates that performance improvements should focus on load latency problems. The reason this is clear to the user is because all costs are expressed in cycles, the relative importance of contributing factors and magnitude of the potential gains in applying individual fixes for each can be compared directly to the bandwidth metric (or any other metrics in HCA for that matter). If the data provided was for instance the number of accesses at each level, it would be far harder for the user to take action on that information.

Table 2: Percentage of program cycles consumed by memory related effects

Level 3	Bandwidth saturation cycles – 13%	Load latency cycles – 87%		
		L3 cycles	Local DRAM cycles	Remote DRAM cycles
Values		4%	26%	53%

5.2 Case 2: SPEC2006 OmnetPP

OmnetPP is a SPEC benchmark with a sizeable (~50%) memory latency component. However, HCA points out another considerable issue of this benchmark, not detected in other methodologies: store resource saturation [9][18]. Gooda quantifies the penalty to be approximately 15% of the cycles. Such a considerable (more than a few %) backlog of stores should be investigated, since it is likely to lead to instruction starvation, draining the reservation station.

One of the culprits is a simple constructor, where storing initial values of the object causes enough pressure on store resources to slow down execution. The values are used as markers in only one place of the code, which suggests that an optimization of this area would save cycles.

Table 3: Relevant HCA branches for a run of OmnetPP

Load Latency	Instruction Starvation	Bandwidth Saturation	Store Resource Saturation	Branch Mispr.
47%	9%	19%	15%	6%

5.3 Case 3: The ATLAS framework

The ATLAS framework is a large, compiled application written in C, C++ and Fortran, spanning 6 million lines of code – a challenge for many profiling tools. It processes data for the ATLAS particle detector at CERN, the European Organization for Nuclear Research. Multiple instances of this scientific application run on a network of 150 datacenters totaling roughly 500'000 cores. The case analyzed in this work is CPU bound and is run on a single core with a single thread.

The experiments described in this sub-section were conducted by experienced scientists who are versed in programming but did not have a formal education in computer science and are not PMU experts [19]. Gooda was used as the profiler on a system with dual Intel Westmere processors, and the example is meant to demonstrate the workflow and outcomes, supported by cycles expressing the cost of issues and thus prioritizing them for tuning non-experts. The metric tree described in Section 2.2 was prepared by a PMU expert earlier, and users received the event documentation distributed with the package.

An initial tuning session reported a large number of stall cycles in one of the top consumers, a part of the code responsible for the handling of the magnetic field generated by the detector. The stall branch of the tree showed a high number of cycles in Load Latency and Instruction Latency in that location. It led to a discovery that the function was still written in Fortran and had not yet been rewritten to C++. A rewrite in C++ doubled the speed of the function thanks to more advanced compiler optimizations being enabled.

A second pass at the newly rewritten piece of code indicated a further problem with stalls taking ~70% of the cycles, and more precisely Instruction Latency caused by the busy divider. 70% of all stalls were accounted to this bin. After an examination of the offending locations, divisions in forms of:

```
const1*dBdphi[1]/r + const2*dBdphi[2]/r…
```

were replaced by inverse multiplications. This optimization produced a 40% speedup in that function, which produced an overall 5-20% improvement for the whole application in this test scenario.

Further, several other functions suffered from further stall problems, coming from Instruction Starvation. Through source code inspection, frequent software vector and matrix operations were discovered in those methods. They were rewritten to support hardware vectorization and achieved a 2.5x speedup per function.

In a last tuning step described here, considerable Call Overhead was signaled on memory allocation routines. Although this issue was already reported earlier by other methods and was therefore known before, the penalty was quantified in the straightforward metric of cycles, and was sufficiently high to trigger an investigation.

5.4 Case 4: High Energy Physics simulation

This case demonstrates how HCA, as implemented in Gooda, correctly identifies and assigns costs to scaling bottlenecks, by showing correct values at Level 3 (supported by detailed metrics at Level 4). It is a generalized and more sophisticated illustration of HCA properties. Geant4 [17] is a complex C++ toolkit for Monte Carlo simulation of particles passing through matter. It is used to build applications in many scientific domains ranging from medicine to aerospace with estimated deployments reaching 300'000 cores world-wide.

