Parallel Computing 51 (2016) 46–55

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

A data-oriented profiler to assist in data partitioning and distribution for heterogeneous memory in HPC

Antonio J. Peña*, Pavan Balaji

Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, United States

ARTICLE INFO

Article history:
Available online 5 November 2015

Keywords:
Data-oriented profiling
Object-differentiated profiling
Heterogeneous memory
Scratchpad memory
Valgrind

ABSTRACT

Profiling is of great assistance in understanding and optimizing an application's behavior. Today's profiling techniques help developers focus on the pieces of code leading to the highest penalties according to a given performance metric. In this paper we describe a profiling tool we have developed by extending the Valgrind framework and one of its tools: Callgrind. Our extended profiling tool provides new object-differentiated profiling capabilities that help software developers and hardware designers (1) understand access patterns, (2) identify unexpected access patterns, and (3) determine whether a particular memory object is consistently featuring a troublesome access pattern. We use this tool to assist in the partition of big data objects so that smaller portions of them can be placed in small, fast memory subsystems of heterogeneous memory systems such as scratchpad memories. We showcase the potential benefits of this technique by means of the XSBench miniapplication from the CESAR codesign project. The benefits include being able to identify the optimal portion of data to be placed in a small scratchpad memory, leading to more than 19% performance improvement, compared with nonassisted partitioning approaches, in our proposed scratchpad-equipped compute node.

1. Introduction

Analyzing an application's performance has been of interest since the early days of computers, with the objective of exploiting the underlying hardware to the greatest possible extent. As computational power and the performance of data access paths have been increasing, such analysis has become less crucial in commodity applications. However, software profiling is still used largely to determine the root of unexpected performance issues, and it is especially relevant to the high-end computing community.

The first modern profiling approaches used code instrumentation to measure the time spent by different parts of applications [1,2]. Recognizing that the same functional piece of code may behave differently depending on the place from where it is called (i.e., its stack trace), researchers devised more advanced tools that use call graphs to organize profiling information. This approach is limited, however, to pointing to the conflicting piece of code consuming large portions of execution time, without providing any hint about its root.

A more informative approach consists of differentiating time spent in computation from the time spent on memory accesses. In current multilevel memory hierarchies, cache hits pose virtually no penalty, whereas cache misses are likely to lead to large

* Corresponding author. Tel.: +34 934137734.
E-mail addresses: apenya@mcs.anl.gov (A.J. Peña), balaji@anl.gov (P. Balaji).

http://dx.doi.org/10.1016/j.parco.2015.10.006
0167-8191/

amounts of processor stall cycles. Cache misses, therefore, are a commonly used metric to determine whether a performance issue is caused by a troublesome data access pattern.

In this regard, cache simulators have been used in optimizing applications [3,4]. The main benefit of these is their flexibility: they enable the cache parameters to be easily changed and the analysis repeated for different architectures. A disadvantage of this profiling approach is the large execution times caused by the intrinsically large cache simulation and system emulation overheads.

The introduction of hardware counters enabled a relatively accurate and fast method of profiling [5–7]. This method is based mainly on sampling the hardware counters at a given frequency or determined by certain events and relating these measurements with the code being executed. Both automated and user-driven tools exist for this purpose, and in some cases code instrumentation is not required. Unlike their simulator-based counterpart, however, these limit the analysis to the real platform on which they are executing on. Although different architectures provide a different set of counters, cache misses are commonly available and a widely used profiling metric.

All these approaches are code focused. They point developers to the place of code showing large execution time or large cache miss rates. However, it is not uncommon to perform accesses to different memory objects from the same line of code. In this case, a code-oriented profiler does not provide detailed information about the root of the problem.

Data-oriented profiling is intended to complement that view. This approach seeks to identify the data showing problematic access patterns throughout the execution lifetime, which may be imperceptible from a traditional approach. This view focuses on data rather than lines of code.

In this paper we present a tool we have developed on top of state-of-the art technologies. Specifically, we have incorporated per-object tracing capabilities into the widely used Valgrind instrumentation framework [8]. Furthermore, we have extended Callgrind [4], a tool from Valgrind's large ecosystem, in order to demonstrate the possibilities of our approach. Memory-object differentiated profiling can be of help in emerging and upcoming heterogeneous memory systems for high-performance computing (HPC) as a means of determining the most appropriate distribution of applications data. In this regard, we introduce a novel technique to assist in data partitioning for these systems based on data-oriented profiling and main memory access histograms.

