Counting Performance: Hardware Performance Counter and Compiler Instrumentation

Jan-Patrick Lehr[1]

Abstract: Analyzing applications for their runtime behavior, especially in the light of efficient resource utilization, involves iterative measurements and the interpretation of the data gathered. For fine grained analysis often hardware performance counters are monitored. Since compiler instrumentation augments the program with calls to a measurement system, it interacts with compiler optimizations. Currently, it is unclear to which degree the instrumentation influences the characteristics of the resulting binary. To determine the behavioral change introduced through instrumentation we conduct a series of measurements on a subset of the SPEC CPU 2006 benchmark suite. We show that although runtime increases up to acceptable 10% of the original runtime some hardware performance counter are significantly perturbed. However, it is also possible that hardware performance counter deviate only slightly from the values measured in the original binary, even though the benchmark's runtime increases substantially. In particular, the program *444.namd* showed an increase in store instructions by $2x$ with only 3% runtime overhead, whereas *450.soplex* did not show significant change in mispredicted branches, when exhibiting an increase in runtime by $3x$. We investigate whether the validity of a hardware performance counter can be determined via static analysis. Therefore, we outline a new tool based on the MAQAO binary analysis framework to compute and compare the static instruction mix of binaries to identify the introduced change in the static instruction mix due to instrumentation. The static instruction mix is comparable to a histogram, denoting how many instructions of a certain category are found in each function of the binary. We conclude that static analysis of the binary's instruction mix may describe induced change for hardware performance counters. Finally, we outline directions of further research in the field of perturbation detection and predictive modelling.

Keywords: High Performance Computing, Performance Analysis, Compiler Instrumentation, Hardware Performance Counter, Binary Analysis

1 Introduction

Modern hardware offers increasing performance at the price of increasing complexity. Especially in the area of high performance computing (HPC), big advances have been made within the last decades. However, the increasing compute capability poses strong demands on software architects in order to make efficient use of the available hardware.

Since many users of HPC are experts in other domains, such as physics or mechanical engineering, a deep understanding of the available hardware cannot be assumed. Bischof et al. pointed out that, based on economic considerations, performance analysis and tuning is important [BaMI11]. Performance analysis and tuning is an iterative process consisting of measurement, analysis and tuning, with the goal to improve the application's performance

[1] TU Darmstadt, Scientific Computing, Mornewegstrasse 30, 64293 Darmstadt

and efficiency. In order to give guidelines and help analysts to carry out the task, the process has been modelled by Iwainsky et al. [Iw11].

To gain insight about the application's behavior a performance analyst (PA) performs a series of measurements, either manually or automatically. Those measurements are, in general, performed using *sampling* or *instrumentation* and are taken with respect to a certain metric. For a more detailed explanation of the two approaches the reader is referred to Morris et al. [Mo10]. The program that is measured is called a performance proxy, as it is used to approximate the original application's behavior in regard of the metrics monitored. As with any measurement, the system under observation is also influenced by the measurement carried out [Ma91]. This influence usually is referred to as runtime overhead.

Besides measuring runtime, often hardware performance counters are monitored. A commonly used library to abstract from the low-level and machine dependent interface of hardware performance counters is the PAPI library [Br00]. State of the art measurement systems like Score-P [Me11] or HPC Toolkit [Ad10] offer support for this interface. However, instrumentation, as used by Score-P, might actually have two types of influence on a program's behavior [MRW92]. Since it changes a program's source code, it might also change the compiler's decisions on optimizations it can apply. Consequently, not only the measurement system's runtime overhead is introduced, but a potentially differently optimized binary is observed.

If a PA uses measurements taken from such an instrumented binary as the basis for a tuning decision, the question arises whether the collected data is obtained from a valid performance proxy of the original application. Current best practice suggests that whenever the measured runtime was within 10% of the original runtime, the performance proxy is considered valid. However, this guidance only reflects runtime and leaves aside other important characteristics of an application.

We conducted a series of experiments regarding the influence of automatic compiler instrumentation on hardware performance counters. The study is carried out using a subset of the SPEC CPU 2006 benchmark suite consisting solely of C/C++ benchmarks as the toolchain that we use is currently only capable of dealing with C/C++ programs.

