Towards Automated Memory Model Generation Via Event Tracing

O.F.J. Perks[1], D.A. Beckingsale[1], S.D. Hammond[2], I. Miller[3], J.A. Herdman[3], A. Vadgama[3], L. He[1], and S.A. Jarvis[1]

[1] Performance Computing and Visualisation
Department of Computer Science
University of Warwick, UK
[2] Scalable Computer Architectures/CSRI, Sandia National Laboratories,
Albuquerque, New Mexico, USA
[3] Supercomputing Solution Centre, UK Atomic Weapons Establishment,
Aldermaston, Reading, UK

Keywords: High Performance Computing; Memory; Multi-core; Tracing; Modelling

Abstract. The importance of memory performance and capacity is a growing concern for high performance computing laboratories around the world. It has long been recognised that improvements in processor speed exceed the rate of improvement in DRAM memory speed and, as a result, memory access times can be the limiting factor in high performance scientific codes. The use of multi-core processors exacerbates this problem with the rapid growth in the number of cores not being matched by similar improvements in memory capacity, increasing the likelihood of memory contention. In this paper we present `WMTools`, a lightweight memory tracing tool and analysis framework for parallel codes, which is able to identify peak memory usage and also analyse per-function memory use over time. An evaluation of `WMTools`, in terms of its effectiveness and also its overheads, is performed using nine established scientific applications/benchmark codes representing a variety of programming languages and scientific domains. We also show how `WMTools` can be used to automatically generate a parameterised memory model for one of these applications, a two dimensional non-linear magnetohydrodynamics (MHD) application, `Lare2D`. Through the memory model we are able to identify an unexpected growth term which becomes dominant at scale. With a refined model we are able to predict memory consumption with under 7% error.

1 Introduction

The importance of memory performance and capacity is a growing concern for high performance computing laboratories around the world. In the five decades since Gordon Moore famously predicted the rate at which transistor counts would increase, the per processor performance of supercomputers has grown by over

four orders of magnitude. Coupled with increasing processor counts, supercomputers have become more powerful; this has created opportunities to significantly advance the potential complexity and size of computing simulations which can be performed, as long as supporting data can be can be fed from memory to the compute cores at a sufficient rate.

The divergence between processor performance and memory access time, highlighted by Wulf and McKee's term 'memory wall' [1] (the point at which poor memory access times become the limiting factor in processing rates), has traditionally been thought of as one of the greatest concerns in computer architecture. The use of multi-core processors presents several other memory-related challenges. One of the most significant of these is the decrease in per-core memory, which is a result of rapid growth in the number of cores per processor without matched improvements in memory capacity. Perhaps more subtle, but nonetheless important, is the rising level of contention for main memory – a direct implication of increasing the number of cores per-processor without significant improvement in the number, or performance, of the memory channels for data transfer. This issue looks set to be more problematic in the future [2]; processors seem likely to increase in core-density and the use of accelerators including graphics-processing units (GPUs) will make understanding memory placement and capacity, per processing element, more important than ever.

Memory also represents a significant upfront cost of machine procurement; power consumption and hardware failures also contribute to the total cost of ownership. The lack of significant improvements means that to maintain current levels of performance and capacity, the proportional cost of memory has escalated.

The act of strong scaling an application, increasing the core count to solve a fixed problem size, is a traditional method used to address insufficient memory per core. This technique is not without problems as the increase in core count has been shown to increase memory consumption in certain circumstances [3, 4]. This is due to the storage of additional core-count-dependent data, for example communication buffers and rank-to-rank lookup data. Although this is primarily a problem with the Message Passing Interface (MPI), it is a fundamental concept replicated in some production codes, with negative implications.

Memory therefore continues to pose a challenge in the design and optimisation of parallel algorithms at scale. Understanding how an application requests, utilises and frees memory during execution remains a key activity associated with optimising application runtime and thereby improving scientific delivery. If memory requirements across an application workflow are sufficiently understood, opportunities may be created to reduce the memory configuration of a machine during procurement, reducing initial capital expenditure.

