Exploiting Spatiotemporal Locality for Fast Call Stack Traversal

O. Perks, R.F. Bird, D.A. Beckingsale and S.A. Jarvis
Performace Computing and Visualisation
Department of Computer Science
University of Warwick, UK
{ofjp, rfb, dab, saj}@dcs.warwick.ac.uk

ABSTRACT

In the approach to exascale, scalable tools are becoming increasingly necessary to support parallel applications. Evaluating an application's call stack is a vital technique for a wide variety of profilers and debuggers, and can create a significant performance overhead. In this paper we present a heuristic technique to reduce the overhead of frequent call stack evaluations. We use this technique to estimate the similarity between successive call stacks, removing the need for full call stack traversal and eliminating a significant portion of the performance overhead. We demonstrate this technique applied to a parallel memory tracing toolkit, `WMTools`, and analyse the performance gains and accuracy.

Keywords

Call Stack, Unwind, Context Calling Tree, Memory, Multicore, Tracing

1. INTRODUCTION

As we transition from the petascale to the exascale generation of supercomputers, obtaining raw performance measured in FLOPS, is just one of the problems facing high-performance computing institutions. Application developers are finding it increasingly challenging to scale their applications in order to fully exploit new platforms and architectures. Tools which support the analysis and maintenance of large parallel applications at scale are crucial in supporting code custodians. Unfortunately, the scalability issues facing parallel applications also impact the tools designed to support them. As such, the development of exascale tools is a key area of research.

Evaluating an application's call stack is a vital technique for a wide variety of profilers and debuggers, and can represent a significant performance overhead. The call stack represents the specific sequence of function calls made by an application to arrive at the current state, and tracking changes in the call stack allows tools to monitor the progression of the application. These call stack changes are represented as transitions within the context calling tree (CCT). The analysis afforded by detailed progression monitoring is a crucial component in a multitude of debuggers, event tracers and performance profilers. On many architectures, obtaining this information is an inherently expensive operation, due to the layout of the call stack in memory. We present a heuristic approach to call stack traversal to reduce this overhead. Through overhead reduction we aim to make frequent call stack traversal a more affordable technique, facilitating fine grained analysis in scalable tools.

Whilst alone this technique does not increase the scalability of parallel tools it can reduce the impact of existing tools, increasing their viability at scale. Currently debugging errors in large scale applications can be a multi-stage process, of first identifying a problem case then attempting to replicate the error conditions at a more manageable scale, where 'heavyweight' tools can be applied. By reducing the overhead of a key component in such heavyweight tools, we are enabling their application at a larger scale.

Our technique is based upon the exploitation of the spatiotemporal temporal locality of calling sites. We employ a heuristic technique to predict the overlap between successive call stacks during traversal. The overlap between two call stacks manifests itself in the form of a shared prefix, representing a shared path in the CCT from the root to the point of divergence. The closer two locations are within the CCT, the larger the shared prefix, and therefore, the larger the performance gain afforded by applying the heuristic. The heuristic uses a number of markers to determine this point of divergence, but can not guarantee the correct inference, and therefore, the validity of the assumed shared prefix. Using additional correctness checks we are able to demonstrate a high degree of accuracy of the heuristic (>90%) in a number of cases, whilst still providing a significant improvement in performance.

The technique is presented in the context of `WMTools`, a lightweight memory tracing toolkit developed at the University of Warwick [7]. The tool uses function interposition of standard POSIX memory management functions to track memory allocations. In previously published literature we have demonstrated the overheads of memory tracing, for example up to an 11.8× slowdown for `phdMesh` during a 256-core run. We attributed a large portion of this overhead to the cost of stack traversal, thus reductions in overheads

would enable the viability of memory tracing at larger scale.

In previously published material we have illustrated that the performance overheads of memory tracing different applications vary significantly. In this paper we demonstrate that the overheads associated with memory tracing are directly related to the volume of memory management calls combined with the depth of the call stack. We propose a metric, traversal event density, to capture the likely impact of call stack traversal on an application. We demonstrate the validity of this metric, and investigate it's behaviour as core count and problem size is changed. By employing a new technique which exploits the similarity in successive call stacks we coerce these previously limiting factors into helping improve the performance of call stack traversal, whilst maintaining a high level of accuracy.

Our heuristic is based on a very simple principle: the higher the number of call stack traversal events within a given period of time (high density), the closer these calls will be in the CCT. This in-turn suggest a large overlap between successive calls, and therefore a large potential saving.

The stack traversal functionality of `WMTools` is provided by `libunwind` through frame pointers, and as such will be the traversal method of choice for our analysis. Whilst we have explored the use of other traversal libraries and methods, the portability and speed of `libunwind` were found to be preferable. Additionally experiments with function prologue and epilogue instrumentation provided poor performance and in many cases an inability to instrument external libraries, such as the Message Passing Interface (MPI). Similarly we focus on the `x86_64` architecture, and do not compare the technique on other architectures with different call stack structures.

