Low-Overhead Detection of Memory Access Patterns and Their Time Evolution

Harald Servat[1,2(✉)], Germán Llort[1,2], Juan González[1], Judit Giménez[1,2], and Jesús Labarta[1,2]

[1] Computer Sciences Department - Barcelona Supercomputing Center, c/Jordi Girona 1-3, 08034 Barcelona, Catalunya, Spain
harald.servat@bsc.es
[2] Computer Architecture Department - Universitat Politècnica de Catalunya, c/Jordi Girona 31, 08034 Barcelona, Catalunya, Spain

Abstract. We present a performance analysis tool that reports the temporal evolution of the memory access patterns of in-production applications in order to help analysts understand the accesses to the application data structures. This information is captured using the Precise Event Based Sampling (PEBS) mechanism from the recent Intel processors, and it is correlated with the source code and the nature of the performance bottlenecks if any. Consequently, this tool gives a complete approach to allow analysts to unveil the application behavior better, and to lead them to improvements while taking the most benefit from the system's characteristics. We apply the tool to two optimized parallel applications and provide detailed insight of their memory access behavior, thus demonstrating the usefulness of the tool.

Keywords: Performance analysis · Address sampling · Data-object analysis · Sampling · Instrumentation

1 Introduction

The memory hierarchy is becoming more and more sophisticated as the processors evolve generation after generation. Its advances respond not only to address the speed divergence between the processor and the memory outside the chip, but also to reduce the energy dissipated by the data movement. Processor manufacturers have typically organized the memory hierarchy in different *strata* to exploit the temporal and spatial localities of reference. The memory hierarchy ranges from the extremely fast but tiny and power-hungry registers to the slow but huge and less energy-consuming DRAM, including multiple cache levels. Still, some processor researchers and manufacturers are looking for opportunities to extend the memory hierarchy to improve the application execution in terms of performance and energy. Their research consider additional integration directions so that the memory hierarchy adds layers as scratchpad memories, stacked 3D DRAM [12], and even non-volatile RAM [24].

J.L. Träff et al. (Eds.): Euro-Par 2015, LNCS 9233, pp. 57–69, 2015.
DOI: 10.1007/978-3-662-48096-0_5

When it comes to performance analysis, traditional performance analysis tools (e.g. gprof [9], Scalasca [25], TAU [20], HPCToolkit [23] and Periscope [7]) have naturally associated performance metrics to syntactical application components such as routines, loops, and even statements. Despite this association has proven valuable and has helped understanding and improving applications, the impact of the memory hierarchy makes necessary to explore the performance from the data perspective, also. A study from this point of view includes, but it is not limited to, unveil which application variables are referenced the most and their access cost, detect memory streams to help prefetch mechanisms, calculate reuse distances, and even identify the cache organization that may improve the execution behavior. To this end, two mechanisms have emerged to address this type of studies. On the one hand, there exists instruction-based instrumentation that monitors load/store instructions and decodes them to capture the referenced addresses. While this approach accurately correlates code statements with data references, it imposes a severe expense, daunting the analysis with large data collections and/or time-consuming analysis; thus not being practical for long in-production executions. On the other hand, several processors have enhanced their Performance Monitoring Unit (PMU) to sample instructions based on a user specified period and associate them with data such as the referenced address. These mechanisms help on delimiting the amount of data captured and the overhead imposed. However, the results obtained are statistical approximations that may require sufficiently long runs so that the results approximate the actual distribution, yet highly volatile metrics may be missed.

The framework described in [18,19] addresses the latter issue and provides accurate and instantaneous performance metrics even using coarse grain sampling and minimal instrumentation. This framework smartly combines sampled and instrumented data by taking benefit of the repetitiveness from the applications. In this paper, we extend this framework by incorporating the application address space perspective to unveil the access patterns and the locality of reference to the application data structures. Such an extension relies on the address sampling mechanisms offered by the PMU extension known as PEBS [4] and to minimize the overhead we use large sampling periods. The result of this enhancement is a framework that provides complete support to gain insight of the application behavior, including the application syntactical level, its data structure organization, and its memory hierarchy usage and achieved performance.

