Compressible Memory Data Structures for Event Based Trace Analysis

Andreas Knüpfer Wolfgang E. Nagel

Center for High Performance Computing (ZHR)
TU Dresden, 01162 Dresden, Germany

Abstract

The article presents a new compressible memory data structure for trace events. Its primary intention is to aid the analysis of huge traces by reducing the memory requirements significantly. Furthermore, customized evaluation algorithms reduce the computational effort. The data structure as well as algorithms for construction and evaluation are discussed in detail. Experiments with real-life traces demonstrate the theoretically derived capabilities of the new approach.

Key words: Performance Analysis, Tracing, Data Structures, Compression

1 Introduction

Today, event based program tracing is the key technology for post-mortem performance analysis. Its biggest challenge is the amount of data, especially in the High Performance Computing (HPC) and Massive Parallel Processing (MPP) areas. The evaluation of this huge amount of data tends to become an application for HPC and MPP itself! This is true for automatic as well as for interactive/manual performance analysis.

This article presents an algorithmic aid to this issue by introducing customized compressible memory data structures and appropriate evaluation algorithms. It is part of ongoing research in this field, continuing the work on data compression of trace files [1,2]. While those former studies focused on trace time analysis time is in the center of interest now. Related work is covered in [3].
The article is structured as follows: Section 2 presents the new data structure accompanied by necessary transformation operations. Sections 3 and 4 explain algorithms for the construction of the data structure resp. how to perform the typical analysis queries. Section 5 shows some practical experiments to demonstrate the capabilities of the approach. Finally, Section 6 draws conclusions and gives an outlook on future work.

2 Complete Call Graphs (CCGs)

Traditionally, trace events are stored sequentially in a set of arrays sorted by timestamp much in the same way as in trace files [4]. Random access and iterative traversal are fast and usually binary intersection search (or any $O(\log n)$ search) is applicable. This is a simple and effective data structure but it doesn’t provide an opportunity for optimization. In particular, there is no way to exploit the inherent redundancy of trace data streams thus obtaining data reduction.

Our new approach replaces this data structure by a Complete Call Graph (CCG). In contrast to an ordinary Call Graph [5–7] that summarizes the function calling hierarchy a CCG contains every instance of every function as a node. Thus, a CCG of a program run consist of a Call Graph Tree for every process (see Figure 1(a)). All vital properties of a function call (e.g. the actual function, run-time informations, performance counter values) are annotated in a graph node. Furthermore, there are special node types for atomic events, i. e. events representing non-function-call events like message passing, input/output, etc. These are always leaf nodes.

All properties of all node types are divided into hard and soft properties. Soft properties are continuous attributes that can vary slightly without changing the meaning. Hard properties are attributes that must not be changed in any way since this would corrupt the meaning. The decision if a property is soft or hard depends on whether it is allowed to be subject to lossy compression or not. For example, a function ID, a process ID or the number of child nodes are hard properties. A typical soft property is the list of timestamps associated with a function call. Here, a slight deviation of a few ticks (units of the timer resolution) or a few percent of the run-time would influence a large scale analysis in a negligible way only.

With these definitions, two nodes are called compatible if all hard properties are equal and all soft properties are similar (regarding to any given comparison function. For the actual test of similarity see Appendix A). In particular, this means two compatible nodes are of the same type. Likewise, compatible subtrees of a CCG are subtrees where all corresponding nodes are compatible.
Fig. 1. The left hand side (a) shows a Complete Call Tree as part of a CCG. The right hand side (b) shows the same graph compressed, based on the fact that all four subtrees with function foo as its top node are compatible. Note that the 3rd appearance of function bar has a slightly different run-time but is nevertheless compatible with the others.

The data compression in CCGs is achieved by replacing a set of subtrees with a reference to a single instance they are compatible to (see Figure 1). Maximum compatible subtrees have to be replaced for maximum compression. This depends on the given deviation bound $T$ for soft properties. The details of CCG construction and subtree comparison and replacement are discussed in Section 3.

