THE UNIVERSITY OF
WARWICK

Original citation:
Smith, J. A., Hammond, Simon D., Mudalige, Gihan R., Davis, J. A., Mills, A. B. and Jarvis, S. A. (2009) hpsgprof : a new profiling tool for large-scale parallel scientific codes. In: UK Performance Engineering Workshop (UKPEW 2009), Leeds, UK, 6-7 July 2009. Published in: 25th UK Performance Engineering Workshop 6-7 July 2009

Permanent WRAP url:
http://wrap.warwick.ac.uk/59779

warwick**publications**wrap
highlight your research

http://wrap.warwick.ac.uk/

hpsgprof: A New Profiling Tool for Large–Scale Parallel Scientific Codes

J.A. Smith*, S.D. Hammond, G.R. Mudalige
J.A. Davis, A.B. Mills, S.A. Jarvis

Abstract

Contemporary High Performance Computing (HPC) applications can exhibit unacceptably high overheads when existing instrumentation–based performance analysis tools are applied. Our experience shows that for some sections of these codes, existing instrumentation–based tools can cause, on average, a fivefold increase in runtime. Our experience has been that, in a *performance modelling* context, these less representative runs can misdirect the modelling process.

We present an approach to recording call paths for optimised HPC application binaries, without the need for instrumentation. A a result, a new tool has been developed which complements our work on analytical– and simulation–based performance modelling. The utility of this approach, in terms of low and consistent runtime overhead, is demonstrated by a comparative evaluation against existing tools for a range of recognised HPC benchmark codes.

1 Introduction

High Performance Computing (HPC) applications represent a significant investment not only on the part of the developing organisation but also by the user community who routinely make use of these codes. The aim is to provide the best performance in an economic and resource efficient manner. To this end there has been considerable research in analysing and improving the performance of these applications. *Performance engineering* has numerous benefits, including supporting decisions in system procurement, guiding routine maintenance and ensuring the effectiveness of upgrades as part of the machine life cycle.

Performance engineering can be divided into four main activities: *Benchmarking* involves measuring the performance of real machines, often with benchmark applications which seek to replicate the behaviour of larger production applications; *statistical methods* allow the prediction of application performance by interpolation from a large number of existing results; *analytical methods* seek to capture application behaviour by building mathematical models of the critical path of an application's control flow and communication structure and *simulation approaches* seek to capture application behaviour by executing a

---

*High Performance Systems Group, Department of Computer Science, University of Warwick, Coventry, CV4 7AL `jas@dcs.warwick.ac.uk`

representation of the application's control flow and communication structure. The High Performance Systems Group at Warwick has, over the last twenty-five years, developed a variety of approaches and tools to support this activity [6, 8, 14, 16, 19].

Key difficulties faced in performance engineering are caused by the scale and complexity of the target applications. High performance scientific computing applications are often developed over a number of years by several contributors, using a variety of programming languages, supporting libraries and legacy code. Understanding the performance behaviour of applications, particularily when applying analytical and simulation techniques, continues to require a large investment of time and effort.

The initial objective when assessing the performance of such codes is finding the hotspots and bottlenecks. *Profiling* encompasses a number of performance analysis techniques related to the measurement and capture of the runtime or dynamic behaviour of an application, and as such are regarded as the primary means for finding hotspots and bottlenecks. These areas of the code will be responsible for the majority of the execution time of the application and hence become prime candidates for performance modelling or optimisation. Therefore, profiler accuracy is important. In a performance modelling context, *overhead* can shift the computation–communications balance. This can change the location and significance of hotspots and misdirect the modelling process.

As the scale of the applications that are under analysis increases, modifying these applications to obtain performance measurements *by hand* becomes increasingly impractical, thus performance analysis tools to automate this process are being developed. Tool support can broadly be divided into groups based on how the tools collect performance data and how that performance data is output.

The collection of data may be performed by either *sampling*, where the the program is paused periodically and (a subset of) its state is stored, or by *instrumentation*, where additional instructions are inserted at points of interest. In terms of output, there is a clear distinction between *profiling*, the collection of a *summary* of the application's behaviour, and *tracing*, recording significant points of a program's execution. Typically either sampling or instrumentation may be used for profiling, whilst only code instrumentation is used for tracing.

