Exploring Space-Time Trade-Off in Backtraces

Jean-Baptiste Besnard, Julien Adam, Allen D. Malony, Sameer Shende, Julien Jaeger, Patrick Carribault, and Marc Pérache

Abstract The backtrace is one of the most common operations done by profiling and debugging tools. It consists in determining the nesting of functions leading to the current execution state. Frameworks and standard libraries provide facilities enabling this operation, however, it generally incurs both computational and memory costs. Indeed, walking the stack up and then possibly resolving functions pointers (to function names) before storing them can lead to non-negligible costs. In this paper, we propose to explore a means of extracting optimized backtraces with an O(1) storage size by defining the notion of stack tags. We define a new data-structure that we called a hashed-trie used to encode stack traces at runtime through chained hashing. Our process called stack-tagging is implemented in a GCC plugin, enabling its use of C and C++ application. A library enabling the decoding of stack locators though both static and brute-force analysis is also presented. This work introduces a new manner of capturing execution state which greatly simplifies both extraction and storage which are important issues in parallel profiling.

J.-B. Besnard (✉) · J. Adam
ParaTools SAS, Bruyères-le-Châtel, France
e-mail: jbbesnard@paratools.fr

J. Adam
e-mail: adamj@paratools.fr

A. D. Malony · S. Shende
ParaTools Inc, Eugene, OR, USA
e-mail: malony@paratools.com

S. Shende
e-mail: sameer@paratools.com

J. Jaeger · P. Carribault · M. Pérache
CEA, 91 297 Arpajon, France
e-mail: julien.jaeger@cea.fr

P. Carribault
e-mail: patrick.carribault@cea.fr

M. Pérache
e-mail: marc.perache@cea.fr

H. Mix et al. (eds.), *Tools for High Performance Computing 2018 / 2019*,
https://doi.org/10.1007/978-3-030-66057-4_8

1 Introduction

The rapid pace at which High-Performance Computing (HPC) hardware is evolving is putting unprecedented pressure on parallel software and its developers. Hybridization such as MPI+X or MPI+CUDA require careful design and lead to potential faults inside codes which have to transition. This state of things explains why the rich tool ecosystem available in HPC—validation tools, profilers, tracing tools and debuggers, alongside their corresponding support APIs is important to address this ever-going challenge. Whether a program is being debugged or profiled, tools generally try to capture the program's state to present it to the end user. To do so, a parallel program can be seen as running in three-dimensional space [12]: over computing resources (space), time and program's code. The ability to capture, explore and present this state in a scalable manner is at the core of the design of any parallel tool. Indeed, due to the potentially large number of MPI processes and threads running on supercomputers, means of capturing and storing points in this execution space have to be carefully designed [7, 20, 23, 29]. In doing so by whatever means, it is generally of interest to capture the state as fast as possible and to encode it using a minimum amount of data.

As far as the two first dimensions are concerned, the locality can be expressed in a relatively compact manner. For example, an event can have a *timestamp* which precisely defines when it happened. When it comes to parallel machines, a timestamp is not necessarily a simple object as a distributed synchronization is required [5, 6], however, the availability of high precision timers such as the TSC combined with elaborated synchronization techniques provides such information in constant time and storage cost (usually a 64 bits integer). Similarly, if we now consider *space*, by capturing execution stream creation and their locality (i.e in which node, process and pinned to which core) it is possible to build an integer identifier table for execution streams. Then, such value can be used to compactly describe a given thread in a massively parallel execution given that associated meta-data are correctly handled. It is not a trivial process but it is solved considering, for example, state-of-the-art trace formats such as OTF2 [10, 20].

However, the last dimension describing which part of the program is being executed is less trivial to extract. It is possible to capture the program counter which points to the precise line of code being executed but by doing so, the hierarchical nature of the call-stack is lost. In such configuration, sampling a program shows "where" you spent time during the execution but does not present you the succession of function calls which led to it or distinguish between them—all costs being summed up. In order to preserve this hierarchy, either a list of addresses has to be kept for each measurement point (the backtrace) or each entry and exit event has to be monitored and replayed to enable a replay of the stack (approach used in traces). Overall, there is currently no method enabling compact call-stacks description at the level of what can be done for other profiling dimensions —a 64-bit integer identifier. Devising such compact descriptor for call-stacks is the object of this paper. In particular, we define the notion of *stack-tag* and explain how it benefits to performance

and debugging tools by (1) optimizing backtrace operation and (2) yielding constant size 64 bits backtrace descriptors.

The rest of this paper is organized as follows, we first present related work. In a second time, we progressively introduce the components enabling *stack tagging*. The associated space-time trade-off we rely on is exposed in the context of the *hashed-trie* data-structure used to encode call-stacks. We then present our runtime implementation relying on a GCC plugin and we detail how our approach compares to "regular" backtraces. Finally, we evaluate *stack-tagging* on representative applications demonstrating that such an approach can provide several advantages.

2 Related Work

The backtrace is one of the most common operations for support tools such as debuggers [24] and profilers [25]. It is the mean of retrieving the current layout of the stack and therefore the nesting of function calls leading to current execution point for a given execution stream. Common methodologies for retrieving backtraces involve walking the stack using a dedicated library [1, 22, 26]. Complex schemes are at sometimes needed to reverse compiler's optimization in order to provide a clear stack description. Such unwinding mechanisms were developed concurrently for exception handling (e.g., C++) and debugging [9], leading to duplicate binary sections with similar purposes (`.eh_frame` and `.debug_frame`) [2].

It is also to be noted that some architectures have dedicated support for backtracing such as through ARM unwind sections [3] or Intel Last Branch Register (LBR) used for example by Linux Perf (with the `-call-graph lbr` flag). Such hardware support despite much faster than any software method generally come with limited resource with respect to maximum stack depth. For example, current Intel Skylake architecture can store only up to 32 branching records. In addition, access to such performance counters might be restricted [27], preventing their direct use.