To help the PA in determining whether the collected values resulted from a valid performance proxy, we investigate whether hardware performance counter perturbation can be correlated to the change in the static instruction mix within the binary. The static instruction mix is comparable to a histogram, denoting how many instructions of a certain category are found in each function of the binary. To this end, we developed a tool, based on the MAQAO binary analysis framework by Djoudi et al. [Dj05], that computes the static instruction mix.

The paper is structured as follows: Section 2 provides a more detailed motivation and background. We present experimentally gathered results for PAPI events in Section 3. Our definition of the static instruction mix as well as a description of our tool is given in Section 4. We conclude our findings in Section 5 and give an outline of possible future work and application areas of instruction mix comparison in Section 6.

2 Background

To investigate whether an application is constrained by a specific machine limitation or does not use a given platform efficiently, an analyst needs to obtain an understanding of the application's runtime behavior. In order to construct a picture of the application's behavior, a series of measurements is carried out, gathering a variety of metrics. The metrics typically include runtime, performance counter and, in the case of parallel applications, communication waiting times.

Inspecting and relating all of the available information leads to a hypothesis why the target application suffers from a specific performance problem. Depending on the identified problem, the analyst decides which limiting factors are to be approached first. If the performance proxy, however, does not reflect the original application's behavior with respect to the metric it was used to monitor, the PA may draw a wrong conclusion. As a result, tuning effort is spent to resolve an artefact which resulted from the measurement and is not necessarily present in the original application. Typically recorded hardware performance counter differ from platform to platform, but may include:

- Level 1 and 2 instruction and data cache accesses / hits / misses
- Performed branch instructions / mispredicted branches
- Load and store instructions
- Stalled cycles / cycles without instruction issue

In this work, the effects of instrumentation will be inspected. The impact of sampling on the various hardware performance counters needs yet to be studied, but will not be part of this work. Instrumentation augments the application with calls to a measurement system. Thus, it guarantees the observation of events and is therefore useful for certain tasks, e.g. discovery of communication patterns in parallel applications. Instrumentation can be carried out manually or automatically; in this work we focus on automatic compiler instrumentation, as provided by major compilers including GCC[2], Clang[3] or the Intel compiler[4].

List. 1: Uninstrumented example code.

```
int factorial(int n){
  if(n == 0){
    return 1;
  }

  return factorial(n-1) * n;
}
```

List. 2: Instrumented example code.

```
int factorial(int n){
  __cyg_profile_func_enter(&factorial,
      __builtin_return_address(1));
  if(n == 0){
    __cyg_profile_func_exit(&factorial,
        __builtin_return_address(1));
    return 1;
  }

  int _tval_ = factorial(n-1) * n;
  __cyg_profile_func_exit(&factorial,
      __builtin_return_address(1));
  return _tval_;
}
```

Automatic compiler instrumentation introduces calls to a measurement interface at the

[2] see http://gcc.gnu.org/
[3] see http://clang.llvm.org/
[4] see http://software.intel.com/en-us/c-compilers

start and exit of every function as can be seen exemplarily in Listing 2. Especially in C++ codes, this can lead to massive runtime overhead, if one considers a modern, object-oriented programming style with many small functions. However, the introduced runtime overhead is only one part, as the additional function calls may also change the compiler's decisions about optimizations, including inline expansion and tail call elimination. Thus, compiler instrumentation might influence the target application's behavior in more subtle ways.

2.1 Related Work

Mytkowicz et al. showed that capturing software metrics using instrumentation may significantly perturb hardware performance counters [My07]. In their work they use sampling to create program traces of vanilla[5] and instrumented versions[6] and capture a selection of hardware performance counters, including level 1 data and level 2 total cache misses. For both traces, the correlation of two performance counters values within the trace is computed. After aligning the traces, these correlations are compared and the results show that adding instrumentation significantly alters the correlation values. In comparison to their approach, our research targets static analysis of the binary instead of the comparison of multiple traces. The possibility to determine the influence of instrumentation statically, would decrease the necessity to perform time and resource consuming baseline measurements.