The specific contributions of this paper are as follows:

- We present a heuristic approach to call stack traversal exploiting the spatiotemporal locality of successive calls, in applications with a heavy use of call stack traversals;
- We motivate the technique by first quantifying the severity of the problem, and investigating other techniques applicable to this domain. We demonstrate how this heuristic technique can be used to reduce call stack traversal overheads in debugging and tracing tools;
- We evaluate the accuracy of the technique under different circumstances, illustrating additional techniques which can be employed to mitigate false predictions.

The remainder of this paper is structured as follows: in Section 2 we catalogue the state of practice within the analysis tool domain, with specific reference to call stack traversal; Section 3 investigates the gravity of the problem, with a practical investigation based on a parallel memory tracing tool; Section 4 illustrates the heuristic traversal technique and existing alternatives within the context of this study. Section 5 demonstrates a performance comparison of these techniques, while Section 6 investigates the notion of trading off accuracy and performance. Finally, we conclude the paper in Section 7 and document further work.

2. RELATED WORK

Call stack tracing plays a crucial role in debuggers and tracers of all shapes and sizes and the ability to analyse program location mid-execution enables a whole range of analysis, techniques, from 'hot spot' analysis to backtracking from abnormal termination. Due to the importance of this feature, it has been the focus of a large body of research, specifically aimed at speeding up the inherently slow backtracking process.

In a recent study, Mytkowicz et al. discuss a technique similar to the one we present in this paper [6]. They propose the evaluation of the first element in the call stack along with the call stack size, then relate this back to a previously generated call graph. They deal with ambiguous mappings through a process they term Activation Record Resizing (ARR), where by they modify the size of some call functions to increase unique mappings. This method is shown to work with an accuracy of 88% (95% for `C` and 80% for `C++`) for offline calculation with pre computation of all context mappings, and 74% (73% for `C` and 75% for `C++`) for offline calculation, with a partial mapping. For this accuracy they incur a minimal runtime cost due to limited intrusion reported as a mean of 0.17% and maximum of 2.1%. A requirement for this methodology is the pre-computation of the call graph; this is performed by dynamically constructing path maps which require offline 'training' runs, thus incurring the full cost of traversal for a single run. As training runs may not produce a comprehensive call graph, because the tool is based on sampling their approach suffered from 'missing' mappings.

Whilst the technique they presented is obviously efficient under certain circumstances, they fail to exploit spatiotemporal locality in the disambiguation of mappings. They also incur the cost of searching all mapped call paths, whereas the technique presented in this paper only focuses on comparison with the previous call stack.

There is a body of research identifying methods of negating the necessity to traverse the full call path by evaluating existing information, a key principle in the methodology presented in this paper. Whaley suggested a technique of masking the return address with an identifier bit to signal that the call frame had not been removed from the call stack [13]. This method suffers ambiguity problems from cycles where a function features multiple times in a previously traversed call stack, making it difficult to identify which instance of the function is the crossover between the new suffix and previous prefix. Despite this he claims to achieve >90% accuracy. Based on sampling, he also demonstrates the effect of increased sampling rates on the accuracy of the predictions for various benchmark suites on different platforms. This relates to the concept of traversal density presented in Section 3.4.

This technique was further elaborated on using the concept of 'trampoline' functions, which remap the return address of a function to a new function which is able to record information, and manage a 'shadow stack' which can quickly be queried to establish the exact call path prefix [3].

Szebenyi et al. developed a suite for mixed mode instrumentation incorporating a hybrid sampling and event driven tracing model designed for `MPI` applications [11]. To address the problem of call path traversal overheads they expand on the concept of trampoline functions, in the form of lightweight 'thunk stacks', which are inserted into the return address of functions. This technique enables the stack walker to identify previously explored stack frames, thus establishing the prefix.

Additional research has been conducted into other aspects of call stack traversal, with specific reference to representation and storage from the perspective of call stack depth [10].

Serrano and Zhuang present a methodology of approximating the CCT through reconstruction of partial call traces [9]. This techniques uses call stack suffixes and approximates the point of overlap to join them together to form a CCT. This is very similar to the technique we employ in this paper, but applied to a different problem domain. The difference between the two techniques is that we only consider the immediately preceding call stack, and we have the information to the root of the CCT and so can better approximate the shared prefix.

3. CALL STACK TRAVERSAL OVERHEADS

Call stack traversal is the process of interrupting a running process to establish the sequence of function calls which brought the code to its current location. This information is stored on the program stack, which can easily be traversed by looking at the return pointers for each function to establish the next function in the sequence.

The `libunwind` library is considered the industry standard for call stack traversals, due to its portability and accessible nature. While it already provides a 'fast unwind' facility, it is inherently a slow operation which, if performed a sufficient number of times, will have a significant performance impact on a code. In this section we quantify the frequency and overheads of this traversal within the context of a parallel memory tracer, `WMTrace`.

3.1 WMTrace

`WMTrace` is a memory tracing tool, designed to perform interposition on standard posix memory management functions to evaluate total heap memory consumption, as part of the `WMTools` tool suite [7]. The software is designed to evaluate parallel MPI-based applications.

Beyond simplistic memory consumption analysis `WMTrace` provides functional and temporal breakdowns of memory usage. A key component of this is relating memory allocations to a specific function, more specifically a unique function call path. As such the tool needs to perform a call stack traversal on every memory allocation, mapped as `malloc` or `calloc` posix function calls.

