Shortened version to appear in Proc. Parallel Computing (ParCo, Málaga, Spain, 12–16 Sept., 2005).

Holistic Hardware Counter Performance Analysis of Parallel Programs

Brian J. N. Wylie[a], Bernd Mohr[a], Felix Wolf[a]

[a]John von Neumann Institute for Computing, Forschungszentrum Jülich, D-52425 Jülich, Germany

The KOJAK toolkit has been augmented with refined hardware performance counter support, including more convenient measurement specification, additional metric derivations and hierarchical structuring, and an extended algebra for integrating multiple experiments. Comprehensive automated analysis of a hybrid OpenMP/MPI parallel program, the ASC Purple sPPM benchmark, is demonstrated with performance experiments on equisized POWER4-II-based IBM Regatta p690+ cluster, Opteron-based Cray XD1 cluster and UltraSPARC-IV-based Sun Fire E25000 systems. Automatically assessed communication and synchronisation performance properties, combined with a rich set of measured and derived counter metrics, provide a holistic analysis context and facilitate multi-platform comparison.

Contents

1. Introduction

Modern microprocessors have integrated event counters which offer low-overhead access to a potential wealth of execution performance information, encompassing the utilisation and efficiency of various functional units and the memory and cache hierarchy. Although microprocessors from different manufacturers, and also within microprocessor families, provide broadly similar functionality, there are often very significant differences: variation in processor architecture and memory/cache hierarchy are reflected in corresponding event provision, and when combined with restrictions on

Dagstuhl Seminar Proceedings 05501
Automatic Performance Analysis
http://drops.dagstuhl.de/opus/volltexte/2006/507

which (and how many) events may be measured simultaneously this greatly complicates performance measurement and analysis.

Various libraries have addressed the measurement issues, providing a portable application programming interface to event counter control and access (e.g., PAPI [7]). Along with interfacing to system libraries, these offer standardised definitions for the most important and universally available events, and mappings to the native events provided by each microprocessor. Additional events may be derived from one or more native events (if the processor supports their simultaneous measurement) and imposed counter time-sharing/multiplexing may provide a means for approximating the measurement of multiple counters within a single program execution. Although these approaches address the goal of acquiring a richer set of measurements in a particular experiment, it is notable that there is corresponding additional complexity which complicates interpretation. There may also be ambiguities in the definitions of events (such as whether speculative instructions are included in event counts or not) which must also be taken into account during their analysis.

Interpretation and analysis of performance counters has therefore been hindered, limited to a very small subset of the potentially usable events, and often specific to particular processor platforms. One goal of our current work has been to investigate the extent that it is possible to incorporate a wider range of counter metrics, both universal and platform-specific, and exploiting multiple measurement experiments where necessary, for holistic analysis of execution performance.

The analysis of parallel applications executing in distributed and shared memory computer systems is of particular interest, due to the additional complexity and opportunities to exploit comprehensive execution information for improved performance.

1.1. Initial KOJAK approach

Previous developments of the KOJAK performance measurement and analysis environment for parallel programs, which supports many current computer systems, offer a suitable vehicle for pursuing this investigation [1]. KOJAK provides semi-automatic instrumentation of user applications and automatic analysis of performance problems arising from inefficient usage of parallel programming interfaces (such as MPI and OpenMP) [2,3]. Performance problems are classified by type and quantified by severity, for investigation via an interactive browser (CUBE) which presents an integrated, hierarchical view of performance behaviour, call path and process/thread of execution.

A basic infrastructure also exists in KOJAK for measuring counter events and their incorporation into hierarchical analyses alongside communication and synchronisation metrics. One approach extended KOJAK's portable execution tracing to directly include counter measurements and incorporate them in its various analyses [4]. Another incorporates hardware counter analysis from separate platform-specific profiling tools with KOJAK's own execution trace analysis [5]. In both cases, counter measurements/metrics are related to program and system entities (i.e., the call tree, processes and threads) and quantified. While the second approach has a limited separated hierarchy of raw counter measurements, the first was an initial attempt to assess corresponding time penalties and integrate these with KOJAK's directly-measured time-based performance properties.

Quantifying time-penalties for event counts was promising, however, further investigation with additional metrics highlit the limitations of the approach. Where KOJAK identified a metric tuple (call-path and thread) with an occurrence rate above or below a certain threshold, it derived a performance penalty as the entire measured execution time of that tuple; in effect it used an upper bound on the actual penalty, for want of a better approximation. Comparing the derived performance penalties with those directly measured from cycles-based stall counters (on platforms which support them, e.g., UltraSPARC [10]), showed that while they were broadly representative, they were also significantly exaggerated. In this case, the measured penalties could have been used to adjust the

performance penalty derivations to improve their accuracy, though the derivations would inevitably be platform-specific (and it would generally not be possible to quantify the actual penalties). Furthermore, the performance of a tuple is ultimately due to multiple causes, manifesting in multiple counter metrics and also non-counter metrics (e.g., communication and synchronisation times), in complex dynamic relationships, such that it is not possible to accurately determine the time penalty related to a single count measurement. Although the exaggeration of particular performance aspects can be broadly in-line with their actual severity, and as such benefit analysis, in practice it was found to have a detrimental impact on the analysis as a whole, by subtly compromising its integrity.

2. Refined design for hardware counter measurement and analysis

A more robust foundation for incorporating event counts from hardware counters into performance experiments is to integrate them in separate metric hierarchies presented alongside that for measured time metrics. This is particularly the case when larger numbers of counters are measured for analysis. Since it is rare that processors support simultaneous measurement of all of the counters of interest, multiple measurements with subsets of counters may be required, with these partial experiments integrated into a single comprehensive analysis. Assistance can also be provided with specification of appropriate sets of counters for measurement, and multiple presentation hierarchies may be valuable during analysis.

These various aspects have been addressed to refine KOJAK support for counter-based analysis within the existing framework of MPI and OpenMP communication and synchronisation analysis.

2.1. Structured analysis via metric hierarchies

Defining hierarchies of related counter events both provides an improved structure for navigating and interpreting the relationships between events (such as data references encompassing loads and stores, or hits and misses at different levels of cache and memory) and assessing their significance (e.g., cache misses as a proportion of references). In some cases, it can be clear that a single natural hierarchy of related events can be defined. Generally, however, a set of event data may profitably be structured in several hierarchies, where it may not be possible to determine in advance which is most valuable: indeed, the various hierarchies are often complementary rather than redundant. Furthermore, while part of a hierarchy may be platform/processor-independent, it is desirable to be able to include available platform/processor-specific events for a more complete and detailed understanding of execution performance, which itself may well be platform-specific.

For example, consider the hierarchy of caches used to improve the performance of data accesses from memory. When a program requires data that's not already in the processors' registers, then that data must be loaded from system memory (which is typically orders of magnitude slower than processor speed) or one of several hierarchical levels of intermediate storage known as cache: caches closer to the processor (and often actually on the processor itself) are faster, but of necessity smaller in capacity, than caches further out and closer to memory. An unsatisfied data access is attempted from each level of cache (perhaps simultaneously), with the cache providing the data registering a 'hit' and lower caches registering 'misses': along with the required data, accompanying data from the same cache line or block is also loaded into the lower cache(s) so that subsequent accesses which are temporally and spatially proximate will also benefit, however, to do so, some (older) data must be evicted from the lower cache.

A general categorisation of data (and instruction) accesses uniquely associates them with the level of cache or system memory from which they are provided, i.e., where they hit:

```
DATA_ACCESS = DATA_HIT_L1$ + DATA_HIT_L2$ + ... + DATA_HIT_MEM
```

It can also be inferred that misses occurred in lower levels of cache. Data accesses to each level can be reads/loads or writes/stores, offering the next general division:

```
DATA_HIT_L1$ = DATA_LOAD_FROM_L1$ + DATA_STORE_INTO_L1$
```

It is worth noting that this general hierarchy, while applying to a variety of processors and systems, contains elements which will not apply on all: e.g., IBM p690+/POWER4-II [8] has three levels of cache whereas Opteron [9] and UltraSPARC-III/IV [10] only have two, and while the latter can register stores into each level of cache (and memory) the former only registers stores into L1 cache which write-through to the rest. This is readily handled with the proposed structuring, as the inapplicable L3 cache measurements can be treated as zero-valued (i.e., equivalent to a non-functional L3 cache).

Provision of hardware counters also varies considerably by processor/system. Opteron has a counter to measure data accesses directly, so an Opteron-specific definition can be used,

```
DATA_ACCESS = DC_ACCESS                                          # Opteron
```

however, data accesses must be derived from the *composition* of other events on UltraSPARC-III/IV and POWER4-II, and such composed metrics are fundamental to the hierarchical structure. L1 cache read and write hits can not be measured directly by the UltraSPARC or POWER4-II counters, however, they can be determined by a *computation*[1] with measured counters:

```
DATA_LOAD_FROM_L1$ = DC_rd - DC_rd_miss                          # US-3/4
DATA_STORE_INTO_L1$ = DC_wr - DC_wr_miss                         # US-3/4
DATA_LOAD_FROM_L1$ = PM_LD_REF_L1 - PM_LD_MISS_L1                # POWER4
DATA_STORE_INTO_L1$ = PM_ST_REF_L1 - PM_ST_MISS_L1               # POWER4
```

Opteron doesn't provide counters which can distinguish L1 cache read and write hits, or even allow their combination to be measured directly, however, this can also be computed instead:

```
DATA_HIT_L1$ = DC_ACCESS - DC_MISS                               # Opteron
```

Similarly, data load hits from L2 cache on UltraSPARC-III/IV requires the computation:

```
DATA_LOAD_FROM_L2$ = DC_rd_miss - EC_rd_miss                     # US-3/4
```

While such computed metrics provide a valuable means for completing the general hierarchies, when compositions are not available, they don't provide the benefit of extending the hierarchies in the way that composed metrics naturally do. For example, data load hits from L2 cache are composed from multiple native events on Opteron and POWER4-II, respectively:

```
DATA_LOAD_FROM_L2$ = DC_L2_REFILL_O + DC_L2_REFILL_E + DC_L2_REFILL_S # Opt
DATA_LOAD_FROM_L2$ = PM_DATA_FROM_L2
                   + PM_DATA_FROM_L25_MOD + PM_DATA_FROM_L25_SHR
                   + PM_DATA_FROM_L275_MOD + PM_DATA_FROM_L275_SHR # POWER4
```

Although these compositions have quite different constituent measured counters, they naturally extend the general hierarchy with additional platform-specific detail, which can offer further insight for performance tuning on the respective platforms. While each (dual-core) POWER4-II processor has its own local L2 cache, it shares this with the other processors on its multi-chip module (MCM,

[1] The term *computation* is defined as a general calculation which can include subtractions (and potentially other arithmetic operations), whereas *composition* is defined to be strictly additive.

L25) and the processors on the other MCMs in its node (L275), all of which are faster than accessing L3 cache (which is similarly shared), so local versus remote L2 cache accesses impact performance.

This process of deriving hierarchies of new metrics from compositions and computations of available measurements is able to create quite comprehensive structured relationships for data, instruction and TLB accesses (and associated hits and misses), with a general structure extended by additional platform-specific components. Metrics which are not applicable, or can't be derived from available measurements can be omitted. When a composition is only partially satisfied by available measurements, it can still be valuable to retain it, but it should be clearly indicated as incomplete, such as by including '~' in its label. (Where a particular set of measurements include such partially satisfied derivations, these may subsequently be completed when experiments are combined.) Partial computations can have negative values or values in excess of their parent, such that it's generally not prudent to retain them: in most cases, measurements can be grouped such that those required for computed metrics are kept in the same group to avoid this.

Similar structuring can also be applied to the types of instruction processed by various functional units and cycles-based counters for related busy/stall and idle periods. In these cases, more of the measurements are platform-specific and while it's still possible to have a hierarchical relationship, there are typically more 'gaps' corresponding to unmeasurable/unaccounted events. There can also be considerable ambiguity regarding particular events and the counters which measure them. For example, since storing floating-point data is typically done by the floating-point unit (FPU), this is often naturally accounted as a floating-point event (e.g., in `PM_FPU_FIN` or `FP_PIPE_COMPLETION`): where this is not desired, the corresponding event measurement (e.g., `PM_FPU_STQ` or `FP_ST_PIPE`) can be relocated to another category, such as *`MEMORY`*. Often, however, it may not be possible to distinguish the different kinds of events counted by particular functional units. There may also be inconsistency between counting instructions issued and those which actually complete.

While a general classification and hierarchy of a variety of processor events can be developed, it is ultimately necessary to refer to the respective processor manuals (and associated documentation of native counter events) to assess their significance [8–10].