### 2.1 Timestamp Transformation

In the traditional way run-time information of events are stored as absolute timestamps, i.e. points in time measured in units of the timer resolution. For our new approach this representation is unfavorable. The compression relies on repeated function call sequences (subtrees of the CCG) to be stored only once. Since timestamp values inside a process-trace are monotone increasing there are no repetitive patterns present.

However, a simple transformation of the run-time information is suitable for uncovering the reasonable assumption that identical function call sequences often have similar run-times: Instead of timestamps duration values have to be stored (i.e. difference of successive timestamps) within CCG nodes as already shown in Figure 1. For a node with $n$ children there are $2n + 1$ duration values: $n$ for the child calls, one before the beginning of the first child, $n – 1$ between two successive child calls and one after the last child call.

While the actual transformation is simple (a single subtraction), the decision whether two subtrees are compatible in terms of run-time behavior is more
complicated. It is no longer possible to compare subtrees node by node since timestamps of a graph node are restored as a sum of duration values from parent, grandparent, grand-grandparent, ... etc. nodes. Thereby, deviations in duration values may accumulate arbitrarily, even if the deviation for every single duration value is bounded. Exactly the same applies for most of the common hardware performance counter samples which are another typical example for soft properties.

2.2 Further Optimization

Unfortunately, memory requirements for uncompressed CCGs are higher than needed for the traditional scheme. Of course, this is more than compensated by applying compression. However, some additional optimizations eliminate this overhead and are beneficial for both, uncompressed and compressed CCGs. The key idea is to respect the actual memory requirement for each value of a variable and not the general one. For example duration values need to be stored as 64 bit integers in general. Whereas, most of the time a 32 bit integer is sufficient, even 16 bit or 8 bit integers are often enough. To put this into practice one has to create different implementations (classes) for the same node type with the same interface. According to the actual variable contents it can then be chosen which implementation to use for a node. This is done most comfortably by employing C++ templates and Template Meta Programming (TMP) [8,9].

Without this optimization uncompressed CCGs need approximately twice the memory the traditional scheme would occupy. Now with this improvement, both are similar in terms of memory requirements. At the same time, the memory consumption for uncompressed CCGs is roughly equal to the trace file size (about ±30%). However, this is very much dependent on the trace file format used and the actual trace.

3 Construction of Compressed CCGs

The construction of a compressed CCG (cCCG) consists of two stages: the creation of the uncompressed CCG and the actual compression. In practice, these two steps are not separated but closely integrated such that no intermediate uncompressed CCG needs to be stored. For the sake of easier explanation both steps are discussed separately.
3.1 CCG Construction

Basically, the CCG construction without compression is a replay of the run-time call stack. The construction algorithm reads through the stream of events and acts according to the current one. For three event types there are three different tasks assigned:

- For **function enter events** a new function-call node is created on top-of-stack (TOS), appended as child of the previous TOS node and made wait for its children to be appended. Both, the new node and its parent need to be informed of the timestamp of the enter event.
- For **function leave events** the current TOS node (that was created by the associated enter event) is finalized. From the LIFO scheme of function calls follows that all child calls have already been finished. So, all timestamps of the current node are available and can be transformed into duration values. It also follows that the whole subtree (with the current node as its root) is finished at this point. At last, the new TOS node needs to be informed about the timestamp that its currently last child is leaving.
- For **atomic events** node creation and finalization can be performed at once: a node of suitable type (send node, receive node, I/O node, etc.) is created, there are no children.

3.2 CCG Compression

The compression of a CCG is achieved by replacing maximum compatible subtrees with references to a single instance. Instead of comparing whole subtrees in full depth it is desirable to design a comparison scheme with constant complexity, independent of the size or depth of the subtree. This is possible given that all nodes have no more than $s = constant$ children$^1$.