In our previous research we demonstrated that profiling a parallel code with `WMTrace` can have a significant impact of application execution time. These slowdown were shown to result in instrumented execution times of up to 11.8× the original execution time. Whilst the performance in many cases is more than acceptable there are many instances where the overheads are too significant to be ignored. Specifically, we demonstrated significant application slowdowns during the profiling of `phdMesh` on all core counts and `miniFE` on core counts above 64. These cases will largely form the basis of this paper.

Table 1: Demonstrator applications and benchmarks used in our analysis

	Language	Description
miniFE[5]	C++	Unstructured finite element solver
phdMesh[5]	C++	Unstructured mesh contact search
AMG[4]	C	Parallel algebraic multigrid solver
LAMMPS[8]	C++	Classical molecular dynamics
LU[2]	Fortran 90	LU PDE solver
FT[2]	Fortran 90	FFT PDE solver
Graph500[1]	C	Graph solver
XML Parse Lib[12]	C	XML parser - serial

3.2 Applications/Benchmarks

We employ the use of `WMTrace` on a variety of parallel applications and benchmarks, illustrated in Table 1, representing a cross section of languages and scientific disciplines, representing the interests of various high performance computing laboratories. All of these applications utilise the Message Passing Interface (MPI) to achieve parallel execution. During this paper we apply a heavy emphasis to HPC codes, as they predominantly form the context to surrounding research, though we make an effort to demonstrate the effects for various scientific domains. As previously discusses the technique presented is applicable to different types of code, though we have not performed a wider study and thus can not comment on performance.

We note that for all the benchmarks a representative problem set has been used to allow for scaling, and the memory and time limits of the available compute resource.

As `XML Parse Lib` is not multithreaded we instead run a single instance of the problem based on the parsing of a 168 MB XML file.

3.3 Testing Platform

The results presented in this paper are based on the Minerva cluster, located at the Centre for Scientific Computing (CSC) at the University of Warwick. This machine comprises 258 dual-socket, hex-core Intel Westmere-EP X5650, nodes connected via QDR-Infiniband. Each node provides 24GB of system memory (2GB per core). The runs presented all use the Intel 11.1 compiler toolkit with OpenMPI 1.4.3, compiled with the `-O3` optimisation flag and debugging symbols.

3.4 Traversal Frequency

Initially to demonstrate the intensity of traversal we present findings on the number of instances an application requires a traversal within a given run. For this we will look at a selection of different codes which formed the basis of previous studies; these codes represent a cross section of tool performance and programming languages. For each code we indicate how many calls were made and the duration of un-instrumented runtime to calculate the traversal density.

This will form a basic metric to evaluate the susceptibility of an application to improved call stack traversal.

Whilst the techniques discussed in this paper are applicable to all situations with a high density of call stack tracing, we focus on parallel applications and specifically the strong scaling of applications. Strong scaling is the process of solving the same problem size on an increased number of processors, thus reducing the per core computation. In the context of memory tracing this is crucial, as in general the number of memory allocations does not reduce at the same rate as the computation, thus the density of call stack traversals increases.

Whilst this metric is demonstrated with memory allocations as the trigger point, it is relevant to all areas where the trigger point can be identified - for example MPI functions for a communication tracing tool. Analysis of this metric during scaling provides a very useful insight into the potential overheads of tracing with call stack traversal.

The technique could also be applied to interval-driven sampling although the results will differ from event-driven tracing, due to natural clustering of events. Using a time interval to trigger call stack traversal will naturally create an even distribution of tracing events throughout the execution. The technique applied in this paper exploits the natural grouping, both in time and location in the CCT to benefit from large shared prefixes. Time based events are unlikely to generate call stacks with naturally large shared prefixes due to the movement in the CCT within that time, thus will not likely benefit from this technique. Other forms of trigger events, such as different function calls or hardware signals would likely share the same clustering effects as the memory management functions we demonstrate in this paper. As these clusters will have a naturally higher traversal density than the code average, we should experience improved performance and accuracy within them. We expand upon this clustering in Section 3.5, where we explore the temporal distribution of allocation evens within different code.

From Table 2 we can clearly see that call stack traversal density varies between different codes. We also verify our previous assertion that the density tends to increase as a code is strong scaled, thus adversely affecting the performance of any such analysis tool.

With memory tracing, the density tends to increases during strong scaling due to a reduction in computation (and therefore runtime), not an increase in the number of allocations. Thus the impact of this density change will have minimal impact on the accuracy of the heuristic. This is due to the fact that the spatial locality, within the CCT, between successive events has not changed, but the time between the calls has decreased. Similarly the time saved by the heuristic will not vary with any significance during scaling, but rather as the un-instrumented times fall this time saving will represent an increased percentage of instrumentation time, thus a larger impact.

In the case of `WMTrace`, which generates trace files based on the events, the impact of this trend is magnified. As the volume of data generated is directly proportional to the number of traced events, the increased density will result in an increased I/O transmission density, which in turn will contribute to an overall slowdown, but is not the focus of this study.