Malony et al. proposed a methodology to automatically detect measurement overhead in time stamps of program trace events [MRW92]. In their approach, they construct a propagation model of incurred runtime overhead and are able to recover trace event time within small error margins. In order to decrease the necessary analysis time, Malony et al. proposed an on-the-fly compensation method for overhead mitigation in [MS05]. They extend their former work to account for parallel applications and the special needs to capture communication correctly. However, both approaches focus solely on time and the relative order between different events in a potentially parallel application. While this is valuable information for the analysis of applications, their approach regards solely runtime.

Kashnikov et al. use the static loop analyzer provided with the MAQAO infrastructure to extract low-level assembly features and characterize loops to discover potential optimization opportunities [Ka13]. While their approach to extract important features, such as vectorization ratio, is similar, the actually extracted features are different and the overall goal differs. Kashnikov et al. focus on improving existing compiler optimization or apply further binary optimization, our work focuses on determining the validity of measured performance counter values.

Moseley et al. focus on the discovery of regions for potential improvements in compiler optimization techniques. While the process involves the comparison of assembler instructions, it focuses on the performance of short instruction sequences in terms of runtime.

[5] Here: the application without any instrumentation applied.
[6] Here: the application compiled with any kind of instrumentation applied.

To this end, they match event logs from program traces and correlate the respective assembly instructions [MGP09a]. Although their correlation approach is interesting, the primary goal is drawing the user's attention to potentially poorly applied compiler optimizations [MGP09b].

Static binary analysis is frequently used for reverse engineering and cross instruction set architecture translation [CE99, CS00, TC02]. Especially today's high-level language abstractions, such as virtual calls, and transformations applied by optimizing compilers pose challenges to the correct reverse engineering of the binary.

To the best of our knowledge static binary analysis has not been applied to determine whether an instrumented binary is a valid performance proxy with respect to a certain hardware performance counter.

3 Experiments

In order to study the impact of instrumentation on hardware performance counters we conduct a series of measurements on a C/C++ subset of SPEC[7] CPU 2006 benchmark programs, using the Clang compiler in version 3.8. The benchmarks are written either in C (*403.gcc*, *429.mcf*, *433.milc*, *456.hmmer*, *458.sjeng*, *462.libquantum*, *464.h264href*, *470.lbm* and *482.sphinx3*) or in C++ (*444.namd*, *447.dealII*, *450.soplex*, *453.povray* and *473.astar*). For a more detailed explanation of the benchmarks, see [He06].

To measure the PAPI counters, we developed a lightweight wrapper library, which allows us to capture PAPI events defined in its high-level API. The library is loaded using *libmonitor* [Kr13], a lightweight mechanism to inject measurement facilities into applications. All measurements are conducted on nodes from phase 2 of the Lichtenberg cluster of TU Darmstadt[8]. Each node is equipped with two Intel Xeon E5-2680 v3 processors, 64GB main memory and is used exclusively for the measurement. The processes are pinned to a specific core and hyper threading as well as frequency scaling are disabled in order to obtain comparable results across runs.

3.1 Measuring PAPI Counters for Vanilla and Instrumented Binaries

In our measurements we obtain values for exactly two PAPI events. The total number of instructions completed (`PAPI_TOT_INS`) is measured in every run. The second metric is one of nine commonly used metrics: Level 1 instruction (L1I) cache misses (`PAPI_L1_ICM`), level 2 instruction (L2I) cache total accesses (`PAPI_L1_ICA`), level 2 instruction and data cache misses (`PAPI_L2_ICM`, `PAPI_L2_DCM`), load and store instructions (`PAPI_LD_INS`, `PAPI_SR_INS`), conditional and mispredicted branches (`PAPI_BR_CN`, `PAPI_BR_MSP`) as well as the number of cycles stalled on any resource (`PAPI_RES_STL`). The counter values are monitored for a vanilla version as well as for an automatically instrumented version of

[7] See `http://www.spec.org`
[8] see `www.hhlr.tu-darmstadt.de`

the benchmark. We do not link the instrumented binary to an actual implementation of a measurement system, but use the empty default implementation in glibc[9]. The goal is to gather an understanding of hardware performance counter behavior in the sole presence of additional calls to a measurement system, without actually capturing data. Thus, the perturbation observed stems from solely the instrumentation instructions and its impact on the compiler's optmization decisions.