2.2. Flexible metric specification and customisation

Metric structuring which specifies (presumed) relationships between events provides a mechanism for helping to navigate and understand those relationships. While generic hierarchies such as those described offer one particular structuring, alternative or complementary structures may also be defined and preferable in some cases. Measured events which fit no hierarchy must simply be listed separately (as is the case when no relationships are associated with a metric).

A flexible approach is therefore taken, which provides the specification of metric relationships in a text file which is read to configure and structure the analysis: specifications shown in the previous subsection are extracts from such a file. The default specification can then be overridden to provide alternative analyses when desired. Examples of generic and platform-specific metric hierarchy specifications are provided in Appendix A.2.

A specification file also offers convenience during measurement collection, providing definitions of groups of counters which can usefully be collected in the same measurement, i.e., taking into account restrictions on the number and types of events that can be counted simultaneously. (As mentioned previously, measuring certain events together is advantageous when they are required to compute derived metrics.) Examples of counter measurement sets for are provided in Appendix A.1: HPM/PMAPI [8] and PAT [13] also provide and use similar specifications of groups of counters, though these can neither be modified nor extended by users. PerfSuite [14] and HPCToolkit [15] also provide configurable eventlist and derived metric specifications in XML for Intel/Linux platforms,

whereas Paraver [16] and ParaProf [17] support interactive specification of derived metrics, with re-use of such specifications in subsequent analyses.

Although it is possible to use PAPI preset names for counters to create notionally-portable groups, it is preferable to specify platform-specific groups directly in terms of native events (provided by PAPI), since many of the relevant native events have no corresponding PAPI preset definition and combination of presets is still subject to the same platform-specific limitations.

2.3. Holistic analysis via integration of multiple experiments

Analysis of hardware counter measurements, and metric derivations therefrom, can take two broad approaches. The first sticks strictly to what can be reliably determined from a single measurement experiment (as is the case for HPM [8] and Apprentice^2 [13]), and as such is significantly limited by the flexibility and capabilities of the actual monitoring hardware provided by the processor. Several, separate experiments with different sets of measurements may be considered, with the implicit understanding that the execution may be quite different in each case. An alternative uses time-sharing or multiplexing to automatically change the events measured throughout the duration of an experiment, and extrapolate from these partial measurements to a larger set of approximate measurements. [11,12] Whereas this has the convenience and benefit of handling a single execution, it can be compromised by variations in behaviour within the execution (though these may be small if the execution is sufficiently regular and long with respect to the time-sharing period).

Requiring multiple executions is a significant overhead, however, it also provides an opportunity to consider possible run-to-run variations and incorporate them in the analysis. While past results are no guarantee of future performance, they can help indicate what range of performance can reasonably be expected. This is particularly useful for deterministic applications when the hardware configuration is unchanged and executions occur in a relatively controlled (dedicated) environment.

KOJAK's CUBE algebra operators [5] allow experiments to be combined to produce the mean of multiple related experiments or to aggregate experiments containing different hardware counter metrics. Combining both approaches can be used to reduce run-to-run variations and extend the metric analyses to the set of experiments. Furthermore, the difference of two experiments can be calculated to examine variations between them.

The existing merge utility produced an experiment with the union of metrics, call-paths and process/thread measurements in input experiments. This was extended to integrate experiments containing identical call-path and process/thread trees, but different sets of measured and derived hardware counter metrics. Measurements replicated in more than one experiment are averaged, however, measurements contributing to metric compositions which are only partially fulfilled in individual experiments are accumulated to allow the compositions to be completed. Where available, measured metric values are also retained in preference to partially computed or accumulated values.

3. Results

To demonstrate these new KOJAK capabilities, three comprehensive sets of experiments consisting of complementary groups of hardware counter measurements were collected on an IBM Regatta p690+ cluster, Cray XD1 cluster and Sun Fire E25000 (SF25k), using the ASC Purple sPPM v1.1 benchmark [18]. This application uses a simplified piecewise parabolic method (PPM) to solve a 3D gas dynamic problem on a uniform Cartesian mesh. It is written mostly in Fortran 77 (with some C utility routines) and can simultaneously exploit multithreading for shared-memory parallelism and domain decomposition with message passing for distributed parallelism: the double-precision (64-bit) hybrid parallelisation tested used 32 MPI processes each with 2 OpenMP threads. The processes

were partitioned $2 \times 4 \times 4$ in the $X \times Y \times Z$ dimensions, a configuration chosen to offer a reasonably close comparison between the experiments on the different systems, rather than being optimised for any particular system.

Preparation of the instrumented application executables was done by prepending kinst-pomp to the commands that invoke the compiler and linker. This runs a source preprocessor to automatically instrument the application's 12 OpenMP parallel DO loops, 41 explicit barriers and various additional single and master blocks, and link instrumented PMPI and POMP libraries along with the PAPI library for hardware counter measurements. To provide additional context for the analysis, while avoiding overheads associated with automatically instrumenting the entry and exits of every application routine, the program's main phases and the key routines using MPI and OpenMP had also previously been manually annotated with POMP region instrumentation directives [6]. When the instrumented applications are executed in the usual fashion (and with optional hardware counter measurements configured through an environment variable), the instrumented events are recorded in per-thread trace buffers which are subsequently merged into a single trace for each execution.

The experiments used two p690+ nodes of an IBM Regatta cluster (running AIX 5.2 and connected via HPS) consisting of 4 MCMs with 4 dual-core POWER4-II processors, 32 nodes of a Cray XD1 cluster (running GNU/Linux 2.6 and connected via RapidArray network) each with two AMD Opteron 248 processors, and a Sun Fire E25000 (running Solaris 9) with dual-core UltraSPARC-IV processors. On the IBM system, 6 experiments were collected (with up to 8 counters in each), whereas 10 experiments (each with 4 counters) on the XD1 and 18 experiments (each with 2 counters) on the E25000 were required to acquire a comparable level of detail. Several additional experiments were collected to investigate platform-specific performance aspects outside the core analysis hierarchies. These sets of experiments were subsequently incorporated into a single composite analysis experiment for each platform.[2]

3.1. Analysis presentation

KOJAK's CUBE browser presents its analysis in three linked trees, for performance properties, call-tree (or code region), and system tree (machines, processes and threads as appropriate), as shown in Figure 1 which has three views of the analysis of the IBM p690+ composite experiment.

Performance properties calculated from patterns of events identified in the execution traces, are organised in the leftmost tree as a hierarchy from most general to most specific. The hierarchy of time-based metrics (*Time*) includes assessments of MPI and OpenMP overheads, with the remainder of the execution time property (*Execution*) considered to be productive user code. Detailed descriptions for metrics are shown in a separate window when requested. Metrics directly measured or derived from performance counters are shown in additional hierarchies below the timed metrics (or listed separately if they are not associated with any specified hierarchy).

The middle pane has the application call tree as found in the trace of the execution, consisting of regions (often routines) containing other called regions, OpenMP parallel regions (and barriers, etc.), and calls to MPI functions. Alternate views are provided for a flat region/routine profile or grouping by source module. From a selected call-site or called-region, corresponding source code can be shown in a separate window.

Finally, the leftmost pane shows the (physical) machines and nodes and (virtual) processes and threads, similarly hierarchically structured: MPI processes are labelled by their rank, and OpenMP threads by their OpenMP thread identifier, with usually only the master thread (number 0) in each process participating in inter-process MPI communication.

[2] Ultimately, 32 experiments were collected and unified for the SF25k.

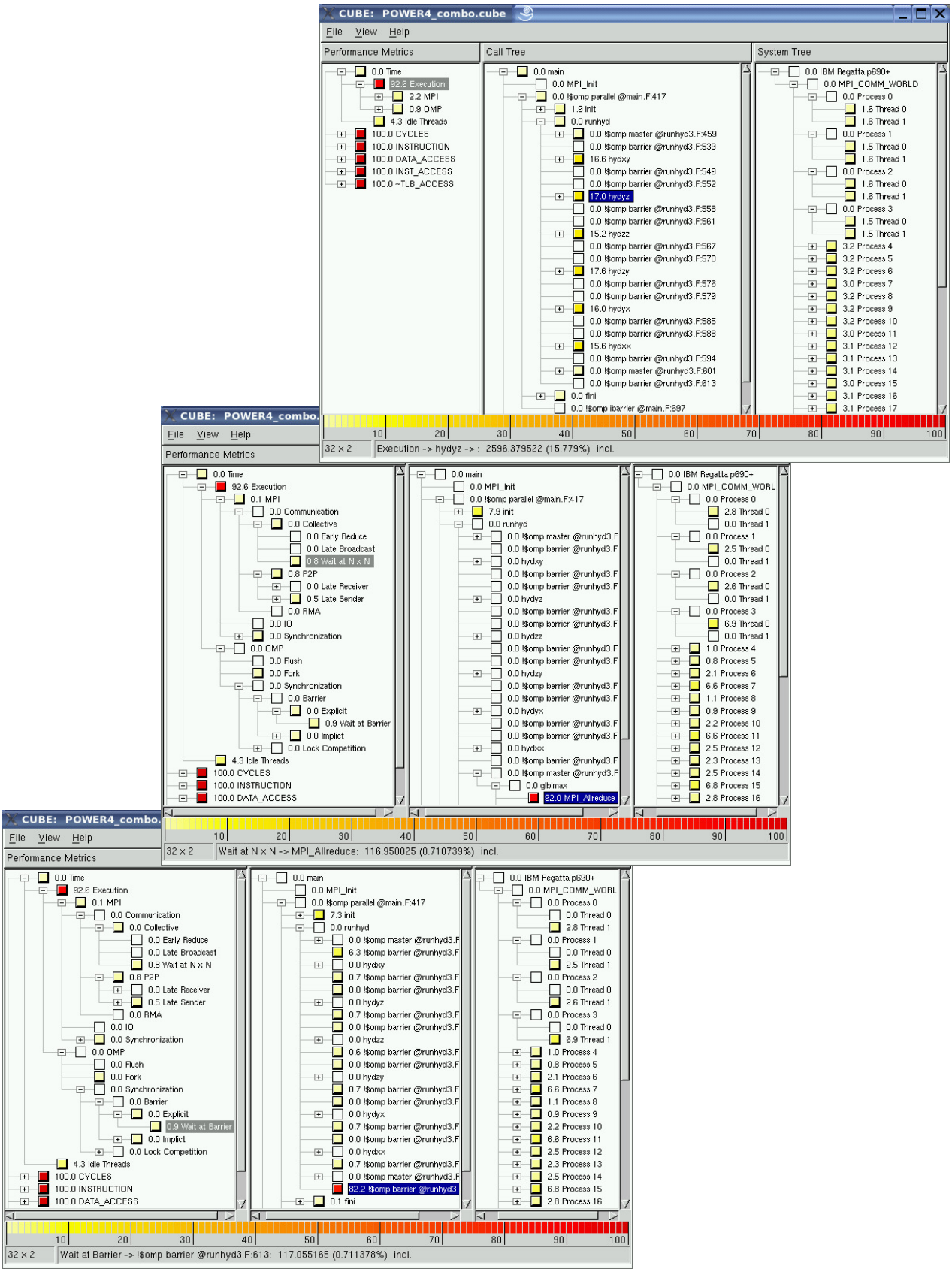

Figure 1. ***KOJAK analyses of hybrid OpenMP/MPI sPPM experiment on IBM p690+ cluster: exclusive Execution Time fairly equally attributed to the six key hydrodynamics routines (upper), MPI Communication Collective Wait at N×N predominantly in*** glblmax MPI_Allreduce ***(middle), and OpenMP Synchronization Explicit Wait at Barrier mostly in final*** runhyd ***barrier (lower).***

The tree-based presentation in each pane allows nodes at any level to be expanded to reveal their children, or closed to conceal them: the metric values shown with nodes are inclusive when they include concealed children or exclusive when expanded and their children are visible and have their own metric values. Selection of a node in the performance metrics tree determines that that metric is shown in the other trees, and the selected node in the call tree further refines the analysis presented in the system tree to only that call-path. Percentage values in the performance metrics pane are relative to the root of their respective hierarchy, whereas the selected metric determines the base value for the percentages in the middle pane, and the call-path/region selected there determines the base value for the percentages in the system tree. Details for the current selection are provided in the status area at the bottom, below the colour scale for the boxes shown with each value.

Selectively opening the nodes in each tree with the most significant values (readily identified by the colours of their associated boxes) provides a straightforward yet powerful mechanism for assessing and refining performance problems, isolating the call-paths where they occur, and reviewing their distribution across processes and threads. An additional graphical display using the virtual or physical topology is available for large-scale applications.