3.5 Traversal Temporal Locality

In this Section we further motivate the cause for a heuristic traversal technique by analysing the distribution of traversal events within normal execution. For this we measure the time at which an allocation event occurs, and visually represent the distribution of memory management calls against time. This representation method allows us to ascertain more information than simple allocation density, as discussed previously, as it allows us to visualise allocation clustering. Whilst our example is based around traversal events caused by memory events, the technique can be replicated for other trigger events. For example one would expect similar clustering from the interception of MPI events.

Figure 1 illustrates the distribution of events for `miniFE` and `phdMesh` for single core runs on a reduced data set, showing the clustering of memory allocation events. From Table 2 we can see that these two codes have significantly different allocation densities; 1220 events / second and 897747 events / second respectively. In the case of `miniFE`, Figure 1(a), we see a very natural clustering around five points in the execution, with varying allocation size. For `phdMesh`, Figure 1(b), we see a very different pattern. Due to the excessively high allocation density there is no real area of clustered allocations. In part this is due to many objects being allocated on a per iteration basis within the code, thus there is a constantly high number of allocations. In both cases there are areas of high density; in `miniFE` they are clustered, and in `phdMesh` they are global, this suggests that potential improvements could be made from the heuristic technique.

As discussed previously, the clustering of traversal events generates small regions of high density, which improves the performance and accuracy of the heuristic. Whilst the technique is still applicable to sampling-driven tracing, we lose this clustering, and thus do not benefit from localised areas of high density. To see a significant benefit from the heuristic technique applied to a sampling based tracer, you would need a very small sampling interval, to keep the traversals as close as possible. Alternatively sampling on hardware events rather than time intervals would potentially provide this natural clustering.

4. HEURISTIC TRAVERSAL

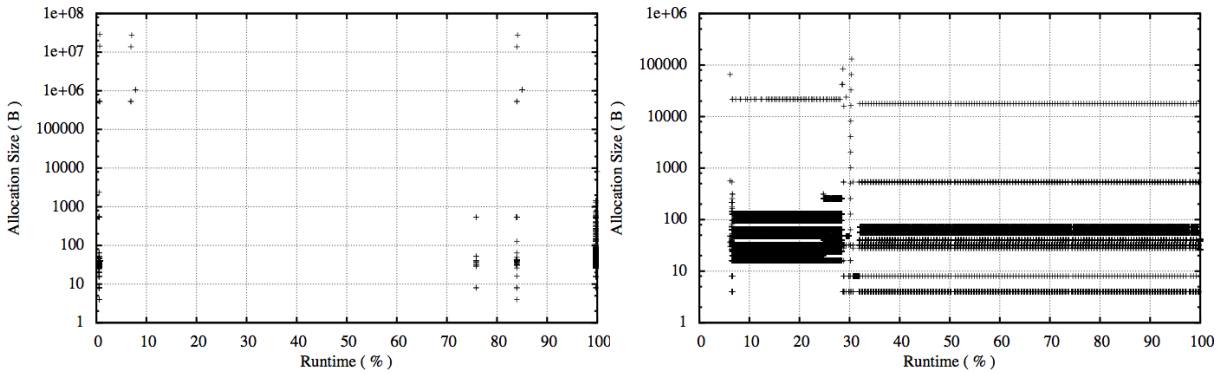
The principal of heuristic traversal is that the higher the density of call stack traversal trigger points, the shorter the probable distance travelled in the CCT between successive events.

The implication of this is that sequential events will share a significant portion of their call stack. Due to the nature of transitions within the call path tree, differences will occur at the top of the call stack. We exploit this similarity to alleviate the need to fully traverse the call stack of the second event, by ascertaining the point of overlap and assuming similarity from there on.

To identify the point of intersection between two succes-

Table 2: Call stack traversal density represented as the average number of allocations (`Malloc` or `Calloc`) per second of un-instrumented execution

	Core Count					
	1	**2**	**4**	**8**	**16**	**32**
miniFE	$1.2x10^{3}$	$3.7x10^{3}$	$1.8x10^{3}$	$6.4x10^{3}$	$1.2x10^{4}$	$1.5x10^{4}$
phdMesh	$9.0x10^{5}$	$2.7x10^{5}$	$6.4x10^{5}$	$1.1x10^{6}$	$1.1x10^{6}$	$8.6x10^{5}$
AMG	$2.1x10^{5}$	$1.1x10^{5}$	$7.8x10^{4}$	$5.1x10^{4}$	$3.7x10^{4}$	$3.3x10^{4}$
LAMMPS	$6.5x10^{0}$	$1.6x10^{3}$	$2.7x10^{3}$	$4.1x10^{3}$	$4.5x10^{3}$	$6.4x10^{3}$
LU	$1.0x10^{-2}$	$4.6x10^{-1}$	$1.2x10^{0}$	$3.4x10^{0}$	$9.2x10^{0}$	$1.4x10^{1}$
FT	$6.1x10^{-1}$	$2.2x10^{0}$	$3.0x10^{0}$	$1.1x10^{1}$	$2.8x10^{1}$	$4.1x10^{1}$
Graph500	$7.7x10^{1}$	$9.5x10^{1}$	$1.9x10^{2}$	$7.1x10^{2}$	$2.8x10^{3}$	$2.9x10^{3}$
XML Parse Lib	$1.6x10^{6}$	-	-	-	-	-



(a) Allocation distribution for `miniFE`

(b) Allocation distribution for `phdMesh`

Figure 1: Traversal event distribution on `miniFE` and `phdMesh` - serial execution

sive call stacks we perform a comparison of the 'instruction pointer' (also known as the 'program counter') and the 'stack pointer' for each function in the call stack. The instruction pointer refers to the function address, which can be resolved to a function name, and the stack pointer indicates the top of the stack, for that function, which can be used to simulate stack size.