The organization of this paper is as follows: Sect. 2 contextualizes our mechanism with respect to previous existing tools. Section 3 introduces the framework used as the basis for our mechanism and the hardware support for the address sampling. Then Sect. 4 describes the extension applied to the framework and exemplifies its results. Section 5 explores the behavior of two applications to demonstrate the usefulness of the resulting framework. Finally, Sect. 6 draws some conclusions and discusses possible future research trends.

2 Related Work

This section describes earlier approaches related to performance analysis tools that have focused to some extent on the analysis of data structures and the

efficiency achieved while accessing to them. We divide this research into two groups depending on the mechanism used to capture the addresses referenced by the load/store instructions.

The first group includes tools that instrument the application instructions to obtain the referenced addresses. MemSpy [13] is a prototype tool to profile applications on a system simulator that introduces the notion of data-oriented, in addition to code oriented, performance tuning. This tool instruments every memory reference from an application run and leverage the references to a memory simulator that calculates statistics such as cache hits, cache misses, *etc* according to a given cache organization. SLO [1] suggests for locality optimizations by analyzing the application reuse paths to find the root causes of poor data locality. This tool extends the GCC compiler to capture the application's memory accesses, function calls, and loops in order to track data reuses, and then it analyzes the reused paths to suggest code loop transformations. MACPO [17] captures memory traces and computes metrics for the memory access behavior of source-level data structures. The tool uses PerfExpert [3] to identify code regions with memory-related inefficiencies, then employs the LLVM compiler to instrument the memory references, and, finally, it calculates several reuse factors and the number of data streams in a loop nest. Tareador [22] is a tool that estimates how much parallelism can be achieved in a task-based data-flow programming model. The tool employs dynamic instrumentation to monitor the memory accesses of delimited regions of code in order to determine whether they can simultaneously run without data race conditions, and then it simulates the application execution based on this outcome. Peña *et al.* have designed an emulator based data-oriented profiling tool to analyze actual program executions in an emulated system equipped with a DRAM-based memory system only [16]. They also use dynamic instrumentation to monitor the memory references in order to detect which memory structures are the most referenced. With this setup, they estimate the CPU stall cycles incurred by the different memory objects to decide their optimal object placement in heterogeneous memory system.

The second group consists of tools that take benefit of hardware mechanisms to sample addresses referenced when processor counter overflows occur and estimate the accesses weight from the sample count. The Sun ONE Studio analysis tool has been extended in [10] by incorporating memory system behavior in the context of the application's data space. This extension brings the analyst independent and uncorrelated views that rank program counters and data objects according to hardware counter metrics, as well as, shows metrics for each element in data object structures. HPCToolkit has been recently extended to support data-centric profiling of parallel programs [11]. In contrast to the previous tool, HPCToolkit provides a graphical user interface that presents data- and code-centric metrics in a single panel, easing the correlation between the two. Giménez *et al.* use PEBS to monitor load instructions that access addresses within memory regions delimited by user-specified data objects and focusing on those that surpass a given latency [8]. Then, they associate the memory behavior with several semantic attributes, including the application context which is shown through the MemAxes visualization tool.

Our proposal belongs to the second group and its main difference from existing tools relies on the ability to report time-based memory access patterns, in addition to source code profiles and performance bottlenecks. The inclusion of the temporal analysis allows time-based studies such as detection of simultaneous memory streams, ordering accesses to the memory hierarchy, and even, code reordering. This data is captured using two independent monitoring tools that are configured to collect data sparsely. While one of the tools capture information regarding the performance bottlenecks, their nature and their association with the code; the other tool samples the references to the process address space.