In this paper we introduce `WMTools`, a memory management analysis suite, comprising a lightweight profiling tool, `WMTrace`, and a post execution analysis tool, `WMAnalysis`. The tools have been developed to trace the memory allocation events of parallel applications, using the MPI library. `WMTrace` is a dynamic shared library which requires no source code modifications or binary instrumen-

tation. `WMAnalysis` can act as both a serial and parallel analysis tool, for interpreting trace data to accurately calculate high-water mark, memory usage per function over time and memory usage per function at peak. The intended use of our tool is to provide sufficient information to developers to enable the diagnosis of memory allocation issues, relating to both overall memory consumption and scalability issues. It also permits an analysis of behaviour over time, and thus the identification of optimisation opportunities. Understanding the behaviour of memory requirements throughout application execution, and on multiple-core configurations, is the first step in assessing or predicting memory usage at scale and a vital precursor to identifying potential memory scaling issues. Our tool presents this opportunity within a framework which is lightweight, accurate and simple to use.

We extend the tool suite to enable parallel analysis of trace files at the point of high-water mark. This analysis, coupled with some additional domain knowledge, can be used to automatically construct a memory model of the code. Such a model enables the prediction of memory consumption at large scale, on increased core counts.

The specific contributions of this paper are as follows:

- We present `WMTools`, a lightweight memory tracing tool and analysis framework for parallel codes. The distinguishing features of this tool are the ability to record memory allocation traces combined with temporal and stack information without the introduction of significant overhead. This data is sufficient to identify peak memory usage and analyse memory usage over time, on a per function basis, for a set of parallel tasks - a pre-requisite in understanding how individual sections of an application utilise memory;
- We present the application of `WMTools` to nine established scientific applications and benchmark codes representing a variety of programming languages and scientific domains. The output of `WMAnalysis` illustrates how the memory allocation traces of these codes scale with increased problem size and core count. We illustrate the importance of static memory analysis for two codes, `LU` and `FT`, which provide minimal allocation information during heap tracing. We demonstrate the ability of the suite to identify key problematic areas within `phdMesh`;
- Finally, we use `WMTools` to generate a model for the memory usage of one of the nine codes: `Lared2D`, based on a comparison of traces from 2- and 4-core executions. This model is then validated against collected data, showing an average error of under 7% in predicting the memory high-water mark of a given run of up to 256 cores. We extend the initial model to capture an alternative growth function which becomes dominant at scale. Additionally we validate this new model against the memory results of a 4× bigger problem set, at scale, observing between -1% and 4% error.

The remainder of this paper is structured as follows: in Section 2 we catalogue a shortlist of previously reported memory analysis tools and summarise techniques which are relevant to this work; Sections 3 and 4 contain the presentation of our tracing tool, `WMTrace`, and the post-processing tool `WMAnalysis`; a case

study applying our tools to the analysis of memory allocation behaviour of several industry-standard applications and benchmarks is described in Section 5, including a comprehensive analysis of overheads and a comparison with alternative tools; in Section 6 we describe the use of `WMAnalysis` to generate a memory model for one of the studied codes and use the model to predict the memory high-water mark for various application runs; in Section 7 we conclude the paper with a summary of our findings and identify areas for future work.

2 Related Work

Memory represents a significant bottleneck within high performance computing (HPC) centres, with regards to both performance and capacity. Feitelson [5] demonstrates the importance of memory consumption considerations in job scheduling and execution on a supercomputer at Los Alamos National Laboratory (LANL). Discussing the benefits of an analytical memory model to accurately predict memory usage, Feitelson concludes that due to the large number of components, an analytical approach to memory modelling would be too complex.

Due to the importance of memory, a collection of tools already exist to assist developers in analysing memory usage and management within their code. Understanding an application's memory management enables code redesign, with the aim of reducing the runtime memory footprint. As the specifics of each particular bottleneck are unique, these existing tools each serve a different purpose, ranging from leak detection to cache alignment.

