View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

Procedia Computer Science
Volume 80, 2016, Pages 1439–1449
ICCS 2016. The International Conference on Computational Science

Inclusive Cost Attribution for Cache Use Profiling

Josef Weidendorfer and Jens Breitbart

Department of Informatics, Technical University of Munich, Germany
{Josef.Weidendorfer|j.breitbart}@tum.de

Abstract

For performance analysis tools to be useful, they need to show the relation of detected bottlenecks to source code. To this end, it often makes sense to use the instruction triggering a problematic event. However for cache line utilization, information on usage is only available at eviction time, but may be better attributed to the instruction which loaded the line. Such attribution is impossible with current processor hardware. Callgrind, a cache simulator part of the open-source Valgrind tool, can do this. However, it only provides *Self Costs*. In this paper, we extend the cost attribution of cache use metrics to inclusive costs which helps for top-down analysis of complex workloads. The technique can be used for all event types where collected metrics should to be attributed to instructions executing earlier in a program run to be useful.

Keywords: Performance Analysis, Cache Simulation, Cost Attribution

1 Introduction

If programmers want to tune their applications, performance analysis tools are crucial for understanding the sources of bottlenecks. The usefulness of such tools heavily depends on their ability to correctly relate events, which are responsible for a given bottleneck, to the source code. While it still may be difficult to find the right modifications reducing the bottleneck, the correct relation as well as an estimation of how much performance can be improved is essential to not waste time on irrelevant optimizations. Due to the large gap between processor and memory performance, it nowadays is essential for good performance to exploit caches. Depending on the application, it may be possible to improve the performance of an application by a factor of 10 just by improving access locality by changing the order in which memory is accessed, or by changing the layout of data structures. More concretely, caches are exploited well when temporal and spatial locality of accesses is high. A metric for the first is the average number of times a cache lines was accessed while residing in cache, and a metric for the latter (in regard to current block-based cache designs) is the number of bytes accessed within a line before it got evicted.

Performance counters in current processors provide cache miss counters as well as information about the memory access which triggered a cache miss. Together with the number of

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

© The Authors. Published by Elsevier B.V.

doi:10.1016/j.procs.2016.05.464

accesses done by the program, we can calculate the above metric for temporal locality. However, relation of low locality to source is difficult. Let us assume two data structures frequently accessed in an interleaved manner, one larger than the cache and accessed in a streaming fashion, the other easily fitting into cache. As the streamed access will evict both data, cache misses happen on both accesses to the high and the low locality data structure. Thus, the cache miss numbers with their relation to source lines is misleading as only the streaming accesses show the bad behavior. If we instead measure a locality metric per cache line, and relate that to the instruction which loaded the line into cache, the tool would show bad locality only for accesses to the stream in our example (touching data of cache lines only once before eviction). While such an attribution technique cannot be implemented with hardware measurement, Callgrind [16] — a cache simulator based on the runtime instrumentation framework Valgrind [13] — provides this feature when switching on the optional *cache use analysis* [17]. It uses the locality metrics given above. However, as we want to highlight bad behavior, low locality should show up on top. Thus, the metrics actually used are the reciprocal of the access count before eviction, and the number of unused bytes at eviction time.

Just pin-pointing at the source lines with bad performance often is not good enough in complex codes, especially when this happens in library functions which cannot be modified, or when the functions themselves are so small that modifications make no sense. In this case, for an analysis tool to be able to tell about the call chains to the bad performing functions often is essential for a tools' usefullness. For example, functions up the call chain may be called unnecessary often, or the order of calls may be changed to result in better cache behavior. To be able to spot the call chains relevant for performance, *inclusive cost* attribution is used: whenever a metric is attributed to an instruction in a function, this metric also is propagated up the call chain as inclusive cost of functions. This way, `main` gets 100% of all metrics collected as its inclusive cost. Users of the profiling tool are guided by inclusive costs in a top-down fashion via the relevant call chains. They can easily spot where modifications are needed/useful, that is, on the relevant call chains involving their own written functions before diving into 3rd-party library code.