3.2. Comparative experiment analysis

Table 1 summarises the three experiment configurations and execution performance measurements, taken from Figure 1 and similar analyses for the XD1 and SF25k platforms. For this analysis, execution times (and other absolute measurements) are less important than relationships between measurements, whether within a set of experiments or between sets.

Wall-clock execution time of 180s (163s in the runhyd computational kernel) on the XD1 compares with 280s (241s in runhyd) on the Regatta and 885s (867s in runhyd) on the SF25k for each experiment. Parallel initialisation overheads (in the init phase) amount to 1.9% of execution time on the Regatta and 1.8% on SF25k versus 0.5% on the XD1, with the balance attributed predominantly to the runhyd computational kernel, within which the six routines responsible for the hydrodynamics each account for roughly equal shares of the total, and have good load balance over the 64 threads (32 processes) on each platform.

The respective proportions of total execution time attributed to MPI are 1.7% on XD1 and 2.2% on Regatta and significantly larger with 7.6% on SF25k. Investigating each case further, this corresponds primarily to point-to-point communication, with (the master threads of) every fourth process responsible for contributing twice as much as the others. The MPI_Allreduce in glblmax at the end of the main computation loop in runhyd is also found to require a significantly higher collective wait time on the Regatta, totalling 505s (1.0%) on SF25k and 117s (0.71%) on Regatta versus 15s (0.14%) on the XD1.

OpenMP runtime costs on the XD1 are attributed 3.3% of total execution time, which is notably higher than the 1.2% on SF25k and 0.9% on Regatta. These are further categorised as explicit barrier synchronisation wait time in each case. Whereas this is mostly attributed to the six hydrodynamics routines on the XD1, with only 4% in the barrier at the end of the computational loop, on the SF25k and Regatta that final barrier is attributed 87% and 82% respectively.

Some potentially important differences in the MPI and OpenMP communication and synchronisation costs can therefore be seen in the three experiments, however, they also demonstrate broadly similar parallelisation efficiencies.

Proceeding beyond the parallel execution, communication and synchronisation times, additional performance metrics are provided by and derived from hardware counters measurements. While subsets of the counter-based metrics are available in individual experiments, in combination they offer comprehensive insight into the processors' execution.

Platform	**IBM p690+ cluster**	**Cray XD1**	**Sun Fire E25000**
Processor	POWER4-II	Opteron-248	UltraSPARC-IV
Core	dual 1700 MHz	single 2200 MHz	dual 900 MHz
Counter registers	8 (restricted)	4 (unrestricted)	2 (restricted)
Cluster network	High Perf. Switch	RapidArray	Fire Link (unused)
Operating System	IBM AIX 5.2	GNU/Linux 2.6	Sun Solaris 9
Compiler	IBM XL 9.1	Portland Group 6.0	Sun Studio 10
MPI	POE 4.2	Cray MPICH 1.2.6	HPC ClusterTools 5
SMP nodes	2 (full, dedicated)	32 (full, dedicated)	1 (partial, shared)
MPI processes	4x4/node	1/node	32/node
OpenMP threads	2/process	2/process	2/process
Time [sec.]			
Wall	280	180	885
> init (inclusive)	5 (1.9%)	1 (0.5%)	14 (1.8%)
> runhyd (inclusive)	241 (97.8%)	163 (99.2%)	867 (97.7%)
Total	16455	10856	49769
> Execution (exclusive)	15240 (92.6%)	9956 (91.7%)	44742 (81.1%)
> MPI	364 (2.2%)	182 (1.7%)	3772 (7.6%)
> > Comm P2P	218	153	3118
> > Comm Coll Wait at NxN	127	17	527
> > > glblmax MPI_Allreduce	117 (92.2%)	15 (89.3%)	505 (95.7%)
> OpenMP	142 (0.9%)	354 (3.3%)	588 (1.2%)
> > Synch Expl Wait at Barrier	142	354	580
> > > last barrier in runhyd	117 (82.2%)	15 (4.2%)	505 (87.1%)
Counter metrics [10^9]			
CYCLES	25911	22090	38789
> STALL		15499 (70.2%)	24581 (63.4%)
INSTRUCTION	15377	17678	21197
> FLOATING_POINT	7208 (46.9%)	11704 (66.2%)	8649 (40.8%)
> BRANCH	1287 (8.4%)	967 (5.5%)	923 (4.3%)
> > BRANCH_MISP	90 (0.6%)	14 (0.1%)	29 (0.1%)
INST_ACCESS	4202	6575	8793
> INST_HIT_L1$	4201 (100%)	6575 (100%)	8789 (100%)
INST_TLB_MISS	0.008	0.002	0.004
DATA_TLB_MISS	5.014	5.753	2.991
DATA_ACCESS	5235	7456	6887
> DATA_HIT_L1$	5092 (97.3%)	7230 (97.0%)	5594 (81.2%)
> DATA_HIT_L2$	129 (2.5%)	194 (2.6%)	1255 (18.2%)
> DATA_HIT_L3$	7 (0.1%)		
> DATA_HIT_MEM	7 (0.1%)	32 (0.4%)	37 (0.5%)
> > DATA_LOAD_FROM_MEM	7	20	11
L2 cache locality	99% of loads		56% of misses
L2 cache store mix		90% of accesses	78% of accesses, 7% of which RTO

Table 1. ***Hybrid OpenMP/MPI sPPM experiment configurations & execution statistics summary.***

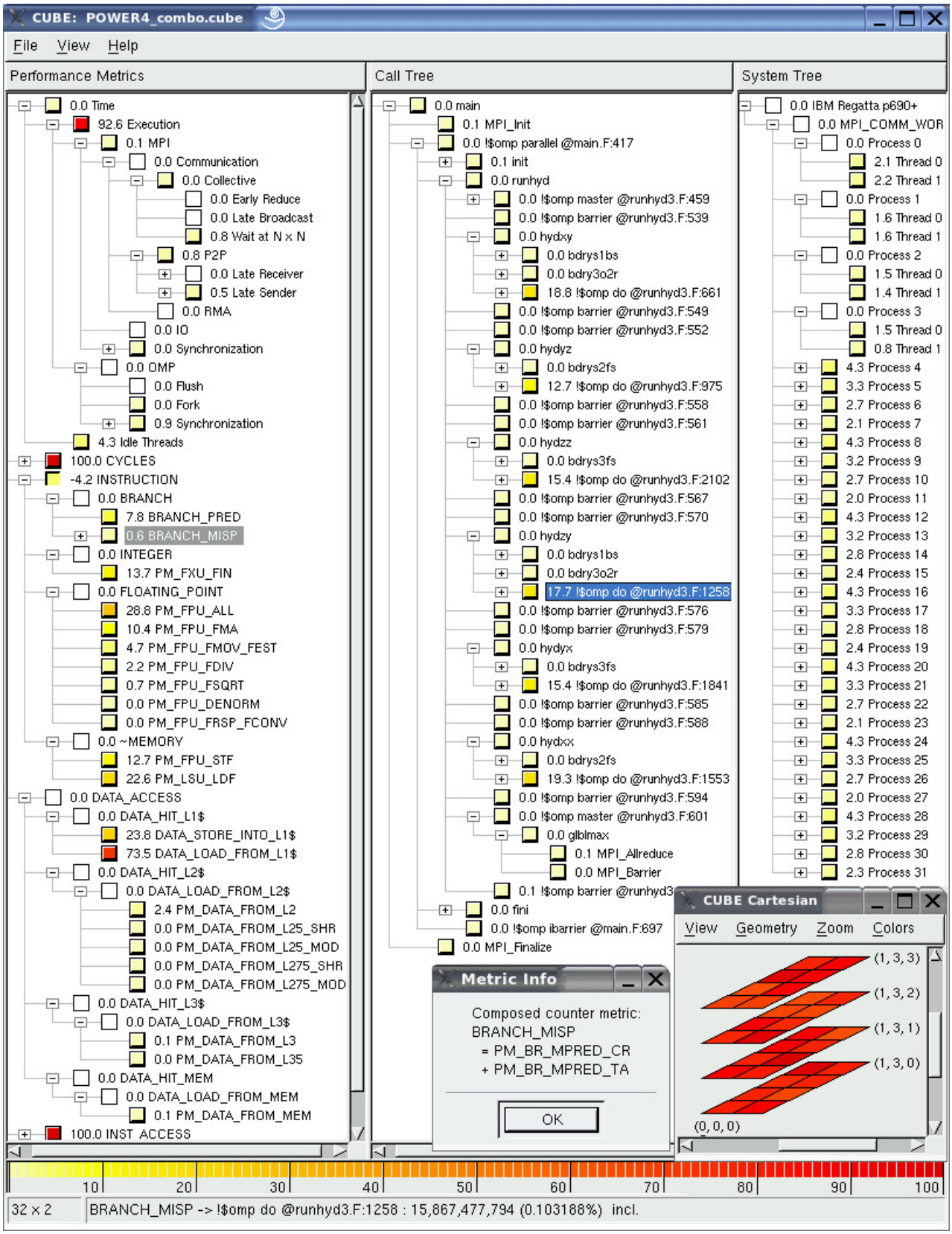

Figure 2. ***KOJAK analysis of 6 combined hybrid sPPM executions on POWER4-II-based IBM Regatta investigating mispredicted branches `BRANCH_MISP` in the parallel loop of one key routine.***

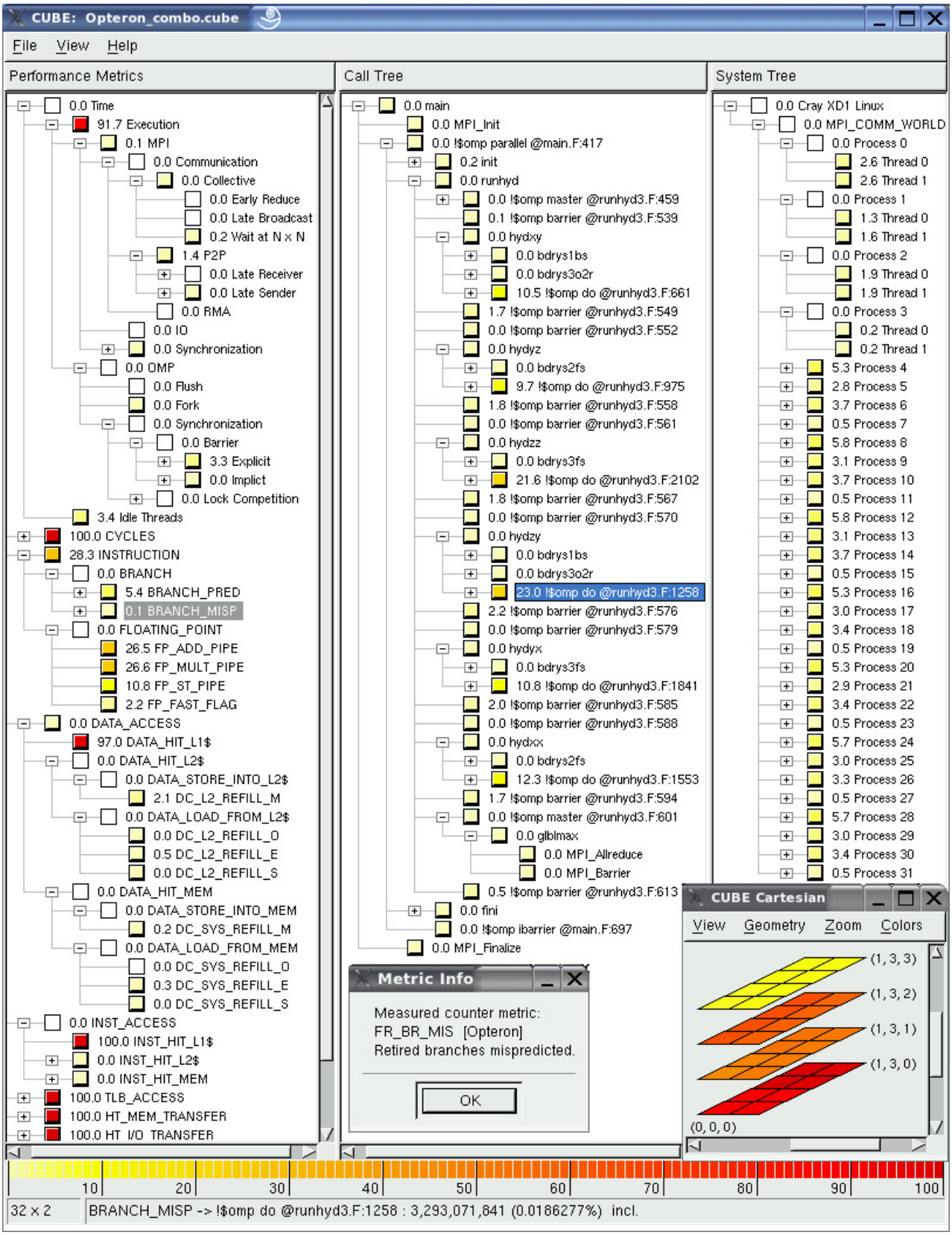

Figure 3. ***KOJAK analysis of 10 combined hybrid sPPM executions on Opteron-based Cray XD1 investigating mispredicted branches*** ***`BRANCH_MISP`*** ***in the parallel loop of one key routine.***