The main contributions of this paper are: (1) we introduce a novel technique for data distribution and partitioning for heterogeneous memory systems targeting performance and based on data-oriented profiling and cache miss histograms, (2) we present the key design and development details of a data-oriented profiler to assist in that process, and (3) we demonstrate the applicability of our proposed methodology by means of a use case. To the best of our knowledge, this is the first time a data-oriented profiling technique based on access histograms is proposed to assist in the data partitioning process towards efficiently using explicitly-managed small but fast memory subsystems in compute nodes.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 introduces the Valgrind instrumentation framework as the basis for our profiling tool. Section 4 describes our modifications to incorporate per-object differentiation capabilities into the Valgrind core and the Callgrind profiler. Section 5 presents and analyzes a use case for data distribution using this tool. Section 6 provides concluding remarks.

2. Related work

In this section we discuss some representative work related to our research. We start by reviewing prior work on data-oriented profiling; we then discuss work related to data distribution among memory subsystems.

2.1. Data-oriented profiling

The Sun ONE Studio compilers and performance tools were extended in [9] to offer object-differentiated profiling based on hardware counters. Although this technique features low overheads, however, it does not provide the flexibility of the software solutions based on system emulators. In addition, because of the intrinsics of the processor architectures, its granularity is limited.

MemSpy [10] was an early "prototype tool" providing object-differentiated profiling. It implemented a technique similar to ours for merging accesses to different memory objects. An interesting feature of this tool was the simulation of the memory subsystem, enabling the possibility of using the processor stall cycles as a performance metric. To the best of our knowledge, however, MemSpy was never made publicly available, and the project was discontinued.

A recent data-oriented profiling Valgrind tool, called Gleipnir [11], provides object-differentiated data access tracing. It is integrated with the Gl_cSim cache simulator instead of Callgrind. This tool analyzes the cache behavior of the different data structures of applications. Our work differs in that our tool does not focus on cache performance issues but, instead, provides object-differentiated access pattern information.

2.2. Data distribution

Most of the related work on data distribution focuses on embedded systems featuring scratchpad memories, deployed as energy savers because of being less power-hungry than caches. Broadly speaking, the data distribution techniques can be divided into either static (compile-time) or dynamic (run-time).

Dynamic distribution schemes require hardware assistance from the memory management unit in order to manage virtual address mapping to different memory subsystems at run time. One of the closest related works on this area is [12], which proposed a dynamic scratchpad memory management technique focused on frequently accessed data targeting energy reduction in embedded systems. The authors used profiling information based on cache misses at page granularity to aid in data distribution in a simulated custom memory architecture.

Static approaches, on the other hand, do not require specific hardware to assist in memory distribution. There is previous work addressing the problem of memory partitioning among applications [13], targeting those cases in which a given system runs a predefined set of applications concurrently. In the case of data distribution within applications, many previous works do not consider splitting memory objects, expecting large overheads from boundary checks [14]. In our previous work targeting HPC environments [15], however, we experienced low contributions to performance from scratchpad memories leveraging nonpartitioned memory object distributions based on data-oriented profiling; and we observed that small-sized memory objects tend to contribute only a low number of cache misses to the overall execution. Verma et al. [16] explored object partitioning oriented to efficient scratchpad usage in terms of energy savings in embedded systems. Their approach involved autogenerating code by the compiler toolchain in a source-to-source compiling stage in order to split the data objects, based on an energy model and code-oriented profiling; thus, iterative profiling was needed in order to determine the best partitioning solution. Additionally, the compiler-generated code always inserted branches for boundary checks, not benefiting from those cases where these can be avoided (as we explain in Section 5.4).

Mixed approaches have also been explored. For example, in [17] the authors use static partitioning based on code analysis plus dynamic placement to migrate objects among memories at run time for streaming applications.

Our work focuses on HPC environments where the hardware resources of a given compute node are usually assumed to be used exclusively by a single application, or at least by a well-known reduced set of applications. Such environments feature much more powerful processors than do embedded systems, having for instance better branch predictors. Moreover, the processor-to-memory performance ratios tend to be much higher, leading to different constraints. Performance and energy concerns about the use of scratchpad memories also differ. Indeed, we demonstrate that the static partitioning approach does not necessarily introduce significant overhead in HPC applications, and avoids the use of complex hardware-assisted placements at run time. To the best of our knowledge, this is the first study proposing the use of object-differentiated profiling in data partitioning for heterogeneous memory in nonembedded environments.