This paper presents an approach that forms a middle ground between 'pure' instrumentation-based tracing and 'pure' sampling-based profiling. This approach was developed to avoid the problems we experienced when profiling large applications, particularily those developed using contemporary development methodologies which feature many calls to many small functions. This approach forms the centrepiece of a new performance analysis tool which provides a promising alternative to instrumentation–based tracing techniques. As a demonstration of the tool's effectiveness, we present its application to well known benchmarks from the Los Alamos National Laboratory (LANL), the Lawrence Livermore National Laboratory (LLNL) and NASA. Special attention is paid to (1) how the tool fits into our performance analysis workflow, and (2) demonstration of the tool's runtime overhead. The three specific contributions of this paper are:

- The presentation of a new performance analysis tool, which uses a sampling approach to provide low runtime overheads;

- The demonstration of the tool on a number of standard HPC benchmarks;

- The comparision of the performance overheads of the tool against existing tools.

The remainder of the paper is organised as follows: Section 2 contrasts performance modelling and performance tuning - this distinction is important for the context of this research, as here we develop a tool to *aid modelling*, not to simply improve the process of profiling; Section 3 overviews related work and how our performance tool relates to this; Section 5 presents a comparison of the output and overheads of the tool against `gprof` [5] and TAU [17] for a variety of benchmarks; finally, in Section 6, we present our conclusions.

2   Contrasting Views of Performance Analysis

A common goal of performance analysis is to provide insights into the behaviour of the application. This is typically achieved by profiling the application and modifying it in line with the insights gained in order to improve application performance; typically in terms of execution speed or memory consumption [10]. Profiling allows program regions (hotspots) that are responsible for large proportions of the application runtime to be identified. The assumption made is that the greatest gains are to be made in the regions that exhibit the largest runtime cost. Profiling can then be used to validate that any modifications to the software aimed at improving performance have indeed improved application performance.

A contrasting use of performance analysis is in the support of performance modelling, which itself includes analytical- and simulation-based methods. Analytical modelling strives to represent the critical path of an application by mathematical formulae; which can be parameterised by measured values. Simulation aims to achieve a similar end, to provide predictions of application performance given measured parameters by executing a representation of the application control flow, implicitly reconstructing the critical path of the application. For both analytical- and simulation-based methods, a side effect of the modelling process is the production of the application/machine behaviour, which itself may be understood and applied in different contexts.

Once a performance model has been produced it can be used, in a similar fashion to a profile, supporting code engineering decisions. Unlike a profile however, a performance model can be used to predict the effects of engineering decisions *ahead of time*. In helping to avoid development effort being wasted in deadends, performance modelling has the potential to reduce development costs and development periods [13]. Additionally, a performance model also has the benefit that it can produce predictions of application performance when the application is running on hardware which may not be available or even when the hardware itself does not exist (the required parameter values can be obtained from vendor datasheets, for example) [7]. Performance models also have the advantage that they are able to produce values for application performance more quickly and cheaply than studies that require repeatedly running the application. In all these scenarios, profiling is used as a first step in the building of these models, and secondly, in order to provide validations to those models once they have been built.

The two major paradigms employed by profilers are program instrumentation and sampling. The decision between the two involves a tradeoff. This tradeoff is informed by the environment the profiler operates within and the process it provides support to. A profiler designed to locate bottlenecks as part of a software performance and tuning process is thus distinct from one designed for performance modelling and prediction. The former requires that the performance data must be produced or converted into a form that is human readable. A reasonable form for this is dictated somewhat by the implementation language of the application that is being profiled (e.g., C, C++ and FORTRAN). Conversely, a profiler intended for use as part of a simulation platform is targetting the different problem of feeding a machine model. As a direct result of the simulation environment being human designed and implemented, it is likely that producing the profile in terms of constructs that are familiar to a human is similarly beneficial. If this is the case, this is a result of the internal model of the simulation environment being based around the implementor's mental model of the executing application. The difference is that the extracted constructs need not map exactly to the constructs encoded in the high-level language representation, but that the profile should be produced in terms of such constructs. This implies that it is not imperative to be able to resolve ambiguious scenarios; for example, the use of a `for` loop or a `while` loop; a distinction that is important to a human, but one which is less likely to be of much concern to a simulation platform. In contrast to the requirements of an analytical modelling targetted profiler, one aimed at simulation places less value on mapping the observed behaviour to abstractions familiar to human users and instead places that value on capturing behaviour in terms that map, or that can be mapped, onto the internal model of the simulation tool.