3 Background

3.1 The Basic Framework

The framework described in [18,19] generates reports of the performance along time for computing regions from trace-files containing instrumented and sampled data. The computing regions can be manually delimited using instrumentation or automatically detected by the framework after the execution based on their performance characteristics. In the latter case, a computing region is defined as the user code in between successive parallel programming calls (such as MPI or OpenMP). These regions are automatically grouped according to their performance metrics (typically number of instructions and instruction rate) through a density-based clustering algorithm. Then, the framework applies a mechanism named folding that combines coarse grain sampled and instrumented information to provide detailed performance metrics within a computing region. In the context of the folding process, the samples are gathered from the computing into a synthetic region by preserving their relative time within their original region so that the sampled information determines how the performance evolves within the region. Consequently, the folded samples represent the progression in shorter periods of time no matter the monitoring sampling frequency, and also, the longer the runs the more samples get mapped into the synthetic instance. The framework has shown mean differences up to 5 % when comparing results obtained sampling frequencies that are two orders of magnitude more frequent (50×10^3 cycles *vs* 10^6 cycles).

3.2 Capturing the Referenced Addresses

The Precision Event Based Sampling (PEBS), and similarly the Instruction Based Sampling (IBS [5]), are respective extensions to the Intel's and AMD's PMU component that allow monitoring instructions at a user-configurable sampling period. These mechanisms periodically choose an instruction from those that enter into the processor pipeline. Then, the selected instruction is tagged, and it is monitored as it progresses through the pipeline while annotating any event caused by the instruction. When the instruction completes, the processor generates a record containing the instruction address, its associated events and

the machine state (without time-stamp), and then the record is written into a previously allocated buffer. Every time the buffer gets full, the processor invokes an interrupt service routine provided by a profiler that collects the generated records. Since instructions are reported at the retirement stage, these mechanisms exclude contributions from speculative execution. For the particular case of load instructions, PEBS collects data such as, but are not limited to: the linear address[1] referenced, the layer of the memory hierarchy that served the reference, and how many cycles did it take to reach the processor. These monitoring mechanisms report linear addresses from the process address space but do not provide information with respect to the physical addresses, thus they do not help understanding memory migrations.

4 Enhancement of the Framework

This section describes the integration of the sampled memory references into the aforementioned framework to display the time evolution of the memory access patterns in addition to other performance metrics. We also provide an example on how to use the output of this framework by applying it to a slightly modified version of a well-known benchmark.

4.1 Capturing Referenced Addresses

The first enhancement involves collecting the referenced addresses during the application execution so that the framework can later display them in the report. We use the Extrae[2] instrumentation package to generate the input for the original framework. Extrae uses PAPI [2] to capture hardware performance metrics, but PAPI does not capture the PEBS generated information[3]. perf [15], on the other hand, is a tool that uses the performance counters subsystem in Linux, and since Linux kernel version 3.11 it benefits from PEBS or IBS to collect memory references from either load or store instructions, but not both at the same time. This tool allocates a 1-entry buffer to store the memory references and then samples the application at a user defined period. Thus, each time the processor reaches the period, it generates a memory reference record, and then perf captures this record and associates a time-stamp to it. This way, perf is capable of generating timestamped trace-files containing sampled memory references even though neither PEBS nor IBS capture a timestamp.

Our approach relies on combining the results of these two monitoring tools when applied on an optimized application binary with debugging information in the same run, as depicted in Fig. 1. In this context, perf collects a time-stamped sequence of references while Extrae collects performance counters and

[1] Linear addresses also refer to logical addresses in x86-64 architectures as segmentation is generally disabled thus creating a flat 64-bit space, according to Sects. 3.3.4 and 3.4.2.1 from Intel®64 and IA-32 Architectures Software Developers Manual.

[2] http://www.bsc.es/paraver - Last accessed June, 2015.

[3] As of PAPI 5.4.0.