Every subtree is delivered to the compression facilities right after completion. The basic problem is to find an already existing subtree (pattern) $P$ that is compatible with the current candidate $C$. If such an instance is found, $C$ can be replaced by a reference to $P$ (Figure 2). Otherwise it remains in the cCCG. In order to find matching patterns the following attributes of the root node are examined in this order:

1. node type
2. child nodes
3. hard properties
4. soft properties

$^1$ Section 3.3 explains how this condition can be established.
For two subtrees to match these attributes need to fulfill the following requirements:

First of all, if root nodes must be of same type. Second, all corresponding child nodes need to be equal. This requirement is deduced from the fact that for compatible subtrees all corresponding child subtrees must be compatible too. Since all child subtrees have already passed through the compression facilities and are found to be compatible (with one another or with a third party) they have been replaced by references to the same instance. Therefore, this second test can be implemented as a plain pointer comparison of the child node references. See Figure 2 for the successive replacement of child (b) and parent (c). Third, the hard properties of the root nodes are compared. Only if they are equal, the two nodes can be compatible.

Finally, the soft properties are examined. As mentioned in Section 2.1 the decision, whether all timestamps in the whole subtree are similar, cannot be made on information from the root node only. However, it is sufficient to propagate the deviation information about a subtree upwards to its parent node - most favorable by interval arithmetic (as shown in Figure 2). Now, the test whether two subtrees are compatible in terms of timestamps can be performed by accessing information of the root nodes only (see Appendix B).

The actual search strategy for a replacement of a given node/subtree can be optimized by arranging all previous nodes in a sorted data structure. The sorting order is determined by the first three node attributes, i.e. node type, child nodes and hard properties. Based on this, the search for a set of possible replacements can be done with \(O(\log n)\) effort. Only the elements in this particular set must be compared according to the soft properties.

The degree of compression achieved can be measured in two ways: Counting the number of final graph nodes or comparing the total memory usage. For both the compression ratio \(R\) is defined as the uncompressed value divided by the compressed value. The compression ratio according to the graph node count is defined as:

\[
R_{\text{nodes}} := \frac{N}{n} = \frac{\text{nodes in uncompressed CCG}}{\text{nodes in cCG}}, \quad n < N. \tag{1}
\]

The compression ratio according to memory usage is defined as:

\[
R_{\text{memory}} := \frac{M}{m} = \frac{\text{memory for uncompressed CCG}}{\text{memory for cCG}}, \quad m < M. \tag{2}
\]
Fig. 2. Successive compression in a CCG. The first graph (a) shows the uncompressed subtree. After the finalization of node bar* it is found to be compatible with node bar and replaced by a reference (b). Note the slight difference in run-times: the deviation arising from this is propagated to the parent node as a deviation interval [-1,0]. After finalizing foo* it is replaced by a reference to foo as both reference the same child node (c).

Fig. 3. The oversized node main has four children (a) although not more than \( s = 3 \) are allowed. So, main is split up into a balanced tree of smaller nodes (b).

3.3 Further Optimization

Although, a trace can have any number of child calls per function it would be of advantage to bound their number to a maximum of \( s \geq 2 \). This criterion can be enforced by introducing artificial nodes, converting an oversized node into a balanced tree of multiple nodes (see Figure 3). This policy pays off when comparing nodes for compatibility, on search operations in CCGs\(^2\) and for compression itself\(^3\).

Given that every node is copied into a new object during finalization\(^4\) the general memory management scheme can be replaced by a simpler one. If

\(^2\) A balanced tree with a lower branching factor can be searched faster.

\(^3\) Shorter nodes are more likely to be compatible because every additional piece of data is a potential source of incompatibility. Two incompatible nodes might share some smaller compatible parts once they are split up.

\(^4\) This is desirable because not all data used during stack replay are needed in the final CCG (see Section 3.2: propagating deviation intervals.).
a newly created node is not permanent (i.e., to be replaced by a reference to another node) then it will be deleted right away, before the next node is created. Thus, it is possible to arrange the node objects consecutively in a pre-allocated memory area. This minimizes the number of memory allocation operations from $O(n)$ to $O(1)$, reducing overhead for memory allocation significantly. This applies especially on platforms without advanced memory management facilities [10].