The closest related-work we found is the notion of *probabilistic backtrace* [8] by Bond et al., applying a similar hashing technique for Java applications. We extend this previous work by fully defining how stack tags could be decoded by storing relevant information in the generated binary. In addition we target GCC which is able to compile languages more relevant to High-Performance Computing (C, C++, Fortran).

Overall, doing a backtrace, despite being a widespread operation, involves complex considerations and therefore is reserved for dedicated libraries generally provide this feature. Our work in this paper offers an alternative approach which involves a space-time trade-off while preserving overall application performance.

The hashed trie we present in this paper is inspired by its reference data-structure the trie [18] (or prefix-tree). It aims at providing features similar to those of a Merkle tree [19] whereas instead of hashing data hierarchically, we hash the path inside the tree. As far as storing data through hashing is concerned, it is a common model in the *block-chain* paradigm. In our work, we apply the principle of hash-chains to

encode paths in arbitrary graph modeled by their prefix-tree (or a subset of it). It is, therefore, a specialized data-structure more focused on encoding the keys leading to the data than minimizing access time. In fact, what we present is not strictly speaking a data-structure but a key-linearization procedure to simplify the storage of metrics attached to a path inside a graph.

3 Hashed Trie

At the core of our contribution, we find a data-structure derived from *prefix trees* or *tries*. Such data-structures are aimed at optimizing the search for elements by encoding their corresponding key in a tree as a succession of nodes leading to the element. In other words, the key defines the successive indexes leading to the data.

Figure 1 presents a simple example of trie indexed with a string key. Such data-structure consists in an M-ary tree representing the choice of an element k_i from the key alphabet. Here, if we consider capital letters, we have $M = 26$ and each k_i going down the prefix tree is one of the 26 capital letters. Such data-structure enables both insertions and searches in $O(l_k)$ with l_k the key length. Given that the key can be split in k_i sub-elements, such trie has a better worst case than for example an hash-table which is $O(N)$ with N the number of entries with a much lower average dependent from the hashing function. However, the trie tends to use much more space than, for example, hash-table. Indeed, in order to encode all possible keys up to length n in an alphabet of M symbols, you need $k = \sum_{i=0}^{i=n} M^i$ nodes in your trie, each node containing potentially M pointer to its children nodes. For example, storing all keys up to a length of three in an alphabet of three characters requires 40 nodes each with three pointers or 320 Bytes whereas a simple list of all the keys would use $\sum_{i=1}^{i=3} i * 3^i + 1$ or 103 Bytes (counting the `NULL` key). A trie can, therefore,

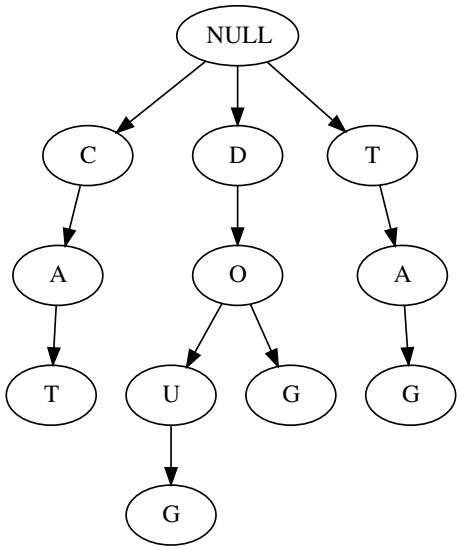


Fig. 1 A trie

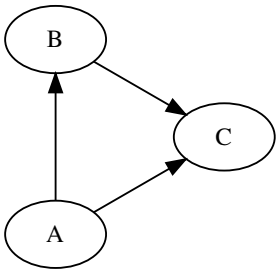


Fig. 2 Sample directed graph

become expensive for large key-spaces, it is consequently often used in combination with other data structures.

Now that we introduced the notion of trie, we propose to consider it not to store an element at a given key but conversely to associate an element with a given key. Extending this idea, we propose to derive from tries a compact descriptor for arbitrary paths in a directed graph. If we define a Graph $G(M, K)$ with M the set of vertices and K the set of edges, we can see that any path in such graph can be encoded in a prefix tree. The reason for this is that each step in the graph is like choosing the edge k linking current element to the next one from M. For example, if we look at Fig. 2, there are two paths from A to C, and they can be expressed as ABC and AC, such keys could be seen as prefixes in a trie covering the whole space for these three nodes. However, we have seen that the trie is less efficient in terms of storage. Our idea is then to further optimize this aspect using the trie as a generative function of the graph, thus storing paths in a radix-tree in a space efficient manner. Enabling, in particular, the probabilistic encoding of the succession of choices in the key space. Then, if each node of a graph is given a unique key, any path in the graph can be encoded with an identifier representing the successive edges from one node to the other.

In order to generate such identifiers with a fixed size, we rely on recursive identifier hashing. Therefore, each step in the prefix-tree generates a new and highly likely unique *tag* associated with this path. For example, if we consider three nodes A, B and C with respectively 1, 2 and 3 as unique identifier we can encode the tags $T_{a\to b\to c}$ for the path $A \to B \to C$ as follows using MIX as hashing function combining the two identifiers into one:

$$\begin{aligned} T_a &= MIX(1, \emptyset) \\ T_{a\to b} &= MIX(T_a, 2) \\ T_{a\to b\to c} &= MIX(T_{a\to b}, 3) \end{aligned} \tag{1}$$

By looking at Eq. 1, we see that thanks to the nested hashing the size of each path remain constant as the size of a given hash. This makes a big difference when compared to a trie which needs to store pointers to each possible suffix and even when compared to a direct key-value array. For the rest of this paper, we now define

the hashed-trie as being composed of two parts, (1) a *model* describing the list of graph nodes and associated keys and (2) a *list of tags* representing arbitrary path between these nodes.