3. Background

To develop our data-oriented profiling tool, we extend Valgrind, a generic instrumentation framework, and Callgrind, its call-graph profiler.

3.1. Valgrind

Valgrind is a generic instrumentation framework. It comprises a set of *tools* that use this generic core, including its default memory debugging tool [18].

Valgrind can be seen as a virtual machine. It performs just-in-time compilation, translating binary code to a processor-agnostic intermediate representation, which the tools are free to analyze and modify by means of a high-level application programming interface (API). Tools tend to use this process for inserting hooks (instrument) to perform different tasks at run time. This code is taken back by the Valgrind core to be executed. The process is performed repeatedly in chunks of code featuring a single entry and multiple exit points, called "super blocks" (SBs), as depicted in Fig. 1. The main drawback of this process is the incurred overhead. Typically the code translation process itself—ignoring the tool's tasks—poses an overhead of $4\times$ to $5\times$.

Valgrind exposes a rich API to its tools for tool-core interactions, plus a client request mechanism for final applications to interact with the tools if needed. In addition, it facilitates the possibility of intercepting the different memory allocation calls, enabling tools to provide specific wrappers for them.

3.2. Callgrind

Callgrind is a Valgrind tool defined as "a call-graph generating cache and branch prediction profiler." By default, it collects the number of instructions and function calls, and the caller–callee relationship among functions (the call graph). All this data is related to the source lines of code. If cache simulation is enabled, cache misses can be used as a performance metric. By default, Callgrind simulates a cache hierarchy featuring the characteristics of the host computer, with the aim of providing an accurate estimation of the host cache behavior. Although in practice the provided measurements diverge from those obtained by actual hardware counters, it is still a useful profiling tool: the fact that the simulated cache does not behave exactly as does the physical cache is not relevant if the profiling goal is to determine cache-unfriendly accesses and fix the application's access pattern not only for a particular cache implementation. Additional features include a branch predictor and a hardware prefetcher.

After Callgrind has profiled an application, one can use KCachegrind as a postprocessing tool that enables the graphic visualization of the collected data.

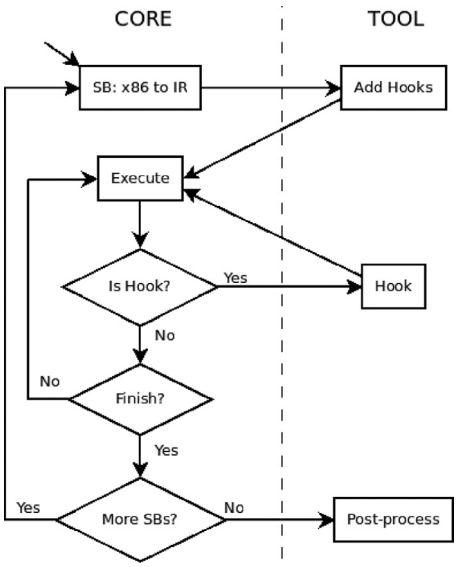

Fig. 1. Simplified high-level view of the interaction between Valgrind and its tools.

```
// Scope #1
// i1 is valid
int i1;

{
  // Scope #2
  // i1 and i2 are valid
  int i2;
}

// Scope #1
// Only i1 is valid again
```

Fig. 2. Example of scopes.

4. Valgrind extensions

Our extensions are based on the development branch of Valgrind 3.10.0. We first introduce the integration of the new functionality not specifically related to a particular tool. Next we describe the way we extended Callgrind to use these new capabilities. Further details and other extended profiling capabilities outside the scope of this study can be found in [19].

4.1. Core extensions

Our modifications focus on enabling the differentiation of memory objects. To this end, we incorporate functionality to locate the memory object constituting a given memory address and store its associated access data. We differentiate two types of memory objects—statically and dynamically allocated—requiring two different approaches.

4.1.1. Statically allocated memory objects

Our changes concentrate on the debug information functionality that Valgrind exposes to its tools. This functionality requires applications to be compiled with embedded debug information (usually by means of the `-g` compiler option). We developed new functionality to locate variables and record their accesses.

