Holistic Hardware Counter Performance Analysis of Parallel Programs

B.J.N. Wylie, B. Mohr, F. Wolf

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 is demonstrated with performance experiments, containing communication and synchronisation metrics combined with a rich set of counter metrics, which provide a holistic analysis context and facilitate multi-platform comparison.

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 which events may be measured simultaneously (and limited numbers of event counters) 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 [6]). 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.

1.1. Initial KOJAK approach

Previous developments of the KOJAK performance measurement and analysis environment for parallel programs, which supports many current computer systems[1], offer a suitable vehicle for pursuing this investigation. KOJAK provides semi-automatic instrumentation of user applications and automatic analysis of performance problems arising from inefficient usage of parallel programming

---

[1]Download available from `http://www.fz-juelich.de/zam/kojak/`

interfaces (such as MPI and OpenMP) [1,2]. 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 [3]. Another incorporates hardware counter analysis from separate platform-specific profiling tools with KOJAK's own execution trace analysis [4]. 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 [9]), 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. 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 [7] has three levels of cache whereas Opteron [8] and UltraSPARC-III/IV [9] 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*[2] 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

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

---

[2]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.

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, 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 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: where this is not desired, the corresponding event measurement 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 [7–9].

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. 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. 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, 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 [7] and Apprentice[2] [10]), 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. 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 [1] 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, and 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 cluster, Cray XD1 cluster and Sun Fire E25000, using the ASC Purple sPPM v1.1 benchmark [11]. 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 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 [5]. 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 single traces 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. These sets of experiments were subsequently incorporated into a single composite analysis experiment for each platform.

3.1. Comparative experiment analysis

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: Figure 1 shows a view of the analysis of the XD1 and Regatta experiments. Due to space limitations, the E25000 experiment is not included in this abbreviated analysis: for additional analyses see [12].

Wall-clock execution time of 180s (163s in the runhyd computational kernel) on the XD1 compares with 280s (241s in runhyd) on the Regatta for each experiment. Parallel initialisation overheads (in the init phase) amount to 1.9% of execution time on the Regatta 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 each has good load balance over the 64 threads (32 processes) on both platforms.

The respective proportions of total execution time attributed to MPI are very similar — 1.7% on XD1 vs. 2.2% on Regatta — and investigating 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 can also be found to require a significantly higher collective wait time on the Regatta, totalling 127s (0.77%) versus 15s (0.15%) on the XD1.

OpenMP runtime costs on the XD1 are attributed 3.3% of total execution time, versus 0.9% on Regatta, 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 Regatta that final barrier is attributed 82%.

Some potentially important differences in the MPI and OpenMP communication and synchronisation can therefore be seen in the XD1 and Regatta experiments, however, they also demonstrate broadly similar parallelisation efficiency. 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.