4 Path Reconstruction in Hashed-Tries

In the previous section, we described how fixed-sized *tags* could be generated to describe paths in a tree. In particular, we unfolded the idea of relying on a trie acting as a *model* for the tree supporting the given paths instead of describing the full corresponding tree. Of course, the choice of repetitively hashing paths to generate such descriptors does not convey the direct nature of the encoded path. In particular, such encoding has to be resolved. This process required for each *tag* and due to the nature of hash functions, it needs to rely on a brute-forcing of all the possible combinations.

At first, the brute-forcing process may look expensive and indeed it is in some conditions. However, this cost has to be mitigated for a tree of (1) small size and (2) cases when the radix can be limited. Table 3 illustrates the time needed to brute-force all the possible path for $|M| \in [2, 1024]$ and $l_k \in [2, 10]$. These times do increase in an exponential manner along the two axes. Looking that the key space complexity $\sum_{i=0}^{i=l_k} |M|^i$, we can see it as a sum of powers which biggest term is the one where $i = l_k$, additionally denoting $|M|^i = e^{i.ln(|M|)}$ shows that the complexity is expected to grow faster with path lengths.

What can be observed is that this cost is indeed very high for larger parameters. However, such cost encompasses the resolution of *all* path up to l_k which could be encountered for the given graph. Second, the results of Fig. 3 are relative to a sequential computation, such key-space exploration being an embarrassingly parallel problem, linear gains are to be expected from parallelization. Consequently, even if

$\|M\|$	$l_k = 2$	$l_k = 4$	$l_k = 8$
8	<1 μs	2.14 μs	3,91 ms
16	<1 μs	12.87 μs	0.40 s
32	1.19 μs	71.04 μs	47.80
64	95.36 μs	411 μs	>1 hour
128	95.37 μs	8.49 ms	>1 hour
256	95.36 μs	60.45 ms	>1 hour
512	1.9 μs	0.46 s	>1 hour
1024	4.05 μs	3.49 s	>1 hour

(a) Sequential computation time for brute-forcing all paths for different tree ($|M|$) and path lengths (l_k).

$\|M\|$	Degree	$l_k = 2$	$l_k = 4$	$l_k = 8$	$l_k = 16$
8	2	<1 μs	<1 μs	1.9 μs	0.54 ms
16	2	<1 μs	<1 μs	1.9 μs	0.54 ms
32	2	<1 μs	<1 μs	1.9 μs	0.54 ms
64	3	<1 μs	1.19 μs	25.99 μs	0.17 s
128	6	<1 μs	1.91 μs	2.57 ms	1693.69 s
256	12	<1 μs	15.02 μs	0.17 s	>8 hours
512	25	<1 μs	69.14 μs	25.62 s	>8 hours
1024	51	<1 μs	0.55 ms	3969.57 s	>8 hours

(b) Sequential computation time for brute-forcing all paths for different tree ($|M|$) and path lenghts (l_k) when considering an output degree of 5% of $|M|$ (forcing at least two outgoing edges).

Fig. 3 Stack-tag brute-force evaluation for both full and sparsely connected graphs

it takes thousands of seconds to explore a given key-space it would be possible to rely on a punctual parallel resolution to reduce the solution time to a few seconds.

Despite the embarrassingly parallel nature of the path resolution scheme, there is another approach for limiting its cost. Indeed, resolving paths using what we described as the *trie model* supposes that the graph is potentially fully-connected and therefore yields the maximum resolution cost. This model is by nature the universal one as it encompasses any directed graph generated by this set of nodes. However, and hopefully, some of the most interesting graphs are sparser. It is then possible to extend our *model* to express the reduced radix of each vertex. To do so, instead of a node-list, the trie model could be an adjacency matrix or some knowledge with respect to connectivity. Again, this supposes a storage-size trade-off, indeed a full adjacency matrix takes up to $|M|^2 \times l_{identifier}$ Bytes and it has to be compared to the associated extra computational cost. Moreover, the more edges there are, the closer we get to the simple trie-model which mimics a fully-connected tree. Therefore, relying on adjacency matrices to describe the source tree is clearly aimed at relatively sparse trees. In order to illustrate the potential gain in performance, we propose to redo the computations from Table 3a with a loosely connected graph. In particular, we generated random graphs for each configuration such as the output degree of each vertex is 5% of the number of nodes with at least two outgoing edges.

By looking at these results as shown in Table 3b, the cost has been reduced by a very important factor, up to millions times faster ($l_k = 8, |M| = 32$), when compared to the brute-force result on a fully-connected tree. More importantly, for relatively small graphs with controlled output degrees, computing every path is a fast operation in the order of the seconds—keeping in mind that we present timings for sequential computation. In summary, the hashed-trie approach is to be applied to either tree with a small number of vertexes (a few hundred), with a low radix (less than 10) and with the expectation of small paths—the latter being the most expensive dimension in terms of cost.

As we will illustrate in the following sections, backtraces do fulfill such constraints and can be practically reconstructed. Next Section will consider the use of this data-structure as an alternative backtracing model. First, evaluating and comparing it with other methodologies before discussing associated trade-off and advantages.

5 Constant Size Backtrace

Now that we have described the *hashed-trie* and space/time trade-off it supposes, this section proposes to apply it to parallel program instrumentation. The overall call-stack of a program can be seen as a tree and the call stack is a path in this tree. Due to the potential creation of threads, all stacks may not share the same root i.e. the main function, and call stack may start with any function.

As far as the structure of the callgraph is concerned, it can be directly inferred from the source code as in most cases (ignoring function pointers) function calls can be resolved statically. Such information is not obvious to retrieve from a compiled

binary, it is however relatively trivial when done at compilation time. For this reason, we proceeded to the implementation of call-stack tracking through a hashed-trie. This was done in a GCC 8.3.0 plugin inserting extra code in every function during program compilation.