The focus here is on providing a tool which supports our established analytical and simulation performance modelling activities[6, 14]. Increases in application and machine size means that we require a tool which outputs logs of application behaviour, that interfaces with our performance modelling tools and which exhibits low overhead. No existing tools have this capability.

3   Related Work

Performance analysis continues to depend on specialised tools to gather information concerning application behaviour. There are a number of existing profiling and tracing tools, all of which aim to capture and summarise runtime application behaviour. Examples include `gprof` [5], TAU [17] and `csprof` [4]. A notable problem with previous approaches is as a consequence of the tracing/profiling split; either the profiler overhead scales with the rate at which events occur, or the behaviour is summarised across the run, and much of the information contained in the structure of the dynamic calling tree is lost.

Methods which use virtual machine and binary-translation-based methods, exemplified by frameworks such as Valgrind [15], tend to be approaches based on the insertion of instrumentation code. Whilst they have advantages in the fine detail they can provide with regard to a program's behaviour (instruction mix, memory usage, cache behaviour), they require that each individual program instruction must be simulated. This means that Valgrind must simulate much of the processor behaviour and, although Valgrind increases performance by a

Just in Time compilation approach, there remains a considerable performance overhead (22.2 times on average for the Memcheck plugin [15] - a component described as heavily optimised).

Instrumentation-based tracing tools such as TAU (Tuning and Analysis Utilities) incur overhead for each instrumented event. Therefore the greater the frequency of events the greater the relative overhead, meaning that the overhead of the profiler is, to a relatively large extent, dependent on the behaviour of the application. Instrumentation-based profilers often have features designed to remove some of the instrumentation they perform in a heuristic manner, for example where a short routine is called frequently. TAU's default is where a routine is called over a million times and each call lasts less than two microseconds.

Depending on the method used to insert the instrumentation, the convenience of this process within a performance analysis workflow can be expected to differ. TAU supports several methods depending on the platform it is deployed on. Methods based on the instrumentation of source code are the least convenient; typically requiring a rebuild of the target application to add, remove or alter instrumentation. This is not of great concern for small, simple applications but becomes increasing problematic as the scale of the applications profiled grows. This is an issue where complex, fragile or bespoke build systems are involved. The difficulties encountered with build systems can be compounded when there is a need to profile across the application-library boundary, such that compilation occurs in a number of separate units, many of which are interdependent.

Instrumentation inserted after compilation, into the compiled object code [11], has the potential to lead to a tighter instrument-profile-analyse cycle. Like all methods that instrument object code, they avoid the need for integration with build systems; thus aiding an automated approach to instrumentation refinement. Additionally, by avoiding the overhead of heavyweight recompilation, it has the potential benefit of reducing the time taken between analysis and performing the next profiling run. Tightening this cycle, hence increasing the speed of iteration, means that performance analysis and modelling activities can be performed more quickly and with lower associated costs. There is also the advantage of a greater degree of language independence; disadvantages include the requirement of working on a semantically poorer program representation as well as less immediate portability between different computing platforms.

The most flexible form of object code instrumentation is that performed dynamically at runtime [3]. The principle benefit is that the level and locations of instrumentation can be adjusted whilst the program is running, in response to the overhead being incurred or directly upon the behaviour and actions of the profiled program; for example, additional instrumentation could be inserted based on entry to a specific part of a program that is under analysis, then later removed after exit from that specific part. There is of course no guarantee that the overhead of the cost of inserting and removing profiling instrumentation at runtime will be less than the cost of having the equivalent instrumentation continuously inserted.

The traditional alternative to instrumentation-based tracing is *sample-based profiling*. Normal program execution is periodically interrupted and the currently executing function is observed. The overhead can be adjusted by changing the sampling rate. Reducing the sampling rate decreases overhead at the

expense of resolution. Similarily, the resolution can be increased by adjusting the sampling rate at the expense of increased overhead.