Thus when the same function appears in both stack traces, with an identical stack size, then it is very probable that the remainder of the stack (the prefix) has not changed, and thus can be assumed from the previous entries. Whilst this assumption is not applicable in every case it still results in a high level of accuracy and a significant performance increase in many cases.

Figure 2 illustrates the circumstance where a partial traversal of two sequential call stacks can result in a reduction of traversal depth. Both call stacks share a common prefix, `Function A -> Function B -> Function C`. Our aim is to identify this commonality at the earliest possible point in this example from the inspection of `Function C`. To establish that the call stacks are the same we look at both the current entry and the stack height. This enables us to eliminate false positives like `Function F`, which appears at the top of both call stacks, thus perhaps indicating a common prefix below. In this case it is clear to see that the depth at this point, shown as X_1 and X_2 are not the same, thus there is not a shared prefix at that point. Whereas in the correct case of `Function C`, we can clearly see that Y_1 and Y_2 are the same, thus correctly identifying our shared prefix.

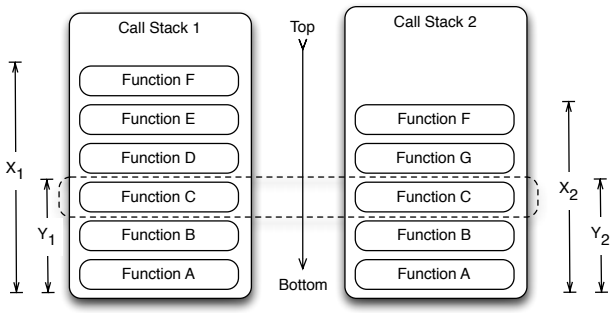


Figure 2: Call stack comparison for partial reconstruction

In Figure 3 we demonstrate the case of a false positive where both of our two metrics indicate success, but our key assumption is incorrect. In this case, `Function E` features in both call stacks at the same height, which would lead us to believe that the remaining prefix is shared. This assumption is incorrect, as whilst the two prefixes are similar they contain a mismatched entry, `Function D` in call stack 1 and `Function G` in call stack 2. The incorrect assumption was based on the height of the two stacks at `Function E` being the same, which in turn implies that `Function D` and `Function G` are the same size.

For this circumstance to have occurred the program must have unwound the call stack to `Function C`, then called

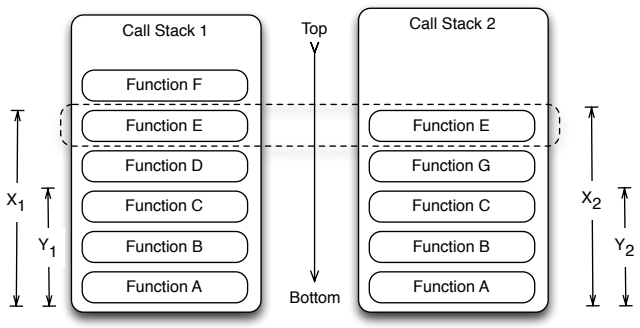


Figure 3: Call stack comparison for partial reconstruction with false positive

Table 3: Overhead analysis of full stack traversal - Slowdown relative to no stack traversal

	Core Count					
	1	**2**	**4**	**8**	**16**	**32**
miniFE	1.00	1.06	1.00	1.03	1.07	1.00
phdMesh	3.51	2.21	2.85	2.98	2.72	2.46
AMG	1.85	1.48	1.30	1.19	1.11	1.05
LAMMPS	1.00	1.02	1.04	1.05	1.05	1.05
LU	1.00	1.00	1.03	1.00	1.03	1.00
FT	1.19	1.00	1.02	1.02	1.46	2.37
Graph500	1.00	1.00	1.00	1.01	1.00	1.00
XML Parse Lib	2.89	-	-	-	-	-

`Function G` which in turn called `Function E`. The probability of such a situation occurring is based on the time between successive traversals, as the smaller the time interval the smaller the distance travelled in the CCT.

In the above example we made use of a single function difference, but the reality is slightly more complex. The method is not only compromised by a single function mismatch, but also when the sum of the stack sizes of the differing functions are the same in both call stacks. This can mean that the same functions in a different order, will cause a false positive.

In the example above referring to functions differing is a further oversimplification. A difference in two call stacks may indicate a difference between two lines within the same function, rather than the function itself. As we resolve function pointers to return addresses, as apposed to function names, we do not distinguish between errors made at a line level or a function level.