The main distinction between memory tracing tools is the level of analysis performed and the granularity of the data collected. We classify memory analysis tools into two classes, *lightweight* and *heavyweight*, depending on their overheads and level of analysis. The tools which provide the most detailed level of data collection have an inherent overhead in either additional memory consumption or runtime – and often both. The large volumes of data which these tools produce also require extensive post-processing to derive any meaningful interpretation from their results. This class of tools often collects data at the hardware counter level, and may require additional code instrumentation.

The alternative class of tools attempt to avoid this overhead and are thus limited in the data they can collect. As lightweight tools, they are often loaded dynamically at runtime and therefore do not require code instrumentation.

The closest tool to `WMTools` is `memP` from Lawrence Livermore National Laboratory (LLNL) [6]. `memP` is a lightweight library designed for collecting basic memory consumption information. The primary use of `memP` is to collect high-water mark usage data from all of the processes in an MPI job, which it achieves by re-implementing the memory management functions as ones which allow tracing. All of this data is collected and stored within in-memory data structures, eliminating the need for a post-processing stage. Maintaining these in-memory data structures introduces some performance overhead and additional memory consumption. The resulting data set is small, providing maximal heap and stack

usage across the processes together with aggregated statistics over this data (*e.g.* standard deviation and coefficient of variation).

Where `WMTools` differs from `memP` is that it does not store the complete dataset in memory, but instead streams it to file, thus minimising overheads and having as little influence on the execution of the program as possible. As statistics are not immediately available upon job completion, `WMTrace` utilises a post-processing phase to extract memory usage statistics. `WMTrace` is also intended to provide the user with a more in-depth analysis of the system during times of peak memory consumption.

`mprof` also provides a lightweight framework for memory analysis with greater granularity than `memP`. `mprof` was written when memory constraints prevented the tool from consuming too much system resource, with the authors claiming to require only 50KB of additional memory and at worst incurring a 10× slowdown [7]. This tool also favours in-memory data structures; the authors considered the storage of all collected information prohibitive. One of the main limitations of `mprof` is that it only provides a stack traversal up to a maximum depth of five, whereas `WMTrace` enforces no such limitation. It is also worth noting that `mprof` was not designed to profile parallel codes, and so has limited use within HPC, without modification.

Whilst we understand the performance implication of recording the full call-stack, we consider it necessary to obtain a complete understanding of the system state to support a full analysis of memory usage in an application. We also provide lossless data collection, where all temporal information is retained, allowing a complete playback of application execution. This generates far more data than tools which focus on just high-water mark analysis, but enables a more thorough analysis of memory to be performed. As our tool does no analysis of memory management during program execution, it is unable to determine whether specific information should be deemed important or irrelevant and thus all data is retained.

The volume of data generated by stack tracing tools has been a topic of detailed research, with a number of methods to reduce or compress stack data being proposed [8, 9]. `WMTools` employes similar techniques to reduce compression overheads and file size, by using knowledge about the repetition of data to build custom dictionaries, as well as leveraging existing compression libraries.

The specific aim of `WMTools` is to bridge the gap between lightweight and heavyweight profiling tools, by providing more detailed information than a typical lightweight tool, but by minimising the overheads of a heavyweight tool.

Many modern heavyweight analysis tools are built upon dynamic binary instrumentation (DBI) frameworks, such as `Valgrind` [10] and `Pin` [11]. These DBI frameworks facilitate the use of shadow memory, where either metadata regarding access frequency or even value representation is stored for every byte of application allocated memory. This is exploited by many different shadow memory tools to detect accesses to un-addressable memory, perform type checking and identify data races, amongst others. The overheads of such tools are widely acknowledged to be large; the authors of `Memcheck` [12] document a mean slow-

down of 22.2x. Tools like `Memcheck` introduce overheads by monitoring memory allocations and accesses. Differences in the functionality of `WMTools` and `Memcheck` mean that `WMTrace` does not incur the same mid-execution processing costs; `WMTrace` does however have additional data generation and storage costs. We address this balance in overhead (between runtime slowdown, amount of data generated, and post-processing cost) in our subsequent application and analysis of `WMTrace`.