5.1 Stack Tags with a GCC Plugin

Just as the `-finstrument-functions` flags does in GCC we added code at both the start and end of each function. As per our description in the previous section, our goal is to enable the systematic hashing of the current stack location before entering each function. As presented in Fig. 4, in order to track call-stacks, the program has to be modified not only by hashing current path when entering a function but also by saving the current *stack tag* in order to restore it when leaving the function. To do so, we dynamically added a temporary variable inside each function to save the current *tag* prior to computing the one of the local function—such saved tag is restored when leaving the function. To simplify retrieval while supporting multi-threaded code, the *stack tag* is stored in a global thread-local variable (named `tls_tag` in Fig. 4) initialized with the value 0. This first initialization defines the "NULL" function at the root of any call-stack being hashed with the first function. This initial state is needed as it enables the support for thread creation (POSIX or OpenMP) which may have a stack starting with an arbitrary function. The direct consequence of this is that in the call-stack model when reconstructing tags, the "NULL" function is the parent to every other function.

Each function is given a pseudo-unique identifier based on a hash of its name, such identifier is statically inserted in each function when proceeding to the hashing. Additionally, two sections are added to the binaries, the first one `.btloc` defines the matching between a function name and a 64 bits identifier. It can be seen as defining the basic *trie-model* as discussed in Sect. 3. Indeed, any call-stack will be a combination of these instrumented function as per definition only instrumented functions and then those listed can alter the *tag*. Naturally, using all the functions

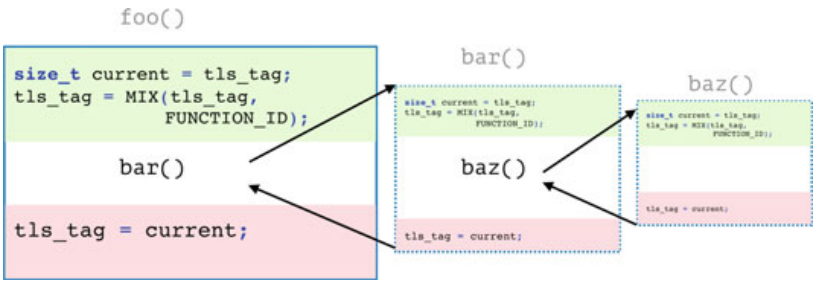


Fig. 4 Illustration of our *stack-tag* implementation using code inserted in successive function calls. `tls_tag` is a global thread-local variable accessible from each function context. The *stack-tag* can be retrieve at any point of the code by reading this same variable

as the sole generative function of the possible call-stacks is suboptimal. It is for this reason that we enriched our plugin with simple static analysis capabilities. Doing so, we were able to extract every function calls from instrumented functions. We then defined a new section called `.btedge` where known transitions are listed as 64bits tuples. This section is used, in a complement of the node-list to optimize *stack-tag* decoding. Note that by default, the linker merges sections it does not "understand", greatly simplifying the gathering of compile-time information inside the final binary.

The hashing function used is very simple to limit its overhead. It aims at *mixing* current path with the new function identifier. It is implemented as $MIX(A, B) = (A * 11) \oplus B$ with A the current path (0 for first element) and B the identifier of the target function (as stored in the `btloc` section). Note that in order to optimize hashing, function identifiers are themselves hashes of the function name, leading to values spread in the 64-bit space and therefore mitigating the need for a complex MIX function.

A global TLS variable handling the *stack-tag* it is inserted by linking a shared library that we called `libbt`. This library is externally referenced by the code injected in each function. Besides, this library implements several functions, including backtrace and stack-tag resolution, making these facilities available at runtime. A tool willing for example to resolve stack-tags in post-mortem may link itself again the same library which is able to open binaries and libraries, extracting section of interest to enable *stack-tags* consumption.

6 Backtrace Comparison

In order to evaluate the stack-tag backtrace implementation, to assessing its gains, this section proposes to benchmark each approach turn-by-turn thanks to a simple recursive code. By doing so, we derive a cost model for each methodology considering both time and memory aspects. In particular, we show that *stack tagging* is competitive in terms of computing cost and efficient in terms of memory usage.

6.1 Stack-Tag Evaluation

In order to describe the cost of stack tagging, it is crucial to account for the transitive overhead linked with the systematic tagging. Retrieving the backtrace state by itself is, in fact, the retrieval of a thread-local variable. The direct consequence of this technique is that there is a cost which can be linked to the backtrace attached to each function call. Therefore, before comparing our technique with others, the overall per-function call cost has to first be computed.

To do so, we relied on a simple recursive function. We want to (1) measure the per-call overhead and (2) ensure that this overhead is actually linear. By using such function with and without stack-tags for various recursion depth, we generated the

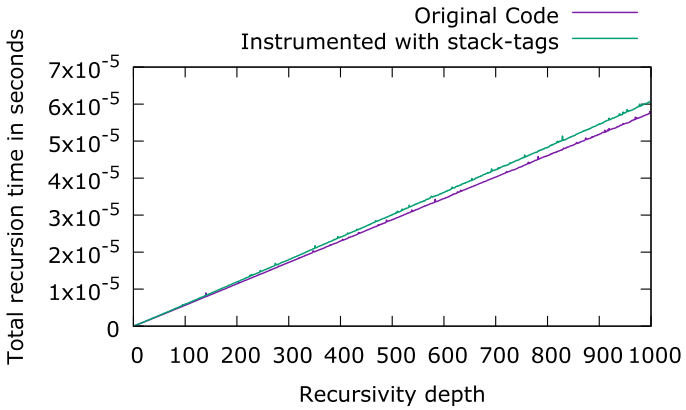


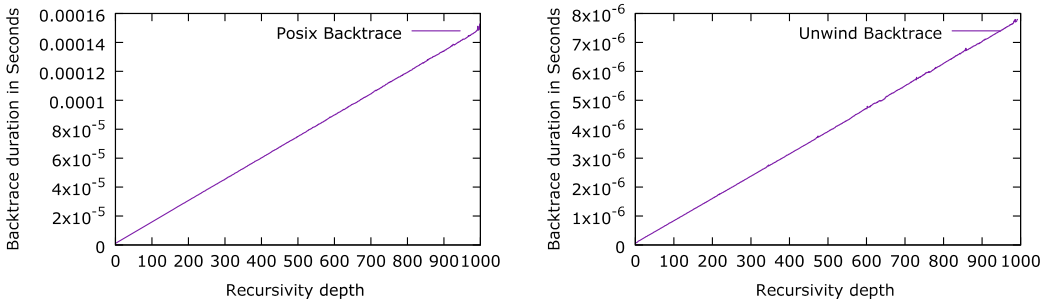
Fig. 5 Total time spent in the recursion with and without stack-tags enabled