If the stack size is not changed between successive traversals, then our technique may assume the calls occurred from the same point within the function. Whilst this is an incorrect prediction - we rarely analyse call stacks at line level granularity, thus it is of minimal concern, but is still included in our evaluation metric as a false prediction.

There are techniques to mitigate these false positives for increased accuracy, but at the cost of some performance, and this trade-off is discussed in more detail in Section 6.

5. PERFORMANCE AND OVERHEADS

Table 4: Overhead analysis of heuristic stack traversal - Slowdown relative to no stack traversal

	Core Count					
	1	**2**	**4**	**8**	**16**	**32**
miniFE	1.00	1.00	1.01	1.00	1.01	1.00
phdMesh	1.62	1.33	1.49	1.55	1.47	1.39
AMG	1.23	1.13	1.05	1.04	1.00	1.07
LAMMPS	1.00	1.00	1.00	1.00	1.00	1.00
LU	1.00	1.01	1.10	1.00	1.00	1.00
FT	1.00	1.00	1.00	1.00	1.00	1.00
Graph500	1.00	1.00	1.00	1.00	1.00	1.00
XML Parse Lib	2.16	-	-	-	-	-

We first motivate the introduction of our call stack traversal technique by analysing the overheads of the existing methodology. Table 3 presents an analysis of the additional overhead of stack traversal without any reduction technique. This method executes a full stack traversal at each trigger point, in this example `Malloc` or `Calloc` events. To establish these overheads we instrument the code with two different configurations of `WMTrace`, which both interpose memory management functions and output the results to file. For each configuration we enable stack tracing, and monitor the difference in runtime, between it and the instrumented time excluding stack tracing.

As we can see from Table 3 the overhead of the traversals can be very expensive for some applications. A single core execution of `phdMesh` with stack tracing enabled can take 3.5× longer than an execution excluding stack traversal, similarly `XML Parse Lib` runs 2.9× slower under the same conditions.

These overheads represent the relative slowdown of the instrumented application using call stack traversal, where 1.00× represented no slowdown at all, and 2.00× represents the codes running twice as slow (or 100% slowdown). From Table 2 we can compare the overheads with the event densities of the code and see a strong correlation between density and overhead, as one would expect.

Some applications do not suffer any observable overhead from the inclusion of stack traversal. In the case of `LU` and `FT` we attribute this to the very low call stack densities as shown in Table 2. Other cases the overhead may be minimal due to short call stacks, which are naturally quicker to traverse. Although the focus of this paper is to minimise traversal overheads we have included these applications to give a broad representation of the effects of our technique. Codes for which there is little motivation for optimisation, due to minimal overheads, provide an opportunity to evaluate the accuracy of the heuristic.

5.1 Heuristic Overheads

To analyse the performance gains of our heuristic technique we repeated the above experiment, but this time using the heuristic stack traversal. Table 4 illustrates the overheads of our technique when compared to an instrumented run without stack traversals. When compared with Table 3 we can see that the overheads of some codes are significantly reduced.

Whilst we experience good performance improvements in some circumstances this is not guaranteed. This technique

Table 5: Percentage of positive call stacks with heuristic stack traversal

	Core Count					
	1	**2**	**4**	**8**	**16**	**32**
miniFE	0.54	66.16	73.37	16.02	13.34	11.09
phdMesh	85.31	83.79	81.48	80.55	79.21	77.58
AMG	4.06	4.95	7.51	7.29	53.96	10.45
LAMMPS	76.03	98.09	98.09	98.09	98.09	98.09
LU	84.62	96.95	97.41	91.39	94.59	69.91
FT	59.52	31.65	32.04	18.42	12.78	12.20
Graph500	85.23	85.54	86.20	85.74	86.62	87.46
XML Parse Lib	66.66	-	-	-	-	-

has obvious limitations, as not only can it be incorrect if two paths are similar, but the additional work can cause a slowdown over full traversal. If traversal events are sparse, with large jumps in the CCT, then there is unlikely to be much commonality between the two traces. In this scenario the effort of looking for a shared prefix may outweigh any benefit from finding it.

5.2 Heuristic Validation

The overheads of our heuristic technique, presented in Section 5.1, present one aspect of the story. As previously discussed, our technique is fallible, and whilst it can perform quicker than full traversal it does not guarantee the correct result. In this section we analyse the percentage of correctly predicted call stacks across all of the processes of execution.

Table 5 compares every heuristically generated call stack with the fully traversed "correct" call stack. As we can see the percentage of correct predictions varies across different codes, and scales. Whilst this is in part related to event density, there are other contributing factors.

For codes where low validation is experiences, the performance displayed in Table 4 is not representative as it is generally faster to make a false prediction than an correct one.

For some codes, for example `LAMMPS`, we experience very good accuracy (a mean accuracy of 94.4%), and also see a slight reduction in overheads. As the overheads of tracing this application were already low, in part due to the low traversal event density, it is likely that for each comparison there was such minimal overlap in call stack prefix that there was very little time saved and few false positives. Whilst this is a positive result, our aim is to speed up the worst performing codes.

If we take the case of `AMG` we see a very poor accuracy (6.5%). Whilst the performance increase of the technique paints an attractive picture the accuracy is too low to make the technique applicable to this code, in its current format.