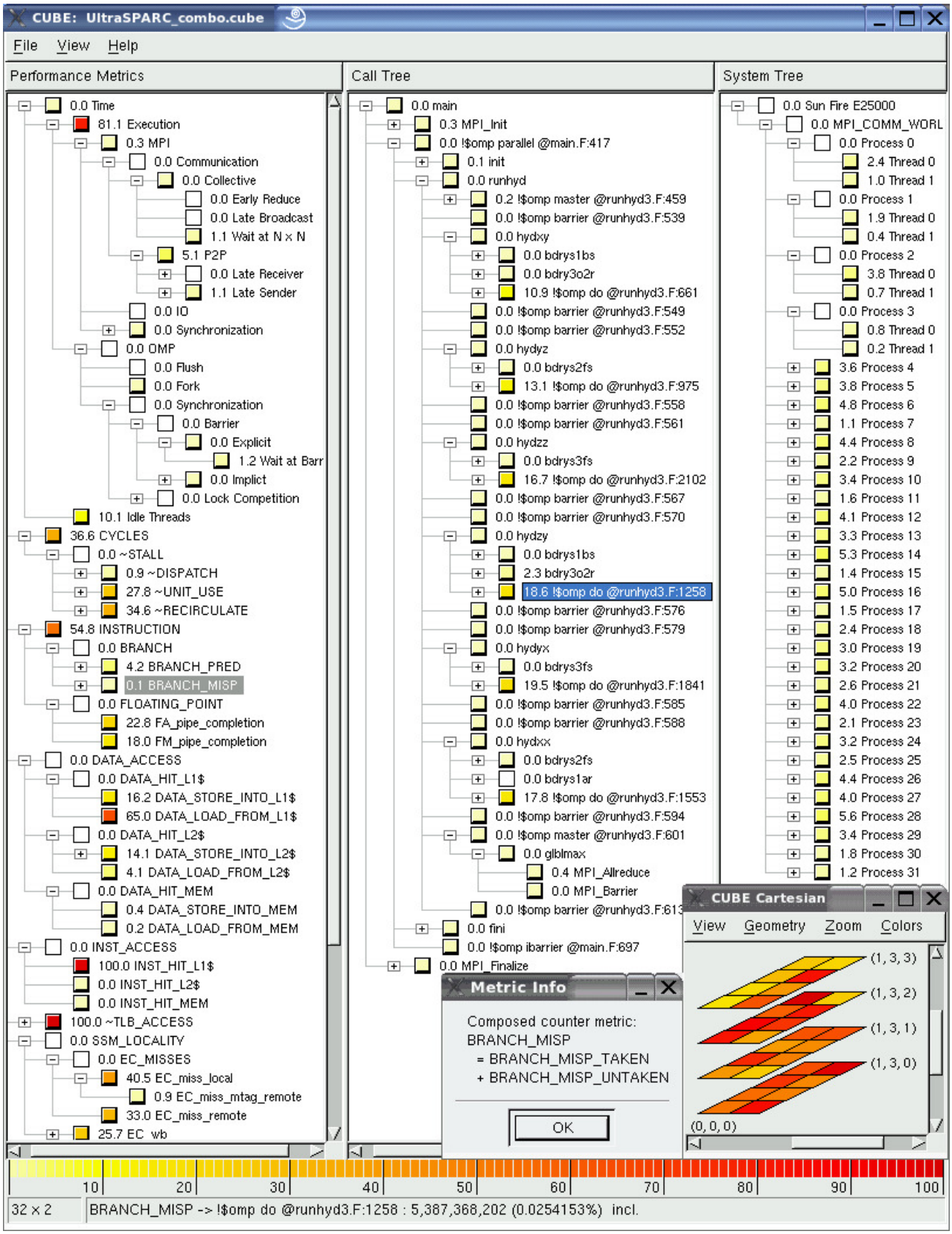

Figure 4. ***KOJAK analysis of combined hybrid sPPM executions on UltraSPARC-IV-based Sun SF25k investigating mispredicted branches `BRANCH_MISP` in the parallel loop of one key routine.***

Figures 2, 3 and 4 show partially expanded integrated metric hierarchies derived from hardware counter measurements for each platform, focussing on the proportion of mispredicted branches (*BRANCH_MISP*). While relatively small in each case, at 0.58% of all instructions (7.0% of branches) it is considerably larger for POWER4-II[3] than the 0.08% (1.5%) of Opteron and 0.14% (3.1%) of UltraSPARC-IV. Depending on the selected call-path, it is also seen to vary considerably by thread, with some threads notably more affected than the others. The distribution is most readily seen from the virtual process topology display: Opteron has a particularly pronounced distribution. Examining the respective counters which measure branch stall cycles allows this to be investigated further.

Figure 5 shows that both POWER4-II and Opteron processors have 97% of data accesses hit L1 cache versus only 81% for UltraSPARC-IV, however, it is the increasingly costly accesses that miss L1 cache and must be satisfied from higher caches and memory that are most significant and warrant further investigation. On Regatta p690+ loads which miss L1 cache are seen to come predominantly from local L2 cache (`PM_DATA_FROM_L2`), which combined with its large L3 cache means that only 0.14% require to come from memory. UltraSPARC-IV and Opteron have smaller, two-level caches, however, and whereas only 0.16% of UltraSPARC-IV data accesses are loads from memory, at 0.27% for Opteron it is almost twice as high as POWER4. Furthermore, only 56% of the UltraSPARC-IV L2 data access misses are found to be local (`EC_miss_local`), with the remainder satisfied from remote processor boards in the SF25k (`EC_miss_remote`).

This integration of platform-specific measurements, within hierarchies of generic metrics derived from the available hardware counters, supports ready identification of performance-critical aspects of parallel execution which can be refined in their detail.

4. Future work

The existing hardware counter metric measurement sets and presentation hierarchies defined for Opteron, POWER4 and UltraSPARC-III/IV should be complemented with similar specifications for other platforms supported by KOJAK and PAPI: e.g., IBM BlueGene/L (PowerPC), Cray X1, MIPS, Alpha, Intel Pentium and Itanium. In each case, the available counters and their relationships need to be carefully investigated to guide the drafting of appropriate specifications.

The current specifications can also be augmented with additional measurement sets and hierarchies, and alternatives compared. These have the potential to provide extra value and insight, though too many could become awkward and confusing. Where appropriate, modification or replacement of those currently provided should be considered.

Cycles-based metrics could be readily converted to times in seconds using the processor clock frequency, if this were recorded in the traces collected. Additional recording of the type of processor would also assist with selection of appropriate measurement sets and hierarchies.

The integration of separate hardware counter measurement analyses into combined analyses could be made more robust by only reading base counter measurements from each input experiment and doing the calculations of derived metrics (according to the default or provided specification) during the merge. Determination of the completeness of computed metrics, and whether incomplete derivations should be retained, is best undertaken with all measurements available. Potential inconsistencies between the preliminary analyses with Expert using different metric hierarchy specifications would also be avoided. These benefits come at the cost of duplicating this analysis in the initial analysis and ultimate integration steps.

[3] A partial explanation for the larger absolute number of mispredictions reported for POWER4 is that the provided counters record issued (rather than completed) branches: the impact on the branch misprediction rate is unclear.

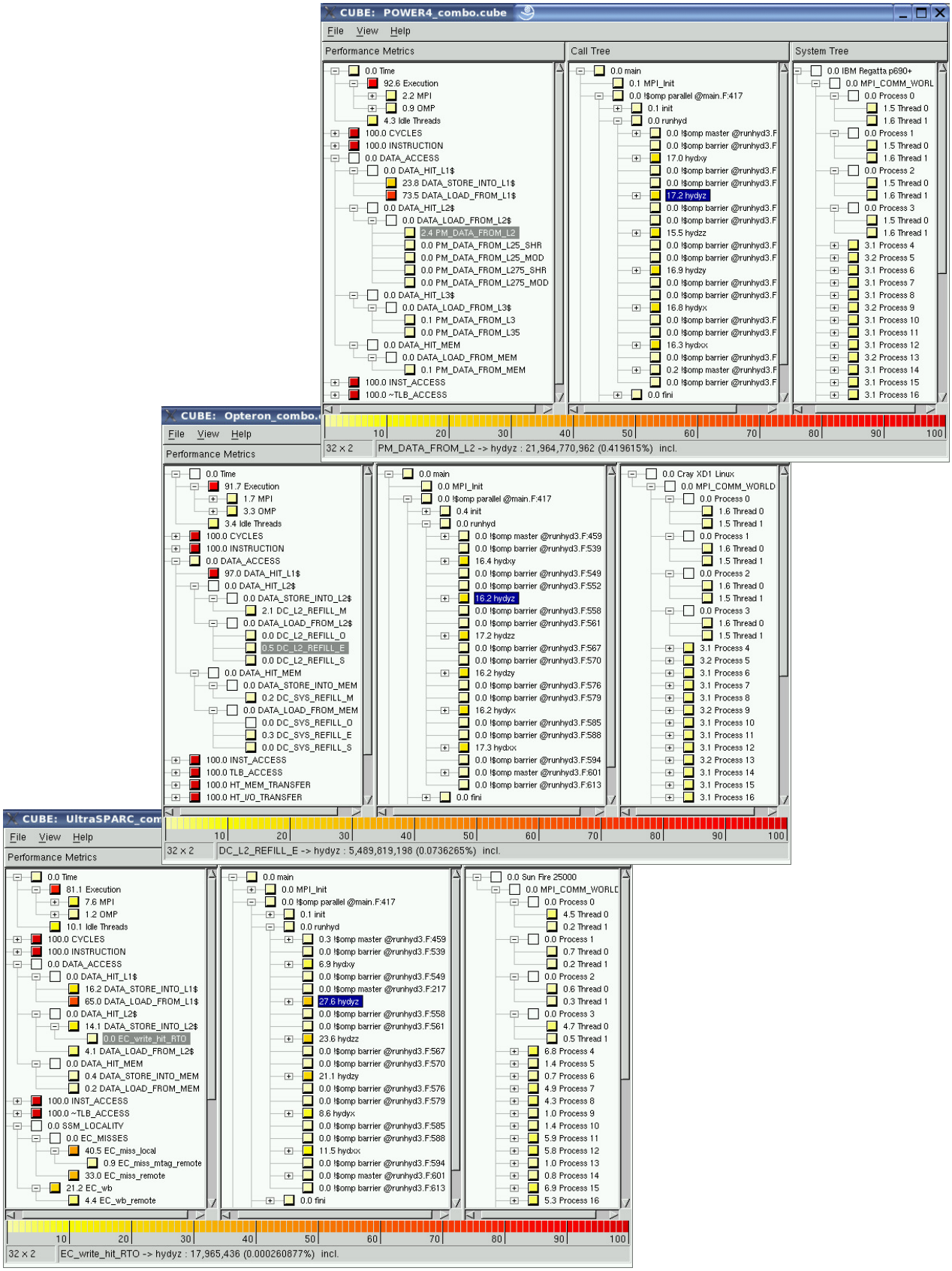

Figure 5. ***KOJAK analyses of hybrid OpenMP/MPI sPPM experiments' L2 data cache accesses, with platform-specific details: IBM p690+ L2 loads from local MCM cache*** `PM_DATA_FROM_L2` ***(upper), Cray XD1 L2 load hits in exclusive coherency state*** `DC_L2_REFILL_E` ***(middle), and Sun Fire 25k L2 store hits with read-to-own bus transaction*** `EC_write_hit_RTO` ***(lower).***

It is highly desirable to include metrics calculated as ratios of counter measurements, e.g., for cache miss rates, instructions per cycle (or cycles per instruction), and floating-point operations per second (FLOPS). These cannot be calculated incrementally as each metric measurement is processed, as currently, and would require instead to be calculated in a second phase after the base measurements are complete (which would be natural if metric derivation is done during integration).