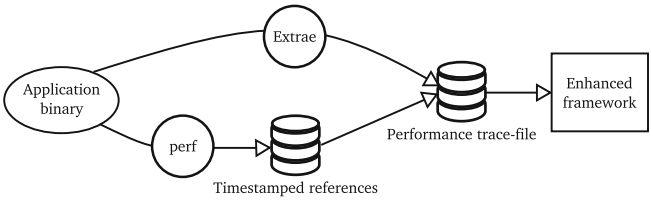


Fig. 1. Combination of two monitoring tools to generate a single trace-file that includes hardware counter performance metrics, call-stack references and data references.

call-stack references, and then a post-process combines them into a single trace-file. Both tools must use the same timing source in order to correlate the data captured. The perf tool uses low-level kernel timing routines and Extrae uses the Posix compliant high precision clock routines by default. Thus, we have adopted a kernel module that exposes the low-level timing routines[4] to the user-space applications, yet there are other possibilities to achieve this goal. After generating the trace-file, we extend the folding mechanism to apply to the memory reference samples and to collocate all the metrics (source code, memory references and node-level performance [such as MIPS rate and L1D miss ratio per instruction]) in one report per region.

4.2 Associating Addresses with Data Structures

When exploring the address space, it is convenient to map the address space to the application data structures in order to let the analyst match the generated results with the application code and also to explore their pattern access type. For that reason, Extrae has been extended to capture the base address and the size of the static variables, as well as, of the dynamically allocated variables. With respect to the static variables, the instrumentation package explores the symbols within application binary image using the binutils library[5] in order to acquire their name, starting address and size. Regarding the dynamic variables, we instrument the `malloc` family related routines and capture their input parameters and output results to determine the starting address and size. As dynamically allocated variables do not have a name, the tool collects their allocation call-stack reference to identify them. Since applications may contain lots of variables, Extrae ignores those smaller than a specified threshold (that defaults to 1 MiB). Finally, it is worth to mention that some languages (such as C and C++) allow declaring local (stack) variables within code blocks that can only be referenced by the inner block statements. While these references are captured by the perf tool, Extrae cannot track their creation; so, their references may appear on the resulting plot but do not have an associated variable name.

[4] https://lkml.org/lkml/2013/3/14/523.
[5] http://www.gnu.org/software/binutils. Last accessed June, 2015.

4.3 Practical Example

We have applied this framework to a modified version of the Stream benchmark [14] in order to show the usability of the described framework when exploring the load references. Since Stream accesses to statically allocated variables through ordered linear accesses, we have modified the code so that: (1) the `c` array is no longer a static variable but allocated by `malloc` and (2) the `scale` kernel loads data from pseudo-random indices from the `c` array. Due to modification (2), `scale` executes additional instructions and exposes lesser locality of reference, thus we have reduced the loop trip count in this kernel to $N/8$ to compensate its duration. The resulting code looks like:

```
for i := 1 to NITERS do                                   ! main loop
    for j := 1 to N do c[j] := a[j]; od                   ! Copy
    for j := 1 to N/8 do b[j] := s * c[random(j)]; od     ! Scale
    for j := 1 to N do c[j] := a[j] + b[j]; od            ! Add
    for j := 1 to N do a[j] := b[j] + s * c[j]; od        ! Triad
od
```

We have instrumented the loop body, compiled it using the GNU suite v4.8.1, and then, we have monitored the execution of the resulting binary on an Intel Core i7 2760QM running at 2.40 GHz and executing Linux 3.11. With respect to the monitoring, the Extrae package has sampled the application at 20 Hz and the perf tool has sampled the application every 250k load instructions, resulting in an overhead below 5 %.

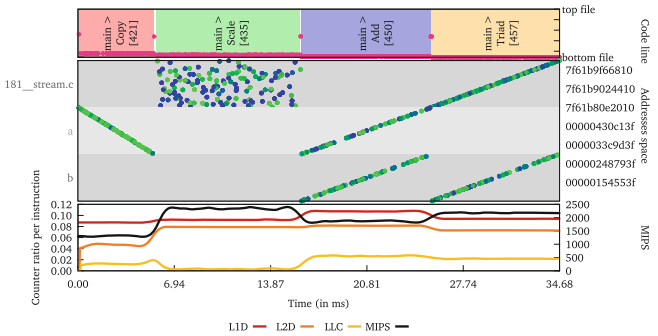


Fig. 2. Analysis of the modified Stream benchmark. Triple correlation time-lines for the main iteration: source code, addresses referenced and performance.