The information about the statically allocated variables is distributed among the different binary objects constituting an application, including the different dynamic libraries an application may use. The internal Valgrind representation holds this information in a linked list of objects—one per binary object. In addition, different scopes may exist in which the different statically allocated variables may or may not be exposed depending on the current program counter (see an example in Fig. 2). Note that the address of each variable is defined only when its scope is active.

We follow the algorithm already employed by Valgrind to locate statically allocated variables, leveraged for instance by the memory debugger tool (Memcheck) to indicate the location of an illegal access (see the pseudocode in Fig. 3). The asymptotic computational cost of this algorithm is

$$O(st \times (dio + sao)),$$

```
for instruction_ptr in stack_trace:
  debug_info = debug_info_list
  while debug_info:
    if instruction_ptr in debug_info.txt_mapping:
      debug_info_list.bring_front(debug_info)
      # From inner to outer
      for scope in debug_info_scopes:
        vars = scope.get_vars(instruction_ptr)
        for var in vars:
          if address in var:
            return var.user_id
    debug_info = debug_info.next
return MEM_OBJ_NOT_FOUND
```

Fig. 3. Algorithm to find statically allocated variables.

where *st* is the maximum stack trace depth, *dio* is the number of debug information objects, and *sao* the maximum number of statically allocated objects defined for a given IP. Since this process is time consuming and since users are likely to focus primarily on their application variables—not including external libraries, by default this new functionality considers only the debug information in the object containing the main entry point (that is, the executable object). Tools can fully control the debug information objects to be considered.

4.1.2. Dynamically allocated memory objects

Taking advantage of Valgrind's capabilities, we intercept the application calls to the memory management routines and provide wrappers for them. Following Valgrind's infrastructure, this feature is implemented on the tool side as a separate module, since the management of the memory handling routines is expected to be tool dependent. Nevertheless, our code can be used by different tools, and the exposed API is similar to the case of statically allocated variables.

The information about the dynamically allocated objects is kept within an ordered set using the starting memory address as the sorting index. This approach enables binary searches whose asymptotic computational cost is

$$O(\log dao),$$

where *dao* is the number of dynamically allocated objects of a given application. This algorithm is enabled by the fact that the dynamically allocated objects reside in the global scope (in other words, they are globally accessible), and hence their addresses do not change among scopes.

We also implemented a merge technique for this kind of memory object, similar to that described in [20]. Merging the accesses of different memory objects, provided that these were created in the same line of code and feature a common stack trace, provides a unique view of objects that, in spite of being created by separate memory allocation calls, are likely to be considered as a single object from an application-level view. As an example, consider a loop allocating an array of lists as part of a matrix or an object being repeatedly created and destroyed when entering and leaving a function. Note that the latter needs to be called from the same line of code (i.e., within a loop) in order to meet the condition of sharing the same stack trace. This feature is optional and can be disabled by the tool.

4.1.3. Access histograms

We have incorporated in our Valgrind core extensions for data-oriented profiling the capability of generating *access histograms*—detailed information regarding the location within the memory object where the access occurs is recorded during the execution. This feature is highly memory-consuming, requiring a 64-bit counter for each location to be monitored. The length of the array of counters depends on the size of the memory object and the desired granularity (for example, the typical granularity specified by a profiler based on cache simulation such as Callgrind would not be smaller than the last-level cache line size). Accordingly, this feature is enabled on-demand only.

Access histograms are intended to help users examine the per-object access distribution, for example, identifying whether any parts of their memory objects are causing a higher rate of cache misses. Users can review their access patterns, looking for bugs when encountering unexpected distributions or, as later discussed in this paper, partitioning their data to accommodate the most-accessed portions of their memory objects to faster but smaller memories.

4.2. Extending Callgrind

We have modified Callgrind to use the functionality described in Section 4.1 in order to offer object-differentiated profiling capabilities. Following the observation that applications tend to feature far fewer dynamically allocated objects than their statically allocated counterpart does and that the latter tends to cover a much larger amount of memory space, we first perform the search in the dynamic set of objects for performance purposes. We have also included support for integration with the KCachegrind visualization tool.

Table 1
Cache configuration in our experiments.

Description	Total size	Associativity	Line size
L1 Instruction	32 KB	8	64 B
L1 Data	32 KB	8	64 B
LL Unified	8 MB	16	64 B

Table 2
Memory setup for experiments.