The results form a statistical profile of the application, including approximate function execution times. This profile is 'flat'; that is it contains no information about calling contexts. Capturing calling contexts is the major contribution of `gprof`, which uses compiler inserted instrumentation to record calling contexts, but continues to use sampling to estimate runtimes. `gprof`, therefore, forms a hybrid between instrumentation and sampling methods, forming one of the many possible tradeoffs. The use of instrumentation allows it to record an exact dynamic call graph; the downside to this is that this instrumentation is a source of application-linked profiler overhead.

A call graph allows parent functions to be allocated the cost of their children; therefore allowing the identification of costly functionality, whether in a single function or split over a number of child functions. Where multiple functions call the same child, `gprof` assumes allocates runtime according to calling frequency. For our purposes, `gprof` has a number of problems; firstly, it assumes that functions have a largely similar runtime, independent of the calling site; secondly, `gprof` represents runtime in terms of profiling time rather than real time, and finally, it produces a summary of the entire application run. This profile does not differentiate between sections of an application run. `gprof`'s weakness with respect to profiling heavily recursive functions (where it combines most of those functions into a single monolithic cycle [12, 18]), is less of a concern as our experience indicates that we do not tend to encounter such programs in our performance modelling activities.

Assuming that function execution times are independent of the call site does not hold for all programs. The canonical example is where execution time is dependent on function arguments, and the function arguments themselves are dependent on the call–site. A common example might be library functions such as those for message passing. A function that sends many short messages is likely to have an exaggerated runtime cost with respect to a function which sends fewer long messages.

The `csprof` profiler performs limited instrumentation, which might be considered an optimisation detail and not intrinsic to its operation, thus we categorise it as a sampling profiler. Periodically, it takes a stack trace of the application. It then uses this stack trace to allocate runtime. This means the allocation of runtime is call–site dependent, thereby eliminating distortion caused by the incorrect attribution of runtime to call–sites.

Representing the runtime in profiling time has the advantage that, in a time sharing system, the profile is largely independent of system load. Profiling time, unlike walltime, does not include the runtime of applications that may be sharing the same machine via a time slicing system. In this sense it is similar to CPU (process) time, however, it also includes time spent by the operating system on behalf of the user. For current high performance computing facilities, which tend to be message passing clusters, it does not tell the whole story; it does not include time spent waiting for messages. Additionally, clusters are less likely to timeslice CPU time. Together, these two factors, make walltime a more accurate representation of application execution time.

Only providing a profile is particularily problematic in many high performance computing benchmarks. These benchmarks follow an input-process-output system model, where data is read by the application, processing oc-

curs on the data and the result is written back to disk. As Sweep3D [20] and AMG2006 [9] demonstrate, the processing section is often made significantly shorter than in the application that the benchmark represents. Therefore a summary might indicate that much of the time is spent in file I/O routines, whereas, as the problem is scaled up, the application hotspot clearly becomes the processing section.

4   `hpsgprof`: HPC callpath extraction

Our solution to the instrumentation overhead problem is to extend the stack sampling model used by `csprof` to form an equivalent trace of the application's behaviour. This allows us to avoid the need, as with `csprof`, to attempt to reconstruct the application behaviour from combining the result of static analysis with a profile. We demonstrate that recording a log of stack traces during execution is practical for a variety of real world benchmarks.
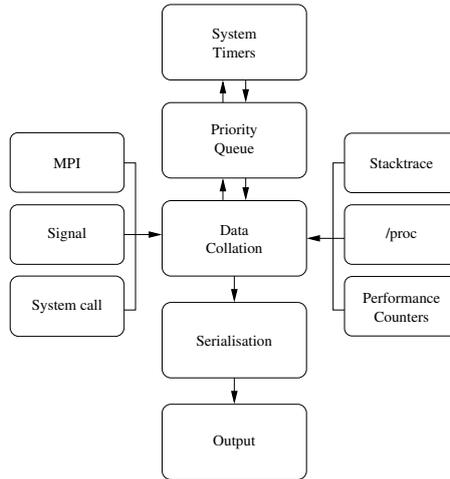

Figure 1: Architectural overview of `hpsgprof`

This focus on periodic behaviour differentiates `hpsgprof`, as in Figure 1 from previous tools, such as TAU [17]. Previous tools are centred around collecting data when an external event occurs within the profiled program, conversely, `hpsgprof` is primarily based on scheduling actions in the future. Recurring behaviour, such as repeated sampling, is based on an action retriggering itself after the data has been collected. The profiler then records the data in a standard format. One of the design goals of our tool is to try and reduce overhead by shifting as much analysis as possible to post-execution. By contrast, TAU performs some analysis during the profiled run, allowing some metrics to be reported more quickly.