Benchmark	Runtime	Load	Store	L1I Misses	L2I Misses	L2D Misses	Cond. Branches	Br. Mispred.
403.gcc	1.405	1.608	1.579	1.091	1.121	0.985	1.000	1.200
429.mcf	1.186	1.422	1.561	1.506	1.508	1.028	1.075	1.611
433.milc	1.052	1.171	1.379	1.250	1.249	0.984	1.000	1.010
444.namd	1.038	1.344	1.959	1.135	1.073	1.002	1.000	1.000
447.dealII	13.136	7.282	14.736	17.479	7.788	0.764	1.428	6.247
450.soplex	3.081	3.271	11.122	2.892	1.932	0.650	1.093	1.033
453.povray	3.170	2.300	2.801	2.657	1.808	1.738	1.062	3.304
456.hmmer	1.024	1.010	1.018	1.007	0.935	1.000	1.000	1.009
458.sjeng	1.488	1.360	1.586	5.557	0.786	1.001	1.005	1.263
462.libquantum	1.002	1.019	1.070	1.449	1.465	1.011	1.000	0.923
464.h264ref	1.374	1.289	2.085	1.140	1.181	0.993	1.000	1.370
470.lbm	1.053	1.000	1.000	0.923	1.015	1.027	1.000	0.997
473.astar	1.818	1.970	2.775	1.653	1.473	0.913	1.000	1.063
482.sphinx3	1.132	1.054	1.325	1.104	1.115	0.998	1.000	1.305

Tab. 1: Measured counts for PAPI load and store instructions as well as L1I, L2I, L2D cache misses and conditional branch instructions as well as conditional branch instructions mispredicted. The values are given as relative factors with the vanilla version's value being the baseline (1.0).

Table 1 lists the benchmarks and their runtime increase for an automatic compiler instrumented version. The factors listed show relative behavior of several PAPI counters when vanilla and instrumented binaries are compared. Taking overhead runtime as an indicator for the validity of the measurement suggests that *433.milc*, *444.namd*, *456.hmmer*, *462.libquantum* and *470.lbm* are valid performance proxies. In contrast, *447.dealII*, *450.soplex* and *453.povray* show an increase of 13*x* and 3*x* in runtime and, thus, would not be considered valid. Both, *429.mcf* and *482.sphinx3* do not exhibit large overhead and are within 20% overhead, whereas *458.sjeng*, *464.h264ref* and *473.astar* show runtime increases between 37% and 82%.

As it can be seen in Table 1 and Figure 1, *444.namd* would be considered valid, although the increase in store instructions is close to 2x. Such a deviation can lead the performance analyst to drawing the wrong conclusions and trying to investigate and fix not an inherent problem of the application but an artefact of the measurement. The same can also be true for *433.milc*, although, the increase in store instructions is not as severe as it is for *444.namd*.

For *462.libquantum*, where nearly no runtime increase is seen, the L1I and L2I cache misses increase by nearly 50%. In the analysis one can conclude that it would be necessary to change code layout in order to improve code locality in the binary. However, as the increase is due to the instrumentation added, it is uncertain whether the vanilla version would benefit from such changes.

[9] see `http://www.gnu.org/software/libc/`

On the other hand, for *403.gcc*, the counted number of L1I cache misses is nearly unchanged when compared to the vanilla version. Additionally, the L2I cache misses are close to a 10% deviation compared to the vanilla version. This suggests that the number of these events counted is usable for analysis purposes, even though the runtime increase is 1.4x.