Another approach to memory profiling, currently being investigated, is virtualisation and sandboxing. Virtual Machine Introspection (VMI) [13], developed at Sandia National Laboratories, enables memory profiling through the use of a system-level virtual machine. The overheads of such a technique are likely to be considerable, but the opportunities for data capture are significantly broadened.

Due to similar data-movement and call-stack traversal problems, many of the techniques and methodologies employed in memory tracing are taken from research into large scale debuggers [14, 15]. Szebenyi et al. demonstrate the benefit of hybrid profiling, using sampling - to capture general application performance - and event-based instrumentation - to capture specific MPI function performance. They also leverage the concept of a 'thunk stack', a marker inserted into the call-stack as a point of reference to reduce call-stack traversal overheads [16]. Issues surrounding the scalability of fine-grained call-stack tracing, from a sampling perspective, are discussed in [17].

In the development of the memory tracing tool we developed a proprietary trace format, as opposed to using existing formats such as the Open Trace Format (OTF) [18]. The primary motivation for this decision is the reduction in data volume, afforded by the application of domain specific knowledge. Whilst the portability of trace files is important, at this stage we have only focused on the movement of the files rather than compatibility with other software.

Woodside and Schramm present a formal notation for supporting the storage and analysis of complex, event-driven parallel traces. Their solution, entitled NICE (Notation for Interval Combinations and Events [19]), provides an automata-based language for querying trace data, which contrasts with other techniques, such as OTF, which focuses primarily on the trace format. As we transition to a single parallel trace file in future iterations of the tool, we hope to exploit this capability to enhance the analysis.

The technical difficulty of trace compression, especially call-stack traces, has been the focus of a large volume of research, due to the impact on the scalability of tracing tools. Many techniques have been developed to reduce data volume, including 'lossy' and 'lossless' compression formats [20, 21]. Like many other tools the success of the compression technique, with respect to both overhead and compression ration, are paramount to the success of `WMTools`.

The construction of the memory model, in the later part of this study, leverages prior work in developing performance models for high performance codes [22, **?**].

3 WMTrace - Memory Event Tracing

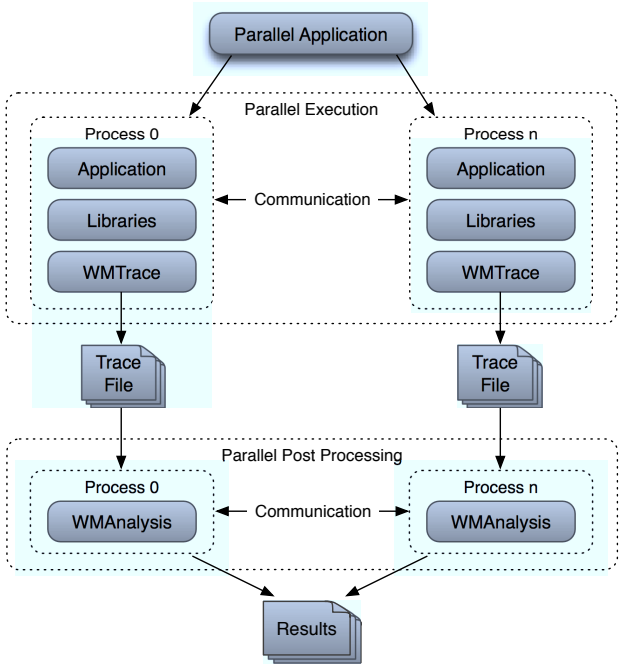

Fig. 1. `WMTools` workflow

Memory allocation analysis through `WMTools` is performed as a two stage process. The first stage, data collection, is performed during code execution by the `WMTrace` library. This library is linked at runtime using the operating system's linker, and intercepts the POSIX memory management functions whilst the code executes. During this execution the library collects vital information regarding the size, time and location of memory allocations. This data is then streamed to file, via an internal buffer, which in turn streams to a compression buffer.