Significantly more awkward is the fact that ratio calculations do not provide a containment property that would allow them to be presented in the CUBE hierarchical display. Furthermore, calculations derived from short intervals and having small denominator values may be unboundedly large, and these will be more likely to be located in the deepest sections of the metric, call-tree and processor/thread hierarchies.[4] This complicates the choice of colour scale used to guide the eye and navigate to the most significant metrics, as these become 'hidden' by lesser values nearer the root.

In addition to ratio metrics, it is worth investigating higher level performance properties based on hardware counter measurements (perhaps in combination with non-counter measurements). For example, a property for "poor cache utilisation" might be ascertained from observations of particularly low cache hit rates, and perhaps further categorised by type according to the predominance of cold, conflict or capacity misses.

This improved understanding of the performance impact of counter-based metrics, might lead to a reliable and accurate determination of their time severity which would allow their integration with the primary hierarchy of time-measured metrics. Severity determination would still need to respect inter-relationships between metrics, such that times represented by hardware counter metrics (such as stalls on cache misses) which occur during communication or synchronisation are not multiply accounted.

Finally, convenience would be provided by a utility which automatically ran a subject application on a target platform with each of the appropriate sets of measurements, performed the preliminary analysis on each measurement and ultimately integrated the results into a comprehensive unified analysis report.

5. Conclusion

Refinement of KOJAK's hardware-counter-based analysis retained much of the existing measurement, recording and analysis infrastructure, with the incorporation of functionality for more convenient counter-metric measurement specification, additional metrics derivable from measured metrics, and customisable structured metric hierarchies. Furthermore, the algebra for integrating multiple experiments was extended to consolidate experiments containing (sub)sets of counter-based metrics and produce unified experiments with all of the available measured and derivable metrics.

Unified experiments, containing communication and synchronisation metrics combined with a rich set of counter metrics, support comprehensive holistic analysis of parallel programs: execution inefficiencies may be isolated to particular processors (or threads) and their various functional units, or found to relate to the use of shared and distributed caches and memory within modern computer systems. The portable CUBE format of analyses also allow fuller comparison between platforms, where architectural differences may be significant. These capabilities contrast those of existing tools which can also offer detailed platform-specific analysis when appropriately directed by knowledgable users, but without a holistic overview and context, or multi-platform comparison.

[4] Zero-valued measurements clearly require special treatment as denominators, however, arbitrarily small measurements can also appear at any level of the hierarchies.

Acknowledgements: Use of the Sun Fire E25000 kindly provided by the Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen Rechen- und Kommunikationszentrum (RZ).

References

[1] Forschungszentrum Jülich GmbH ZAM and University of Tennessee Innovative Computing Laboratory: "KOJAK: Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks," `//www.fz-juelich.de/zam/kojak/`

[2] Felix Wolf and Bernd Mohr: "Automatic Performance Analysis of Hybrid MPI/OpenMP Applications," J. Systems Architecture, 49(10–11):421–439, Elsevier, Nov. 2003.

[3] Felix Wolf: "Automatic Performance Analysis on Parallel Computers with SMP Nodes," PhD dissertation (RWTH Aachen, Germany), NIC Series, Vol. 17, Forschungszentrum Jülich, 2003.

[4] Felix Wolf and Bernd Mohr: "Hardware-Counter based Automatic Performance Analysis of Parallel Programs," Proc. Conf. on Parallel Computing (ParCo'03, Dresden, Germany), Parallel Computing: Software Technology, Algorithms, Architectures & Applications, pp. 753–760, Elsevier, 2004.

[5] Fengguang Song, Felix Wolf, Nikhil Bhatia, Jack Dongarra, and Shirley Moore: "An Algebra for Cross-Experiment Performance Analysis," Proc. Int'l Conf. on Parallel Processing (ICPP'04, Montreal, Canada), pp. 63–72, Aug. 2004.

[6] Bernd Mohr, Allen D. Malony, Hans-Christian Hoppe, Frank Schlimbach, Grant Haab, Jay Hoeflinger, and Sanjiv Shah: "A Performance Monitoring Interface for OpenMP," Proc. 4th European Workshop on OpenMP (EWOMP 2002, Roma, Italy), Sept. 2002.

[7] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci: "A Portable Programming Interface for Performance Evaluation on Modern Processors," Int'l J. High Performance Computing Applications, 14(3):189–204, 2000.

[8] Luis A. DeRose: "The Hardware Performance Monitor Toolkit," Proc. 7th Int'l Euro-Par Conf. (Manchester, UK), Lecture Notes in Computer Science, Vol. 2150, pp. 122–131, Springer-Verlag, Aug. 2001.

[9] Advanced Micro Devices, Inc.: "BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors," Pub.#26094, Rev. 3.14, Apr. 2004.

[10] Sun Microsystems, Inc.: "UltraSPARC Processors," `//www.sun.com/processors/manuals/`

[11] John M. May: "MPX: Software for Multiplexing Hardware Performance Counters in Multithreaded Programs," Proc. 15th Int'l Parallel & Distributed Processing Symp. (IPDPS'01, San Francisco, USA), IEEE Computer Society, Apr. 2001.

[12] Frédéric Parienté: "Performance Analysis and Monitoring using Hardware Counters," `//developers.sun.com/solaris/articles/hardware_counters.html`, Dec. 2001.

[13] Cray, Inc.: "Cray Performance Analysis Tool-set (PAT & Apprentice^2)," `/opt/xd-tools`, Feb. 2005.

[14] Rick Kufrin: "PerfSuite: An Accessible, Open Source Performance Analysis Environment for Linux," Proc. 6th Int'l Conf. on Linux Clusters (LCI-05, Chapel Hill, USA), Apr. 2005.

[15] John Mellor-Crummey, Robert Fowler, Gabriel Marin, and Nathan Tallent: "HPCView: A Tool for Top-down Analysis of Node Performance," Journal of Supercomputing, 23(1):81–101, 2002

[16] Jordi Caubet, Judit Gimenez, Jesús Labarta, Luiz DeRose, and Jeffrey Vetter: "A Dynanmic Tracing Mechanism for Performance Analysis of OpenMP Applications," Proc. Workshop on OpenMP Applications and Tools (WOMPAT'01, Purdue, USA), July 2001.

[17] Robert Bell, Allen D. Malony, and Sameer Shende: "ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis," Proc. 9th Int'l Euro-Par Conf. (Klagenfurt, Austria), Lecture Notes in Computer Science, Vol. 2790, pp. 17–26, Springer-Verlag, Aug. 2003.

[18] John Engle: "The ASC Purple sPPM Benchmark Code," Lawrence Livermore National Laboratory, USA `//www.llnl.gov/asc/purple/benchmarks/limited/sppm/`, Feb. 2002.

[19] PAPI Group: "PAPI Standard Events by Architecture," University of Tennessee at Knoxville, Innovative Computing Laboratory, USA, `//icl.cs.utk.edu/projects/papi/presets.html`, Oct. 2004.

A. Definition of counter measurement sets and derived metric hierarchies

KOJAK provides specifications of convenient measurement sets which account for hardware restrictions and intended derivation of metrics and structuring hierarchies. Both can be readily modified and extended.

A.1. Hardware counter measurement sets

The following tables present sets of native counters (named according to PAPI v3.0) grouped together for convenient measurement on Opteron, UltraSPARC-III/IV and POWER4-II platforms. Sets are selected to maximise the ability to derive further metrics, with particular emphasis on being able to complete compositions and especially computations: this also naturally groups related counters, either as peers or hierarchically. A further aim is to minimise the total number of sets required to complete the core metric hierarchy and complement those with additional optional metrics.

In the tables, derived metrics measured directly with a native counter are shown bold (and related with the '=' operator), and when a native counter contributes to a composed metric and provides a platform-specific extension to the hierarchy it is also shown bold (related with the '+' operator). Computed metrics (and the associated computation) with native counters are shown below in *slanted* font (and distinguished with the '$\doteq$' operator). Each group of derived metrics is preceded by the root metric of the hierarchy to which they contribute to via composition, unless the root metric is measured directly. (For brevity, intermediates in these hierarchies are omitted.) Where derived metrics cannot be fully completed by the native counters in the set, they are distinguished by prepending '**~**' to their names: these metrics may be completed when several sets of measurements are accumulated.

Table 2. ***Opteron counter sets (maximum of 4 in group, unrestricted assignment)***

Set Name	Derived Metrics	op.	Native Counters / *Computations*
OPTERON_TLB	***TLB_ACCESS***		
	DATA_TLB_MISS	=	DC_L1_DTLB_MISS_AND_L2_DTLB_MISS
	DATA_TLB_HIT	=	DC_L1_DTLB_MISS_AND_L2_DTLB_HIT
	INST_TLB_MISS	=	IC_L1ITLB_MISS_AND_L2ITLB_MISS
	INST_TLB_HIT	=	IC_L1ITLB_MISS_AND_L2ITLB_HIT
OPTERON_IC	**INST_ACCESS**	=	IC_FETCH
			IC_MISS
	INST_HIT_L2$	=	IC_L2_REFILL
	INST_HIT_MEM	=	IC_SYS_REFILL
	INST_HIT_L1$	$\doteq$	*IC_FETCH – IC_MISS*
OPTERON_DC1	**DATA_ACCESS**	=	DC_ACCESS
			DC_MISS
			DC_L2_REFILL_I
			DC_SYS_REFILL_I
	DATA_HIT_L1$	$\doteq$	*DC_ACCESS – DC_MISS*
OPTERON_DC2	**~***DATA_ACCESS*		
	DATA_STORE_INTO_L2$	=	DC_L2_REFILL_M
	DATA_LOAD_FROM_L2$	+	**DC_L2_REFILL_O**
	DATA_LOAD_FROM_L2$	+	**DC_L2_REFILL_E**
	DATA_LOAD_FROM_L2$	+	**DC_L2_REFILL_S**
OPTERON_DC3	**~***DATA_ACCESS*		
	DATA_STORE_INTO_MEM	=	DC_SYS_REFILL_M
	DATA_LOAD_FROM_MEM	+	**DC_SYS_REFILL_O**
	DATA_LOAD_FROM_MEM	+	**DC_SYS_REFILL_E**
	DATA_LOAD_FROM_MEM	+	**DC_SYS_REFILL_S**

Set Name	Derived Metrics	op.	Native Counters / *Computations*
OPTERON_ETC	**INSTRUCTION**	=	FR_X86_INS
	CYCLES	=	CPU_CLK_UNHALTED
			FR_HW_INTS
OPTERON_BR	~ *INSTRUCTION*		
	BRANCH	=	FR_BR
	BRANCH_MISP	=	FR_BR_MIS
			FR_BR_TAKEN
	BRANCH_MISP_TAKEN	=	FR_BR_TAKEN_MIS
	BRANCH_MISP_UNTAKEN	≐	*FR_BR_MIS – FR_BR_TAKEN_MIS*
	BRANCH_PRED	≐	*FR_BR – FR_BR_MIS*
	BRANCH_PRED_TAKEN	≐	*FR_BR_TAKEN – FR_BR_TAKEN_MIS*
	BRANCH_PRED_UNTAKEN	≐	*FR_BR – FR_BR_MIS – FR_BR_TAKEN + FR_BR_TAKEN_MIS*
OPTERON_FP	~ *INSTRUCTION*		
	~ *FLOATING_POINT*	+	**FP_ADD_PIPE**
	~ *FLOATING_POINT*	+	**FP_MULT_PIPE**
	~ *FLOATING_POINT*	+	**FP_ST_PIPE**
	~ *FLOATING_POINT*	+	**FP_FAST_FLAG**
OPTERON_ST1	~ *CYCLES*		
	~ *CYCLES*	+	**FP_NONE_RET**
	~ *CYCLES*	+	**IC_FETCH_STALL**
	~ *IC_FETCH_STALL*	+	**FR_DECODER_EMPTY**
	~ *FR_DISPATCH_STALLS*	+	**FR_DISPATCH_STALLS_QUIET**
OPTERON_ST2	~ *CYCLES*		
	~ *FR_DISPATCH_STALLS*	+	**FR_DISPATCH_STALLS_FULL_FPU**
	~ *FR_DISPATCH_STALLS*	+	**FR_DISPATCH_STALLS_FULL_LS**
	~ *FR_DISPATCH_STALLS*	+	**FR_DISPATCH_STALLS_FULL_REORDER**
	~ *FR_DISPATCH_STALLS*	+	**FR_DISPATCH_STALLS_FULL_RESERVATION**
OPTERON_ST3	~ *CYCLES*		
	~ *FR_DISPATCH_STALLS*	+	**FR_DISPATCH_STALLS_BR**
	~ *FR_DISPATCH_STALLS*	+	**FR_DISPATCH_STALLS_SER**
	~ *FR_DISPATCH_STALLS*	+	**FR_DISPATCH_STALLS_SEG**
	~ *FR_DISPATCH_STALLS*	+	**FR_DISPATCH_STALLS_FAR**
OPTERON_FPU			FR_FPU_X87
	SIMD	+	FR_FPU_MMX_3D
	SIMD	+	FR_FPU_SSE_SSE2_PACKED
	SIMD	+	FR_FPU_SSE_SSE2_SCALAR
OPTERON_MMX	*HT_MEM_XFER*	+	HT_LL_MEM_XFR
	HT_MEM_XFER	+	HT_LR_MEM_XFR
	HT_MEM_XFER	+	HT_RL_MEM_XFR
OPTERON_IOX	*HT_I/O_XFER*	+	HT_LL_IO_XFR
	HT_I/O_XFER	+	HT_LR_IO_XFR
	HT_I/O_XFER	+	HT_RL_IO_XFR