4 Trace Data Analysis

The main purpose of collecting traces is the post-mortem analysis of a program’s run-time behavior. In a very coarse classification there are three types of analysis algorithms applied to the data. All of them will benefit from a much smaller memory consumption\(^5\)

- Statistic evaluation and summarization.
- Search for events or time intervals, iterative traversing of events.
- Visualization of events and their relationships.

**Statistic evaluations** are important for interactive performance investigations as well as for automatic performance analysis. Generally, this involves all data that can be gathered by simple profiling [5,11]. In addition to profiling with tracing these statistics can be calculated for a single process/thread or any desired interval of time. This kind of algorithms is subject to further improvements as described below. **Search and traverse algorithms** are employed primarily for automatic performance analysis [12]. There are $O(\log n)$ search tasks (when searching for timestamps) and $O(n)$ search tasks (when searching for events with particular features). The latter tasks can be accelerated in a fashion equal to the statistic summarizing - however, this is not covered here for conciseness. **Visualization algorithms** are essential for interactive performance investigation and debugging [13,14]. Internally, they utilize the search and traverse algorithms. However, this topic would go beyond the scope of this article. So, this whole matter is avoided here by the statement that there is no disadvantage of using CCGs instead of traditional data structures since the traditional algorithms are applicable too.

Now, the main focus is directed towards the statistic evaluation algorithms. All statistics are cumulative, i.e., subtree statistics can be accumulated from statistics of child-subtrees (plus properties of the current node). When calculating statistics in this way for the cCCGs root nodes one can take advantage

\(^5\) Especially if compression makes the data fit into main memory when it would not otherwise.
of multiple referenced subtrees, i.e. computing intermediate results for every node only once and re-using it later on. While this requires some additional memory for intermediate results it provides a reduction in computational effort in the same order of magnitude as the nodes compression ratio $R_{nodes}$.

### 4.1 Further Optimization

For very small compression ratios $R_{nodes}$ the cumulative way of computing statistics may be slower than an algorithm traversing the trace in a linear way. An unnecessary penalty can be avoided by a simple heuristic decision. It will apply the more promising of the algorithms according to the actual compression ratio $R_{nodes} < R^*$ with $R^*$ determined experimentally.

### 5 Experimental Results

For the experiments demonstrating the capabilities of cCCGs four different traces were processed. The first one is a sequential trace of a Navier-Stokes solver used in the area of Computational Fluid Dynamics (CFD) [15]. The trace file has a size of 22 MB containing about 1 million events. The second one is a trace of a linear systems solver using a semicoarsening multigrid algorithm (SMG) [16,17]. This is a benchmark of ASCI/LLNL. The trace file has a size of 20 MB, contains 4 processes and has about 1.7 million events. The third example is a sequential trace file from a Nonlinear Optimization Algorithm (FMC) [18] with a size of 45 MB and 1.1 million events. The last one is a trace from the Implicit Radiation Solver (IRS) [19] run with 8 MPI processes. This is another benchmark code from LLNL. Containing 177 million events and having a size of 2 GB this example is a real challenge for analysis.

#### 5.1 cCCG Construction

For the construction process two characteristics are critical: the compression ratios and the run-time. The compression ratios for the test traces are plotted in Figures 4 and 5. Figure 4 shows $R_{nodes}$ (a) and $R_{memory}$ (b) depending on the absolute deviation bound for timestamps $T_{abs}$. Figure 5 shows the same depending on the relative deviation bound for duration values $T_{rel}$. The

---

6 Advanced Simulation and Computing Initiative at Lawrence Livermore National Laboratory
common perception from all test cases is that wider deviation bounds yield larger compression ratios $R$. This is no strict correlation but very likely.

Furthermore, the actual compression ratio obtained depends on the regularity of the trace. More regular function call patterns and more uniform run-time behavior will obviously result in better compression. So, the test cases (CFD) and (FMC) behave much better than the (SMG) example (Figures 4 and 5). Likewise the rate of increase in $R$ for expanded deviation bounds is depending on characteristics of the trace. The (CFD) and (FMC) examples reveal a substantial growth in $R$ for larger $T_{abs}$ resp. $T_{rel}$: (CFD) sooner and (FMC) later. The (SMG) example shows only small increase in $R$ (Figures 4 and 5).