Memory	Latency	Size
L1 Instr.	0 c	32 KB
L1 Data	0 c	32 KB
L2	20 c	8 MB
SP	20 c	32 MB
Main	200 c	32 GB

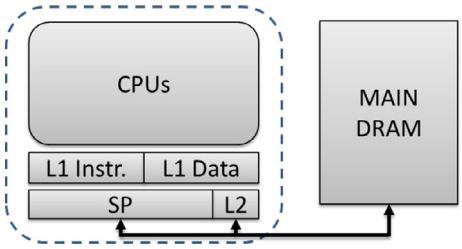

Fig. 4. Target system architecture.

5. Data partitioning for heterogeneous memory

In this section we discuss the usability of our new profiling tool to assist in data partitioning in order to efficiently use explicitly managed small but fast memory subsystems. After presenting our target system, we introduce our use case application. Next we analyze the profiling information gathered by means of our tool. We then discuss the performance improvements obtained.

Our evaluations are carried out in a quad-core Intel Core i7-3940XM CPU running at 3 GHz, with the first and last levels of cache as detailed in Table 1, and equipped with 32 GB of main memory. Where applicable, the results presented are the average of three executions and are accompanied with relative standard deviation (RSD) information.

5.1. Target system

Our proposed target system, as depicted in Fig. 4, is composed of an 8-core processor equipped with a two-level cache hierarchy as in Table 1, a scratchpad memory (SP) at the L2 cache level, and a traditional DRAM-based main memory. Since neither mainstream systems nor compute nodes equipped with scratchpad memories are available today, we base part of our analysis on cache simulations and cycle estimations. Table 2 specifies the sizes of the different memory subsystems as well as their respective average access latencies on which we base our estimations. We use the average latency estimations of [15], where readers can find more details about the different memory technologies of our target system along with a justification for these estimations. In our execution cycle estimations we consider our processor to leverage one instruction per cycle per core, serialized main memory accesses, and no stall cycles caused by hazards. Branch mispredictions are assumed to pose a cost of 15 CPU cycles (consistent with many modern processors), and data prefetching is also enabled.

5.2. XSBench

XSBench [21] is part of the CESAR Monte Carlo neutronics miniapps. It targets the typically most computationally intensive piece of Monte Carlo transport algorithms leveraged by nuclear reactor simulations: the computation of macroscopic neutron cross-sections. According to its documentation, in a typical simulation this poses approximately 85% of the total runtime of OpenMC [22]—the fully featured application, and this miniapplication mimics its data access patterns. The user-defined input of XSBench includes the number of nuclides to simulate, the number of grid points per nuclide, and the number of lookups of cross-section data to perform.

Table 3
Last-level cache miss statistics for XSBench.

Object	Size	Absolute	Relative
`nuclide_grids`	184 MB	$1.7 \cdot 10^9$	88.8%
`energy_grid`	61 MB	$1.3 \cdot 10^8$	6.6%
`energy_grid.xs`	5,434 MB	$9.1 \cdot 10^7$	4.6%

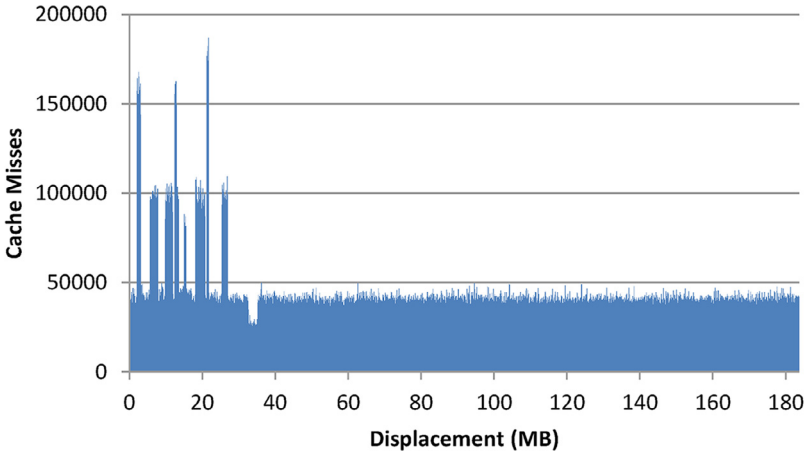

Fig. 5. LL cache miss distribution for the `nuclide_grid` memory object at 4 KB granularity.