graph of Fig. 5 which presents the total time spent in the recursion with and without instrumentation. Due to the small duration in play, we averaged each measurement 10^4 times and used the Timestamp Counter (TSC) as a time source. What is important to observe is that this overhead is clearly linear (as it could be expected) and therefore that it is meaningful to model the cost of stack-tagging on a per function call basis. In addition, using these two series it is possible to compute the pure per-call overhead of the stack-tagging logic as the difference divided by recursion depth. The per-call overhead associated with stack-tagging is relatively constant and close to 4.76 nanoseconds per function call on this given platform (Intel Core I7 Haswell CPU running at 3.6 GHz). In addition to the computation upon each function call, the cost of accessing the TLS to retrieve current hash has to be considered, we measured it at 8.9 ns (averaged 10^4 times) with the first access at 12μs on the same system as a call the `libbt` shared-library returning the current *stack-tag*. These two measurements allow us to parametrize the cost-model of our stack tracking implementation such as $C_{ht}(d) = 8.9 + 3.2d$ with d the backtrace depth and C_{ht} in nanoseconds.

6.2 GLIBC and Libunwind Backtrace Evaluation

Now that we measured the total cost of our backtrace implementation, we can compare it with others. We start with the glibc (version 2.24-11) implementation [1] as part of the `backtrace` function from the `execinfo.h` header. We measured its per call cost at the various depth of the same recursive function. Figure 6a presents these results. If we now compute the per frame cost of the GLIBC backtrace we measure 1.95 μs per frame. Therefore we can model the GLIBC backtrace cost as $C_{glib} = 1950d$ with d the call-stack depth and C_{glib} in nanoseconds.

Another implementation of the backtrace function is provided by the `libunwind` [22]. This function called `unw_backtrace` has the exact same interface than the one from the GLIBC. Again, we studied the same recursive function in Fig. 6, we



(a) GLIBC Backtrace. (b) Libunwind Backtrace.

Fig. 6 Measurement of GLIBC `backtrace` and libunwind `unw_backtrace` in function of call-stack depth. All values were averaged 10^3 times

measure a cost of 7.79 ns per frame, we can then model $C_{uw} = 7.79d$ with d the call-stack depth and C_{uw} in nanoseconds. It is to be noted that this value is on par with the one stated by the libunwind documentation "around 10ns/frame" [21].

6.3 *Performance Modeling Summary*

In previous subsections, we have compared two backtracing approaches to *stack-tagging*. A notable difference between *stack-tagging* and regular backtrace is that the later returns precise program counters when walking up the stack. This means that not only the parent function can be extracted but also the calling line after resolving the address. This is a supplement of information which can be of importance, for example, if a function calls the same routine multiple time—it would not be captured by *stack-tagging*. This mechanism cannot, therefore, be a full replacement for actual backtraces. Besides, we also measured some gains with *stack-tagging*. First dealing with the storage size. As detailed at the beginning of our paper, a stack-tag is always 64bits whereas backtraces grow linearly with a storage size of 64bits per frame. Therefore, a regular backtrace takes more storage space which is an expensive resource, for example, in trace formats. Besides, having such a compact call-stack descriptor enables fixed-size events which can be an advantage, for example, to do a timestamp search in files through a dichotomy [17], assuming events are ordered.

Moreover, in terms of backtracing overhead, the POSIX implementation was clearly not designed for usage in a performance constrained environment. The two other methods are however closer to each other. As presented in Fig. 7, the libunwind backtrace is faster up to two frames due to the cost measured for TLS access. However, another aspect of the *stack-tagging* model is that it is always on. Consequently, where libunwind pays the cost at each point of interest, the linear part of our approach is always paid. This shows that the use of *stack-tagging* is a matter of trade-off, if verbose events are to be attached to deep stack locations and that linearized storage of these backtraces (indexed by tag) is preferable, for example doing a profile, *stack*

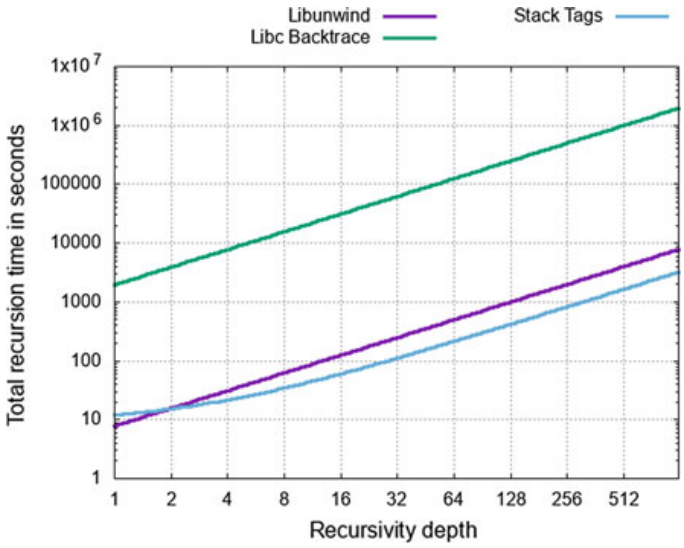


Fig. 7 Backtrace performance-models

tags can be of interest. Conversely, if punctual events are to be located precisely and that the efficient storage of this data is not crucial (e.g. not a trace) libunwind is probably the best pick.