The accuracy of the heuristic on codes such as `FT` is inline with our hypothesis that a low traversal event density can result in reduced ability to correctly estimate shared prefixes.

Additionally the validation in Table 5 presents a worst case scenario for the heuristic technique, as it incorporates 'drift'. Drift is where an error in the prefix propagates through the successive call stacks un-noticed. There are many techniques to tackle this problem and limit the effect of drift. Seeding predictions from an accurate call stack will reduce the opportunities for drift, and better facilitate its identification. This ensures accurate predictions which can increase the validity of future predictions. Alternatively enforcing a full traversal at either a regular interval, or after large gaps between traversal events, can force the program to seed from a correct prefix.

When we eliminate drift, by seeding each traversal from the correct previous call stack, we experience a significant improvement in accuracy for some applications. The average accuracy for `AMG` rose from 6.5% to 99.8% suggesting that if drift is eliminated we will experience higher levels of accuracy for this code.

6. PERFORMANCE ACCURACY TRADEOFF

In Section 5 we presented an overview of both the performance gains and the accuracy of the heuristic technique presented in this paper. Whilst the technique is designed to improve call stack traversal overheads it does so at the cost of perfect accuracy. At each traversal a best estimate is made as to the commonality between the current call path and the previously traversed one. As outlined in Section 4, this technique can generate false positives when discrepancies between the underlying call stacks are unidentifiable at such a high level. To increase accuracy we must include more error checking, which will have an additional overhead associated with it. In this section we build upon the basic heuristic technique to include a simple accuracy enhancement, and evaluate it for accuracy versus runtime tradeoffs.

6.1 Second Match

The fundamental basis of the heuristic technique identifies similarity in call stack return addresses for a corresponding match in stack size. The performance gain from the technique is dependent on the identification of a large shared call stack prefix. The existence of a large shared prefix is an indication of a small transition in the CCT. When errors in comparison occur they stem from similarities in the underlying call stack, but without an absolute match being present. This scenario is demonstrated in Figure 4, where a second match would identify a problem that a first match would not.

Our simple accuracy improvement requires a second match. This modification requires two successive entries in the call stack to match both on return location and stack size. When a (previously identified) positive is identified this new method must traverse one level deeper to confirm. When a false positive is identified you must continue to traverse until another match is found. Whilst this is likely to result in slower performance than the original heuristic in the majority of cases there are circumstances where it will afford both an accuracy and performance gain. As the accuracy of predictions increase, it can become quicker, in some circumstances, to find the true intercept, as there may be little commonality between the previous false prediction and the current one.

The impact of this enhancement is that it is far less likely

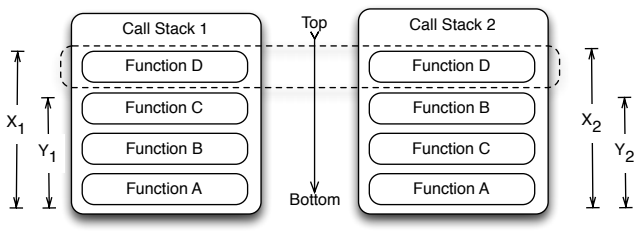


Figure 4: False positive identified by second match criteria

Table 6: Overhead analysis of second match heuristic stack traversal - Slowdown relative to no stack traversal

	Core Count					
	1	**2**	**4**	**8**	**16**	**32**
miniFE	1.00	1.00	1.00	1.00	1.02	1.00
phdMesh	1.75	1.16	1.61	1.05	1.24	1.50
AMG	1.39	1.22	1.14	1.17	1.00	1.01
LAMMPS	1.00	1.00	1.00	1.00	1.00	1.00
LU	1.00	1.00	1.02	1.02	1.00	1.00
FT	1.00	1.00	1.00	1.00	1.00	1.00
Graph500	1.00	1.00	1.00	1.00	1.03	1.01
XML Parse Lib	2.60	-	-	-	-	-

to return false positives, where there is a small modification in the call path, generally in the form of a function swap. Again we rely on spatial-temporal locality to assume that the majority of transitions in the CCT will be at the leaves (suffix) rather than the root (prefix). Thus a second point of validation at the suffix can prove crucial in accuracy improvements.

Figure 4 demonstrates the case there the order of `Function B` and`Function C` are rotated. Thus the top of the stack remains the same, `Function D`, and the stack size is unaffected. With a first match the technique would not identify this problem, but by requiring the next entry to also match, we identify this problem. Obviously this technique only identifies errors in a single entry lower than the previous match, and this function rotation could have occurred at any location in the call stack, rather than just near the top.

6.1.1 Second Match Overheads and Validation

Table 6 illustrates the overheads of the second match heuristic stack traversal, over a an instrumented execution without stack traversal. When compared with Table 4 we can see a reduction in performance, attributed to the additional work of validation.

The additional overheads are a cost incurred for an increase in accuracy. Table 7 illustrates the new accuracy rates for the second match heuristic traversal technique. For the minor increase in overheads we experience a significant improvement in accuracy, when compared to Table 5.