While the above mentioned cache use metric in Callgrind is very useful for analysis, it does not provide inclusive costs, rendering the metrics difficult to use in complex codes. In this paper, we extend the collection technique for inclusive costs used in Callgrind to include metrics which are only available long after an instruction was executed that should get the metrics attributed. We call these *back-dating metrics* in contrast to *immediate metrics* which directly are available at the same time the instruction is executed which should get the metrics attributed. Cache use metrics are examples of back-dating metrics while FLOP count, load/store operations or cache misses are immediate metrics. We show the resulting overhead of cache use analysis with our new attribution technique, both in simulator runtime and memory overhead.

The paper is structured as follows: in the next section, we explain the current collection technique in Callgrind and show the difficulty of getting inclusive costs of cache use metrics. In Sect. 3, we describe our novel strategy. In Sect. 4, overhead numbers are given, and before conclusion, we provide some related work.

2 Inclusive Cost Collection in Callgrind

Callgrind uses instrumentation of binary code at runtime. Before a piece of machine code is executed for its first time, an instrumentation pass injects measurement code into the original code. This instrumented version is executed instead of the original code and is stored into a so-called *code cache*. Any further request for executing the original code will be redirected to the previ-

ously instrumented version in the code cache. Re-instrumentation may happen on eviction as the code cache has limited size. The technique is very similar to *Just-In-Time Compilers*, which translate higher-level code piece-wise (often already pre-compiled into architecture-independent byte code) into machine code directly before it is to be executed for the first time. The instrumentation pass detects instructions accessing memory, function calls and returns. For each instruction which potentially has a cost, Callgrind allocates space for corresponding counters. For example, memory load instructions get counters for the number of loads and the number of cache misses for each cache level of the simulated processor model. Further, the instrumentation pass injects calls into handlers which then forward runtime events to the cache simulator or maintain a stack of currently active calls. At every point in time, Callgrind knows the exact call chain from `main`[1]. We note that there is a separate call stack for every thread running as part of the application. Inclusive costs are not maintained per function, but per call arc, ie. per $(caller_i, callee)$ tuple. Here, a $caller_i$ is an instruction address of a call instruction inside a function *caller* and *callee* is the called function. Whenever a specific call arc is traversed for the first time within a thread, a call-arc object including space for the inclusive costs "below this call arc" is allocated. The entries of the call stack maintained by Callgrind actually consist mainly of pointers to such call arc objects. The structure for a call arc object is called `JCC` for "jump cost center" [2].

In the introduction, we already provided a rough idea of how inclusive costs can be collected. We want to aggregate event costs such as the number of last level cache misses. Whenever an event (here a last level cache miss) happens, we increment (adding a cost of "1" for this event) a cost counter keyed by the instruction address of the instruction whose execution triggered the event[3]. Such cost counters aggregate *Self costs*, that is, costs triggered by the execution of the instruction. The distinction to inclusive cost gets clearer on the function level: self costs of a function are the sum of all self costs of instructions belonging to this function, while inclusive costs of a function are the self cost of the functin *plus* the sum of inclusive costs of called functions (a recursive definition).

For events such as floating point operations, execution of load/store actions, memory accesses, or cache misses, the cost is directly available when the instruction triggering the cost is executed (i.e. *immediate metrics* according to the definition in the introduction). Thus, to update any inclusive costs of functions for the cost stemming from execution of an instruction, we have to maintain the current call chain down from `main` to the function which includes the instruction executed and propagate the new generated cost to all inclusive cost counters up the call chain. We can do this directly after updating the self cost counter.