Figure 2 shows the result of the extended framework. The Figure consists of three plots: (1) source code references (top), (2) address space load references (middle), and (3) performance metrics (bottom). In the source code profile each color indicates the active routine (identified by a label of the form X >Y [n], where Y and X refer to the active routine and its ancestor, and n indicates the most observed line). Additionally, the purple dots represent a time-based profile of the sampled code lines where the top (bottom) of the plot represents the begin (end) of the container source file. This plot indicates that the application progresses through four routines and that most of the activity observed of

each of these routines occurs in a tiny amount of lines. The second plot shows the address space. On this plot, the background color alternates showing the space used by the variables (either static or dynamically allocated), and the left and right Y-axes show the name of the variables referenced and the address space, respectively. The dots show a time-based profile of the addresses referenced through load instructions and their color indicate the time to solve the reference based on a gradient that ranges from green to blue referring to low and high values, respectively. We want to outline several phenomena observed in this plot. First, as expected, the access pattern in the `Scale` routine to the variable allocated in line 181 of the file `stream.c` (formerly `c`) shows a randomized access pattern with most of the references in blue (meaning high latency). The straight lines formed by the references in the rest of the routines denote that they progressively advance and thus expose spatial locality, and also the greenish color indicates that these references take less time to be served. Second, the `Copy` routine accesses to the array `a` downwards despite the loop is written so that the loop index goes upwards. This effect occurs because the compiler has replaced the loop by a call to `memcpy` (from glibc 2.14) that reverses the loop traversal, unrolls the loop body and uses SSSE3 vector instructions. A linear regression analysis indicates that approximately each instruction references five addresses in `Copy` and since SSSE3 vector instructions may load up to 16 bytes, this translates into a 31.25 % vector efficiency. Finally, the instructions within routines `Add` and `Triad` reference two addresses per variable in average, the loaded data comes from two independent variables (or streams) simultaneously, and their accesses go from low to high addresses honoring the code. The third plot shows the achieved instruction rate (referenced on the right Y-axis) within the instrumented region, as well as, the L1D, L2D and LLC cache misses per instruction (on the left Y-axis). While we would expect a large cache miss ratio per instruction in `Scale`, we observe that they behave similarly to the rest of the kernel routines. This occurs because `random()` executes instructions to compute its results without accessing to the memory, thus reducing the cache miss ratio per instruction.

5 Usage Examples

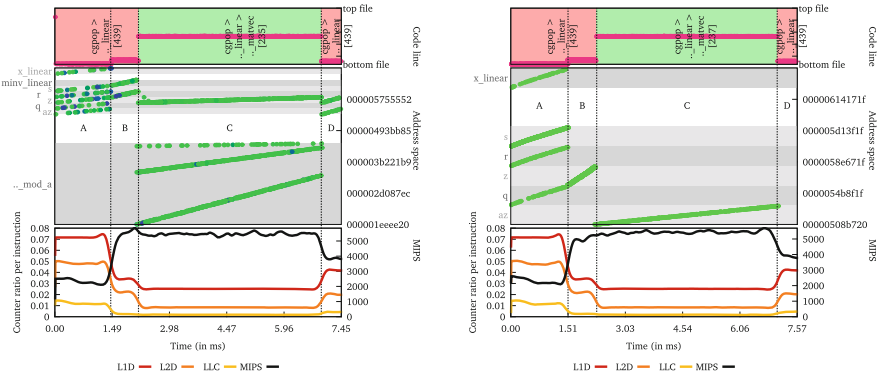
We have applied the extended framework to two parallel applications to demonstrate its usefulness. Table 1 provides details of the application, execution and monitoring characteristics. With respect to the systems, we have used a Core i7 system that includes the kernel module that allows the two monitoring mechanisms use the same clock source. Since the system only has four cores and we do not want to overload the system, we have used an additional Xeon system to execute the remaining processes that do not fit on the former machine. The Core i7 system has three levels of cache with a line size of 64 bytes: level 1 are two 8-way 32 KiB caches for instructions and data, level 2 consists of a 8-way unified 256 KiB cache, and level 3 is a 12-way unified 6,144 KiB cache.

