A Framework for Tracking Memory Accesses in Scientific Applications

Antonio J. Peña
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
Email: apenya@anl.gov

Pavan Balaji
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
Email: balaji@anl.gov

***Abstract*—Profiling is of great assistance in understanding and optimizing applications' 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 pair of tools we have extended to complement the traditional algorithm-oriented analysis. Our extended tools provide 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. Memory objects found in this way may have gone unnoticed with the traditional profiling approach. This additional view may lead developers to think of different ways of storing data, leveraging different algorithms, or employing different memory subsystems in future heterogeneous memory systems.**

I. Introduction

Analyzing applications' 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 largely used to determine the root of unexpected performance issues, and it is highly relevant to the high-end computing community, where code is expected to be highly tuned.

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 has been 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; it does not provide any hint about the root of that time. For instance, a highly optimized, compute-intensive task may legitimately be spending a considerable amount of time, hiding other possible candidates for optimization.

A more informative approach consists of differentiating time spent in computation from that spent on memory accesses. In current multilevel memory hierarchies, cache hits pose virtually no penalty, whereas cache misses are likely to lead to large 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, since they enable the cache parameters to be easily changed and the analysis repeated for different architectures. A disadvantage of this profiling approach, however, is the large execution timings 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], [6], [7]. These are based mainly on sampling the hardware counters at a given frequency or deter-

mined 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 amounts of cache misses. However, it is not uncommon to perform accesses to different memory objects from the same line of code. For instance, a matrix-matrix multiplication implementation is very likely to access all the three memory buffers in the same line of code. In this case, a code-oriented profiler would not provide enough information to determine the root of the problem.

The approach pursued in this paper is intended to complement that view, differentiating those performance metrics by *memory object*[1] [8], [9]. The intent of this approach is to expose particular objects showing problematic access patterns throughout the execution lifetime, which may be imperceptible from a traditional approach. This view focuses on data objects rather than lines of code.

In this paper we present a pair of tools we developed on top of state-of-the art technologies. We incorporated per-object tracing capabilities into the widely used Valgrind instrumentation framework [10]. Next, we extended two of the tools from its large ecosystem to demonstrate the possibilities of this approach: Lackey [11] and Callgrind [4]. We describe the intrinsics of our developments with the hope that they prove useful for subsequent extensions and research, such as profilers and runtime systems targeting heterogeneous memory systems. Indeed, we envision memory-object differentiation to be a basic technique in these emerging systems as a means of determining the most appropriate memory subsystem on which to host the different memory objects according to their access pattern.

The rest of the paper is organized as follows. Section II introduces the Valgrind instrumentation framework as the basis for our developed tools. Sections III and IV describe our modifications to incorporate per-object differentiation capabilities into the Valgrind core and the two tools we target. Section V reviews related work, and Section VI provides concluding remarks and ideas for future work.

II. VALGRIND

Valgrind is a generic instrumentation framework. A set of *tools* making use of this generic core constitute the Valgrind ecosystem. Valgrind's most-used tool is *Memcheck* [12], a memory debugger, executed by default on Valgrind's invocation.

Valgrind can be seen as a virtual machine. It performs just-in-time compilation, translating the code to a processor-agnostic intermediate representation (IR), which the tools are free to analyze and modify.[2] 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 Figure 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 around 4x to 5x.

Valgrind exposes a rich API to its tools, plus a client request mechanism for final applications to interact with the tools if needed. The API enables tools to interact with the Valgrind core, for instance, to inform the core of the required capabilities at initialization time, get debug information about the executed code generated by the compiler (typically code compiled with the `-g` option), manage stack traces, or get information about the thread status. It

[1] We refer as "memory object" to every memory entity as seen from the code level, that is, statically and dynamically allocated variables and buffers.

[2] The tools do not directly interact with the IR itself but, rather, with a high-level application programming interface (API) and processed data structures to manage it.