Large server code based on toolkits such as this one often suffers from side effects of compiled C++: memory latency, instruction starvation and branch mispredictions, all accounted for by HCA. Indeed, in this sample, which is a proprietary particle physics simulation, HCA shows several such problems of comparable importance.

Table 4: Selected metrics in Simulation

Level 3	**Instruction Starvation – 15%**	**Load Latency – 14%**	**Branch Mispred. – 12%**
Level 4	**R.S. empty – 15%**		

Table 5: Physics simulation: percentage of program cycles for selected HCA metrics and events (starred events are explanatory and not a fractional part of the HCA branch)

Level 3	**Instruction Starvation – 15%**			**Load Latency – 14%**		
Level 4	RS empty	L2 code RD miss*	L2 code RD hit*	miss to DRAM	L2 data RD hit	L2 data RD miss
% of cycles	15%	8%	13.5%	0.04%	6.7%	6.2%
% of parent	100%	N/A	N/A	0.001%	48%	44%

A condition in which no uops are delivered (e.g. idq_uops_not_delivered >= 4) is commonly used to highlight frontend fetch problems. In this benchmark, other methods would display a value of 40% for the number of such cycles, and this might lead the developer to investigate them as a first priority. However, HCA instead constructs the Instruction Starvation metric out of a more appropriate event – RS empty (see 3.3), and thus the value for Instruction Starvation is only 15%. While it is a bottleneck here, it does not stand out on its own, which drives towards a more detailed investigation demonstrating that the issue is actually more subtle than a pure uop delivery problem (see Table 5).

Still in Instruction Starvation, we look deeper to note a relatively high percentage of code read demands active as well as a considerable percentage of code read misses from L2. At the same time, the Load Latency branch shows a high percentage of data related L2 time. This comparison, enabled by the systematic approach enforced by HCA, directly leads to the conclusion that the code and data keep evicting each other, and that the program would benefit from a processor with a larger L2 cache. Examining the counters only, without the context provided by HCA, would not have led to this conclusion as quickly or would have required more expert knowledge from the user. This picture is also supported by observing some bandwidth saturation, but only to L3 and not to DRAM. This case also presents a difficulty – in the Ivy Bridge architecture, instruction fetches (being a frontend event) cannot be reliably localized, and would need a dedicated precise event to be truly traceable.

Further, looking at the Mispredicted Branch component: because all penalties in HCA are expressed as cycles, Branch Misprediction penalties can be ordered by their significance (by function) and dealt with like standard hotspots in a profile, which is not the case in other methods. In a prototype multi-threaded version of this benchmark, a high number of branch non-predictions (see 2.2.1), not monitored by other methods, was traced to conditional statements and asserts used for debugging. Hiding these statements behind compiler ifdefs produced a speedup of roughly 15-18% at 24 threads.

5.5 Case 5: The Intel HT PMU bug

Internal consistency metrics in HCA calculate at times the same value using data from different sources – in this example

Table 6: Intel Ivy Bridge cross-talk between counters

	Westmere	Ivy Bridge HT on	Ivy Bridge HT off
Loads from local DRAM on ht #0	33.5 million	5.1 million	33.8 million
LBR inserts on ht #1	N/A	11.9 million (0 expected)	0 (expected)

we demonstrate the usefulness of this approach. Especially memory events can be complex to work with – for example, the sum of L2 load misses should correspond to the number of L3 load hits and misses. Similarly, the number of L3 load misses should correspond to the number of memory loads retired sourced from L3 misses plus remote cache forwards.