Table 3. ***UltraSPARC-III/IV counter sets (maximum of 2 in group, restricted assignment)***

Set Name	Derived Metrics	op.	Native Counters / *Computations*
US3_CPI	**CYCLES**	=	Cycle_cnt
	INSTRUCTION	=	Instr_cnt
US3_SCD	**CYCLES**	=	Cycle_cnt
	~ *RECIRCULATE*	+	**Re_DC_miss**
US3_SCO	~ *CYCLES*		
	~ *DISPATCH*	+	**Dispatch0_br_target**
	~ *Re_DC_miss*	+	**Re_DC_missovhd**
US3_SCE	~ *CYCLES*		
	~ *DISPATCH*	+	**Dispatch0_2nd_br**
	~ *Re_DC_miss*	+	**Re_EC_miss**
US3_SCP	~ *CYCLES*		
	~ *DISPATCH*	+	**Dispatch0_IC_miss**
	~ *RECIRCULATE*	+	**Re_PC_miss**
US3_SMP	~ *CYCLES*		
	~ *DISPATCH*	+	**Dispatch_rs_mispred**
	~ *DISPATCH*	+	**Dispatch0_mispred**
US3_SUS	~ *CYCLES*		
	~ *UNIT_USE*	+	**Rstall_IU_use**
	~ *UNIT_USE*	+	**Rstall_FP_use**
US3_SST	~ *CYCLES*		
	~ *UNIT_USE*	+	**Rstall_storeQ**
	~ *RECIRCULATE*	+	**Re_RAW_miss**
US3_SCX	~ *CYCLES*		
			SI_ciq_flow
	~ *RECIRCULATE*	+	**Re_FPU_bypass**
US3_DCR	~ *DATA_ACCESS*		
			DC_rd
			DC_rd_miss
	DATA_LOAD_FROM_L1$	÷	*DC_rd – DC_rd_miss*
US3_DCW	~ *DATA_ACCESS*		
			DC_wr
	DATA_STORE_INTO_L2$	=	DC_wr_miss
	DATA_STORE_INTO_L1$	÷	*DC_wr – DC_wr_miss*
US3_ECM	~ *DATA_ACCESS*		
	DATA_LOAD_FROM_MEM	=	EC_rd_miss
			EC_misses
US3_ECI	~ *DATA_ACCESS*		
	~ *DATA_STORE_INTO_L2$*	+	**EC_write_hit_RTO**
	~ *INST_ACCESS*		
	INST_HIT_MEM	=	EC_ic_miss
US3_ICH	**INST_ACCESS**	=	IC_ref
			IC_miss
	INST_HIT_L1$	÷	*IC_ref – IC_miss*
	~ *INST_ACCESS*		
	~ *INST_HIT_L2$*	÷	*IC_miss – EC_ic_miss*
	~ *DATA_ACCESS*		
	~ *DATA_LOAD_FROM_L2$*	÷	*DC_rd_miss – EC_rd_miss*
	~ *DATA_STORE_INTO_MEM*	÷	*EC_misses – EC_rd_miss – EC_ic_miss*

Set Name	Derived Metrics	op.	Native Counters / *Computations*
US3_FPU	~ *INSTRUCTION*		
	FLOATING_POINT	+	**FA_pipe_completion**
	FLOATING_POINT	+	**FM_pipe_completion**
US3_BMS	~ *INSTRUCTION*		
	BRANCH_MISP_TAKEN	=	IU_Stat_Br_miss_taken
	BRANCH_MISP_UNTAKEN	=	IU_Stat_Br_miss_untaken
US3_BCS	~ *INSTRUCTION*		
			IU_Stat_Br_count_taken
			IU_Stat_Br_count_untaken
	~ *BRANCH_PRED_TAKEN*	$\doteq$	*IU_Stat_Br_count_taken – IU_Stat_Br_miss_taken*
	~ *BRANCH_PRED_UNTAKEN*	$\doteq$	*IU_Stat_Br_count_untaken – IU_Stat_Br_miss_untaken*
US3_ITL	**INSTRUCTION**	=	Instr_cnt
	~ *TLB_ACCESS*		
	INST_TLB_MISS	=	ITLB_miss
US3_DTL	**CYCLES**	=	Cycle_cnt
	~ *TLB_ACCESS*		
	DATA_TLB_MISS	=	DTLB_miss
US3_ECW			
			EC_ref
	~ *SSM_LOCALITY*	+	**EC_wb**
US3_ECL	~ *SSM_LOCALITY*		
	EC_MISSES	+	**EC_miss_local**
	EC_MISSES	+	**EC_miss_remote**
US3_ECX	~ *SSM_LOCALITY*		
	~ *EC_wb*	+	**EC_wb_remote**
	~ *EC_miss_local*	+	**EC_miss_mtag_remote**
US3_PCR	*P$_READS*	+	PC_port0_rd
		+	PC_port1_rd
US3_ETC			SI_snoop
	P$_READS		
	~ *PC_port0_rd*	+	PC_MS_misses
US3_ECS			EC_snoop_inv
			EC_snoop_cb
US3_WCM			SI_owned
			WC_miss
US3_SM1	~ *MC_STALLS*	+	MC_stalls_0
		+	MC_stalls_1
US3_SM2	~ *MC_STALLS*	+	MC_stalls_2
		+	MC_stalls_3
US3_MC0	~ *MC_READS*	+	MC_reads_0
	~ *MC_WRITES*	+	MC_writes_0
US3_MC1	~ *MC_READS*	+	MC_reads_1
	~ *MC_WRITES*	+	MC_writes_1
US3_MC2	~ *MC_READS*	+	MC_reads_2
	~ *MC_WRITES*	+	MC_writes_2
US3_MC3	~ *MC_READS*	+	MC_reads_3
	~ *MC_WRITES*	+	MC_writes_3

Table 4. ***POWER4-II counter sets (maximum of 8 in group, restricted assignment)***

Set Name	Derived Metrics	op.	Native Counters / *Computations*
POWER4_LX	~ *TLB_ACCESS*		
	INST_TLB_MISS	=	PM_ITLB_MISS
	DATA_TLB_MISS	=	PM_DTLB_MISS
			PM_LD_REF_L1
			PM_LD_MISS_L1
			PM_ST_REF_L1
			PM_ST_MISS_L1
	DATA_ACCESS	≐	*PM_ST_REF_L1 + PM_LD_REF_L1*
	DATA_STORE_INTO_L1$	≐	*PM_ST_REF_L1 – PM_ST_MISS_L1*
	DATA_LOAD_FROM_L1$	≐	*PM_LD_REF_L1 – PM_LD_MISS_L1*
POWER4_DC	~ *DATA_ACCESS*		
	DATA_HIT_L2$	+	**PM_DATA_FROM_L2**
	DATA_HIT_L2$	+	**PM_DATA_FROM_L25_SHR**
	DATA_HIT_L2$	+	**PM_DATA_FROM_L25_MOD**
	DATA_HIT_L2$	+	**PM_DATA_FROM_L275_SHR**
	DATA_HIT_L2$	+	**PM_DATA_FROM_L275_MOD**
	DATA_HIT_L3$	+	**PM_DATA_FROM_L3**
	DATA_HIT_L3$	+	**PM_DATA_FROM_L35**
	DATA_HIT_MEM	=	PM_DATA_FROM_MEM
POWER4_IC	***INST_ACCESS***		
	INST_PREFETCH	=	PM_INST_FROM_PREF
	INST_HIT_L1$	=	PM_INST_FROM_L1
	INST_HIT_L2$	+	**PM_INST_FROM_L2**
	INST_HIT_L2$	+	**PM_INST_FROM_L25_L275**
	INST_HIT_L3$	+	**PM_INST_FROM_L3**
	INST_HIT_L3$	+	**PM_INST_FROM_L35**
	INST_HIT_MEM	=	PM_INST_FROM_MEM
POWER4_BRT	**INSTRUCTION**	=	PM_INST_CMPL
	BRANCH	=	PM_BR_ISSUED
	BRANCH_MISP	+	**PM_BR_MPRED_CR**
	BRANCH_MISP	+	**PM_BR_MPRED_TA**
	CYCLES	=	PM_CYC
	~ *BUSY*	+	**PM_BIQ_IDU_FULL_CYC**
	~ *BUSY*	+	**PM_BRQ_FULL_CYC**
	~ *BUSY*	+	**PM_L1_WRITE_CYC**
POWER4_IFP	**INSTRUCTION**	=	PM_INST_CMPL
	INTEGER	=	PM_FXU_FIN
	FLOATING_POINT	=	PM_FPU_FIN
	FLOATING_POINT	+	**PM_FPU_FMA**
	FLOATING_POINT	+	**PM_FPU_FDIV**
	FLOATING_POINT	+	**PM_FPU_FSQRT**
	FLOATING_POINT	+	**PM_FPU_FMOV_FEST**
POWER4_MFP	**INSTRUCTION**	=	PM_INST_CMPL
	~ *FLOATING_POINT*	+	**PM_FPU_ALL**
	~ *FLOATING_POINT*	+	**PM_FPU_DENORM**
	~ *FLOATING_POINT*	+	**PM_FPU_FRSP_FCONV**
	~ *MEMORY*	+	**PM_FPU_STF**
	~ *MEMORY*	+	**PM_LSU_LDF**
	CYCLES	=	PM_CYC
	~ *STALL*	+	**PM_FPU_STALL3**

A.2. Counter metric hierarchy definition

A generic counter metric classification hierarchy, using KOJAK or PAPI preset counter names, is possible, but in practise it is of limited value. Counter availability is platform-dependant, and interpreting measured counter values even more so. While PAPI provides a number of presets, which attempt to provide general platform-independant definitions, this is only partially successful and introduces additional abstraction which can be undesirable.

PAPI preset event definitions for each supported processor architecture are summarised in [19]. Note that the table is not entirely up-to-date, as event definitions are occassionally added, changed or deleted: this variability aspect also presents a challenge to interpreting what a PAPI definition really counts or will count on any given platform. What the table clearly shows is the disparity of event definition provision between platforms: only `PAPI_TOT_CYC` and `PAPI_TOT_INS` are available on every supported platform, and most events are available on fewer than half. (Curiously, some PAPI presets are not available on any platform, and indeed probably don't make too much sense, e.g., `PAPI_L[123]_ICW` defined as "writes to instruction caches.")

In addition to the fundamental limitation of the native events provided by each processor, it should also be noted that PAPI presets are limited to events that can be collected simultaneously, i.e., subject to platform-specific numbers of counter registers and mapping restrictions.

The current set of PAPI preset specifications are a somewhat curious mixture of general (widely available) events and relatively obscure events only applicable to one or a few platforms, while many of the most valuable events on other platforms have no PAPI preset specification. This is especially evident when examining different types of processor stalls and locality events.

Somewhat understandably, the set of native events provided for each processor (and impacting on its development and validation costs) is highly customised to its particular characteristics and key performance indicators. Some consideration is also made for compatibility with the events provided by earlier processor generations. Provision of hardware counters and associated infrastructure was primarily (at least initially and perhaps even exclusively) intended for internal (i.e., non-customer) use by the processor developers, and only latterly made available to customers: it is typically neither intended nor designed with customer needs foremost in mind.

Defining a generic hierarchy of the most important (available) metrics, and determining a spanning set of the corresponding native metrics to capture, are open areas of research. KOJAK therefore provides a flexible mechanism for experimenting with different measurement set and metric hierarchy definitions.

The approach taken for creating an initial generic hierarchy of KOJAK metrics was based on consideration of the most widely available, unambiguous and familiar PAPI metrics: i.e., those for different types of instruction counts, TLB and cache/memory accesses (including hits and misses).