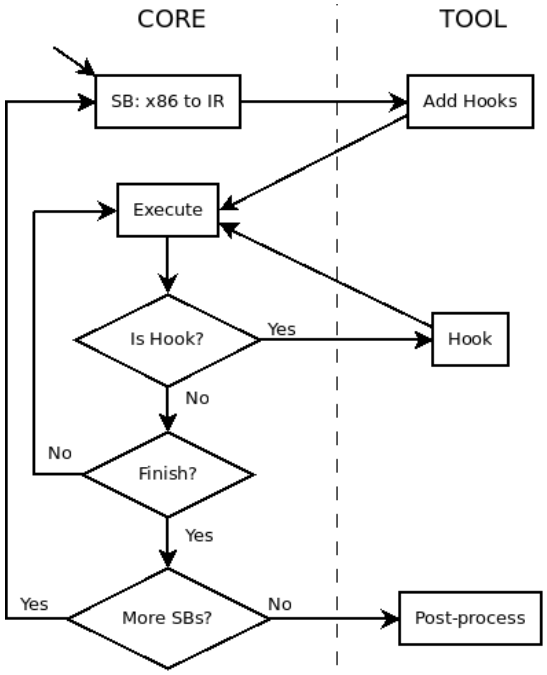


Fig. 1. Simplified high-level view of the interaction between Valgrind and its tools.

Listing 1. Sample output from Lackey.

```
I  04e9f363,5
 S 7fefff4b8,8
I  04f23660,3
I  04f23663,7
 L 05215e60,8
I  04f2366a,6
I  04f23670,6
I  04f23676,2
I  04f23691,3
I  04f23694,3
I  04f23697,2
==16084==
==16084== Counted 1 call to main()
==16084==
==16084== Jccs:
==16084==   total:         567,015
==16084==   taken:         312,383 ( 55%)
==16084==
==16084== Executed:
==16084==   SBs entered:   511,667
==16084==   SBs completed: 357,784
==16084==   guest instrs:  3,279,975
==16084==   IRStmts:       18,712,277
==16084==
==16084== Ratios:
==16084==   guest instrs : SB entered  = 64 : 10
==16084==        IRStmts : SB entered  = 365 : 10
==16084==        IRStmts : guest instr = 57 : 10
==16084==
==16084== Exit code:       0
```

also provides a set of high-level containers such as a arrays, sets, or hash tables. In addition, it facilitates the possibility of intercepting the different memory allocation calls to enable tools to provide specific wrappers for them. The client request mechanism enables capabilities such as starting and stopping the tool instrumentation on demand to save execution time on uninteresting portions of code.

In the following we introduce the two tools we have extended: *Lackey* and *Callgrind*.

A. Lackey

Lackey is an example tool designed to show Valgrind's capabilities and its interaction with tools. Its main functionality is to provide basic statistics about the executed code, such as number of calls to user-specified functions, number of conditional branches, amount of superblocks, and number of guest instructions. It can also provide counts of load, store, and ALU (arithmetic logic unit) operations. Furthermore, it offers the possibility of memory tracing, that is, printing the size and address of (almost) every memory access made during the execution. We will focus on this last capability. Listing 1 shows an excerpt of a sample output from this tool with the memory tracing functionality enabled.

Our extensions enable Lackey to differentiate memory accesses to different objects, which will provide application developers and hardware designers with information about raw access patterns to the different memory objects.

B. Callgrind

Callgrind is autodefined as "a call-graph generating cache and branch prediction profiler" and is intended to be a profiling tool. By default, it collects the number of instructions, the number of function calls, and the caller-callee relationship among function calls (the call graph). All this data is related to their source lines of code. If the cache simulation is enabled, cache misses can be used as a performance metric. By default, it 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 separate tools based on hardware counters, it is still a useful profiling tool: the fact that the simulated

cache does not behave exactly as the physical one 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 Callgrind's features include a branch predictor and a hardware prefetcher. The call-graph functionality differentiates this tool from Cachegrind [13], from which it borrows some code.

After Callgrind has profiled an application, one can use KCachegrind as a postprocessing tool that enables the graphic visualization of the collected data. It loads an output file from Callgrind and displays the collected counters in an interactive way, enabling the user to navigate through this information. Figure 2 shows a snapshot from the visualization tool for a sample run of the MiniMD [14] code.[3]

We have extended this profiler to differentiate per-object cache misses. We have also fully integrated our developments with KCachegrind in order to provide the information graphically.