We measured memory events and generated HCA consistency metrics for many workloads with Hyper Threading on and off. An expected result would have been identical values for all metrics regardless of the conditions, such as HT state. Indeed with HT off, that was the case. We observed that the consistency metrics for a group of memory events (MEM_*), similar to those just described, did not align for many workloads with Hyper Threading on, and depending on the architecture used. Our results, obtained while monitoring a steady state workload in parallel to a sleep loop (Table 6) on the same core but on two SMT threads, showed that events from one counter would "leak" to another – decreasing the count in the first counter and increasing it in the second. In the Ivy Bridge "HT on" scenario, a part of the DRAM event "leaks" to the sibling hyperthread of the same core, programmed to count an unrelated event, that should yield exactly 0. The overall sum is exactly half of the real event count, as only one hardware thread is programmed to count the local DRAM event, instead of two.

The net result of this investigation is bug BT243 [20] filed against the Intel Xeon E5 processor family, for which there is no fix as of the time of writing. As a workaround, we measured with Hyper Threading turned off when profiling. A straightforward consequence of the issue is that event counts will be incorrect for certain combinations of counters and events. At the same time, interrupt-driven profiling results will suffer from large positional inaccuracies.[1]

Similar techniques have been used to cross-compare different architectures and same architecture machines with different BIOS configurations, leading to the discovery of PMU related bugs in the Linux kernel, affecting any PMU measurements based on certain memory events (not only HCA).

6. Related work and conclusions

In this work, we demonstrated HCA – a structured, architecture-agnostic methodology for working with performance issues on modern processors. Three other major reference points exist for such activities: the cycle breakdown offered by perf-stat, the "classic" variants of cycle accounting and the Top-Down method published by Yasin [2].

Perf-stat's [1] cycle breakdown is a non-hierarchical approach, based on generic events, not generic metrics. In a standard case, only five-six metrics directly mapped to architecture-specific events are reported, and they often measure different phenomena on different architectures despite bearing the same name. To add yet another example, "last-level cache misses" measures quite different things depending on whether the cache hierarchy ends at the L3 (e.g., Ivy Bridge) or L4 (e.g., Haswell). Any additional information has to come from additional architecture-specific events explicitly programmed by the user.

Also, PAPI [21] works on many architectures and provides facilities for "Event Sets", but does not offer a methodology similar to HCA, and is exposed to risks of generic events. Its strength lies in a universal, portable measurement interface, which Gooda could potentially use in the future.

"Classic" cycle accounting [6][7] introduces structure on a level constrained to halted/unhalted cycles and stalled/unstalled cycles. Stall decomposition is flat and does not attempt to account for such effects as stall overlaps. "Classic" cycle accounting approaches from the past were architecture-specific, although the concept itself does not necessarily have to be – the property that we exploit in HCA to enable cross-architectural comparisons.

Yasin's Top-Down method [2][22] is an impressive and much needed work, similar in spirit to HCA. Top-Down uses retirement slot utilization to assess efficiency and issues. A key difference in HCA is that cycles, an intuitive metric, are uniformly used throughout all levels of the tree and all levels of code granularity, because HCA metrics are constructed as bottom-up sums. This enables straightforward comparisons of metrics from different branches of the HCA tree, allows to quickly assess the relative importance of problems the programmer is presented with, and enables organizing optimization work into ordered hotspots – much like in any commonly used profiler. With that, HCA delivers the same results regardless of the level of analysis – module, function or assembly line. Further, because cycles are used as a base, higher levels of the HCA metric tree are predominantly simple sums of those below. This also allows the controlled mixing of penalties coming from different stages of the pipeline. Overall, HCA has slightly more high-level metrics, which is why it could be considered slightly more general, while Top-Down puts more effort into describing x86-like architectures in detail. This means that HCA has higher chances for support in non-x86 architectures, like IBM's Power series [15] or future versions of ARM. Using cycles as the main unit has one further advantage – informed comparisons between architectures can be made with relative ease (which has already been shown to give results in practice, e.g., in the case of the HT bug).