Although PAPI defines various total counts for different levels of cache, i.e., aggregating counts for instruction and data accesses, it was found more appropriate to consider the two types separately: even though the caches themselves may be unified, most analyses consider them separately (and generally one will dominate the other). Similarly, metrics for each level of cache could have been structured independantly, but it was prefered to combine them into a single hierarchy to emphasise the important relationship between levels: this exploits the exclusive nature of each hit being uniquely satisfied from one particular level of cache or memory, rather than the otherwise more direct measure of misses (which can miss in multiple cache levels). An advantage of this scheme is that the absence of a third level cache is naturally represented as having zero hits in that cache level, while hits from memory are unaffected.

While PAPI v3.0 provides various hit counts, it is unfortunately missing definitions for load (read)

and store (write) hit counts for the various data caches, even though these are often available as native counters or readily derivable. A further PAPI complication is the lack of preset definitions for counts of instructions and data accesses which are served from memory, having missed in all levels of cache. Presets `PAPI_L3_ICM` and `PAPI_L3_DCM` (and `PAPI_L3_LDM` and `PAPI_L3_STM`) have the appropriate definitions on platforms with three levels of cache, however, for platforms with only two levels of cache they are undefined and the corresponding `PAPI_L2` miss definitions need to be used instead.

Although the resulting definitions of access hit hierarchies for data and instruction caches are reasonably portable, in the absence of portable PAPI metric specifications, they require customised platform-specific mixtures of direct measurements, simple compositions and computations. Even if portable PAPI metric specifications were provided for the generic hierarchy, it would be desirable to augment the hierarchies with platform-specific extensions, such as distinction of cache coherency states or accesses to local versus remote caches.

Definition of a generic hierarchy for different instruction counts is even less satisfactory, largely due to the more substantial variation in functional unit provision. A secondary complicating factor is distinction between dispatched/issued and completed/retired instruction counts, with speculatively issued instructions that don't complete being overcounted. Definitions of floating-point operations (i.e., 'flops') performed by processors have always tended to be ambiguous, and while counts of floating-point instructions would appear unambiguous, in practise there is considerable disparity in which are counted by hardware and which aren't: overcounting arises when processors count all instructions executed by notional floating-point units (e.g., including block copy instructions) or include floating-point stores and/or loads. Fused multiply-add instructions can lead to further discrepancies between the logical/expected and actual counts.

One of the most valuable hierarchies is also unfortunately the most platform-specific: that which classifies cycles into busy, stall or idle states. Some processors provide rich hierarchical breakdowns of stall costs, however, these are typically only applicable to one particular platform (or family). Since cycles can be readily converted into times, using processor (or memory) frequency, they are easily interpreted and directly quantify the significance of otherwise hard to judge (and typically highly variable) events.

The following listings provide examples of generic and platform-specific metric hierarchy specifications. The PAPI-based specification primarily offers guidance in determining which native counters best fit in a platform-specific hierarchy, rather than being useful as is.

POWER4 Specification has three-level cache hierarchies, though since stores write-through there are only stores into L1, with platform-specific extensions distinguishing locality and sharing of the caches. `FLOATING_POINT` has been determined by aggregation rather than direct measurement to explicitly exclude `PM_FPU_STF`, which is instead moved to `MEMORY`. No counters are available to split predicted and mispredicted branch instructions into taken and untaken counts, and the limited number of cycles-based counters in that hierarchy might be improved. Many of the available counters have not been considered, which could provide additional platform-specific metric hierarchies.

Opteron Specification has only two-level cache hierarchies, with additional breakdown of L2 and memory accesses by coherency state. `FP_ST_PIPE` and `FP_FAST_FLAG` have been included in `FLOATING_POINT` but could alternatively be moved to `MEMORY` or other locations. The hierarchy of cycles-based metrics includes comprehensive breakdown of dispatch stalls. Platform-specific hierarchies include HyperTransport memory and I/O transfers.

UltraSPARC-III/IV Specification also has only two-level cache hierarchies, complemented with additional platform-specific SSM locality (on systems supporting scalable shared memory), prefetch cache and memory controller request hierarchies. The hierarchy of cycles-based metrics includes comprehensive breakdown of dispatch, unit use, and recirculation stalls. Several derived metrics require computations with counter measurements that can't be collected together due to limitations of the processor hardware. Consequently, partially complete computations from individual collection experiments need to be aggregated to complete their derivations. Alternatively, some derivations may be replaced with approximations that ignore measurements that only provide minor contributions to computations, e.g., `DATA_STORE_INTO_MEM` and `INST_HIT_L2$` can be calculated without the generally insignificant contribution of `EC_ic_miss`.

Generic Metric Hierarchy Specification

```
### cycles (including stalls)
compose CYCLES = BUSY + STALL + IDLE
compose STALL = DISPATCH + UNIT_USE + RECIRCULATE

### instructions
compose INSTRUCTION = BRANCH + INTEGER + FLOATING_POINT + MEMORY
compose BRANCH = BRANCH_PRED + BRANCH_MISP
compose FLOATING_POINT = FP_ADD + FP_MUL + FP_FMA + FP_DIV + FP_INV + FP_SQRT +
                         FP_MISC
compose MEMORY = LOAD + STORE + SYNCH

### data accesses (to cache hierarchy & memory)
compose DATA_ACCESS = DATA_HIT_L1$ + DATA_HIT_L2$ + DATA_HIT_L3$ + DATA_HIT_MEM
compose DATA_HIT_L1$ = DATA_STORE_INTO_L1$ + DATA_LOAD_FROM_L1$
compose DATA_HIT_L2$ = DATA_STORE_INTO_L2$ + DATA_LOAD_FROM_L2$
compose DATA_HIT_L3$ = DATA_STORE_INTO_L3$ + DATA_LOAD_FROM_L3$
compose DATA_HIT_MEM = DATA_STORE_INTO_MEM + DATA_LOAD_FROM_MEM

### instruction accesses (to cache hierarchy & memory)
compose INST_ACCESS = INST_HIT_PREF +
                      INST_HIT_L1$ + INST_HIT_L2$ + INST_HIT_L3$ + INST_HIT_MEM

### TLB accesses (instruction & data)
compose TLB_ACCESS = DATA_TLB_ACCESS + INST_TLB_ACCESS
compose DATA_TLB_ACCESS = DATA_TLB_HIT + DATA_TLB_MISS
compose INST_TLB_ACCESS = INST_TLB_HIT + INST_TLB_MISS
```