III. Core Extensions

Our extensions are based on the development branch of Valgrind 3.10.0. In this section we introduce the integration of the new functionality not specifically related to a particular tool. Next section will cover the way we extended two Valgrind tools, Lackey and Callgrind, to make use of these new capabilities.

Our modifications are focused on enabling the differentiation of memory objects. For that, we incorporate functionality to locate the memory object comprising a given memory address, and store its associated access data. For our purpose, we differentiate two types of memory objects: statically and dynamically allocated. These require two different approaches as explained below.

A. Statically-allocated Memory Objects

Our changes concentrate in the debug information functionality that Valgrind exposes to its tools. Hence, this capability requires applications to be compiled with embedded debug information (usually by means of the `-g` compiler option). We developed a new function to locate and record a variable access (if found), along with auxiliary functions to handle the debug information objects to be considered or ignored, get the debug information from a variable handler, retrieve or print the gathered access information, trace a user-defined set of variables only, and associate tool-defined identifiers to the variable objects. In addition, we extended the variable data structure with two triplets of 64-bit counters to represent the load, store, and modify information relative to the number of accesses and bytes accessed. The "modify" counters are exposed for performance purposes in case the tool performs that distinction. Fields to store whether the variable has been explicitly requested to be traced and a user identifier are also incorporated.

Listing 2. Example of scopes.

```
// Scope #1
// i1 is valid
int i1;
{
  // Scope #2
  // i1 and i2 are valid
  int i2;
}
// Scope #1
// Only i1 is valid again
```

The information about the statically allocated variables is distributed among the different binary objects comprising 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 on which the different statically-allocated variables may or may not be exposed depending on the current program counter (see an example in Listing 2). Note that the address of each variable is only defined when its scope is active.

We follow the algorithm already employed by Valgrind to locate statically-allocated variables, leveraged for instance by Memcheck to inform of the location of an illegal access. This algorithm (see pseudocode in Listing 3) performs the following steps:

[3]MiniMD is a reduced version of the LAMMPS molecular dynamics simulator [15], [16], [17].