However, this strategy is slow. The length of the call chain from `main` (that is, the call stack maintained by Callgrind) down to the currently executed function can be long. But traversing this chain whenever an event is happening actually is unnecessary. Instead, we can create a new counter for each call into a function, aggregating all self cost within this function. When exiting the function, we propagate these aggregated self costs up the call chain. The cost counter per function invocation can be part of an entry of the call stack. We call this strategy "inclusive cost aggregation method 1".

We can refine this method: there is no need to traverse the full call chain whenever a function exit happens. Instead, we only update the inclusive cost for the call arc we return

[1] Actually not `main` but the function that is started with the first instruction executed in a binary, including runtime linker and shared library initializations.

[2] Actually, *caller* and *callee* which identify call arc objects can be more than functions. They may represent more detailed calling contexts identifying call chains up to a given length themselves. However, this is not relevant to the discussion in this paper.

[3] The cost counter further is keyed by the current thread number.

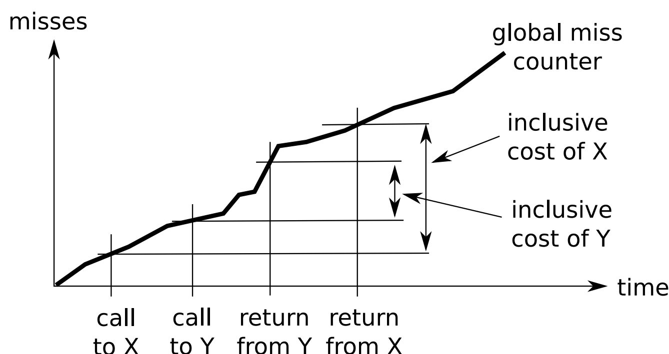

Figure 1: Inclusive cost aggregation method 3. To get inclusive cost for a function invocation, we take the difference of values from a global cost counter at start and end.

from. Further, we add the cost counter of the called function to the cost counter of the calling function on return. This results in automatic propagation of inclusive costs at function exits. We note that in this lazy scheme the inclusive costs are only correct at program termination. We call this strategy “inclusive cost aggregation method 2”.

There is a much simpler way. We do not even have to propagate inclusive costs. Instead, we use a thread-global cost counter which is incremented whenever a self cost counter is incremented. When a function is entered, we remember the value of this global counter, and push both a pointer to the corresponding call-arc object and this value on our call stack. On returning from the function, we calculate the difference of the current value of the global counter and the remembered value. This gives us the inclusive cost of the function invocation, consisting of all events which happened between entering and leaving the function. It is enough to add this difference to the inclusive cost counter of the call arc object which represents the call to the function. This is shown in Figure 1. Against a time axis, a monotonically increasing global cost counter for misses is shown. We show calls and returns of functions X and Y with Y being called nested from X. It is obvious that the inclusive cost of X can be determined without requiring the inclusive cost of Y. We call this strategy “inclusive cost aggregation method 3”. This actually is used in Callgrind.

The Difficulty with Back-Dating Metrics

Both collection methods 2 and 3 only work with immediate metrics. In regard to method 2, the cost counter for the invocation of a function may already be destroyed and propagated at the time a metric is generated for an instruction which executed as part of that function invocation. Regarding method 3, we could have a global counter for back-dating metrics. But we will get increments of this counter for instructions which were executed in the past, mixing up the increments for different functions. There is no way to get back meaningful inclusive costs per function.

Method 1 actually would work, but is hopeless inefficient. Further, as the current call chain from `main` at instruction execution time probably is different from the call chain at metrics generation for this instruction, we have to remember the set of active functions in the call chain for each instruction executed, as we need this set of functions for adding the cost when the metric becomes available.