Metric Hierarchy Specification with PAPI Presets

```
### cycles (including stalls)
compose CYCLES = BUSY + STALL + IDLE
measure CYCLES = PAPI_TOT_CYC
compose IDLE = PAPI_BRU_IDL + PAPI_FPU_IDL + PAPI_FXU_IDL + PAPI_LSU_IDL
compose STALL = DISPATCH + UNIT_USE + RECIRCULATE
measure STALL = PAPI_RES_STL
measure DISPATCH = PAPI_STL_ICY
measure UNIT_USE = PAPI_FP_STAL
measure RECIRCULATE = PAPI_MEM_SCY
compose PAPI_MEM_SCY = PAPI_MEM_RCY + PAPI_MEM_WCY

### instructions
compose INSTRUCTION = BRANCH + INTEGER + FLOATING_POINT + MEMORY
measure INSTRUCTION = PAPI_TOT_INS                          # or PAPI_TOT_IIS
compose BRANCH = BRANCH_PRED + BRANCH_MISP
measure BRANCH = PAPI_BR_INS
measure BRANCH_PRED = PAPI_BR_PRC
measure BRANCH_MISP = PAPI_BR_MSP
measure INTEGER = PAPI_INT_INS
compose FLOATING_POINT = FP_ADD + FP_MUL + FP_FMA + FP_DIV + FP_INV + FP_SQRT + FP_MISC
measure FLOATING_POINT = PAPI_FP_INS
measure FP_ADD = PAPI_FAD_INS
measure FP_MUL = PAPI_FML_INS
measure FP_FMA = PAPI_FMA_INS
measure FP_DIV = PAPI_FDV_INS
measure FP_INV = PAPI_FNV_INS
measure FP_SQRT = PAPI_FSQ_INS
compose MEMORY = LOAD + STORE + SYNCH
measure LOAD = PAPI_LD_INS
measure STORE = PAPI_SR_INS
measure SYNCH = PAPI_SYC_INS

### data accesses (to cache hierarchy & memory)
compose DATA_ACCESS = DATA_HIT_L1$ + DATA_HIT_L2$ + DATA_HIT_L3$ + DATA_HIT_MEM
compose DATA_HIT_L1$ = DATA_STORE_INTO_L1$ + DATA_LOAD_FROM_L1$
compose DATA_HIT_L2$ = DATA_STORE_INTO_L2$ + DATA_LOAD_FROM_L2$
compose DATA_HIT_L3$ = DATA_STORE_INTO_L3$ + DATA_LOAD_FROM_L3$
compose DATA_HIT_MEM = DATA_STORE_INTO_MEM + DATA_LOAD_FROM_MEM
measure DATA_ACCESS = PAPI_L1_DCA
measure DATA_HIT_L1$ = PAPI_L1_DCH
compute DATA_STORE_INTO_L1$ = PAPI_L1_DCW - PAPI_L1_STM
compute DATA_LOAD_FROM_L1$ = PAPI_L1_DCR - PAPI_L1_LDM
measure DATA_HIT_L2$ = PAPI_L2_DCH
compute DATA_STORE_INTO_L2$ = PAPI_L2_DCW - PAPI_L2_STM
compute DATA_LOAD_FROM_L2$ = PAPI_L2_DCR - PAPI_L2_LDM
measure DATA_HIT_L3$ = PAPI_L3_DCH
compute DATA_STORE_INTO_L3$ = PAPI_L3_DCW - PAPI_L3_STM
compute DATA_LOAD_FROM_L3$ = PAPI_L3_DCR - PAPI_L3_LDM
measure DATA_HIT_MEM = PAPI_L3_DCM  # or PAPI_L2_DCM if no 3rd-level D-cache
measure DATA_STORE_INTO_MEM = PAPI_L3_STM  # or PAPI_L2_STM if no L3 D-cache
measure DATA_LOAD_FROM_MEM = PAPI_L3_LDM   # or PAPI_L2_LDM if no L3 D-cache

### instruction accesses (to cache hierarchy & memory)
compose INST_ACCESS = INST_HIT_L1$ + INST_HIT_L2$ + INST_HIT_L3$ + INST_HIT_MEM
measure INST_ACCESS = PAPI_L1_ICA
measure INST_HIT_L1$ = PAPI_L1_ICH
measure INST_HIT_L2$ = PAPI_L2_ICH
measure INST_HIT_L3$ = PAPI_L3_ICH
measure INST_HIT_MEM = PAPI_L3_ICM # or PAPI_L2_ICM if no 3rd-level I-cache

### TLB accesses (instruction & data)
compose TLB_ACCESS = DATA_TLB_ACCESS + INST_TLB_ACCESS
compose DATA_TLB_ACCESS = DATA_TLB_HIT + DATA_TLB_MISS
compose INST_TLB_ACCESS = INST_TLB_HIT + INST_TLB_MISS
measure DATA_TLB_MISS = PAPI_TLB_DM
measure INST_TLB_MISS = PAPI_TLB_IM
```

```
### cycles (including stalls)
compose CYCLES = BUSY + PM_FPU_FULL_CYC + PM_FPU_STALL3 + PM_BIQ_IDU_FULL_CYC
                      + PM_BRQ_FULL_CYC + PM_L1_WRITE_CYC
measure CYCLES = PM_CYC

### instructions
compose INSTRUCTION = BRANCH + INTEGER + FLOATING_POINT + MEMORY
measure INSTRUCTION = PM_INST_CMPL

compose BRANCH = BRANCH_PRED + BRANCH_MISP
measure BRANCH = PM_BR_ISSUED
compute BRANCH_PRED = PM_BR_ISSUED - PM_BR_MPRED_CR - PM_BR_MPRED_TA
compose BRANCH_MISP = PM_BR_MPRED_CR + PM_BR_MPRED_TA

compose INTEGER = PM_FXU_FIN

compose FLOATING_POINT = PM_FPU_ALL + PM_FPU_DENORM + PM_FPU_FDIV + PM_FPU_FMA
                         + PM_FPU_FMOV_FEST + PM_FPU_FRSP_FCONV + PM_FPU_FSQRT
#measure FLOATING_POINT = PM_FPU_FIN # includes PM_FPU_STF!

compose MEMORY = PM_FPU_STF + PM_LSU_LDF + SYNCH

### data accesses (to cache hierarchy & memory)
compose DATA_ACCESS = DATA_HIT_L1$ + DATA_HIT_L2$ + DATA_HIT_L3$ + DATA_HIT_MEM

## level 1 data cache
compose DATA_HIT_L1$ = DATA_LOAD_FROM_L1$ + DATA_STORE_INTO_L1$

compute DATA_STORE_INTO_L1$ = PM_ST_REF_L1 - PM_ST_MISS_L1
compute DATA_LOAD_FROM_L1$ = PM_LD_REF_L1 - PM_LD_MISS_L1

## level 2 data cache
compose DATA_HIT_L2$ = DATA_LOAD_FROM_L2$

compose DATA_LOAD_FROM_L2$ = PM_DATA_FROM_L2 + PM_DATA_FROM_L25_MOD
         + PM_DATA_FROM_L25_SHR + PM_DATA_FROM_L275_MOD + PM_DATA_FROM_L275_SHR

## level 3 data cache
compose DATA_HIT_L3$ = DATA_LOAD_FROM_L3$

compose DATA_LOAD_FROM_L3$ = PM_DATA_FROM_L3 + PM_DATA_FROM_L35

## memory/system data
compose DATA_HIT_MEM = DATA_LOAD_FROM_MEM

compose DATA_LOAD_FROM_MEM = PM_DATA_FROM_MEM

### instruction accesses (to cache hierarchy & memory)
compose INST_ACCESS = INST_PREFETCH +
                      INST_HIT_L1$ + INST_HIT_L2$ + INST_HIT_L3$ + INST_HIT_MEM
compose INST_PREFETCH = PM_INST_FROM_PREF
compose INST_HIT_L1$ = PM_INST_FROM_L1
compose INST_HIT_L2$ = PM_INST_FROM_L2 + PM_INST_FROM_L25_L275
compose INST_HIT_L3$ = PM_INST_FROM_L3 + PM_INST_FROM_L35
compose INST_HIT_MEM = PM_INST_FROM_MEM

### TLB accesses (instructions & data)
compose TLB_ACCESS = DATA_TLB_ACCESS + INST_TLB_ACCESS
compose DATA_TLB_ACCESS = DATA_TLB_HIT + DATA_TLB_MISS
measure DATA_TLB_MISS = PM_DTLB_MISS
compose INST_TLB_ACCESS = INST_TLB_HIT + INST_TLB_MISS
measure INST_TLB_MISS = PM_ITLB_MISS
```

```
### cycles (including stalls)
compose CYCLES = BUSY + IC_FETCH_STALL + FP_NONE_RET
measure CYCLES = CPU_CLK_UNHALTED
compose IC_FETCH_STALL = FR_DECODER_EMPTY + FR_DISPATCH_STALLS
compose FR_DISPATCH_STALLS = FR_DISPATCH_STALLS_BR + FR_DISPATCH_STALLS_FAR +
     FR_DISPATCH_STALLS_FULL_FPU + FR_DISPATCH_STALLS_FULL_LS +
     FR_DISPATCH_STALLS_FULL_REORDER + FR_DISPATCH_STALLS_FULL_RESERVATION +
     FR_DISPATCH_STALLS_SER + FR_DISPATCH_STALLS_SEG + FR_DISPATCH_STALLS_QUIET

### instructions
compose INSTRUCTION = BRANCH + INTEGER + FLOATING_POINT + MEMORY
measure INSTRUCTION = FR_X86_INS

compose BRANCH = BRANCH_PRED + BRANCH_MISP
measure BRANCH = FR_BR
compose BRANCH_PRED = BRANCH_PRED_TAKEN + BRANCH_PRED_UNTAKEN
compute BRANCH_PRED_TAKEN = FR_BR_TAKEN - FR_BR_TAKEN_MIS
compute BRANCH_PRED_UNTAKEN = FR_BR - FR_BR_MIS - FR_BR_TAKEN + FR_BR_TAKEN_MIS
compose BRANCH_MISP = BRANCH_MISP_TAKEN + BRANCH_MISP_UNTAKEN
#measure BRANCH_MISP = FR_BR_MIS
compute BRANCH_MISP_TAKEN = FR_BR_TAKEN_MIS
compute BRANCH_MISP_UNTAKEN = FR_BR_MIS - FR_BR_TAKEN_MIS

compose FLOATING_POINT = FP_ADD_PIPE + FP_MULT_PIPE + FP_ST_PIPE + FP_FAST_FLAG

compose MEMORY = LOAD + STORE + SYNCH

### data accesses (to cache hierarchy & memory)
compose DATA_ACCESS = DATA_HIT_L1$ + DATA_HIT_L2$ + DATA_HIT_L3$ + DATA_HIT_MEM
measure DATA_ACCESS = DC_ACCESS

## level 1 data cache
compute DATA_HIT_L1$ = DC_ACCESS - DC_MISS

## no L1$ load/store alias definitions possible for opteron
#compose DATA_STORE_INTO_L1$ = NA_OPTERON
#compose DATA_LOAD_FROM_L1$ = NA_OPTERON

## level 2 data cache
compose DATA_HIT_L2$ = DATA_STORE_INTO_L2$ + DATA_LOAD_FROM_L2$
compose DATA_STORE_INTO_L2$ = DC_L2_REFILL_M
compose DATA_LOAD_FROM_L2$ = DC_L2_REFILL_O + DC_L2_REFILL_E + DC_L2_REFILL_S

## no L3$ data cache alias definitions appropriate for opteron
#compose DATA_HIT_L3$ = DATA_STORE_INTO_L3$ + DATA_LOAD_FROM_L3$
#compose DATA_STORE_INTO_L3$ = NA_OPTERON
#compose DATA_LOAD_FROM_L3$ = NA_OPTERON

### memory/system data
compose DATA_HIT_MEM = DATA_STORE_INTO_MEM + DATA_LOAD_FROM_MEM
compose DATA_STORE_INTO_MEM = DC_SYS_REFILL_M
compose DATA_LOAD_FROM_MEM = DC_SYS_REFILL_O + DC_SYS_REFILL_E + DC_SYS_REFILL_S

### instruction accesses (to cache hierarchy & memory)
compose INST_ACCESS = INST_HIT_L1$ + INST_HIT_L2$ + INST_HIT_MEM
#measure INST_ACCESS = IC_FETCH

compute INST_HIT_L1$ = IC_FETCH - IC_L2_REFILL - IC_SYS_REFILL

compose INST_HIT_L2$ = IC_L2_REFILL
#compose INST_HIT_L3$ = NA_OPTERON
compose INST_HIT_MEM = IC_SYS_REFILL
```

```
### TLB accesses (instructions & data)
compose TLB_ACCESS = DATA_TLB_ACCESS + INST_TLB_ACCESS
compose DATA_TLB_ACCESS = DATA_TLB_HIT + DATA_TLB_MISS
compute DATA_TLB_HIT = DC_L1_DTLB_MISS_AND_L2_DTLB_HIT
compute DATA_TLB_MISS = DC_L1_DTLB_MISS_AND_L2_DTLB_MISS
compose INST_TLB_ACCESS = INST_TLB_HIT + INST_TLB_MISS
compute INST_TLB_HIT = IC_L1ITLB_MISS_AND_L2ITLB_HIT
compute INST_TLB_MISS = IC_L1ITLB_MISS_AND_L2ITLB_MISS

### HyperTransport memory & I/O transfers
compose HT_TRANSFER = HT_MEM_XFER + HT_I/O_XFER
compose HT_MEM_XFER = HT_LL_MEM_XFR + HT_LR_MEM_XFR + HT_RL_MEM_XFR
compose HT_I/O_XFER = HT_LL_IO_XFR + HT_LR_IO_XFR + HT_RL_IO_XFR

### doesn't fit in Instruction hierarchy
compose SIMD = FR_FPU_MMX_3D + FR_FPU_SSE_SSE2_PACKED + FR_FPU_SSE_SSE2_SCALAR
```

UltraSPARC-III/IV-specific Metric Hierarchy Specification

```
### **** hierarchy requires aggregation of partial/incomplete computations ****

### cycles (including stalls)
compose CYCLES = BUSY + IDLE + STALL
measure CYCLES = Cycle_cnt
compose STALL = DISPATCH + UNIT_USE + RECIRCULATE
compose DISPATCH = Dispatch0_IC_miss + Dispatch0_mispred + Dispatch_rs_mispred
                                     + Dispatch0_br_target + Dispatch0_2nd_br
compose UNIT_USE = Rstall_IU_use + Rstall_FP_use + Rstall_storeQ
compose RECIRCULATE = Re_RAW_miss + Re_PC_miss + Re_DC_miss + Re_FPU_bypass
compose Re_DC_miss = Re_DC_missovhd + Re_EC_miss + RECIRC_EC_HIT

### instructions
compose INSTRUCTION = BRANCH + INTEGER + FLOATING_POINT + MEMORY
measure INSTRUCTION = Instr_cnt

### requires computations with measurements that can't be collected together!
compose BRANCH = BRANCH_PRED + BRANCH_MISP
compose BRANCH_PRED = BRANCH_PRED_TAKEN + BRANCH_PRED_UNTAKEN
compute BRANCH_PRED_TAKEN = IU_Stat_Br_count_taken - IU_Stat_Br_miss_taken
compute BRANCH_PRED_UNTAKEN = IU_Stat_Br_count_untaken - IU_Stat_Br_miss_untaken
compose BRANCH_MISP = BRANCH_MISP_TAKEN + BRANCH_MISP_UNTAKEN
compute BRANCH_MISP_TAKEN = IU_Stat_Br_miss_taken
compute BRANCH_MISP_UNTAKEN = IU_Stat_Br_miss_untaken

compose FLOATING_POINT = FA_pipe_completion + FM_pipe_completion

compose MEMORY = LOAD + STORE + SYNCH

### data accesses (to cache hierarchy & memory, no L3$)
compose DATA_ACCESS = DATA_HIT_L1$ + DATA_HIT_L2$ + DATA_HIT_MEM

### level 1 data cache (D$)
compose DATA_HIT_L1$ = DATA_STORE_INTO_L1$ + DATA_LOAD_FROM_L1$
compute DATA_STORE_INTO_L1$ = DC_wr - DC_wr_miss
compute DATA_LOAD_FROM_L1$ = DC_rd - DC_rd_miss

### level 2 data cache (E$)
compose DATA_HIT_L2$ = DATA_STORE_INTO_L2$ + DATA_LOAD_FROM_L2$
compose DATA_STORE_INTO_L2$ = EC_write_hit_RTO + EC_WRITE_HIT_ETC
measure DATA_STORE_INTO_L2$ = DC_wr_miss
compute DATA_LOAD_FROM_L2$ = DC_rd_miss - EC_rd_miss

### memory/system data
compose DATA_HIT_MEM = DATA_STORE_INTO_MEM + DATA_LOAD_FROM_MEM
compute DATA_STORE_INTO_MEM = EC_misses - EC_rd_miss - EC_ic_miss # 3 PICs!
compute DATA_LOAD_FROM_MEM = EC_rd_miss
```

```
### instruction accesses (to cache hierarchy & memory, no L3$)
compose INST_ACCESS = INST_HIT_L1$ + INST_HIT_L2$ + INST_HIT_MEM
measure INST_ACCESS = IC_ref

compute INST_HIT_L1$ = IC_ref - IC_miss
compute INST_HIT_L2$ = IC_miss - EC_ic_miss # both require PIC1!
compute INST_HIT_MEM = EC_ic_miss

### TLB accesses (instructions & data)
compose TLB_ACCESS = DATA_TLB_ACCESS + INST_TLB_ACCESS
compose DATA_TLB_ACCESS = DATA_TLB_HIT + DATA_TLB_MISS
compute DATA_TLB_MISS = DTLB_miss
compose INST_TLB_ACCESS = INST_TLB_HIT + INST_TLB_MISS
compute INST_TLB_MISS = ITLB_miss

### SSM locality (only for scalable shared memory systems)
compose SSM_LOCALITY = EC_MISSES + EC_wb
compose EC_MISSES = EC_miss_local + EC_miss_remote
compose EC_miss_local = EC_MISS_MTAG_LOCAL + EC_miss_mtag_remote
compose EC_wb = EC_WB_LOCAL + EC_wb_remote

### Prefetch-cache (P$)
compose P$_READS = PC_port0_rd + PC_port1_rd
compose PC_port0_rd = PC_MS_misses + PC_PORT0_ETC

### memory controller requests
compose MC_READS  = MC_reads_0 + MC_reads_1 + MC_reads_2 + MC_reads_3
compose MC_WRITES = MC_writes_0 + MC_writes_1 + MC_writes_2 + MC_writes_3
compose MC_STALLS = MC_stalls_0 + MC_stalls_1 + MC_stalls_2 + MC_stalls_3
```