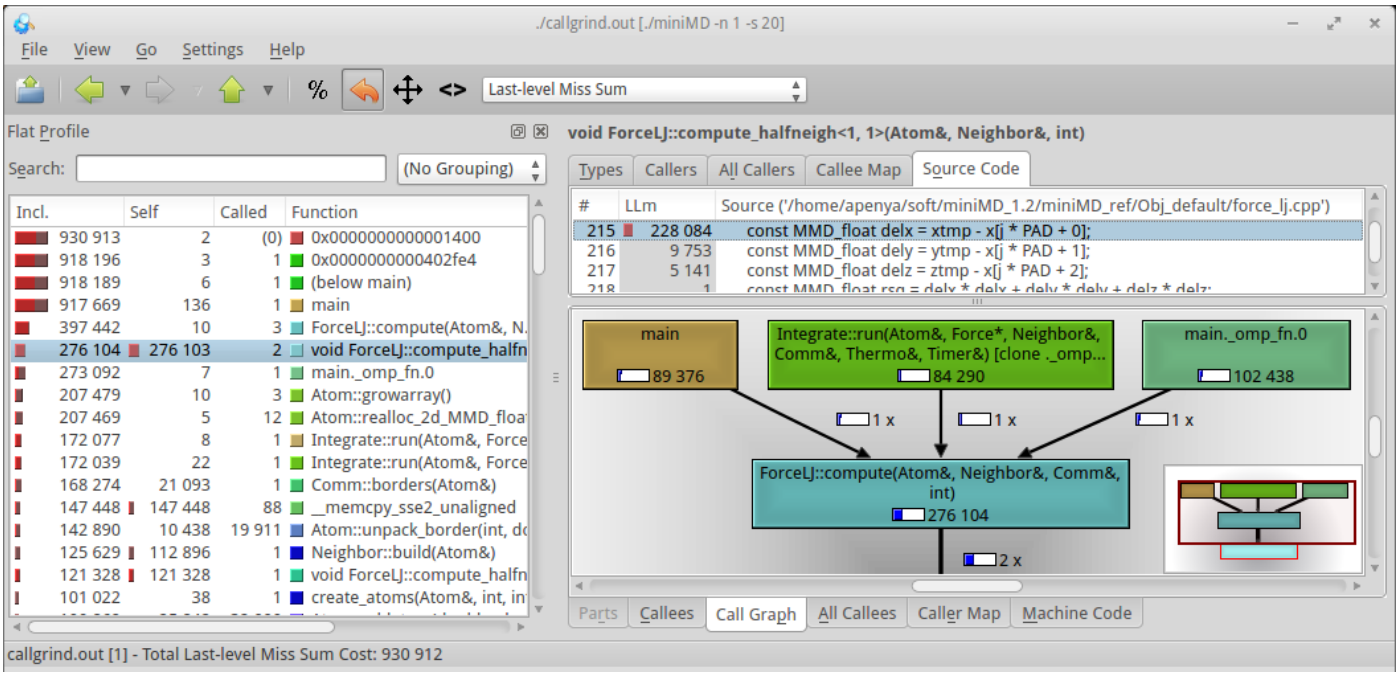


Fig. 2. Snapshot of KCachegrind for a simple run of MiniMD.

Listing 3. Algorithm to find statically-allocated variables.

```
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_variables(instruction_ptr)
        for var in vars:
          if address in var:
            return var.user_id
    debug_info = debug_info.next
return MEM_OBJ_NOT_FOUND
```

1) Iterate through the stack trace and get the instruction pointer (IP).
2) Traverse the debug information object list to find that corresponding to IP.
3) Optimization: bring the found debug information object to the front of the list.
4) Iterate through the debug information scopes, working outwards.
5) Extract the set of variables in the current scope (valid for the current program counter on the current stack trace level).
6) Iterate over them until we find one comprising the target address.

The asymptotic computational cost of this algorithm is:

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

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.

Note that this distribution of the variables—derived from the the DWARF format [18] designed mainly for debuggers—permits performing a search through the statically-allocated variables defined for the current IP only. On the other hand, the address on which a variable is defined is computed on demand following Valgrind's original approach, precluding from a binary search within the scope. In future work we plan to explore the viability of precomputing the addresses of the different variables of a scope every time they become active in order to enable scope-wise binary searches.

As this process is time consuming, and users are likely to focus primarily on its applicationl variables—not including external libraries, by default this new functionality will only consider the debug information comprised on the object containing the main entry point (that is, the executable object). Tools can fully control the debug information

objects to be considered.

B. Dynamically-allocated Memory Objects

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

The information about the dynamically-allocated objects is kept within an ordered set using the starting memory address of the memory objects as the sorting index. This 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 objects, similar to that described in [8]. 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 (see Listing 4), or an object being repeatedly created and destroyed when entering and leaving a function (Listing 5). 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.

Listing 4. Example of objects eligible to have their accesses merged: matrix creation.

```
float *vector = (float *) malloc(HEIGHT);
for(int i=0; i<n; i++) {
   vector[i] = malloc(WIDTH);
}
```

Listing 5. Example of objects eligible to have their accesses merged: temporary buffers.

```
void auxiliary_function(size_t size) {
   int *tmp_buf = (int *) malloc(size);
   // ...
   free(tmp_buf);
}

int main(void) {
   size_t size;
   for(int i=0; i<N; i++) {
      // ...
      auxiliary_function(size);
      // ...
   }
}
```

IV. Extending Valgrind Tools

This section covers the extensions we made to the Lackey and Callgrind tools from the Valgrind ecosystem, as well as sample use cases.

A. Lackey

As introduced above, Lackey instruments the code to get different statistics about the executed application. Besides, it provides memory access tracing capabilities which report the type (instruction, store, load, or modify), address, and size of the access (see Listing 1). We focus on this last feature, extending it to identify the memory object accessed.

We modified the memory tracing feature to make use of the functionality described in Section III. Following the observation that applications tend to feature far lesser dynamically-allocated objects than their statically-allocated counterpart, and that they tend to cover a much larger amount of memory space, we first perform the search in the dynamic set of objects for performance purposes.