Regarding the applications, each has been executed twice: the first execution captures information regarding the load references, while the second run collects

Table 1. Applications analyzed.

	CGPOP	**BigDFT** 1.7.5.13
# Processes	24	21
Processor type	Intel Core i7-2760QM @ 2.40 GHz Intel Xeon E5-2620v2 @ 2.10 GHz (*max:* 2.60 GHz)	
Application size	6 Klines 20 files	496 Klines 769 files
Compiler	GNU compiler suite 4.8.1	
Compiler flags	-O3 -g	-O2 -g
MPI implementation	OpenMPI v1.6.5	
Sampling period	20 ms	
Data sampling period	10^6 load, store instructions	

store references. The resulting plots are shown side-by-side for comparison purposes. Regarding the collecting, Extrae has been used to monitor MPI activity and it has sampled using a period coarser than the gprof sampling frequency (10 *vs* 20 ms). perf has been instructed to sample memory references every 10^6 load (or store) instructions. These coarse grain sampling frequencies ensured that the applications suffered a time dilation below 5 %.



(a) Load references.

(b) Store references.

Fig. 3. Analysis of CGPOP mini-application.

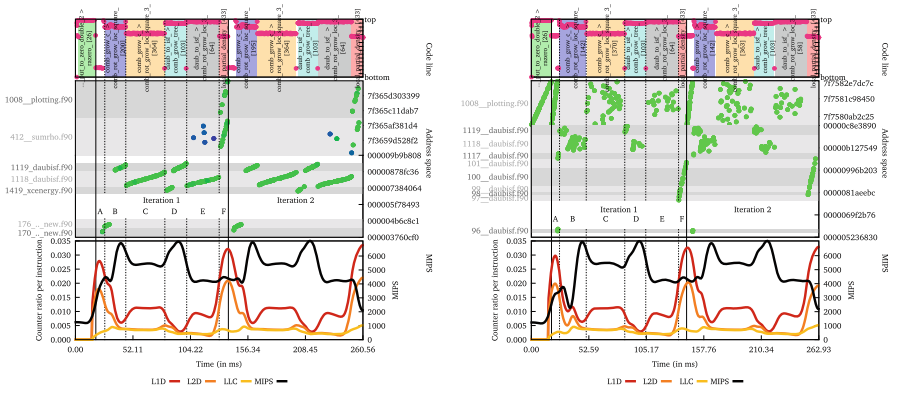
5.1 CGPOP

CGPOP [21] is a proxy application of the Parallel Ocean Program application. POP is a three-dimensional ocean circulation model designed primarily for studying the ocean climate system and it is a component within the Community Earth System Model. Figure 3 shows the obtained plots depicting the load and store references for the most time-consuming region of this execution. Note that both plots have its own address space depending on the accessed variables and that

the store memory references are shown in green because in this architecture the store instructions are inserted into a store buffer and these instructions are no longer under control of PEBS thus not having latency information for them. The Figure indicates that the region faces two routines: `pcg_chrongear_linear` (in red) and `matvec` (from the matrix module, in green), but we have also manually added labels (A-D) in the plot to ease the referencing. The latter routine takes most of the execution time within the region and also achieves the highest MIPS rate (above 5,000 MIPS). With respect to the load instructions within the data structures, we observe that phase C accesses to variables `z` and `a` (from the matrix module). The plot shows that the load references to variable `a` are partitioned into three disjoint portions that are accessed linear and simultaneously by the processor. The analysis of the source code shows that this variable represents a sparse row matrix that includes three arrays (one for double precision values and two for integer indices). When analyzing phase A, we observe that references require more time to be served (blue colored) and this is also related to the highest ratio of cache misses (1 out of every 14 instructions miss at L1D). The code in this phase loads data from six arrays (`x_linear`, `s`, `r`, `z`, `q` and `az`) and stores data to four arrays (`x_linear`, `s`, `r` and `q`). We have tested whether the code using an array of structures (AoS) improves the performance; however, our results indicate that using AoS does not offer performance improvements because the LLC miss ratio, as well as, the number of instructions doubles. With respect to the stores, we observe several effects: phase B generates the data for the array `z` and it is used immediately after in phase C, the `a` variable keeps unchanged during this region, and phase D does not expose stores because it reduces a vector into a scalar (`sumN2`).