[1] For counting mode, as long as the events on the two HTs are aligned and added, we have noted that the total is correct. Thus manually multiplexing the events four at a time can produce the correct answers if done carefully as a workaround with HT on.

Acknowledgments

The authors gratefully acknowledge contributions to this work and the Gooda implementation coming from Roberto Vitillo (LBNL), Paolo Calafiura (LBNL), Sami Kama (ATLAS), Rolf Seuster (ATLAS) and Graeme Stewart (ATLAS). We would also like to thank Stephane Eranian (Google) for his extensive support.

References

[1] A. Carvalho de Melo, "The New Linux 'perf' tools." Linux Kongress, 2010.

[2] A. Yasin, "A top-down method for performance analysis and counters architecture," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, 2014, pp. 35–44.

[3] D. Chen, N. Vachharajani, R. Hundt, S. Liao, V. Ramasamy, P. Yuan, et al., "Taming hardware event samples for FDO compilation," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, New York, NY, USA, 2010, pp. 42–52.

[4] V. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, 2013.

[5] J. Diamond, M. Burtscher, J. D. McCalpin, B. D. Kim, S. W. Keckler, and J. C. Browne, "Evaluation and optimization of multicore performance bottlenecks in supercomputing applications," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, 2011, pp. 32–43.

[6] D. Levinthal, "Cycle Accounting Analysis on Intel Core 2 Processors," Intel Corporation, Jan. 2007.

[7] Intel Corporation, "Introduction to Microarchitectural Optimization for Itanium2 Processors (251464-001)," 2002. [Online]. Available: http://cache-www.intel.com/cd/00/00/21/93/219348_software_optimizatio n.pdf. [Accessed: 23-Sep-2014].

[8] Intel Corporation, "Intel64 and IA-32 Architectures Software Developer's Manual," *Volume-1: Basic Architecture*, 64.

[9] D. Levinthal, "Performance Analysis and software optimization for HPC on Intel Core i7, Xeon 5500 and 5600 family Processors," CERN, Jul-2010.

[10] ARM, "Cortex-R4 and Cortex-R4F - Technical Reference Manual (revision r1p4)." 2011.

[11] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, et al., "Continuous profiling: where have all the cycles gone?," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 357–390, 1997.

[12] *Gooda - a pmu event analysis package.* https://github.com/David-Levinthal/gooda, 2012.

[13] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, W. Jalby, et al., "Maqao: Modular assembler quality analyzer and optimizer for itanium 2," in *The 4th Workshop on EPIC architectures and compiler technology, San Jose*, 2005.

[14] Intel Corporation, "Intel VTune Amplifier XE 2013," 2012. [Online]. Available: http://software.intel.com/en-us/intel-vtune-amplifier-xe. [Accessed: 22-Nov-2012].

[15] B. Elkin and V. Indukuru, "Commonly Used Metrics for Performance Analysis POWER7," *IBM Systems and Technology Group*, 2011.

[16] ATLAS Collaboration, "The ATLAS Experiment at the CERN Large Hadron Collider," *Journal of Instrumentation*, vol. 3, no. 08, pp. S08003–S08003.

[17] J. Apostolakis, "Geant4—a simulation toolkit," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 506, no. 3, pp. 250–303, Jul. 2003.

[18] D. Levinthal, "Cycle Accounting Performance Analysis and Software Optimization." Google, 06-Apr-2012.

[19] S. Kama, R. Seuster, G. A. Stewart, and R. A. Vitillo, "Optimizing ATLAS code with different profilers," *Journal of Physics: Conference Series*, vol. 523, p. 012036, Jun. 2014.

[20] Intel Corporation, "Intel Xeon Processor E5 Family Specification Update, 326510-017." Sep-2014.

[21] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proc. Dept. of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[22] A. Kleen, "Pmu-tools, part II: toplev." [Online]. Available: http://halobates.de/blog/p/262. [Accessed: 26-Sep-2014].