A further goal was to integrate with existing tools for much of the functionality. For example, we use platform–specific debugging features to map our program addresses back to source code, allowing us to take advantage of robust, compiler–supported methods that are provided by the platform. We also have

```
main:??:0:0x40da8c
 driver:/home/jas/sweep3d/driver.f:185:0x4011a9
  inner_auto:/home/jas/sweep3d/inner_auto.f:2:0x40384a
   inner:/home/jas/sweep3d/inner.f:95:0x404a47
    source:/home/jas/sweep3d/source.f:39:0x4082eb
main:??:0:0x40da8c
 driver:/home/jas/sweep3d/driver.f:185:0x4011a9
  inner_auto:/home/jas/sweep3d/inner_auto.f:2:0x40384a
   inner:/home/jas/sweep3d/inner.f:116:0x404dc4
    sweep:/home/jas/sweep3d/sweep.f:120:0x408c1c
```

Figure 2: Log of Sweep3D stacktraces captured by `hpsgprof`

Benchmark	Sweep3D	MG	SP	AMG2006
Source Lines of code	2279	2600	4200	68373
Functions	30	32	31	1026
Language	Fortran 77	Fortran 77	Fortran 77	C90

Table 1: Benchmark comparison

the ability to do a coarser analysis using only the symbol table of an application; which allows the tool to operate without requiring debugging symbols.

Figure 2 shows two stacktraces captured from Sweep3D by `hpsgprof`, and converted into ASCII format. Each line is of the form *function*: *file*: *line*: *address*, indicating the return point within the program for each function. The file and line are missing for the `main` function. This is due to an implementation detail, where the FORTRAN compiler inserts a false main function, which handles FORTRAN initialisation, before jumping to the user defined main program.

From this output, we can reconstruct the calling tree using an approach suggested by Arnold et al., [1], where CPU time is allocated to the last edge in the call stack. For the example, we are able to determine that `inner` calls both `source` and `sweep`, a case that can be confirmed by inspection of the Sweep3D source code. We also assume, like the authors of `csprof`, that the effect of short, infrequently called leaf functions is minor in terms the overall accuracy of the tool output, especially with respect to the benefits.

One of the problems with many applications is that they can have unexpected behaviour: One particular example is that FORTRAN programs, unlike C programs, tend not to call destructors of dynamic libraries when they exit; instead calling `sysexit` when they reach a `STOP` statement. When writes are buffered, the program suddenly exiting will cause a loss of data. Our workaround to this is to intercept the appropriate system call allowing our tool to flush its buffers and cleanly shutdown.

5 Comparison with existing tools

In order to validate our new approach, we use a variety of High Performance Computing benchmarks from LANL, LLNL and NASA (see Table 1) to demon-

Processors		Original	gprof	hpsgprof			TAU
				1Hz	10Hz	100Hz	
2	wall	293.486	297.868	293.689	295.895	299.521	290.880
	cpu	293.278	297.643	293.332	294.386	296.503	290.562
4	wall	156.098	158.706	156.102	156.806	158.140	156.268
	cpu	149.731	152.090	149.889	149.863	150.503	149.685
8	wall	83.776	84.932	83.787	83.858	85.152	84.634
	cpu	75.923	76.943	75.907	76.017	76.557	76.161
16	wall	49.570	50.310	49.561	49.677	50.172	49.734
	cpu	43.149	43.687	43.209	43.305	43.419	42.905
32	wall	27.387	27.769	27.375	27.452	27.783	27.946
	cpu	21.725	22.051	21.777	21.773	21.815	21.867