The four main POSIX memory management functions of interest to `WMTrace` are `malloc`, `calloc`, `realloc` and `free`, the motivation behind this decision is the availability of such functions on different platforms, thus increasing the portability of the library. The choice of functions also allows us to target multiple programming languages, specifically `Fortran`, `C` and `C++`, which remain the dominant programming languages in high performance scientific computing.

The tracing library, `WMTrace`, is specifically targeted at parallel codes based on the Message Passing Interface (MPI). The operating systems linker attaches a

different instance of WMTrace to each process of the parallel job. Each instance of WMTrace monitors its own process, generating a single trace file for that specific process. As illustrated in Figure 1, the parallel execution of the application, with the tracing library attached, can be followed by a parallel post-processing phase, discussed in Section 4. Due to the segmentation of the library instances, WMTrace and additionally WMAnalysis, can also be used on serial codes with no MPI environment.

The WMTools suite is written in the C programming language, and makes use of third party libraries, such as Libunwind, Libelf and ZLib; further details of the framework used can be found in previous work [23, ?].

3.1 Event Tracing

At the core of WMTrace is the notion of an 'event', which is triggered by the interception of a POSIX memory management function call. These function calls are intercepted through the use of function interposition, through the dlsym library, allowing WMTrace to collect additional information at the point of occurrence. The information gathered for each event is dependant on both the event type and user preference. For memory allocations using malloc and calloc, the tool records the time and size of the allocation, and the return pointer, which is used to map a future de-allocation back to this event. For deallocation events the time and memory address is retained, but not resolved at runtime, to reduce overheads.

One optional feature of WMTrace is to leverage the functionality of the third party Libunwind, to traverse the function call-stack from any given point. The call-stack collected from this procedure enables WMTrace to attribute a memory allocation to a given function, and furthermore to a unique call path. This is particularly useful for functions containing allocations which are referenced from multiple points in the code, for example local allocation functions. The cost of such call-stack traversals dramatically increases the overheads of the library, due to the frequency of calls, and the context switch incurred by making the call. In some circumstances, for instance C++ codes with repeated object instantiation and destruction, this overhead may be undesirable, we therefore allow this feature to be disabled via an environment variable. By disabling call-stack traversal, functional breakdown analysis is no longer feasible, but the libraries ability to calculate high-water mark and perform temporal analysis remains.

3.2 Trace File Compression

The compression library attached to WMTrace is essential to boost overall performance, as the volume of data generated by the I/O and storage may otherwise limit the scalability of the tools. Despite separate files being maintained for each process (which limits the impact of file locking), the size of these files increases with runtime (along with other factors).

One of the most important techniques for overcoming this data challenge is to use compression. Due to the repetition found within an execution, a very

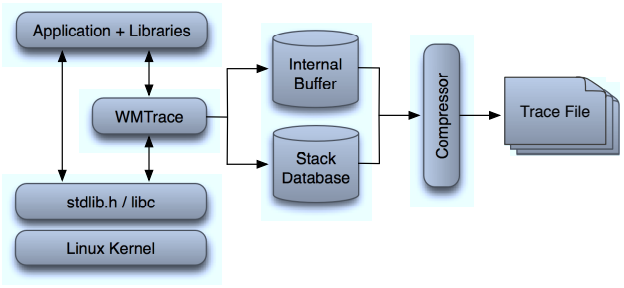

Fig. 2. The `WMTrace` compression process

simplistic compression algorithm is sufficient to achieve significant compression ratios, in the order of 13×. `WMTrace` utilises the third party compression library `ZLib`, as it is a highly portable library with minimal overheads and yet is able to yield significant gains in storage capacity (see Section 5.5).