7. Appendix

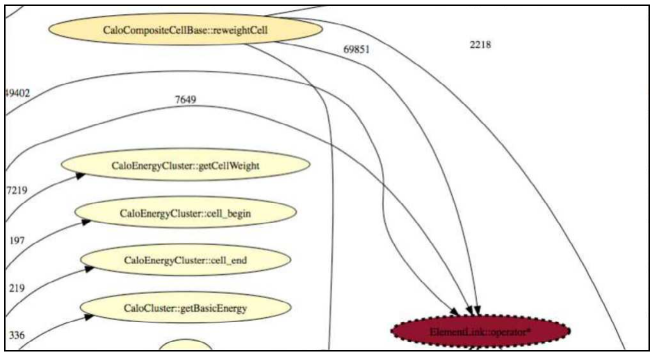
Table 7: Level 1 and 2 HCA metrics on Ivy Bridge

Level	HCA metric	Metric construction	Point of measurement
0	Total cycles	Halted + Unhalted	OS supplied metric
1	Halted cycles	Total - unhalted	
1	Unhalted cycles	cpu_clk_unhalted	
2	Stalled cycles	µops_retired:any:c=1:i=1	retirement
2	Unstalled cycles	µops_retired:any:c=1	retirement

Table 8: Level 3 HCA metrics on Ivy Bridge

HCA metric	Metric construction	Point of detection
Port Saturation	Port usage cycles for each port	Output of RS
Function Call overhead	3*br_instr_retired:near_call	retirement
Instruction Serialization	Cannot be measured at this time	N/A
Exception Handling, ucode	Microcode sequencer active cycles	Front End
Load Latency	Mem*retired:data_source* penalty(data_source) + dtlb_load_cost details shown later	Mostly at retirement
Bandwidth Saturation	Offcore_requests_outstanding:data_rd:c=6	Superqueue
Instruction Starvation	RS_events:rs_empty	RS
Instruction Latency	Arith:div_busy	Execution
Store Resource Saturation	Resource_stalls:st	Allocation

Multithread Collisions	Cannot be measured at this time	N/A
Branch Misprediction	6*(Baclears:any + br_mispred_retired:any) + µops_issued:any – µops_retired:slots	FE, BE, and Allocation



CaloCompositeCellBase::reweightCell
69851
2218
49402
7649
CaloEnergyCluster::getCellWeight
7219
CaloEnergyCluster::cell_begin
197
CaloEnergyCluster::cell_end
219
CaloCluster::getBasicEnergy
336
ElementLink::operator*