7 Performance Evaluation

In the previous section, we evaluated the cost of a backtrace alone. However, the *stack-tagging* methodology incurs as aforementioned trade-offs and yields information which are subject to potential collisions due to the nature of hashing functions. These conflicts should be minimized to enable the reasonable use of such descriptor for performance analysis. Second, stack tags need to be resolved in order to be converted back to a call-chains, this has a cost which has to be characterized and limited to make the approach practical. Eventually, unlike other methodologies, our approach is not associated with a transitive cost upon measurement but to a constant per function overhead. Again, this overhead has to be considered in the choice of backtracing methodology. This section proposes to evaluate and discuss these aspects, assessing the productive use of *stack-tagging* in performance tools.

7.1 Stack Tags and Collisions

One of the main trade-off supposed by the stack-tag model is that the process supposes the possibility of multiple stacks colliding in the same tag. Indeed, when considering combinatory spaces such as graph traversals and the fixed 64 bits of the stack tag, collisions are inevitable. However, it is important to ensure that the number of

Table 1 Call-graph metric comparison for various applications. Stacks (d = 10) is the number of stacks at depth 10. Names in **bold** are those which led to collisions when exploring stacks up to d = 10

Program	Function Count	Max. Degree	Stacks (d = 10)
vim	5970	110	216 377 858
gdb	17017	159	198 150 110
gnuplot	12335	122	31 696 396
nano	1429	83	2 732 509
xeyes	4454	41	1 180 195
htop	1365	27	416 237
tar	1355	45	292 740
amg2013	1048	105	115 813
IMB-MPI1	97	17	888
lulesh	186	9	536

collisions remains low to enable a meaningful usage of stack descriptors. Indeed, performance tools require reliable data to correctly guide the user. In order to measure this propensity for collisions, we relied on the Spack [11] package manager and inserted a compiler wrapper invoking our backtrace plugin. Doing so, Spack enabled use to conveniently compile several tools and all their dependencies with stack tags enabled providing us with maximal coverage.

In order to detect collisions, we walked all the combinations in the static call-tree up to a given depth storing all the keys and counting collisions. We did not explore a pure brute-force model due to its computational cost and the difficulties of storing all the keys for duplication check. Still, despite being a lower-bound on collisions, we think that data derived from the static analysis are representative of the collision rate, especially as all libraries are instrumented and no transition has to be "guessed". To proceed, we picked a range of common programs available in Spack, first, common development tools such as GDB, Nano and VIM to have relatively large code-bases. We also targeted some utilities such as Gnuplot, htop, tar and a simple graphical application xeyes. Eventually, we considered common HPC benchmarks such as Lulesh, AMG2013 and the Intel MPI Benchmarks.

As presented in Table 1, we considered a relatively diverse corpus of common applications and asked ourselves how many collisions there were in the complete stack tag set up to a resolution depth of 10. Note that in such configuration, the generated callgraph is much larger than what is produced by executing the program as every function from libraries are also unfolded. In the first approach, one can see that the number of combinations is highly dependent on the program. Second, most programs did not yield conflicts. In fact and as it could be expected, we observed conflicts only in larger codes or more precisely codes which call-graph can generate a large number of stacks.

From these results, we can conclude that the collision rate is relatively low and often null for smaller projects. When the graph is small, a tool could mitigate the cost of conflict by presenting the possible stacks for a given tag, still extracting some information. This is, of course, an empirical observation that should have to be confirmed on a much larger set of applications, a point that we would like to address as future work.

7.2 Backtrace Resolution Cost

Now that we have seen in the previous section that *stack tags* were relatively good stack descriptors with a low collision rate, we would like to characterize their decoding cost. This operation consists in starting from a 64 bits stack tag to generate the corresponding backtrace as a list of functions in order of calling. As previously detailed, we rely on two-phase decoding, the first one relies on static analysis by exploring the call-graph stored by a dedicated section of the binary (and its libraries). The second model is much less efficient and is simply a brute-forcing of all the possible transitions until finding the right stack. This last approach can be of use, for example, when a portion of the code is not instrumented and therefore unknown transitions are emitted. Of course, it would not be practical to indefinitely brute-force a given combination as there is no guarantee that the result is attainable in an acceptable time. Therefore, the resolution process is implemented with a *time budget* so that if the brute-forcing takes too much time the symbol resolution fails. Figure 8 presents the percentage of resolved symbols for all the stacks observed during a simple execution of the respective binaries—data were retrieved using the `-finstrument-functions` profiling support from GCC. Obviously, increased resolution time yields improved accuracy. C programs appear to yield better results than C++ ones, GDB (≈60%) is an example of such program where resolution is difficult, we think that it is caused by an incomplete static analysis and therefore a

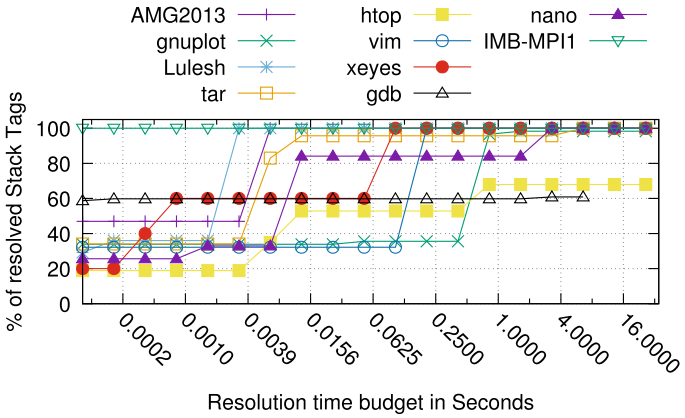


Fig. 8 Percentage of resolved stack-tags for a given per tag resolution time budget

lack of our plugin. Still, we can see that brute-forcing stack-tags in a constrained time is possible and yields acceptable results, enabling tools to decode such tags in a practical manner.