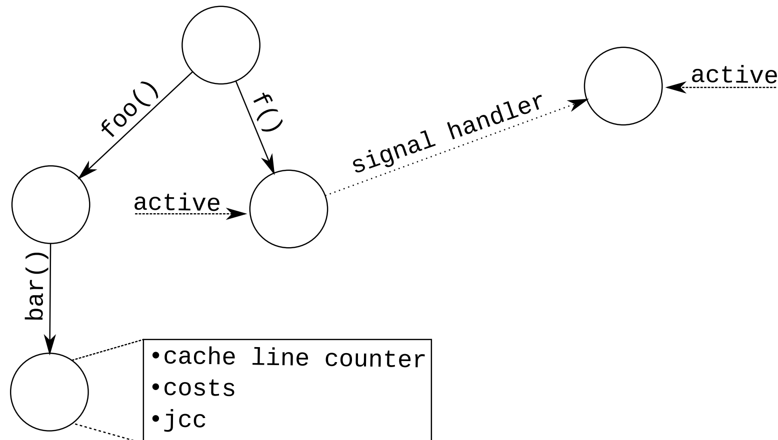

Figure 2: Snapshot of the back-dating tree of one thread with one signal handler running. The signal handler starts a new isolated back-dating tree, resulting in two active nodes for this scenario. Active nodes are not pruned.

3 Inclusive Cost for Back-Dating Metrics

In this section, we describe the strategy for inclusive cost aggregation for back-dating metrics, based on the quite inefficient idea from the last section. We need to remember the state of the call chain from `main` for all instructions where we may have metrics generated afterwards. With the cache use metrics as example, this means that we need to remember the chain of call arcs for every instruction which loaded a cache line that is not evicted yet.

We use a reference counted tree to store all relevant parts of the call chain, being a partial call tree. A node represents a function that was called from a specific other function within the observed program. Thus, each node can be related to an call arc object (`JCC`), and actually has a reference to the corresponding `JCC`. Nodes exist at least as long as there are cache lines residing in the cache simulator which got loaded from instructions in the corresponding function. Such cache lines get associated with corresponding nodes. Each node has cost counters which lazily collect metrics from associated cache lines as they get evicted. Only leafs can be pruned. Then, the cost counters are propagated to the parent node as well as to the cost counters of the related `JCC`. Each thread (and each signal handler within a thread) has its own tree. In each tree, one node is marked as being the *active* node reflecting the currently executing function within this thread (or signal handler). Active nodes are not allowed to be cleaned up. In alignment with the purpose of supporting the back-dating of metrics, we call such a tree in the following a *back-dating tree*.

Our back-dating tree is constructed by adding a new node to our tree as child of the current active node (if not already existing). The new node becomes the active node. For every return observed, we go back to the parent and set it to be the active node[4]. For every cache line that is loaded by an instruction, we increase a reference counter within the active node by one. The cache simulator associates the cache lines by storing a pointer to that tree node. Whenever a cache line is evicted, we accumulate the metric collected for this cache line to the cost counters in the tree node and reduce the reference counter by one.

[4]Actually, the implementation is more complex, as the simulator and call/return observation can be turned on and off during the runtime of the simulation, which requires us to be able to deal with 'missing' parents.

Once the reference counter reaches zero, i.e. all cache lines loaded during the execution of that function have been evicted, the node becomes a candidate for pruning. However, only leafs are actually pruned and must not be an active node. If the node is to be pruned, the aggregated costs are both propagated to the parent node and the related JCC, and the node is deleted. This algorithm is applied recursively. That is, we check the parent of the deleted node. If that is not the active node, it is a leaf node, and no cache lines are associated with it, we prune it as well.

Signals and signal handlers have to be handled carefully. As they are called asynchronously to program execution, Callgrind makes sure they do not show up as being called from the function currently running. Instead, called signal handlers get new roots of isolated call graphs. To this end, whenever a signal handler is called, we also need to create a new isolated root node of a new back-dating tree. Thus, we maintain potentially multiple trees per thread, as even within a signal handler, another signal handler may be called.

Figure 2 shows a snapshot of a back-dating tree of one thread with a signal handler currently running.