Table 2: Runtimes (seconds) for the LANL Sweep3D benchmark

Function	gprof	hpsgprof	TAU
sweep	97.80	97.764	97.694
source	1.96	1.988	2.037
flux_err	0.22	0.234	0.227
inner	0.01	0.006	0.008
initialise	0.01	0.005	0.014
initxs	0.01	0.005	0.014

Table 3: Top function contributions (%) for Sweep3D

state that the result and the overheads of `hpsgprof` are acceptable on a variety of real applications when compared to existing tools. `hpsgprof` was configured to obtain similar data to `gprof`, but with the benefit that we produce a log of the application behaviour rather than a summarized profile. All the codes we present are parallel application benchmark codes using MPI for parallel execution. The codes were all compiled under high levels of optimisation (-O2 or -O3), and executed on a 16 node cluster. Each node has dual 2GHz AMD Opteron 246 processors, 1GiB of RAM per processor and is connected by Gigabit Ethernet. The results presented are the median of ten runs.

We first report on the ASCI kernel application Sweep3D [20] 2.2b from LANL. Sweep3D is a 3D discrete ordinates neutron transport code. The parallelisation strategy is a wavefront code, a 2D array of processors is used to form the wavefront, which forms a hyperplane that sweeps from corner to corner of the array. Sweep3D runs under a variety of parallel environments; all our testing was performed using the MPI implementation. We used the median of a number of runs to minimise the effect of unusually fast or slow runs due to caching or system noise effects. The timing was performed internally by the benchmark application, and represents the time taken for the S_n solution process only.

We compare the results from TAU and the `hpsgprof` and `gprof` tools. Table 3 shows that all three produce comparable function runtime attributions. The differences between the times are generally within the square root of the number

Processors		Original	gprof	hpsgprof			TAU
				1Hz	10Hz	100Hz	
1	-O2	30.155	30.290	30.185	30.165	30.180	29.815
	-O3	29.965	30.130	29.955	29.960	29.975	29.845
2	-O2	14.300	14.350	14.395	14.375	14.395	14.395
	-O3	14.155	14.200	14.165	14.190	14.010	14.330
4	-O2	9.470	9.520	9.360	9.445	9.425	9.645
	-O3	9.295	9.605	9.165	9.295	9.295	9.365
8	-O2	4.985	4.790	5.240	4.865	5.045	4.930
	-O3	4.735	4.935	4.805	4.920	4.935	4.925
16	-O2	4.190	4.150	4.400	4.250	4.310	3.900
	-O3	3.905	3.890	3.870	4.000	3.935	4.730
32	-O2	3.930	4.010	4.205	3.920	3.960	4.210
	-O3	3.915	3.920	3.855	3.880	4.175	4.075

Table 4: Runtimes (seconds) for the NAS MG benchmark

of samples variation, which can be expected for sampling-based methods.

We also compared the overhead with an unprofiled version of the code. Overall, TAU provides the lowest overhead for Sweep3D. The Sweep3D structure is such that instrumentation-based methods work well; it makes few function calls with respect to the computation it performs. In terms of cpu time, `hpsgprof` clearly demonstrates a significantly lower overhead than `gprof` on the Sweep3D benchmark at all three sample rates we tested. Overhead can be seen to scale with sample rate, however, the situation changes when overhead is considered in terms of wall time. `hpsgprof` exhibits lower overhead than `gprof` at both 1Hz and 10Hz sampling frequencies. At 100Hz the overhead is very similar to that of `gprof`. The overhead of our tool is shown to be low (a maximum of around 2%), well within the variation of a single run. We expect that with further tuning, especially in terms of data buffering, the overhead of `hpsgprof` can be further reduced.

The next set of benchmark that we test are from from the NAS Parallel Benchmark suite [2] version 2.4. MG is used to find an approximate solution to a discrete Poisson equation in three dimensions; SP (Scalar Pentagonal) solves non-linear Partial Differential Equations. Testing is performed using the MPI benchmark and the medians of the repeated runs are presented.

All three tools produce very similar runtimes, as seen in Figure 4. The similarily of the overheads to `gprof` suggests that the runtime of all the profilers is acceptable. It is noted that the runtimes for TAU have a tendency to be higher than the runtimes for all the other tools, especially at more aggressive levels of compiler optimisation. We believe that this is due to the heavy instrumentation and hence the greater degree of data collected. It is not clear why for certain job sizes, profiling can decrease application runtime (although inappropriate compiler optimisations and cache effects are suspected).