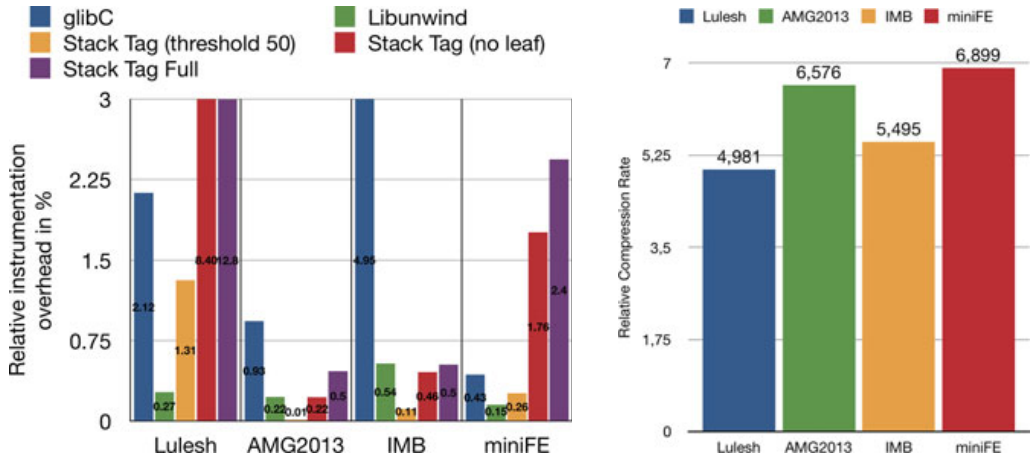
7.3 Overhead for Event-Based Sampling

Now that we have seen that it was possible to (1) assess the low collision rate when using stack-tags and (2) decode tags back to backtraces with an acceptable efficiency we now have to characterize our approach in terms of performance. This section evaluates stack-tagging in the context of Event-Based Sampling (EBS) over MPI calls. It simply consists in tagging each MPI calls with its associated stack to split the profile on the actual source code. It is a common approach for performance tools such as mpiP [28], TAU [23] and Scalasca [13]. What is interesting is that it challenges our approach as previously described it leads to a "constant" overhead as all functions are instrumented whereas other backtraces methodologies impede the program only on sampling points. It is, therefore, one of the most adverse cases, generating a full event traces being more suitable to our model.

We, therefore, ran the Lulesh [16], miniFE [14], AMG 2013 [4] and Intel MPI benchmarks [15] while capturing every MPI calls (through the PMPI, MPI profiling interface) and backtracing at each call before forwarding the call to the MPI runtime. Note that we solely measured the backtracing overhead and did not account for any further processing. Measurements were done with glibc backtrace, libuwind, and stack-tagging. As far as stack-tagging is concerned, multiple cases are presented according to the level of instrumentation. Indeed, unlike other methods, it is possible to partially instrument the binary, therefore leading to partial stack descriptions. This leads to three configurations, (1) full instrumentation, (2) filtering leaf functions (containing no calls) and (3) avoiding instrumenting small functions (minimal length threshold of 50 Gimple statements). This leads us to five measurements per benchmarks comparing our three stack tagging configurations and the two other backtrace implementations to reference execution time.

Figure 9a presents the results of MPI event-based sampling of stacks. It can be seen that C++ benchmarks (Lulesh and miniFE) are challenging targets to our stack tagging model. Indeed, these languages favor small accessors and overall smaller functions leading to increasing overhead, 12.5% for Lulesh and 2.4% for miniFE. However, it can also be seen that selective instrumentation is an efficient manner of reducing this overhead, thresholding by function size being the most efficient model at the cost of backtrace accuracy. However, for C codes, stack-tagging was able to reach levels of performance comparable to the libunwind.

Figure 9 presents the compression ratio achieved by stack tagging when compared to storing arrays of addresses for each backtrace. It can be seen that the average depth of MPI calls in these various codes is such as our 64 bits tags are between 4.96 and 6.89 more efficient in terms of storage size, clearly demonstrating the space-time trade-off we previously mentioned where gains in size are translated to a later decoding cost.



(a) Overhead relative to the non-instrumented case for MPI event-based sampling using various backtrace model. Note that the scale truncates higher values for readability.

(b) Relative compression rate for stack-tagging with respect to regular backtraces (list of addresses).

Fig. 9 Comparison of the stack-tagging approach with other backtracing methodologies on a set of representative benchmarks

In summary, we have seen that our approach could provide compact stack descriptors with generally acceptable overhead. Caution has to be taken when instrumenting small functions which are common in C++. In this case, we have seen that selective instrumentation could partially solve this issue. As part of future work we would like to explore further the possibility of detecting inline function to selectively disable stack-tagging—particularly for C++ codes. Consequently, when considering stack instrumentation from medium to high frequency, our method can have its advantage when compared to other backtraces, as shown in Fig. 7. Conversely, if events of interest are infrequent, the punctual backtrace methodology is to be preferred.

8 Conclusion

In order to flatten the call stack, we proposed a new abstraction based on chained-hashing that we called the *hashed-trie*. This data structure provides a way to model all the possible transitions in a graph and is then used to encode paths as repetitive hashes. It is, of course, important to be able to reverse this encoding and we, therefore, explored the brute-force method but we also proposed a more optimal approach relying on finer modeling of the possible transitions, greatly reducing resolution time.

Eventually, hashed-tries were used to describe stacks. We presented our implementation of compile-time instrumentation using a GCC plugin in charge of injecting stack tags and supplementary sections inside binaries. Using an empirical set

of applications we measured reduced collision rates maximum 0.5% and generally much lower ($\leq$0.01%). Then, after instrumenting the programs and running them, we resolved 85% of collected stack-tag in a constrained resolution time. Eventually, we showed that stack-tag instrumentation led to acceptable overhead with the important aspect that this cost can be mitigated through selective instrumentation, part of this being targeted in future work.

Overall, this paper developed a new mean of encoding path in a graph through chained-hashing opening new opportunities for space-time trade-offs in the instrumentation and storage of backtraces with potential application in support and performance tools. Such a method can advantageously replace regular backtraces methodologies in particular for verbose measurements with acceptable overhead and unprecedented space efficiency.