Listing 6 shows an excerpt from the output of our extended tool. For each access to a known memory object, it specifies its type (G: global, L: local, D: dynamic). In the static case (G or L), the name of the variable, the offset from the beginning of the buffer, and the name and line of the file defining

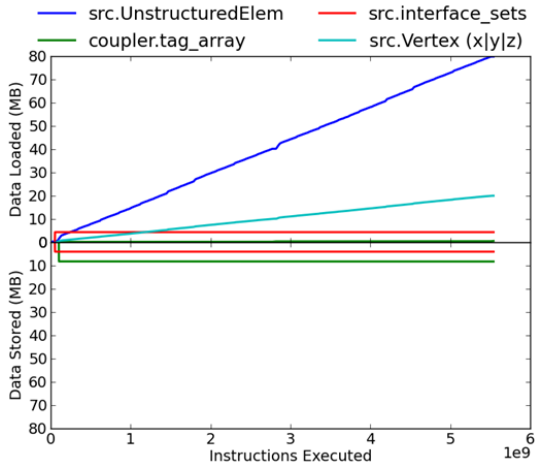
Listing 6. Sample memory trace from our extended Lackey.

```
I  0040b532,3
 S ffeffef20,8  L  timer+0  [ljs.cpp:271]
I  0040b535,4
 S 057acd10,8  D  +0  1392
I  0040b539,5
 S 057acd18,8  D  +8  1392
I  0040b53e,5
 S 057acd20,8  D  +16  1392
I  0040b543,5
 S 057acd28,8  D  +24  1392
I  0040b548,5
 S 057acd30,8  D  +32  1392
```

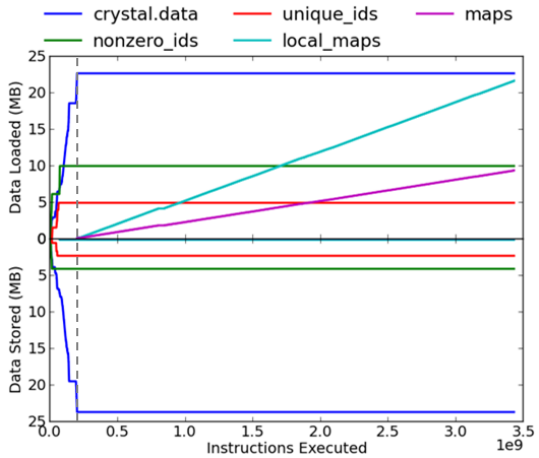
it are printed. For the dynamic case, an execution context identifier is provided. Listing 7 shows the final summary from our tool, including the totals per object as well as the grand totals. The section for dynamically-allocated objects contains the execution context identifier used in the memory trace. This output can be sorted by any column as requested by the user.

The output showed in listing 6 is intended to be visualized for (very) short executions. To facilitate post-processing analysis, we also provide CSV-formatted output. This output additionally includes the instruction counter and the execution context identifier of the line of code of each memory access. It does not include instruction accesses, as they are out of the scope of our extensions. Another useful capability we included is the option to annotate the output from the user source (using Valgrind's client request capabilities), to facilitate the identification of different parts of the code of interest to users.

To demonstrate the possibilities of this tool, we have developed a simple script to analyze and plot the CSV output from our extended Lackey tool. Figure 3 shows a pair of examples of this analysis from the CESAR mini-apps [19]. These are from short runs of CIAN (CESAR Integrated Analytics) in Figure 3a and Nekbone in Figure 3b. In this case we selected the most used dynamically-allocated variables from the summary of the accesses. In addition, we included an annotation in the Nekbone case to plot a vertical line to identify a point of interest in the graph.



(a) CIAN Coupling raw access pattern to the four most accessed memory buffers.



(b) Nekbone raw access pattern to the five most accessed memory buffers.

Fig. 3. Example of raw access patterns from the extended Lackey tool.