The final benchmark is AMG2006 [9]. AMG2006 forms part of the Sequoia benchmark from LLNL. AMG2006 is an unstructured grid algebraic multigrid

Processors		Original	gprof	hpsgprof			TAU
				1Hz	10Hz	100Hz	
1	-O2	796.805	796.915	797.135	801.370	796.020	797.180
	-O3	775.655	774.210	774.485	775.715	774.570	783.350
4	-O2	234.700	237.415	233.020	233.980	234.475	243.155
	-O3	234.035	230.125	232.985	233.305	233.330	231.315
9	-O2	157.800	154.830	139.480	137.630	138.960	142.735
	-O3	144.065	144.355	143.580	145.535	143.095	156.770
16	-O2	117.370	117.740	118.225	118.205	117.775	116.760
	-O3	122.555	117.655	117.410	115.940	115.960	125.390
25	-02	110.985	112.540	106.795	107.125	107.615	108.575
	-O3	109.970	109.435	106.780	108.180	108.225	107.860

Table 5: Runtimes (seconds) for NAS SP benchmark

linear system solver. Once again, MPI is used as the communications library and we calculate the median time, as measured by the benchmark, of ten runs. This benchmark features sections of code that have a much higher function call density (see Table 1, a feature we believe will become increasingly prevalent in future HPC applications as it is a notable feature of contemporary programming practices, featured by languages such as C++.

We compare the overhead of our tool with both `gprof` and automatically instrumented TAU. Table 6 compares the overhead of the tools, for the `SStruct` section of AMG2006. Both `gprof` and and `hpsgprof` show low overhead, in the region of a few percent. With the higher level of optimisation (-O3) TAU demonstrates a much higher overhead (with a runtime from 2 to 8 times, and a median of 5.6 times). This overhead is also unpredictable and changes with processor count, representing a much less attractive result. At a lower optimisation level (-O2) the overhead is more predicable, but tends to increase with processors from around 70% with 5 cores to about 180% at 30 cores. These results demonstrate the advantages of a sampling approach. This is especially true of systems, other than `gprof`, which do not benefit from `gprof`'s combination of extremely lightweight measurements and very high levels of compiler integration.

6   Conclusion

In this paper, we present a new performance analysis tool, `hpsgprof`, which takes a sampling approach to gathering an approximate performance trace, in order to provide for low runtime overheads. We have demonstrated the performance of the tool on a number of standard HPC benchmarks Sweep3D, NAS MG, NAS SP and AMG2006. This has demonstrated that the approach, in comparision to existing tools, can provide detailed program traces and associated timing information with low runtime overheads. Furthermore, we have demonstrated the use our tool is especially beneficial when applied to appli-

Program section			Original	gprof	hpsgprof			TAU
					1Hz	10Hz	100Hz	
SStruct	-O2	min	0.96165	0.99400	0.95137	0.96804	0.97316	1.67813
		ave	—	1.06859	0.99605	1.00097	1.01345	2.40221
		max	1.07257	1.16725	1.10211	1.08445	1.08676	4.11770
	-O3	min	0.96340	0.99965	0.92659	0.95644	0.95549	4.79997
		ave	—	1.04032	0.99345	0.98249	0.97929	5.64947
		max	1.06236	1.09274	1.19902	1.12843	1.10757	10.54840
Setup	-O2	min	0.96355	0.86412	0.97891	0.98059	0.97399	0.91452
		ave	—	0.90518	1.00388	0.99645	1.00150	1.00106
		max	1.05710	0.99632	1.06910	1.11604	1.11928	1.15254
	-O3	min	0.97128	0.91308	0.96665	0.97127	0.97162	1.17664
		ave	—	0.94335	1.00350	0.99171	0.98856	1.21786
		max	1.02830	1.06012	1.09543	1.08309	1.08803	1.29021
Solve	-O2	min	0.79290	0.91459	0.90792	0.85498	0.89894	0.87110
		ave	—	0.99354	1.00032	0.94651	0.97379	0.94831
		max	1.10430	1.20045	1.23946	1.20011	1.38638	2.70966
	-O3	min	0.89343	0.75587	0.88467	0.90489	0.88302	0.88434
		ave	—	0.99837	0.99577	0.95227	0.93945	0.93678
		max	1.11802	1.17770	1.08809	1.37356	1.65606	2.89272

Table 6: Relative runtimes for AMG2006

cations that exhibit increasing function call density, a trait that applications such as AMG2006 highlight and that is likely to become increasingly common in HPC applications.