9 Future Work

As far as the principles behind our MIX function and chained-hashing are concerned we clearly miss a more theoretical approach in the complement of the practical one that we unfolded in this paper. For example, hashes could be split to encode the depth greatly minimizing the search space. Eventually, we solely explored the TLS model to attach context data to our execution streams. It would be of interest to use registers to explore the difference in terms of performance. This would open interesting consideration such as register-based context for runtimes and tools with possible hardware support.

References

1. The gnu libc. https://www.gnu.org/software/libc/
2. Linux standard base core specification 5.0 (2015). http://refspecs.linuxfoundation.org/lsb.shtml
3. ARM: Exception handling abi for the arm®architecture. http://infocenter.arm.com/
4. Baker, A.H., Falgout, R.D., Kolev, T.V., Yang, U.M.: Multigrid smoothers for ultraparallel computing. SIAM J. Sci. Comput. **33**(5), 2864–2887 (2011)
5. Becker, D.: Timestamp Synchronization of Concurrent Events. Dr. (fh), T. Aachen, Jülich (2010). http://juser.fz-juelich.de/record/10841. Record converted from VDB: 12.11.2012; Aachen, TH, Diss., 2010
6. Becker, D., Rabenseifner, R., Wolf, F., Linford, J.C.: Scalable timestamp synchronization for event traces of message-passing applications. Parallel Comput. **35**(12), 595–607 (2009)
7. Besnard, J.B., Malony, A.D., Shende, S., Pérache, M., Jaeger, J.: Gleaming the cube: Online performance analysis and visualization using malp. In: Knüpfer, A., Hilbrich, T., Niethammer, C., Gracia, J., Nagel, W.E., Resch, M.M. (eds.) Tools for High Performance Computing 2015, pp. 53–66. Springer International Publishing, Cham (2016)
8. Bond, M.D., McKinley, K.S.: Probabilistic calling context. In: Acm Sigplan Notices, vol. 42, pp. 97–112. ACM (2007)
9. Committee, D.D.I.F. et al.: Dwarf debugging information format, version 5. Free Standards Group (2017)

10. Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W.E., Wolf, F.: Open trace format 2: The next generation of scalable trace formats and support libraries. PARCO **22**, 481–490 (2011)
11. Gamblin, T., LeGendre, M., Collette, M.R., Lee, G.L., Moody, A., de Supinski, B.R., Futral, S.: The spack package manager: bringing order to hpc software chaos. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 40. ACM (2015)
12. Geimer, M., Kuhlmann, B., Pulatova, F., Wolf, F., Wylie, B.J.N.: Scalable collation and presentation of call-path profile data with CUBE. In: Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007, Forschungszentrum Jülich and RWTH Aachen University, Germany. Accessed 4–7 Sept 2007, pp. 645–652 (2007)
13. Geimer, M., Wolf, F., Wylie, B.J., Ábrahám, E., Becker, D., Mohr, B.: The scalasca performance toolset architecture. Concurr. Comput.: Pract. Exp. **22**(6), 702–719 (2010)
14. Heroux, M.: Minife documentation
15. Intel: Intel®mpi benchmarks user guide. URL https://software.intel.com/en-us/imb-user-guide
16. Karlin, I., Keasler, J., Neely, R.: Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973 (2013)
17. Knüpfer, A., Brunst, H., Nagel, W.E.: High performance event trace visualization. In: 13th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2005). Accessed 6–11 Feb 2005, Lugano, Switzerland, pp. 258–263 (2005). https://doi.org/10.1109/EMPDP.2005.24
18. Knuth, D.E.: The art of computer programming: sorting and searching, vol. 3. Pearson Education (1997)
19. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Conference on the Theory and Application of Cryptographic Techniques, pp. 369–378. Springer (1987)
20. Mey, D.a., Biersdorf, S., Bischof, C., Diethelm, K., Eschweiler, D., Gerndt, M., Knüpfer, A., Lorenz, D., Malony, A., Nagel, W.E., Oleynik, Y., Rössel, C., Saviankou, P., Schmidl, D., Shende, S., Wagner, M., Wesarg, B., Wolf, F.: Score-p: A unified performance measurement system for petascale applications. In: Bischof, C., Hegering, H.G., Nagel, W.E., Wittum, G. (eds.) Competence in High Performance Computing, pp. 85–97 (2010). Springer, Berlin (2012)
21. Mosberger, D., Watson, D., et al.: libunwind documentation (2011). https://www.nongnu.org/libunwind/docs.html
22. Mosberger, D., Watson, D. et al.: The libunwind project (2011). https://www.nongnu.org/libunwind/
23. Shende, S.S., Malony, A.D.: The tau parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006)
24. Stallman, R., Pesch, R., Shebs, S., et al.: Debugging with gdb. Free Softw. Found. **51**, 02110–1301 (2002)
25. Szebenyi, Z., Gamblin, T., Schulz, M., d. Supinski, B.R., Wolf, F., Wylie, B.J.N.: Reconciling sampling and direct instrumentation for unintrusive call-path profiling of mpi programs. In: 2011 IEEE International Parallel Distributed Processing Symposium, pp. 640–651 (2011). https://doi.org/10.1109/IPDPS.2011.67
26. Taylor, I.L.: libbacktrace. https://github.com/ianlancetaylor/libbacktrace
27. Treibig, J., Hager, G., Wellein, G.: Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In: 2010 39th International Conference on Parallel Processing Workshops, pp. 207–216. IEEE (2010)
28. Vetter, J., Chambreau, C.: mpip: Lightweight, scalable mpi profiling (2005)
29. Wagner, M., Knüpfer, A., Nagel, W.E.: Hierarchical memory buffering techniques for an in-memory event tracing extension to the open trace format 2. In: 2013 42nd International Conference on Parallel Processing, pp. 970–976 (2013). https://doi.org/10.1109/ICPP.2013.115