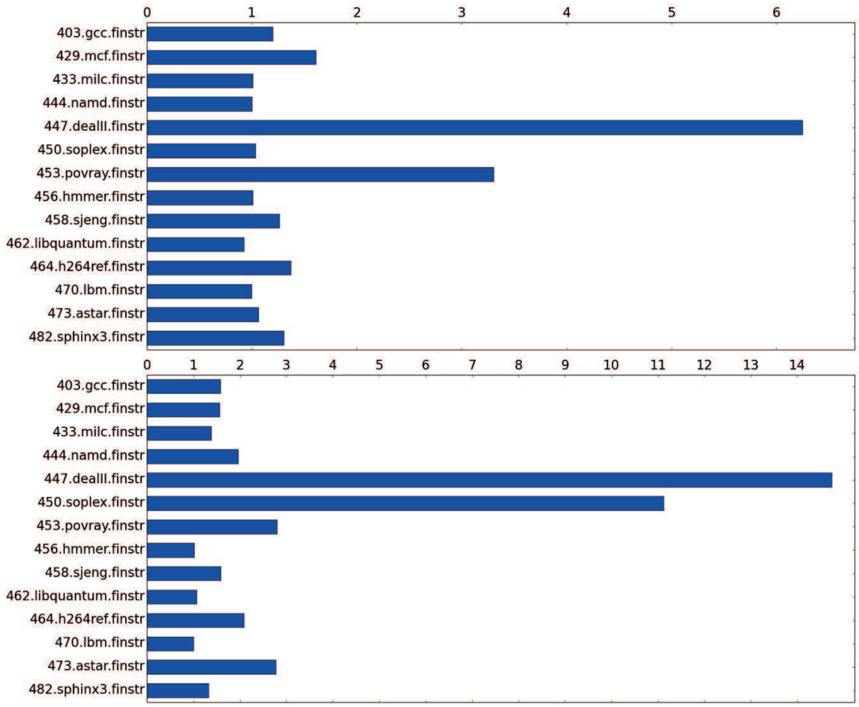

Fig. 1: Depicting the increase for mispredicted branches (top) and store instructions (bottom) for the instrumented versions of the benchmarks.

The results also show that a measurement, which would not be considered a valid performance proxy, as it lies outside of 10% of the vanilla runtime, can indicate the original's behavior with respect to a certain hardware performance counter. All programs, except *447.dealII*, show no or very little change with respect to the measured number of conditional branch instructions commited. Additionally, the counted number of mispredicted branches is nearly unchanged for some of the benchmarks for which the measurement would not be considered valid, see Figure 1. For example, *450.soplex* shows nearly no change with respect to committed, as well as mispredicted, conditional branches, but would not be considered valid, as its runtime increased by 3x.

In summary, we conclude that requiring only runtime to be within 10% of the vanilla version's runtime seems to be valid as a general guideline. However, especially the results for *444.namd* and *450.soplex* motivate a detailed evaluation of the subtle interplay between instrumentation and hardware performance counters.

4 Static Instruction Mix

In the last section we showed that instrumentation does influence some of the PAPI performance counters considerably and that runtime increase alone is not always a good measure for hardware performance counter perturbation. To be able to investigate the correlation between our findings and changes at the binary level, this section illustrates our concept of a *static instruction mix* for binaries. It also outlines challenges when inspecting binaries as one may need to reconstruct properties, such as transformed call sites. In addition, we present our approach for computing the static instruction mix based on the MAQAO infrastructure.

The static instruction mix is comparable to a histogram, denoting how many instructions of a certain category are found in each function of the binary. We chose seven categories: `Arith`, `Mem`, `Calls`, `Branches`, `Unconditional Branches`, `Stack Ops` and `Unclassified`, with the latter containing operations which are not clearly of any other category. The `Mem` category in general captures memory related operations like `MOV` operations, which correlate to PAPI load and store events. Consequently, we do not consider register to register move operations as `MEM` operations, as PAPI does not count these as load or store events. `Calls`, `Branches` and `Unconditional Branches` should influence the PAPI cycles stalled counter as well as branch predictor related metrics. The other categories are chosen to be able to inspect the impact on stack memory, compute arithmetic to memory ratio and possibly missed optimization opportunities to eliminate arithmetic operations.

We use MAQAO's call graph analysis to build the static call graph. Since compiler optimization may turn a function call to a jump, we fix up lost call sites doing a jump-to-function-label analysis. The analysis iterates over all jump instructions in the binary with a function name as jump target. If the label does not have an additional offset, the jump is considered to be an optimized call.