5.2 BigDFT

BigDFT [6] is a massively parallel code based on density functional theories. Our analysis focuses on a computing region that corresponds to approximately 16 %



(a) Load references. (b) Store references.

Fig. 4. Analysis of BigDFT.

of the total execution time, and Fig. 4 shows the outcome of the extended framework for this region. The results indicate that the region consists of two iterations at all levels (source code, references, and performance). We have added labels to identify the iterations, as well as, the phases (routines) within the iterations (A-F). The first thing we notice is that load references expose better spatial locality than the store references, and that phase E shows a random access behavior in the load references and these references take more time to be served. We also outline that phase A traverses completely the array allocated in `plotting.f90` (line 1,008) to store values on it, and that happens immediately after executing the `razero` routine (depicted in green) which may be redundant because there aren't loads in between.

This report also shows some insights on the chances of making this region parallel using a task-based programming model. For instance, phases B and D store data in the data allocated in `daubis.f90` (line 1,118) and this data is used in phases C and E, begetting true (RAW) dependencies between these pair of phases. Also, phases B and D load and store data from the region allocated in `daubis.f90` (line 1,119) causing true (RAW) and output (WAW) dependencies between these phases. Finally, phase F mainly depends on the data located by `plotting.f90` (line 1,008) which is written by phases C and E. Due to the described dependencies, only phases A and B might safely run in parallel.

6 Conclusions and Future Work

We have presented an extension to a framework that displays the memory access patterns of computing regions and their time evolution along the source code and the performance behavior. This extension relies on the ability of recent hardware mechanisms available in current processors to sample instructions based on a user-defined period and attributes to each sample several performance metrics, including the addresses referenced. This enhanced framework has proven valuable to give detailed insight regarding several optimized application binaries, such as detecting the most dominant data streams and their temporal evolution along computing regions. For instance, we have seen that the compiler has replaced the source code by a call in Stream, that CGPOP accesses multiple memory streams simultaneously, and that there may exist redundant work in BigDFT. All this information has been captured using minimal instrumentation and coarse grain sampling periods, thus keeping a low expense during the measurement.

We believe that there are research opportunities using these hardware memory sampling techniques. For instance, we consider using the outcome of this extended framework to capture the store access patterns and then search for those variables that are not used shortly. The access to these variables may benefit from non-temporal instructions because these instructions do not write data into the cache hierarchy, nor fetches the corresponding line; thus not polluting the cache hierarchy. Another direction would include studying data dependencies for porting an application to a task-based data-flow programming model using partial data. Finally, it would be valuable to extend the memory monitoring mechanism to multiplex in order to capture load and store references in one run.

Acknowledgments. We would like to thank Damien Caliste and Luigi Genovese for their insightful comments on BigDFT. We thankfully acknowledge the support of the *Comisión Interministerial de Ciencia y Tecnología* (CICYT) under contract No. TIN2012-34557 which has partially funded this work.