Figure 4: A zoomed-in fragment of an LBR-based call graph

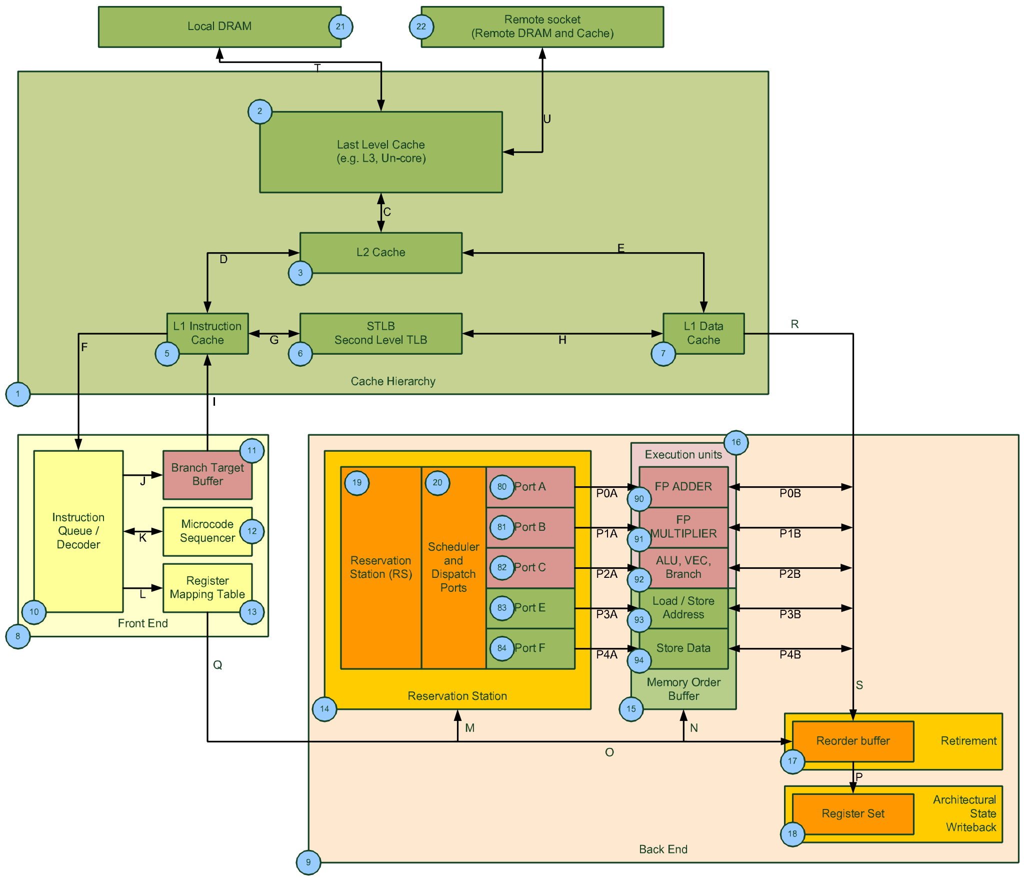
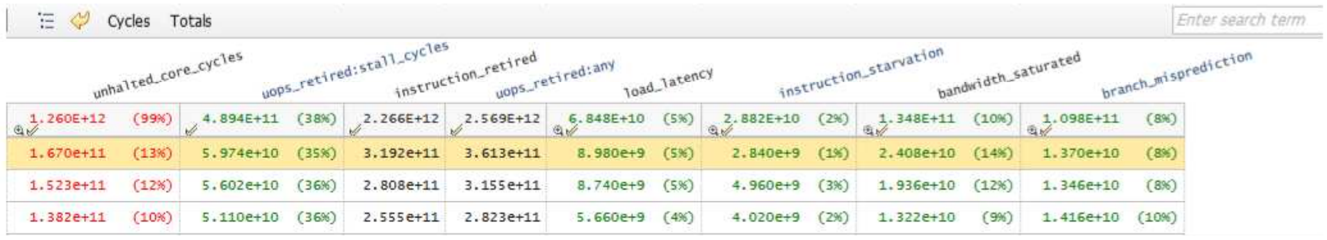


Figure 5: A simplified block diagram of an out-of-order processor core (modeled after "Computer Architecture – A Quantitative Approach" by Hennessy, Patterson and CPU manufacturer material)



unhalted_core_cycles	uops_retired:stall_cycles	instruction_retired	uops_retired:any	load_latency	instruction_starvation	bandwidth_saturated	branch_misprediction
1.260E+12 (99%)	4.894E+11 (38%)	2.266E+12	2.569E+12	6.848E+10 (5%)	2.882E+10 (2%)	1.348E+11 (10%)	1.098E+11 (8%)
1.670e+11 (13%)	5.974e+10 (35%)	3.192e+11	3.613e+11	8.980e+9 (5%)	2.840e+9 (1%)	2.408e+10 (14%)	1.370e+10 (8%)
1.523e+11 (12%)	5.602e+10 (36%)	2.808e+11	3.155e+11	8.740e+9 (5%)	4.960e+9 (3%)	1.936e+10 (12%)	1.346e+10 (8%)
1.382e+11 (10%)	5.110e+10 (36%)	2.555e+11	2.823e+11	5.660e+9 (4%)	4.020e+9 (2%)	1.322e+10 (9%)	1.416e+10 (10%)

Figure 6: A selection of Gooda headers (for illustration only), expandable ones are marked with a magnifying glass