In theory the tree can increase (almost) infinite, but our practical results suggest the size of the tree is within the order of the largest cache size currently simulated for most applications. Compared to method 1 the back-dating tree provides an efficient way to compute the inclusive costs without the need to traverse the call chain.

4 Example and Overhead Evaluation

HYDRO is an application proxy benchmark that is been used to benchmark European Tier-0 HPC systems. HYDRO serves as a proxy for RAMSES[5] [15], which is a Computational Fluid Dynamics application developed by the astrophysics division in CEA Saclay. HYDRO contains all performance relevant algorithms and communication patterns of the original application, but it is simplified and trimmed down to only about 1500 lines of code (compared to about 150,000 lines of code of the original RAMSES). Subsequently, HYDRO was ported to various programming languages and parallel programming models including Fortran, C / C++, OpenMP, MPI, hybrid MPI/OpenMP, CUDA, OpenCL and OpenACC [10]. Our experiments are based on the OpenMP C99 implementation. Hydro is available at GitHub[6].

In the following, we used the input configuration `input_500x500_center.nml` provided with HYDRO but set the number of iterations to 10. While our inclusive collection technique works well with multi-threaded code, there is no benefit to use more than 1 thread as Valgrind (and thus, Callgrind) does not run multiple threads simultaneously. A single MPI task is used. The code was compiled with GCC 5.2.1 with `-O3 -march=corei7-avx` and OpenMP enabled on Ubuntu 15.10. The runtime numbers are measured on an Intel Core i7-3740QM CPU with nominal clock frequency of 2.7 GHz. This CPU has four cores with Hyperthreading and Turbo-boost enabled. Cache sizes of core-private caches are 32 kB (both L1 data and instruction cache) and 256 kB for L2. The shared L3 cache is 6 MB. Associativity of L1/L2 is 8 and 12 for L3 cache.

The 10 iterations of this configuration natively take 1.12s to run. Callgrind (from Valgrind 3.11.0) with cache simulation requires 110 seconds, showing a slowdown of 98x. Switching on cache use analysis with the original Callgrind without inclusive cost collection requires 154 seconds, adding 40% to the simulation with resulting slowdown of 137x. This is expected, as for every access to a cache line, we have to update an access counter and a byte usage mask. The

[5] http://www.itp.uzh.ch/~teyssier/ramses/RAMSES.html
[6] https://github.com/jbreitbart/Hydro

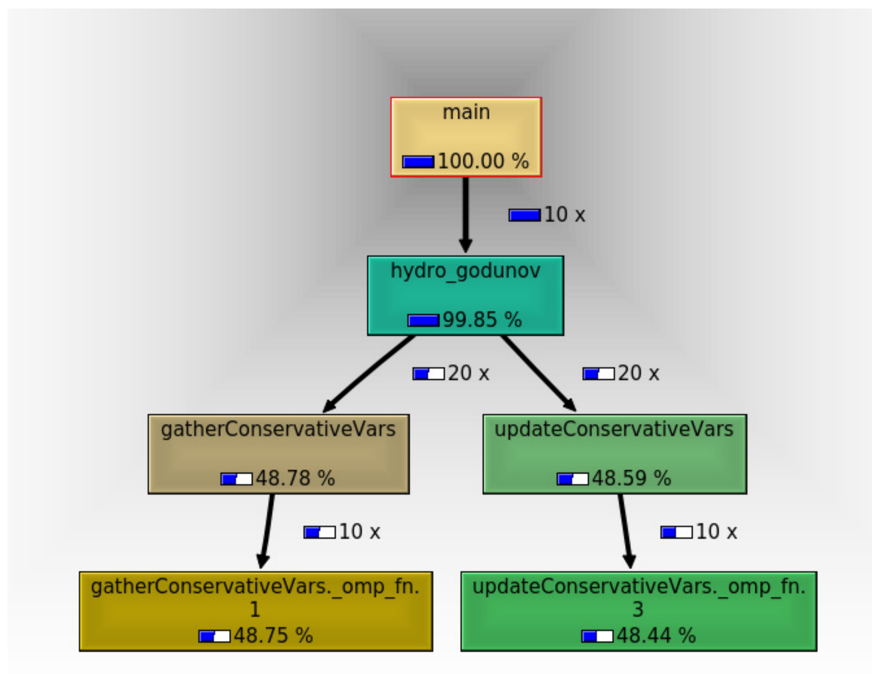

Figure 3: Top-down visualization of the "Spatial Loss for L1" metric of Callgrinds cache use analysis for HYDRO.