We base most of our analyses on a simulated "large"[1] reactor with the default configuration of 355 nuclides, 11,303 grid points per nuclide, and 15 million lookups. Eight threads are deployed by means of OpenMP, and over 5.5 GB of memory are used. In these executions, we find over 70% of the last-level cache accesses resulting in cache misses, which implies that a significant amount of the execution time is spent in CPU stall cycles waiting for data, what would result in over 80% of the execution time in our target architecture.

We use XSBench version 13, dated May 2014, and limit our profile to the actual cross-section simulation lookup loop, omitting from our optimization study the initialization and finalization stages.

5.3. Analysis

Our goal is to determine what part of the data used by our application to place in the SP memory in order to maximize performance. To accomplish this, we profile the application making use of our extended Callgrind profiler.

Table 3 shows the three memory objects featuring most last-level cache misses in our execution. These represent over 99.9% of the overall accesses to main memory. The first object, `nuclide_grids`, is an array of data structures holding the matrix of nuclide grid points. The second object, `energy_grid`, is an array of data structures containing an energy value and a pointer to the third object, `energy_grid.xs`, permitting the access to the latter contiguous array of integers by rows in a matrixlike arrangement. Our profiling reveals no write misses to these objects during our profiled portion of the execution.

As Table 3 shows, the majority of the cache misses of the execution occur in data objects that are too large to fit in our target scratchpad memory. Hence, we extract the *main memory access histogram* for these objects to assess whether there is some smaller portion of them in which the accesses are concentrated in order to place it in the scratchpad memory subsystem. While both histograms for the `energy_grid` objects show a uniform distribution, the remaining histogram reveals interesting results for our purposes.

Fig. 5 shows that over 23% of the cache misses for the `nuclide_grids` object occur in the contiguous region of the buffer comprising less than 15% from its beginning, being up to 80% higher than other regions of the same size. This phenomenon is explained by the physical distribution of the materials in the core of the reactor, causing some of them to be hit with a higher probability.

Next we explain how we benefit from these findings in order to partition and distribute the data efficiently for our target system.

[1] According to XSBench naming.

```
size_t elems[2] = { ELEMS1, ELEMS - ELEMS1 }
int * split_buffer[2];
split_buffer[0] = (int *) malloc(elems[0] * sizeof(int));
split_buffer[1] = (int *) malloc(elems[1] * sizeof(int));

// Example of code needing branching
int get_elem(int **split_buffer, size_t *elems, size_t nth) {
  if(nth < elems[0]) return split_buffer[0][nth];
  else               return split_buffer[1][nth-elems[0]];
}

// Example of code not needing branching
int sum(int **split_buffer, size_t *elems) {
  int sum=0;
  for(int b=0; b<2; b++)
    for(int i=0; i<elems[b]; i++)
      sum += split_buffer[b][i];
  return sum;
}
```

Fig. 6. Example of split buffer code and branching requirements.

Table 4
Performance (Lookups/s) overhead of memory object splitting in XSBench.

Size	Original	Modified	Overhead
Small	3,306,869	3,282,984	0.7%
Large	984,213	975,245	0.9%

Table 5
Profiling data of memory object splitting versus original version.

Source	Size	Instructions	LL Misses	Branches	Mispred.
Callgrind	Small	+5.1%	−0.6%	+16.4%	−0.6%
	Large	+7.7%	+0.1%	+32.1%	−0.4%
PAPI	Small	+5.7%	+0.1%	+17.4%	−0.2%
	Large	+7.6%	−0.1%	+32.4%	−0.1%

5.4. Improvements

We modify the XSBench miniapplication according to the presented analysis so that the identified memory object is now divided. We leave 61 array elements in the first portion, which constitute 31.6 MB and now fit into our SP memory.

Fig. 6 illustrates the sort of modifications required in the code to leverage the buffer partitioning. Note that we avoid branching to check buffer limits in those cases requiring buffer traversal. Studying the feasibility of automatizing this process is left for future work.

5.4.1. Buffer splitting overhead

We next assess the performance impact of our code modifications for object partitioning. Table 4 shows the measured performance in lookups/s of XSBench for both the small (246 MB dataset) and large simulation sizes in our evaluation testbed, obtaining under 1% penalty in both cases (RSDs under 0.5%).