The (IRS) trace (Figure 4) is the only example for which our new approach is really necessary. With 2 GB it could neither be loaded into main memory in the traditionally way nor compressed with very small deviation bounds. With larger deviation bounds $R_{nodes} = 34...93$ and $R_{memory} = 21...44$ are reached. That is a reduction from 89,182,738 to 956,720 graph nodes and 2104 MB of

---

$^7$ Due to the computational complexity and temporary memory requirements no optimal clustering can be performed for nodes differing only in soft properties.
Fig. 6. Run-time of cCCG construction depending on actual compression ratio for nodes (a) and memory (b). (The IRS example has a much higher run-time.)

(hypothetical) memory usage to 48 MB.

Altogether, compression ratios $R_{\text{nodes}} = 5...80$ and more and $R_{\text{memory}} = 2...44$ can be observed. Even values produced with $T_{\text{abs}} = T_{\text{rel}} = 0$ range over $R_{\text{nodes}} = 5..14$ and $R_{\text{memory}} = 2..8$. This means that, it is possible to obtain considerable compression without accepting any deviation in soft properties. Another observation from all test cases is that $R_{\text{memory}} < R_{\text{nodes}}$. This is due to the fact that a replaced node is count as 0 instead of 1 while the memory consumption is only decreased from $\text{sizeof(Node)}$ to $\text{sizeof(Node*)}$.

Finally, the Figures 4 and 5 show the experiments for (SMG) in two flavors with the split parameters $s = 10$ and $s = 30$. The influence of a larger $s$ on $R_{\text{nodes}}$ and $R_{\text{memory}}$ is: $R_{\text{nodes}}$ increases, $R_{\text{memory}}$ decreases. The behavior of $R_{\text{memory}}$ is the expected one (see Section 3.3). The effect on $R_{\text{nodes}}$ arises from the fact that the newly inserted artificial subtree is deeper for smaller $s$.

The run-time consumed by cCCG construction is plotted in Figure 6 vs. $R_{\text{nodes}}$ (a) and vs. $R_{\text{memory}}$ (b). Both plots are very similar and show that an larger compression ratio accelerates the construction process. This implies that wider deviation bounds $T_{\text{abs}}$ and $T_{\text{rel}}$ reduce the construction effort. Furthermore, the split parameter $s$ has no impact on the construction run-time as the plots for the (SMG) examples show.

### 5.2 Statistics Evaluation

As one example of evaluation algorithms working on cCCGs the computation of statistics is examined. The matter of interest is the errors in result values due to the allowance of deviations in soft properties and the run-time behavior.

---

8 A memory object is replaced by a reference which still requires some memory.

9 This can be reproduced easily with a toy example like in Figure 3.
Fig. 7. Errors and speed of statistics evaluation: (a) shows the actual relative errors in exclusive run-times of some functions vs. $T_{rel}$, (b) shows the run-time of the calculation algorithm in comparison to the traditional one as used by VAMPIR. For both experiments the (CFD) example was used.

The resulting values derived from hard properties only are never subject to deviations. For example the computation of occurrences of functions is always correct regardless of the compression. Result values basing on soft properties may introduce certain errors. In order to be useful it is important that these deviations are bounded which they are. As an example the computation of exclusive run-times per function\textsuperscript{10} is analyzed (see Figure 7). The exclusive run-time of functions is evaluated as sum over duration values. Since the deviation for every duration value is bounded by $T_{rel}$ the result is as well. As Figure 7(a) shows the actual relative error is often much smaller than the input deviation bound due to the probability distribution of the deviations.

Figure 7(b) gives the run-time of the calculation algorithm which decreases for wider deviation bounds, i.e. higher compression ratios. The time ranges from 0.64s with deviation bound 0 ($R_{nodes} = 7.5$) to 0.12s with deviation bound 100% ($R_{nodes} = 42$). As comparison the constant run-time of 1.4s for a traditional algorithm working on a flat data structure is shown - taken from the implementation in VAMPIR [13].