B. Callgrind

In Callgrind we followed a similar approach as in Lackey to identify the accesses to the different memory objects. This tool helps on targeting cache misses, that is, accesses to the main memory rather than raw accesses. With these capabilities, users are able to explore the per-object cache misses. We included the same tracing capabilities as in Lackey, enabling the analysis of the per-object cache misses along the execution of the program. Similar graphs to those shown in Figure 3 can be obtained from our extended Callgrind, although comprising main memory accesses (cache misses) rather than raw accesses. In addition, we included support for integration with the KCachegrind visualization tool, by making use of the existing infrastructure. Figure 4 shows a snapshot of a KCachegrind output for

Listing 7. Excerpt of a summary of an execution by our extended Lackey.

```
==10322== 1.- Access to Statically-Allocated Memory
==10322==
==10322==   Loads  Stores  Accesses  Load Bytes  Store Bytes  Total Bytes  T   Size  Location
==10322== ------------------------------------------------------------------------------------------------
==10322==
==10322== -- /home/user/soft/miniMD_1.2/miniMD_ref/miniMD --
==10322==       0       1         1           0            8            8  L      8  argv  [ljs.cpp:72]
==10322==       2       0         2          16            0           16  L      8  comm  [integrate.cpp:71]
                        --- Some contents skipped from this sample excerpt ---
==10322== ------------------------------------------------------------------------------------------------
==10322==  21,033   1,148    22,181  126,080    5,445    131,525  T      3,016
==10322==
==10322== 2.- Access to Dynamically-Allocated Memory
==10322==
==10322==   Loads  Stores  Accesses  Load Bytes  Store Bytes  Total Bytes  T       Size  EC
==10322== ------------------------------------------------------------------------------------------------
==10322==
==10322==   5,400   5,400    10,800      43,200       43,200       86,400  D     16,000  ECU #1384
==10322==                malloc  [ buf_send = (MMD_float*) malloc((maxsend + BUFMIN) * sizeof(MMD_float)); ]
==10322==                Comm::Comm()  [/home/user/soft/miniMD_1.2/miniMD_ref/Obj_default/comm.cpp:47]
==10322==                main  [/home/user/soft/miniMD_1.2/miniMD_ref/Obj_default/ljs.cpp:270]
==10322==
==10322==      18      15        33         144          120          264  D         40  ECU #1392
==10322==                malloc  [ array = (double*) malloc(TIME_N * sizeof(double)); ]
==10322==                Timer::Timer()  [/home/user/soft/miniMD_1.2/miniMD_ref/Obj_default/timer.cpp:38]
==10322==                main  [/home/user/soft/miniMD_1.2/miniMD_ref/Obj_default/ljs.cpp:271]
                        --- Some contents skipped from this sample excerpt ---
==10322== ------------------------------------------------------------------------------------------------
==10322==  34,370  95,659   130,029     248,772      771,408    1,020,180  T  2,333,792
==10322==
==10322== 3.- Grand Totals
==10322==
==10322==   Loads  Stores  Accesses  Load Bytes  Store Bytes  Total Bytes          Size
==10322== ------------------------------------------------------------------------------------------------
==10322==
==10322==  55,403  96,807   152,210     374,852      776,853    1,151,705     2,336,808
```

MiniMD from our extended Callgrind, displaying the read cache misses for the memory object with execution context identifier 1636.

V. Related Work

A previous study on hardware-assisted object-differentiated profiling [9] extends the Sun ONE Studio compilers and performance tools to offer object-differentiated profiling based on hardware counters. While this technique features low overheads, it does not provide the flexibility of the software solutions based on system emulators. In addition, because of the intrinsics of the hardware architectures, its granularity is limited.

MemSpy [8], [20] was an early "prototype tool" providing object-differentiated profiling. It implemented a technique similar to ours to merge accesses to different memory objects. It was based on the Tango [21] system simulator. To the best of our knowledge, this tool was never made publicly available, and both the Tango and MemSpy projects were discontinued. An interesting feature of this tool is the simulation of the memory subsystem, enabling the possibility of using the processor stall cycles as a performance metric.

In our research we attempt to revive the work started by these previous approaches by providing object differentiation capabilities to a state-of-the-art framework: Valgrind. We aim our tools to be useful for today's application profiling needs and the basis for advanced functionality and research (see Section VI).