References

1. Beyls, K., D'Hollander, E.H.: Refactoring for data locality. IEEE Comput. **42**(2), 62–71 (2009). http://dx.doi.org/10.1109/MC.2009.57
2. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. **14**(3), 189–204 (2000). http://icl.cs.utk.edu/papi
3. Burtscher, M., et al.: PerfExpert: an easy-to-use performance diagnosis tool for HPC applications. In: Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–11 (2010). http://dx.doi.org/10.1109/SC.2010.41
4. Corporation, I.: Intel 64 and IA-32 architectures software developer's manual. Volume 3B: System Programming Guide, Part 2, January 2015
5. Drongowski, P., et al.: Incorporating instruction-based sampling into AMD CodeAnalyst. In: Performance Analysis of Systems Software, pp. 119–120 (2010)
6. Genovese, L., Neelov, A., Goedecker, S., Deutsch, T., Ghasemi, S.A., Willand, A., Caliste, D., Zilberberg, O., Rayson, M., Bergman, A., Schneider, R.: Daubechies wavelets as a basis set for density functional pseudopotential calculations. J. Chem. Phys. **129**(1), 014109 (2008)
7. Gerndt, M., Fürlinger, K., Kereku, E.: Periscope: advanced techniques for performance analysis. In: PARCO, pp. 15–26 (2005)
8. Giménez, A., et al.: Dissecting on-node memory access performance: a semantic approach. In: International Conference for High Performance Computing, Networking, Storage and Analysis, vol. 2014, pp. 166–176 (2014)
9. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: a call graph execution profiler. In: SIGPLAN 1982: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, pp. 120–126 (1982). http://doi.acm.org/10.1145/800230.806987
10. Itzkowitz, M., et al.: Memory profiling using hardware counters. In: SC 2003: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, p. 17 (2003)
11. Liu, X., Mellor-Crummey, J.M.: A data-centric profiler for parallel programs. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2013, pp. 28:1–28:12 (2013). http://doi.acm.org/10.1145/2503210.2503297
12. Loh, G.H.: 3D-stacked memory architectures for multi-core processors. In: 35th International Symposium on Computer Architecture, pp. 453–464 (2008). http://dx.doi.org/10.1109/ISCA.2008.15
13. Martonosi, M., Gupta, A., Anderson, T.E.: MemSpy: analyzing memory system bottlenecks in programs. In: SIGMETRICS, pp. 1–12 (1992). http://doi.acm.org/10.1145/133057.133079
14. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. In: IEEE Computer Society Technical Committee on Computer Architecture Newsletter, pp. 19–25 (1995)
15. de Melo, A.C.: The new linux "perf" tools. In: Linux Kongress (2010)
16. Peña, A.J., Balaji, P.: Toward the efficient use of multiple explicitly managed memory subsystems. In: 2014 IEEE International Conference on Cluster Computing, pp. 123–131 (2014)

17. Rane, A., Browne, J.: Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics. In: International Conference on Parallel Architectures and Compilation Techniques, pp. 147–156 (2012). http://doi.acm.org/10.1145/2370816.2370838
18. Servat, H., et al.: Unveiling internal evolution of parallel application computation phases. In: ICPP, pp. 155–164 (2011)
19. Servat, H., et al.: Bio-inspired call-stack reconstruction for performance analysis. Technical report, UPC-DAC-RR-2014-20, Department of Computer Architecture, Universitat Politècnica de Catalunya (2014)
20. Shende, S.S., Malony, A.D.: The TAU parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006)
21. Stone, A., Dennis, J., Strout, M.M.: The CGPOP miniapp, version 1.0. Technical report, CS-11-103, Colorado State University (2011)
22. Subotic, V., Ferrer, R., Sancho, J.C., Labarta, J., Valero, M.: Quantifying the potential task-based dataflow parallelism in MPI applications. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 39–51. Springer, Heidelberg (2011)
23. Tallent, N., et al.: HPCToolkit: performance tools for scientific computing. J. Phys.: Conf. Ser. **125**(1), 012088 (2008)
24. Wang, C., et al.: NVMalloc: exposing an aggregate SSD store as a memory partition in extreme-scale machines. In: 26th IEEE International Parallel and Distributed Processing Symposium, pp. 957–968 (2012)
25. Wolf, F., et al.: Usage of the SCALASCA for scalable performance analysis of large-scale parallel applications. In: Tools for High Performance Computing, pp. 157–167 (2008)