Comparing the proportion of mispredicted branches (BRANCH_MISP), while relatively small in both cases, at 0.58% of all instructions (7.0% of branches) it is considerably larger for Regatta than the 0.08% (1.5%) of Opteron, and depending on the selected call-path is also seen to vary considerably by thread, with some threads notably more affected than the others. The significance can be investigated further by examining the respective counters which measure branch stall cycles.
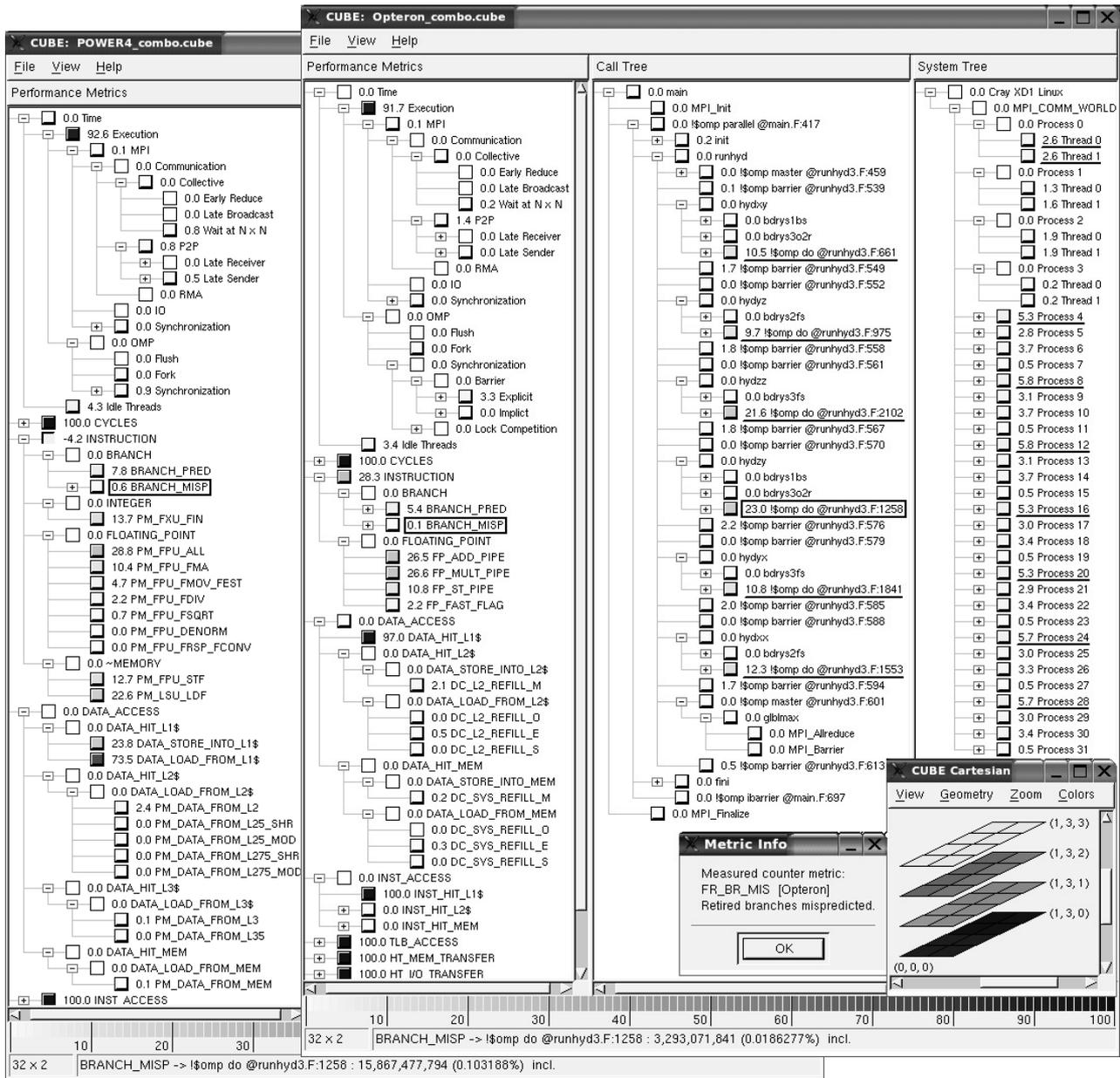
Figure 1. Two KOJAK analyses of combined hybrid OpenMP/MPI sPPM benchmark executions on equisized Cray XD1 Opteron and IBM Regatta p690+ POWER4-II clusters (in front and below). Performance metrics (left pane) and their distribution over the program's call tree (middle pane) and process/thread tree (right pane) are presented hierarchically. Metric values have been expressed as percentage of total execution time or root counter value, and shown with squares coloured according to the scale at the bottom. Selectively expanding or collapsing nodes in each of the three linked trees allows analysis at different levels of granularity. The currently selected metric for branch misprediction rates (with its derivation visible in the pop-up) and the call-tree path ending with the parallel loop in one of the six key hydrodynamics routines are shown boxed (and corresponding details provided in the area at the bottom). Important processes/threads and other call-paths are shown underlined and appear darkest in the lower right virtual process topology display. Each set of experiments has identical call-tree and system tree hierarchies, and only performance metrics which are platform-specific counters differ, yet the broad similarity facilitates comparisons between them.

Both processors are seen to have 97% of data accesses hit L1 cache, 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 p690+ these are seen to be predominantly from local L2 cache (`PM_DATA_FROM_L2`), with only 0.14% requiring to come from memory. With its smaller, two-level caches, Opteron must load twice as much (0.27%) of its data from memory.

4. 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.

References

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

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

[3] 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.

[4] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore: "An Algebra for Cross-Experiment Performance Analysis," Proc. Int'l Conf. on Parallel Processing (ICPP'04, Montreal, Canada), pp. 63–72, Aug. 2004.

[5] B. Mohr, A. Malony, H.-C. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger, and S. Shah: "A Performance Monitoring Interface for OpenMP," Proc. 4th European Workshop on OpenMP (Roma, Italy), Sept. 2002.

[6] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci: "A Portable Programming Interface for Performance Evaluation on Modern Processors," Int'l J. HPC Applications, 14(3):189–204, 2000.

[7] 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.

[8] 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.

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

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

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

[12] Brian J. N. Wylie, Bernd Mohr, and Felix Wolf: "Holistic Hardware Counter Performance Analysis of Parallel Programs," Technical Report FZJ-ZAM-IB-2005-14, Forschungszentrum Jülich, Oct. 2005.