VI. Conclusions and Future Work

In this paper we have presented a pair of tools providing object-differentiated analysis based on a state-of-the-art and widely used technology: the Valgrind instrumentation framework. We have described our design and the implementation details, and we have illustrated its use from two of the tools

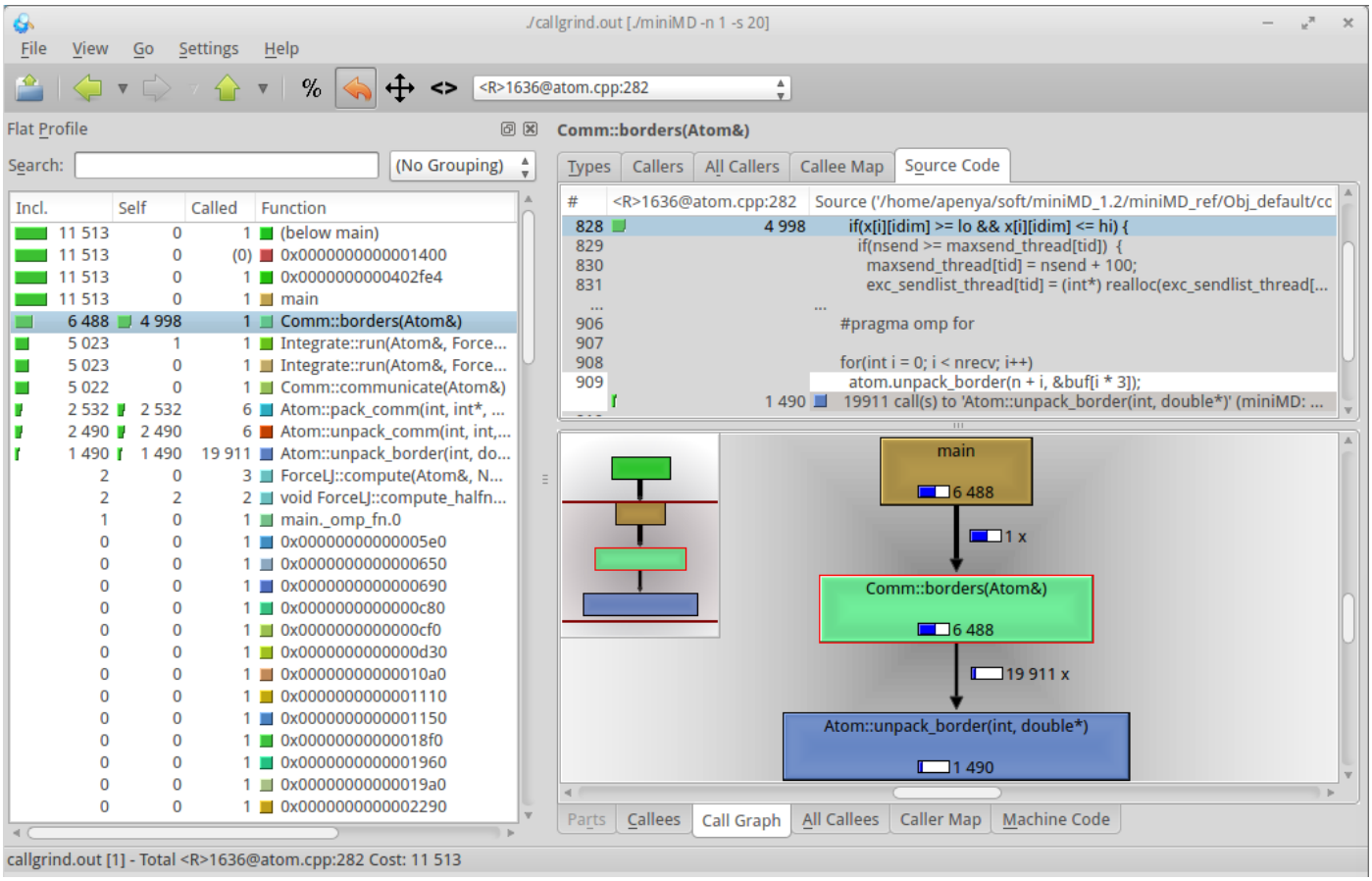


Fig. 4. Snapshot of the Kcachegrind tool with extended per-object information.

of the Valgrind ecosystem—Lackey and Callgrind—that we extended for this purpose. Our extensions to Lackey enable the analysis of the raw access patterns to the different objects, which can be useful for detecting unexpected accesses. The per-object capabilities we incorporated into Callgrind provide an additional profiling view to developers, by exposing memory objects presenting consistently troublesome access patterns, which may have been hidden in traditional profiling approaches; and this additional view may help in designing different algorithmic approaches.