One challenge during the reconstruction of the call graph are virtual calls, as can be found in C++. Another challenge are indirect calls through function pointers, which could not be resolved at compile time but depend on runtime values. Since virtual calls are usually implemented using jump tables, to be able to perform the dispatch based on the runtime type of an object, it is not easy to reliably reconstruct them [TC02]. Our implementation currently ignores this fact and does reconstruction based only on calls and jumps to function labels. However, this is not a limitation of the general aproach but of the current technology used.

After the construction of the call graph, we compute the *inclusive instruction mix*, which is defined as the sum of all instruction mixes for functions $\hat{f}$ reachable from a given function f. Doing so for the *main* function results in the instruction mix for the binary, without static initialization. Having both, the instruction mix per function as well as the inclusive instruction mix for a function and all reachable descendents, it is possible to compare the change on a function to function level and on a subtree to subtree level. This allows to evaluate the effects of instrumentation on inline expansion and the resulting static instruction mix.

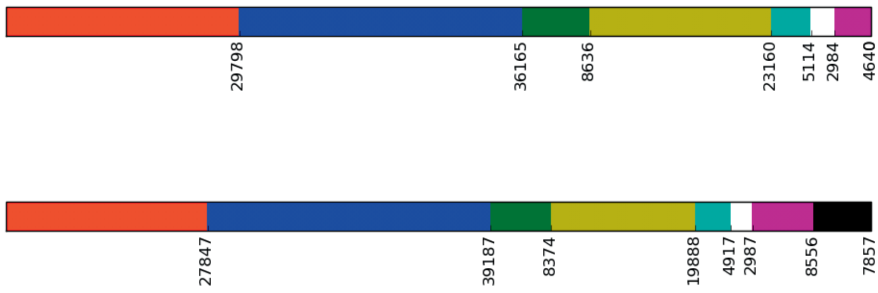

Fig. 2: The instruction mix change between vanilla (upper) and instrumented (lower) version of 453.povray benchmark. `Arith` is shown in red, `Mem` is shown in blue, `Calls` are shown in green, `Branches` are shown in beige, `Unconditional Branches` are given in tourquoise, `Unclassified` are shown in white, `Stack Ops` are colored pink. For the instrumented version, calls to the measurement interface are given in black. The numbers are total counts of the respective instruction mix category.

In general, an increase towards the `Mem` category can be seen for all of the benchmark binaries. As an example, Figure 2 depicts the inclusive static instruction mix for the *453.povray* benchmark with the vanilla binary at the top and the instrumented binary at the bottom, respectively. It can be seen that the value for `Branches` as well as `Arith` within the binary decreases for the instrumented version. These two artefacts, in our opinion, do correlate as the computation of a condition for a `Branch` instruction itself counts as `Arith` operation. Thus, if fewer branches exist, then less conditions must be evaluated leading to a decresing number of `Arith` operations.

In addition, the decreasing number of `Arith` operations can be explained by missed optimization opportunities. Usually, if the compiler's vectorizer hits a computation it is able to vectorize, it emits a vectorized as well as a scalar version of the computation, thus increasing the number of total arithmetic operations. Seeing a decrease in this category is likely to show missed opportunities for such optimization. The lower number of conditional branches can be seen as another indicator of this change, as within an optimized binary, the compiler needs to emit runtime checks, whether the vectorized computation or the scalar version needs to be called.

At the same time the number of `Stack Ops` increases for the instrumented binary when compared to the vanilla version. This correlates with more functions requiring local stack space being present in the instrumented binary (2025) than in the vanilla version (1545). In the stack frame, the compiler has to allocate space for local variables. Further, it needs to save live registers[10] before calling any function. If the register save was not necessary in the vanilla version, the additional calls to the measurement system, most likely, demand

[10] A register is said to be live if its value is read within any subsequent instruction in the instruction sequence without an interpositional write operation.

the saving of at least one register. This leads to either more stack operations or to additional memory operations.