6 Conclusion and Future Work

The proposed data structure for trace events reduces memory usage significantly as the most important advantage over the traditional scheme. This makes much larger traces accessible by automatic and interactive performance analysis techniques. Besides this, the new method enables some advanced evaluation algorithms that decrease the computational effort in the same order of magnitude as the memory reduction. These advantages become possible by exploiting redundancies present in the traces and by allowing certain deviations

\textsuperscript{10} Sum of run-times over all occurrences of a function excluding all child calls.
in selectable elements of the data structure. These deviations are controllable by absolute and relative bounds. Limits for errors in computed results can be derived from the input deviation bounds. The actual compression capabilities as well as the acceleration of evaluation algorithms have been demonstrated with real-life examples.

There are several subjects of further research. While a trace can be compressed with given deviation bounds, another desirable scenario would be to make a trace fit into a given amount of memory. Thus, the actual deviation bounds are to be determined adaptively. This leads to the problem of re-processing an already established cCCG.

Furthermore, ways of storing a cCCG persistently have to be explored. Here, the challenge is to find such parameters or facilities that only a minimum of re-computation is necessary on the re-creation with the actual deviation bounds still controllable. The next major step in program development will be the integration into the VNG [20,21] analysis engine which is the software design complement to this algorithmic work.

A Test for Similarity of Timestamps

Let \((t^A_0, ..., t^A_{2n+1})\) and \((t^B_0, ..., t^B_{2n+1})\) be the timestamps of function calls \(A\) and \(B\) with \(n\) child calls each. Then \((d^X_0, ..., d^X_{2n})\) with \(d^X_i := t^X_{i+1} - t^X_i\) \((X \in \{A, B\})\) are the associated duration values. The test for similarity \(A \sim B\) is:

\[
A \sim B :\Leftrightarrow \left[ |t^A_i - t^B_i| \leq T_{abs} \right]_{i=0}^{2n+1} \land \left[ \left| 1 - \frac{d^A_i - d^B_i}{d^B_i} \right| \leq T_{rel} \right]_{i=0}^{2n} \tag{A.1}
\]

with a non-negative integer \(T_{abs}\) (in ticks) that defines the absolute deviation bound and a non-negative relative deviation bound \(T_{rel}\) (in %).

B Estimating Timestamp deviations based on Duration Values

Let \(A\) and \(B\) be graph nodes with \(n\) children each. Let \((d^A_0, ..., d^A_{2n})\) resp. \((d^B_0, ..., d^B_{2n})\), \(d^X_i \in \mathbb{R}\) be the \(2n+1\) duration values of \(A\) resp. \(B\) and \((e^A_1, ..., e^A_n)\) resp. \((e^B_1, ..., e^B_n)\), \(e^A_i, e^B_i \in \mathbb{R}\) the deviation intervals of the subtrees. Then

\[
t^X_{i=0;..2n+1} = \sum_{j=0}^{i-1} d^X_j, \; X \in \{A, B\} \tag{B.1}
\]
are the timestamps of the nodes and

\[ s_{i=0..2n+1} = \sum_{j=0}^{i-1} (d^A_j - d^B_j) = t^A_{i=0..2n+1} - t^B_{i=0..2n+1} \] (B.2)

are the deviations between the duration values of A and B. Now, when node B is to be replaced by node A the new deviation interval \( E \in \mathbb{R} \) is derived from the timestamp deviations and the intervals of child deviation for B:

\[ E = \bigcup_{i=0}^{2n+1} s_i \cup \bigcup_{i=1}^{n} (s_i + e^B_i) \] (B.3)

Note that there is always \( 0 \in E \) since \( s_0 \equiv 0 \).

References


URL http://www.llnl.gov/asci/applications/SMG98README.html


URL http://www.llnl.gov/asci/purple/benchmarks/limited/irs/


15