extension for inclusive cost collection presented in this paper makes the HYDRO run taking 168 seconds, adding around 9% of runtime.

Statistics output shows that the back-dating tree structure used for inclusive cost collection in this run has an average tree size of 217 nodes with 987 being the maximum. The latter approximates the number of distinct functions executed which is 997. This includes not only functions within HYDRO, but also all functions in shared libraries which got executed during the run. Further we note that 1142 distinct call arc objects (`JCCs`) were allocated.

Callgrind simulates a 2-level cache hierarchy, with size/associativity parameters defaulting to the parameters of the L1 and last-level caches of the real hardware, respectively. Thus, Callgrind uses 32 KB and 6 MB sizes with associativities 8 and 12, respectively. The short HYDRO run does around 2.5 billion accesses, resulting in 97 million L1 misses and 69 million last-level misses, that is 6.2 GB of data transfer into L1 and 4.4 GB into last-level cache from main memory; due to read-for-ownership transactions, every write miss also is a read miss — the numbers are only in direction to the core. Cache use results show that from the 4.4 GB data loaded from main memory into L3 cache, only 17 MB were never actually used. However, from the 6.6 GB of data loaded into L1, 1.2 GB (19 %) are never accessed (this metric is called `SpLoss1` in Callgrind, i.e. *Spatial Loss of L1 cache*). Fig. 3 shows the top-down visualization of `SpLoss1` in our GUI visualization (`KCachegrind`). As both the relevant functions `updateConservativeVars` and `gatherConservativeVars` use OpenMP, GCC's OpenMP implementation adds a call to `GOMP_parallel` in between these functions and the OpenMP loop bodies extracted into separate functions. We use "–fn-skip=GOMP_parallel" to remove this function from the visualization. This works by attributing costs of functions not to be shown to the caller.

Here is the code extract of one of the identified functions with the high "spatial loss" (`conservar.c`, lines 103-111):

```
    #define IHU(i,j,v) ((i)+Hnxt*((j)+Hnyt*(v)))
    ...
103 #pragma omp parallel for private(j,s)
104 for(s=0;s<slices;s++) {
105   for(j=Hjmin;j<Hjmax;j++) {
106     u[ID][s][j]=uold[IHU(rowcol+s,j,ID)];
107     u[IU][s][j]=uold[IHU(rowcol+s,j,IV)];
108     u[IV][s][j]=uold[IHU(rowcol+s,j,IU)];
109     u[IP][s][j]=uold[IHU(rowcol+s,j,IP)];
110   }
111 }
```

This nested loop actually does a transposition of four matrices. From Callgrind results, we see that the loops relate to the input size 500x500; lines 106–109 are executed $500 * 500$ times. These transpositions cannot exploit the L1 cache. All loads are misses, and only use 8 bytes from a 64 byte cache line before it gets evicted. The optimization is beyond the scope of this paper, but loop blocking (that is, transpose sub-blocks which can fully use L1 cache lines) probably increases the performance.

It may be that the issue detected by Callgrind's cache use analysis is hidden by main memory latency in our case (with a size of 500x500) and that such an optimization does not really result in much improvement. However, we note that for larger input sizes, the transposition may not fit into L2. The traffic for the shared L3 may result in poor scaling of the code within a multi-core chip.