By comparing the results of Tables 7 and 7 we get an idea of the real improvement gained by enforcing this second match restriction. These results show that the second match policy significantly increased the accuracy of predictions for a selection of applications, inline with our prediction of accurate if drift is eliminated. The additional overheads of this technique for `AMG` are minimal, and still present a significant improvement over the full stack traversal technique.

Table 7: Percentage of positive call stacks with second match heuristic stack traversal

	Core Count					
	1	**2**	**4**	**8**	**16**	**32**
miniFE	99.45	99.75	99.78	99.92	99.91	99.87
phdMesh	100.00	99.04	99.75	95.80	95.90	94.12
AMG	99.99	99.99	99.97	99.94	54.18	77.24
LAMMPS	97.49	98.16	98.16	98.15	98.15	98.16
LU	100.00	97.29	97.51	88.76	95.89	73.75
FT	64.29	58.86	39.83	58.23	56.65	56.85
Graph500	86.03	86.87	87.84	87.81	87.80	88.16
XML Parse Lib	100.00	-	-	-	-	-

On average our second match heuristic is only 0.4% slower than first match, for a mean increase in accuracy of 28.1%. In some cases second match is faster that first match, such as certain executions of `phdMesh`, this is due to the reduced effort to find the correct call stack when seeded from the correct previous call stack. Due to the significantly higher accuracy of second match, each prediction is more likely to be bases on a correct value, which can reduce the work involved in finding the pointer of interception between the two call stacks.

Given the high accuracy of this second match technique (on average 89.1%) we do not implement any further drift elimination techniques, though we do anticipate a periodic full stack traversal could be beneficial in many circumstances. Whilst we exploit the spatiotemporal locality of trade events, we do not apply a threshold on the use of the heuristic. To achieve increased accuracy this full stack traversal could be timed to occur after a duration of inactivity, thus to seed predictions in a new cluster from a correct trace.

7. CONCLUSION

In this paper we have used a heuristic-based technique to improve the performance of call stack traversal. Using a metric for traversal density, the average number of trace events per second of execution, we identify codes which are amenable to our heuristic-driven technique. We extended the traversal density metric to consider the clustering of memory allocations, showing that clusters of allocations are likely to have very similar call stack prefixes, ensuring the high performance of our technique.

The accuracy of our technique is varied, with accuracy of less than 10% in some cases, and an average of 58%. Using a 'second match' policy on call stack comparisons our heuristic technique is able to achieve a significantly higher lever of accuracy than a single match policy, resulting in an average accuracy of 89%.

Across the codes we found our technique to improve the performance of stack traversal by an average of 12.5% for first match and 12.1% for second match, over the full stack traversal technique. The most significant performance gains came from applications which suffered most from full call stack traversal. We experienced a 46.5% and 49.9% improvement over full stack traversal for first and second matches respectively for `phdMesh`.

Future Work

In future work, we plan to integrate validation techniques similar to those presented in [13] into the tracing component of our memory toolkit. We also plan to migrate the tool to the StackwalkerAPI. This would improve the flexibility of the tool, as well as allowing direct comparison between other call stack traversal techniques.

8. REFERENCES

[1] D. A. Bader, J. Berry, S. Kahan, R. Murphy, J. Riedy, and J. Willcock. 'The Graph 500 List: Graph 500 Reference Implementations'. Graph500, 2010. `http://www.graph500.org/reference.html` (20 June 2012).

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks - Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE International Conference on Supercomputing*, SC '91, pages 158–165, New York, NY, USA, 1991. ACM.

[3] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*. ACM Request Permissions, June 2005.

[4] V. E. Henson and U. M. Yang. BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner. *Applied Numerical Mathematics*, 41(1):155 – 177, 2002.

[5] M. A. Heroux, D. W. DoerïňĆer, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving Performance via Mini-applications (Mantevo Overview). Technical Report SAND2009-5574, Sandia National Laboratories, Albuquerque, NM, USA, September 2009.

[6] T. Mytkowicz, D. Coughlin, and A. Diwan. Inferred call path profiling. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. ACM Request Permissions, Oct. 2009.

[7] O. Perks, D. Beckingsale, S. Hammond, I. Miller, J. Herdman, A. Vadgama, L. He, and S. Jarvis. Towards Automated Memory Model Generation Via Event Tracing. *The Computer Journal*, 2012 in press.

[8] S. Plimpton. Fast Parallel Algorithms for Short-range Molecular Dynamics. *J. Comput. Phys.*, 117:1–19, March 1995.

[9] M. Serrano and X. Zhuang. Building Approximate Calling Context from Partial Call Traces. In *CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 221–230. IEEE Computer Society, Mar. 2009.

[10] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 525–534. ACM Request Permissions, May 2010.

[11] Z. Szebenyi, T. Gamblin, M. Schulz, B. R. de Supinski, F. Wolf, and B. J. N. Wylie. Reconciling Sampling and Direct Instrumentation for Unintrusive Call-Path Profiling of MPI Programs. In *IPDPS '11: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, pages 640–651. IEEE Computer Society, May 2011.

[12] Unknown. XML Parse Library, 2010. `http://xmlparselib.sourceforge.net/` (20 June 2012).

[13] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*. ACM, June 2000.