To understand the reasons behind the low performance impact of our modifications, we profiled the executions of both the original and the modified versions of our use case. Our results are summarized in Table 5, showing the differences of the modified version with respect to the original code. We present instruction counts, last-level cache misses, number of executed branches, and branch mispredictions. We use both the Callgrind profiler and the PAPI support incorporated in XSBench in order to obtain actual hardware counters. The RSDs are under 0.5% except in the case of branch mispredictions, which reach a maximum of 3.0%. Our results reveal up to a moderate increase of over 7% in instruction references, which pose a low performance impact because of the large number of last-level cache misses this application produces, and up to a relatively large increase of over 30% in the number of branches. Low misprediction rates of under 6% in our simulator and 2% in our real hardware, however, alleviate the concern regarding the performance impact of the increased number of branches. Last-level cache misses and branch misprediction rates do not show a significant variation. Note that both profiling methods show close results, being within 1% difference in all cases, thus validating our simulator-based technique.

Table 6
Estimated performance improvements for different object partition choices.

Approach	Misses	Cycles	Execution
Biggest	$5.4 \cdot 10^5$	$-9.7 \cdot 10^7$	−0.0%
Misses-Worst	$2.7 \cdot 10^8$	$-4.8 \cdot 10^{10}$	−11.7%
Misses-End	$2.8 \cdot 10^8$	$-4.9 \cdot 10^{10}$	−12.0%
Optimized	$4.4 \cdot 10^8$	$-7.9 \cdot 10^{10}$	−19.5%

The relatively small overheads caused by the code-based array partitioning approach, mainly enabled by the enhanced branch predictors present in current mainstream and HPC processors in contrast to those of embedded systems, are easily overcome by the benefit of being able to efficiently use a faster memory subsystem, as analyzed next.

5.4.2. Performance improvements

From the average access latency times of our target platform's memory subsystems and the cycle estimations as introduced in Section 5.1, we estimate the benefits of efficiently using the scratchpad memory aided by our profiler. Since the memory objects we consider cover over 99.9% of the main memory accesses of the simulation, we consider negligible the performance impact of using this memory to host small memory objects fitting without data partitioning. We also compare our optimized distribution with the following approaches.

1. *Biggest* Placing the first 32 MB of the biggest data object (`energy_grid.xs`) in the SP memory. This could be a naive approach if performing data partitioning without assistance. Placing the last 32 MB of data would lead to similar results, since this memory object presents a uniformly random data access pattern. Since this object is accessed through row pointers in a matrix-like distribution, it does not require any change in its access code, as long as entire rows are allocated within the same memory region. Hence, this partitioning does not involve any loop-time overhead.
2. *Misses-Worst* Populating the SP memory with the object featuring the largest number of cache misses (`nuclide_grids`), but in the region causing the lower number of them (see Fig. 5). Although in this case the programmer probably would not have chosen such an arbitrary offset, we include this case for comparison.
3. *Misses-End* Allocating the last part of the object exposing the largest number of cache misses in the SP memory. This could be a choice when one has a data-oriented profiling tool without access histogram capabilities.
4. *Optimized* Optimized partitioning of the `nuclide_grids` memory object into the main and SP memories as described in Section 5.3.

Table 6 summarizes the performance improvements attained by the different partition approaches with respect to a nonpartitioned data distribution in which the scratchpad memory would be profoundly underutilized. The estimated cycle differences include the memory access savings derived from the average latencies presented in Table 2. In those cases partitioning the `nuclide_grids` object, the cycle differences also include the increase on executed instructions and mispredicted branches from the profile information summarized in Section 5.4.1. As we can see in the table, the immediate approach of placing part of the biggest data object in the scratchpad memory (Biggest) does not lead to any noticeable improvement because of the low rate of cache misses per byte this memory object features. If we had made a better decision assisted by a data-oriented profiler without the access histogram feature (Misses-Worst and Misses-End), we would have attained gains in the execution time close to 12%. If we make the decision assisted by an analysis of the access histograms, however, the Optimized partitioning results in over 19% execution improvement with respect to the baseline case, thanks to the placement of the region of memory with highest density of cache misses in the scratchpad memory subsystem, hence minimizing the CPU stall cycles caused by memory accesses during execution.

6. Conclusions

In this paper we have presented a tool providing object-differentiated analysis capabilities based on a state-of-the-art and widely used technology: the Valgrind instrumentation framework. We have described our key design and implementation details, based in the Callgrind tool of the Valgrind ecosystem that we extended for this purpose. The added per-object access histogram capabilities can help developers partition their data and place those parts of their memory objects whose accesses cause a large number of CPU stall cycles in small but fast memory subsystems in heterogeneous memory systems. Branch predictors implemented in current mainstream and HPC processors help overcome the increased code branching introduced by our approach.