References

[1] M. Arnold and P.F. Sweeney. Approximating the Calling Context Tree via Sampling. Research Report RC 21789 (98099), IBM, July 2000.

[2] D. Bailey, E. Barszcz, J Barton, D. Browning, R. Carter, L. Dagnum, R. Fatoohi, S. Finebergm, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. RNR Technical Report RNR-94-007, NASA, March 1994.

[3] B. Buck and J.K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14:317–329, 2000.

[4] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low–overhead Call Path Profiling of Unmodified, Optimized code, booktitle = ICS '05: Proceedings of the 19th annual international conference on Supercomputing, year = 2005, isbn = 1-59593-167-8, pages = 81–90, location = Cambridge, Massachusetts, doi = http://doi.acm.org/10.1145/1088149.1088161, publisher = ACM, address = New York, NY, USA,.

[5] S.L. Graham, P.B. Kessler, and M.K. McKusick. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, 17(6):120–126, 1982.

[6] S.B Hammond, G.R. Mudalige, J.A Smith, S.A Jarvis, J.A Herdman, and A. Vadgama. WARPP: A Toolkit for Simulating High Performance Parallel Scientific Codes. In *SIMUTools09*, Rome, Italy, March 2009.

[7] S.D. Hammond, G.R. Mudalige, J.A. Smith, and S.A. Jarvis. Performance Prediction and Procurement in Practice: Assessing the Suitability of Commodity Cluster Components for Wavefront Codes. In *UK Performance Engineering Workshop*, Imperial College, London, UK, July 2008.

[8] John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. Analytical Modeling of Set-Associative Cache Behavior. *IEEE Trans. Comput.*, 48(10):1009–1024, 1999.

[9] V.E. Henson and U.M. Yang. BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner. *Applied Numerical Mathematics*, 41:155–177, 2000.

[10] S. A. Jarvis, J. M. D. Hill, C. J. Siniolakis, and V. P. Vasilev. Portable and Architecture Independent Parallel Performance Tuning using BSP. *Parallel Computing*, 28(11):1587 – 1609, 2002.

[11] J.R. Larus and E. Schnarr. EEL: Machine–independent Executable Editing. *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, 30(6):291–300, June 1995.

[12] R.G. Morgan and S. A Jarvis. Profiling Large–scale Lazy Functional Programs. *Journal of Functional Programming*, 8(3):201–237, 1998.

[13] G.R. Mudalige, S.D. Hammond, J.A Smith, and S.A Jarvis. Predictive Analysis and Optimisation of Pipelined Wavefront Computations. In *IPDPS09*, Rome, Italy, May 2009.

[14] G.R. Mudalige, M.K. Vernon, and S.A. Jarvis. A Plug and Play Model for Wavefront Computation. In *International Parallel and Distributed Processing Symposium 2008 (IPDPS'08)*, Miami, Florida, USA, April 2008.

[15] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[16] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. PACE – A Toolset for the Performance Prediction of Parallel and Distributed Systems. *Int. J. High Perform. Comput. Appl.*, 14(3):228–251, 2000.

[17] S.S. Shende and A.D. Malony. The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[18] J.M. Spivey. Fast, Accurate Call Graph Profiling. *Softw. Pract. Exper.*, 34(3):249–264, March 2004.

[19] D.P. Spooner, S.A. Jarvis, J. Cao, S. Saini, and G.R. Nudd. Local Grid Scheduling Techniques using Performance Prediction. *IEE Proceedings Computers and Digital Techniques*, 150(2):87–96, Mar 2003.

[20] Sweep3D. The ASCI Sweep3D Benchmark. http://www.ccs3.lanl.gov/PAL/software/sweep3d/.