In future work we will explore the possibility of adding memory simulators such as DRAMSim2 [22] to our framework, to enable the measurement of processor stall cycles. Our goal is to use the extended tools along with the memory simulator in the study of heterogeneous memory systems, in order to determine the best memory subsystem in which to place the different objects according to their out-of-die access patterns.

Acknowledgments

This work is part of the "System Software for Scalable Applications" PRAC allocation support by the National Science Foundation (award number OCI-1036216). This work was also supported in part by the U.S. Dept. of Energy under contract DE-AC02-06CH11357.

References

[1] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *ACM Sigplan Notices*, vol. 17, no. 6, pp. 120–126, Jun. 1982.

[2] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," *ACM SIGPLAN Notices – Best of PLDI 1979–1999*, vol. 39, no. 4, pp. 528–539, Apr. 2004.

[3] A. Jaleel, R. S. Cohn, C.-K. Luk, and 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 (MoBS)*, 2008, pp. 28–36.

[4] J. Weidendorfer, M. Kowarschik, and C. Trinitis, "A tool suite for simulation based analysis of memory access behavior," in *Computational Science-ICCS 2004*. Springer, 2004, pp. 440–447.

[5] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A scalable cross-platform infrastructure for application performance tuning using hardware counters," in *Supercomputing, ACM/IEEE 2000 Conference*, Nov 2000, pp. 1–42.

[6] A. C. de Melo, "The new Linux 'perf' tools," in *Linux Kongress*, 2010.

[7] W. E. Cohen, "Tuning programs with OProfile," *Wide Open Magazine*, vol. 1, pp. 53–62, 2004.

[8] M. Martonosi, A. Gupta, and T. Anderson, "MemSpy: Analyzing memory system bottlenecks in programs," *ACM SIGMETRICS Performance Evaluation Review*, vol. 20, no. 1, pp. 1–12, 1992.

[9] M. Itzkowitz, B. J. Wylie, C. Aoki, and N. Kosche, "Memory profiling using hardware counters," in *Supercomputing, 2003 ACM/IEEE Conference*. IEEE, 2003, pp. 1–13.

[10] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan Notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[11] Valgrind™ Developers, "Lackey: an example tool," http://valgrind.org/docs/manual/lk-manual.html, 2013.

[12] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision." in *USENIX Annual Technical Conference, General Track*, 2005, pp. 17–30.

[13] N. Nethercote, "Dynamic binary analysis and instrumentation," Ph.D. dissertation, University of Cambridge, 2004.

[14] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep., Sep. 2009, http://www.sandia.gov/~maherou/docs/MantevoOverview.pdf.

[15] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1–19, 1995.

[16] S. Plimpton, R. Pollock, and M. Stevens, "Particle-mesh ewald and rRESPA for parallel molecular dynamics simulations," in *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[17] Sandia National Laboratories, "LAMMPS molecular dynamics simulator," http://lammps.sandia.gov, 2014.

[18] M. J. Eager, "Introduction to the DWARF debugging format," http://www.dwarfstd.org/doc/Debugging using DWARF.pdf, 2007.

[19] Argonne National Laboratory, "Center for exascale simulation of advanced reactors," http://cesar.anl.gov, 2014.

[20] M. Martonosi, A. Gupta, and T. E. Anderson, "Tuning memory performance of sequential and parallel programs," *Computer*, vol. 28, no. 4, pp. 32–40, 1995.

[21] H. Davis, S. R. Goldschmidt, and J. L. Hennessy, "Tango: a multiprocessor simulation and tracing system," in *Proceedings of the International Conference on Parallel Processing*, Aug. 1991, pp. 99–107.

[22] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, jan.-june 2011.