Further Overhead Results

In the following, we give some kind of worst-case scenario of the additional memory requirements of the back-dating tree. The memory consumption at a given point in time depends on the number of nodes in the call graph trees of the active threads. In the worst case, each loaded cache line may be referenced by another leaf node, and call chains only may share a common root. Thus, the memory consumption depends on (1) the size of the cache (more exactly, the number of cache lines) and (2) an application characteristic which tells about how much various call chains differ from each other.

To present a realistic worst case, we run the initialization phase of Mozilla Firefox 44.0 with Callgrind's extended cache use analysis; we waited until the program window showed up with a page involving a search field, and we closed the window afterwards. This results in almost 80 thousand functions being executed in over 60 threads. The average of summing up the size of all required back-dating trees is 37625 nodes, with the maximum being 89486 nodes for all trees. The maximum almost matches the number of cache lines in the L3 cache, which is $6MB/64 = 98304$. We conclude that the number of cache lines in the largest cache configuration of the simulated architecture seems to be some kind of upper bound for the number of nodes in the back-dating trees. As said before, one could create an almost infinite tree by for example carefully constructing function calls that do hardly any loads from main memory into cache.

5 Related Work

Alternative Inclusive Cost Collection Methods

When programmers add their own profiling to their source code, either using time or performance counters with PAPI [12], they usually read the cost counters before and after an interesting piece of code and print out the difference. This is exactly what Callgrind does with immediate metrics. However, reading cost counters actually can be very time consuming if done with high frequency. This is bad when we want to profile the runtime behavior because measurement error is in the same order as the measurement overhead. With manual profiling, such problems are easily seen and worked around. However for a generic tool, it is difficult to estimate the measurement overhead from inserted measurement code, as it is not known how often given code will be executed. For example, simply add cost counter reads at start and end of every function already may result in the doubled runtime, rendering results invalid.

Sampling by e.g. only looking at every n-th event instead of every event is better[7]. We can get the distribution of events over program instructions in a statistical way, which approximates the real self costs regarding to that event. Overhead is tunable; for more exact results, we just have to run the sampling for a longer time. A drawback is that we do not get inclusive costs. For that, we can propagate every single sample up a maintained call chain. Indeed, most sampling profilers such as the Linux Perf Events[5] can do that. However on every sample, the tool typically back-traces the stack frames which takes quite some time and therefore, samples should not be done with high frequency to keep the overhead below a given threshold. There are ways to reduce the stack back-tracing overhead. For example, the HPC Toolkit [14] replaces return addresses to own trampoline code at sample points for lower overhead. Further, it uses sophisticated binary analysis to get a higher quality of back-traces.

The older GProf [7] uses a different approach: it just collects self costs by sampling, and uses minimalistic instrumentation at function entries to collect enough information for heuristic offline propagation of self costs. However, the resulting approximated inclusive costs can go wrong, and even the minimalistic instrumentation can have a high overhead for small functions. Callgrind has no issue with overhead, as all costs come from a simulated machine model. For back-dating metrics, we actually could use the approach from GProf, heuristically propagating self costs. However, the inclusive costs collection method for immediate metrics are exact in contrast to any heuristic. We do not want to mix such different result qualities, as this would be very difficult for a user to understand.

We note that to the best of our knowledge, for real measurement tools, it currently is impossible to collect back-dating metrics at all with available hardware.

Cache Simulation

Architecture simulation is used in the design process for processor hardware, such as SimpleScalar [3] and Gem5 [2]. For this purpose, cycle-accurate simulation using an exact machine model is important. However, this results in huge slowdown factors which are prohibitive for using them as performance analysis tools by application programmers, even with sophisticated techniques such as interval simulation in Sniper [4]. To reduce simulation time, it may be fine to only simulate a processor component. This especially is the case for caches whose good exploitation can easily make a difference in the order of a magnitude. An example for simulation of complex cache designs is CMP$im [9], which is used to study novel replacement policies. Examples for earlier proposed cache simulators are SIGMA [6] or MICA [8].