Acknowledgments

This material was based upon work supported by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357.

References

[1] S.L. Graham, P.B. Kessler, M.K. Mckusick, Gprof: a call graph execution profiler, a call graph execution profiler vol. 17 (6) (1982) 120–126.
[2] A. Srivastava, A. Eustace, ATOM: a system for building customized program analysis tools, ACM SIGPLAN Notices – Best of PLDI 1979–1999 vol. 39 (4) (2004) 528–539.
[3] A. Jaleel, R.S. Cohn, C.-K. Luk, B. Jacob, CMP$im: A Pin-based on-the-fly multi-core cache simulator, in: Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation, 2008, pp. 28–36.
[4] J. Weidendorfer, M. Kowarschik, C. Trinitis, A tool suite for simulation based analysis of memory access behavior, in: Proceedings of the Computational Science-ICCS, Springer, 2004, pp. 440–447.
[5] S. Browne, J. Dongarra, N. Garner, K. London, P. Mucci, A scalable cross-platform infrastructure for application performance tuning using hardware counters, in: Proceedings of the ACM/IEEE Supercomputing Conference (SC), 2000.
[6] A.C. de Melo, The new Linux 'perf' tools, in: Proceedings of the Linux Kongress, 2010.
[7] W.E. Cohen, Tuning programs with OProfile, Wide Open Mag. 1 (2004) 53–62.
[8] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, in: Proceedings of ACM Sigplan Notices, vol. 42, 2007.
[9] M. Itzkowitz, B.J. Wylie, C. Aoki, N. Kosche, Memory profiling using hardware counters, in: Proceedings of the ACM/IEEE Supercomputing Conference (SC), 2003.
[10] M. Martonosi, A. Gupta, T.E. Anderson, Tuning memory performance of sequential and parallel programs, Computer 28 (4) (1995) 32–40.
[11] T. Janjusic, K. Kavi, Gleipnir: A memory profiling and tracing tool, ACM SIGARCH Comput. Archit. News 41 (4) (2013) 8–12.
[12] H. Cho, B. Egger, J. Lee, H. Shin, Dynamic data scratchpad memory management for a memory subsystem with an MMU, in: Proceedings of the ACM SIGPLAN Notices, vol. 42, ACM, 2007, pp. 195–206.
[13] S. Phadke, S. Narayanasamy, MLP aware heterogeneous memory system, in: Proceedings of the Design, Automation Test in Europe Conference Exhibition, 2011.
[14] P.R. Panda, N.D. Dutt, A. Nicolau, On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems, ACM Trans. Des. Autom. Electron. Syst. 5 (3) (2000) 682–704.
[15] A.J. Peña, P. Balaji, Toward the efficient use of multiple explicitly managed memory subsystems, in: Proceedings of the IEEE Cluster, 2014.
[16] M. Verma, S. Steinke, P. Marwedel, Data partitioning for maximal scratchpad usage, in: Proceedings of the Asia and South Pacific Design Automation Conference, 2003.
[17] I. Issenin, E. Brockmeyer, M. Miranda, N. Dutt, DRDU: A data reuse analysis technique for efficient scratch-pad memory management, ACM Trans. Des. Autom. Electron. Syst. 12 (2) (2007) 1–28.
[18] J. Seward, N. Nethercote, Using Valgrind to detect undefined value errors with bit-precision, in: Proceedings of the USENIX Annual Technical Conference, 2005.
[19] A.J. Peña, P. Balaji, A framework for tracking memory accesses in scientific applications, in: Proceedings of the 43nd International Conference on Parallel Processing Workshops, 2014.
[20] M. Martonosi, A. Gupta, T. Anderson, MemSpy: Analyzing memory system bottlenecks in programs, ACM SIGMETRICS Perform. Eval. Rev. 20 (1) (1992) 1–12.
[21] J.R. Tramm, A.R. Siegel, XSBench — the development and verification of a performance abstraction for Monte Carlo reactor analysis, in: Proceedings of the PHYSOR 2014 — The Role of Reactor Physics toward a Sustainable Future, 2014.
[22] P.K. Romano, B. Forget, The OpenMC Monte Carlo particle transport code, Ann. Nucl. Energ. 51 (2013) 274–281.