5 Conclusion

Performance analysis and tuning is important for efficient use of HPC resources. Since monitoring a system necessarily introduces perturbation into the system, the analyst needs to be aware of the change that was introduced by the instrumentation. For instrumentation, we showed that solely relying on the usually accepted guideline that a measurement is valid (in the sense that the instrumented program properly reflects the characteristics of the uninstrumented program) when its runtime does not deviate more than 10% of the original program's runtime, is not always true. In particular, hardware performance counters show significant differences. Consequently, additional information for determining the validity of hardware performance counter measurements is needed.

We propose to inspect the static instruction mix at the binary level to determine the perturbation of hardware counters due to instrumentation, as the binary more than the source code reflects the really running program. To this end, we developed a static binary analysis tool based on the MAQAO framework. The tool classifies every instruction into one of seven categories to reveal changes within the mix of assembly instructions. The classification is performed at the granularity of functions, thus, it draws a clearer picture of the application than simply counting assembly instructions from start to end. Additionally, the call graph is constructed and the inclusive instruction mix is computed, to give a high-level view on the changes in the instruction mix of the binary.

We showed that instrumentation, in general, gears the instruction mix more towards memory operations and increases the number of stack operations. The number of conditional branch instructions decreased for a subset of the benchmarks, as well as the number of arithmetic operations. Thus, the change in the static instruction mix may be helpful to determine the validity of gathered hardware performance counter values.

6 Future Work

Some of the artefacts that can be seen in the static instruction mix may serve as a descriptor of introduced change. However, further investigation is needed to quantify the relation between properties of the binary and perturbation of hardware performance counters and in this section, we outline avenues for future research.

An interesting property of the binary is the distance between function calls within the vanilla and instrumented versions as it indicates the compiler's ability to inline functions. In addition the number of functions and the distance between the code of the respective bodies is of interest, as it can serve as an indication for level one and two instruction cache misses.

Although in current architectures the floating point operation counters are disabled, these counters are of interest in older architectures. To determine the impact of instrumentation on other compiler optimizations, the kind of arithmetic operations which are found in the binary is of interest. To this end, the operand-type mix and differences in the vectorization ratio can be computed between the binary, between single functions, or between subtrees of the respective call graphs.

Since static analysis cannot determine where an application spends most of its time, using profile information to weight the instruction mixes can also benefit the analysis. Especially loop driven programs, like many scientific applications, spend most of their runtime in a limited amount of code. Thus, a tool used for performance data assessment can use already available profiling information to weight the respective static information.

To further investigate the change induced by instrumentation, experiments with more sophisticated instrumentation techniques are also of interest. For example, Score-P offers support to ignore `inline` marked functions from instrumentation. Since inlining is a powerful compiler optimization, it is worth investigating whether this reduces the induced change.

Finally, it is worth exploring whether the instruction mix can be used to predict the likelyhood that the instrumented binary captures a certain metric to a sufficient degree.

In general, further research is neccessary to investigate whether the static instruction mix can be used as a descriptor for other attributes, such as energy consumption. Being able to correlate the static instruction mix, maybe with added profile information, with the energy behavior of an application, would provide opportunities for energy-aware work scheduling systems.

Acknowledgements

I would like to thank Christian Iwainsky for his guidance and valuable discussions.

References

[Ad10] Adhianto, Laksono; Banerjee, Sinchan; Fagan, Mike; Krentel, Mark; Marin, Gabriel; Mellor-Crummey, John; Tallent, Nathan R.: HPCToolkit: Tools for performance analysis of optimized parallel programs. Concurrency and Computation: Practice and Experience., 22(6):685–701, 2010.

[BaMI11] Bischof, Christian; an Mey, Dieter; Iwainsky, Christian: Brainware for green HPC. Computer Science - Research and Development., 27(4):227–233, 2011.

[Br00] Browne, S.: A Portable Programming Interface for Performance Evaluation on Modern Processors. Intl. Journal of High Performance Computing Applications., 14(3):189–204, 2000.

[CE99] Cifuentes, Cristina; Emmerik, Mike Van: Recovery of jump table case statements from binary code. In: Proc. of the 7th Intl. Workshop on Program Comprehension. pp. 192–199, 1999.