[7] Aliasing effects still have to be taken into account.

For performance studies, simulators with simple cache models are usually enough. This way, MemSpy [11] analyses memory access behavior related to data structures, and SIP [1] provides metrics for spatial locality. SIP only calculates metrics at one cache level. These simulators are quite similar to Callgrind. However, most are academic projects not as useable and stable as the tools found in the Valgrind project.

We do not know of any cache simulators (including the mentioned ones) which collect back-dating metrics.

6 Conclusion

In this paper, we presented our extension to Callgrind for collecting exact inclusive costs for cache use metrics. These metrics reflect temporal and spatial locality behavior of accesses to cache lines and are collected at cache line eviction time. Callgrind relates these metrics to the instruction which loaded the cache lines. We classify such metrics as back-dating metrics as they get attributed to an instruction executed earlier in time. The presented technique which uses a partial call tree produces additional overhead of around 9% with neglectable memory overhead for an example HPC code. The extension makes a top-down approach to cache use analysis possible and thus improves usability. It will be merged into the next Valgrind release.

As cache use is about access behavior to data structures, we will research options to add attribution of cache use metrics to data structures. However, being another dimension of attribution of metrics related to memory accesses, this will also need an extension of the visualization.

References

[1] E. Berg and E. Hagersten. SIP: Performance tuning through source code interdependence. In *Proceedings of the 8th International Euro-Par Conference (Euro-Par 2002)*, pages 177–186, Paderborn, Germany, August 2002.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, T. Hower, D. R. an d Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[3] D. Burger, T. M. Austin, and S. W. Keckler. Recent extensions to the simplescalar tool suite. *SIGMETRICS Perform. Eval. Rev.*, 31(4):4–7, Mar. 2004.

[4] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 52:1–52:12, New York, NY, USA, 2011. ACM.

[5] A. C. de Melo. The new Linux perf tools. Presentation at Linux Kongress 2010, Sep 2010.

[6] L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia. SIGMA: A simulator infrastructure to guide memory analysis. In *Proceedings of SC 2002*, Baltimore, MD, November 2002.

[7] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 39(4):49–57, Apr. 2004.

[8] H. C. Hsiao and C. T. King. MICA: A memory and interconnect simulation environment for cache-based architectures. In *Proceedings of the 33rd IEEE Annual Simulation Symposium (SS 2000)*, pages 317–325, April 2000.

[9] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A Pin-based on-the-fly multi-core cache simulator. In *Proc. Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, pages 28–36, Beijing, China, June 2008.

[10] P.-F. Lavallée, G. C. de Verdière, P. Wautelet, D. Lecas, and J.-M. Dupays. Porting and optimizing HYDRO to new platforms and programming paradigms lessons learnt. http://www.prace-project.eu/IMG/pdf/porting_and_optimizing_hydro_to_new_platforms.pdf, 2012.

[11] M. Martonosi, A. Gupta, and T. E. Anderson. MemSpy: Analyzing memory system bottlenecks in programs. In *Measurement and Modeling of Computer Systems*, pages 1–12, 1992.

[12] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[13] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[14] N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, New York, NY, USA, 2009. ACM.

[15] R. Teyssier. Cosmological hydrodynamics with adaptive mesh refinement-a new high resolution code called ramses. *Astronomy & Astrophysics*, 385(1):337–364, 2002.

[16] J. Weidendorfer, M. Kowarschik, and C. Trinitis. A tool suite for simulation based analysis of memory access behavior. In *ICCS 2004: 4th International Conference on Computational Science*, volume 3038 of *LNCS*, pages 440–447. Springer, 2004.

[17] J. Weidendorfer and C. Trinitis. Collecting and exploiting cache-reuse metrics. In *ICCS 2005: 5th International Conference on Computational Science*, volume 3515 of *LNCS*, pages 191–198. Springer, May 2005.