[CS00] Cifuentes, Cristina; Simon, Doug: Procedure abstraction recovery from binary code. In: Proc. of the 4th Europ. Software Maintenance and Reengineering. pp. 55–64, 2000.

[Dj05] Djoudi, Lamia; Barthou, Denis; Carribault, Patrick; Lemuet, Christophe; Acquaviva, Jean-Thomas; Jalby, William et al.: Maqao: Modular assembler quality analyzer and optimizer for itanium 2. In: The 4th Workshop on EPIC architectures and compiler technology. volume 200, 2005.

[He06] Henning, John L.: SPEC CPU2006 benchmark descriptions. ACM SIGARCH Computer Architecture News., 34(4):1–17, 2006.

[Iw11] Iwainsky, Christian; Altenfeld, Ralph; an Mey, Dieter; Bischof, Christian: Enhancing brainware productivity through a performance tuning workflow. In: Euro-Par 2011: Parallel Processing Workshops. Springer, pp. 198–207, 2011.

[Ka13] Kashnikov, Yuriy; de Oliveira Castro, Pablo; Oseret, Emmanuel; Jalby, William: Evaluating architecture and compiler design through static loop analysis. In: Intl. Conf. on High Performance Computing and Simulation, HPCS. pp. 535–544, 2013.

[Kr13] Krentel, Mark W.: Libmonitor: A tool for first-party monitoring. Parallel Computing, 39(3):114–119, 2013.

[Ma91] Malony, Allen D.: Event-based Performance Perturbation: A Case Study. In: Proc. of the 3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. PPOPP '91, ACM, New York, NY, USA, pp. 201–212, 1991.

[Me11] an Mey, Dieter; Biersdorf, Scott; Bischof, Christian; Diethelm, Kai; Eschweiler, Dominic; Gerndt, Michael; Knüpfer, Andreas; Lorenz, Daniel; Malony, Allen; Nagel, Wolfgang E.; Oleynik, Yury; Rössel, Christian; Saviankou, Pavel; Schmidl, Dirk; Shende, Sameer; Wagner, Michael; Wesarg, Bert; Wolf, Felix: Score-P: A Unified Performance Measurement System for Petascale Applications. In: Competence in High Performance Computing 2010., pp. 85–97. Springer Science + Business Media, 2011.

[MGP09a] Moseley, Tipp; Grunwald, Dirk; Peri, Ramesh: Chainsaw: Using Binary Matching for Relative Instruction Mix Comparison. In: 18th Intl. Conf. on Parallel Architectures and Compilation Techniques, PACT. pp. 125–135, 2009.

[MGP09b] Moseley, Tipp; Grunwald, Dirk; Peri, Ramesh: OptiScope: Performance Accountability for Optimizing Compilers. In: Intl Symp. on Code Generation and Optimization. Institute of Electrical & Electronics Engineers (IEEE), 2009.

[Mo10] Morris, Alan; Malony, Allen D.; Shende, Sameer; Huck, Kevin: Design and Implementation of a Hybrid Parallel Performance Measurement System. In: 39th Intl. Conf. on Parallel Processing. Institute of Electrical & Electronics Engineers (IEEE), 2010.

[MRW92] Malony, Allen D.; Reed, Daniel A.; Wijshoff, Harry A. G.: Performance measurement intrusion and perturbation analysis. IEEE Transactions on Parallel and Distributed Systems., 3(4):433–450, 1992.

[MS05] Malony, Allen D.; Shende, Sameer S.: Models for On-the-Fly Compensation of Measurement Overhead in Parallel Performance Profiling. In: Euro-Par 2005 Parallel Processing., pp. 72–82. Springer Science + Business Media, 2005.

[My07] Mytkowicz, T.; Diwan, A.; Hauswirth, M.; Sweeney, P. F.: Understanding Measurement Perturbation in Trace-based Data. In: IEEE Intl. Parallel and Distributed Processing Symposium, IPDPS. pp. 1–6, 2007.

[TC02] Troger, J.; Cifuentes, C.: Analysis of virtual method invocation for binary translation. In: Proc. of the 9h Working Conference on Reverse Engineering. pp. 65–74, 2002.