`WMTrace` utilises an internal buffer, of user defined size, to store the trace stream. When this buffer is full, it passes the data through the compression stream, which in turn outputs to disk. The motivation behind this design choice is to minimise I/O operations, but at the same time minimise memory consumption. Figure 2 illustrates how data flows out of `WMTrace` into the final trace file.

To improve the compression ratio, `WMTrace` eliminates as much basic repetition as possible. One of the most significant enhancements is the inclusion of a call-stack dictionary. Rather than recording the full call-stack in the trace file, for each memory allocation, the library maps each unique call-stack to an ID, which is written to file rather than the call-stack. The dictionary then systematically writes the new map entries to the trace file at intervals, within a special frame from enhanced post processing. This technique lessens the burden on the compression library by dramatically reducing the volume of data it must process, which in turn reduces the runtime overheads of the compression library and improves the overall compression rate.

Other techniques used to improve compression are targeted at the complexity of the data to be stored. The timestamp for events represents a significant source of variance in the trace output, contributing a high amount of entropy to the compression stream. By simplifying the timestamp data, storing it in a reduced precision format (`float` rather than `double`), we observe a significant reduction in compressed file size of up to 18%. We also improve compression by recording time as deltas, the difference in time between two events, rather than the total elapsed time. The advantage of this is the increase in repetition (there will clearly be no repetition in time, whereas repetition in deltas is common). This technique can reduce the compressed file size by up to 24%, and can reduce the time taken to compress the output stream.

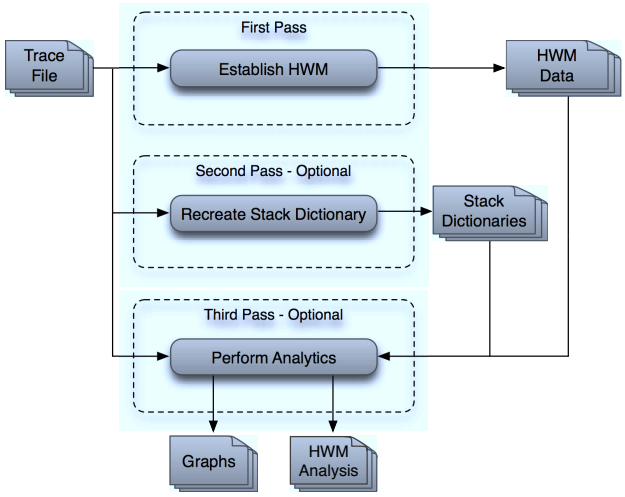

Fig. 3. `WMAnalysis` process diagram

3.3 Static Allocation Analysis

One of the limitations of event-driven memory consumption analysis is the failure to capture static memory allocations. In `Fortran`, for example, if a statically sized array is allocated to the stack, but is too big to fit, it will be pushed to the heap. This allocation will happen at compile time, rather than runtime, and so is difficult to identify during runtime.

To gain a more complete picture of memory consumption, `WMTrace` attempts to capture this static memory by analysing the binary executable. This is performed using the `Libelf` library, a lightweight library for reading and writing Executable and Linkable Format (ELF) files. Using this tool `WMTrace` can analyse the binary headers and look for memory allocations from compilation. Whilst `WMTrace` can not determine the source of the allocation, to assist with the functional breakdown, it treats the consumption as persistent through the whole execution, thus adding a constant to the high-water mark.

4 WMAnalysis - Memory Trace Analysis

The tracing component, `WMTrace`, is merely a data collection agent. Most of the complexities of `WMTrace` arise from minimising the overheads and reducing the impact on the executing code. `WMAnalysis` allows the resulting data to be manipulated for different purposes, to address different memory problems in code.

`WMAnalysis` provides a suite of analysis tools, to process the trace files in different ways with varying complexity. At its simplest the analysis will produce

a memory high-water mark. More complex analysis includes point-to-point trace file comparisons to evaluate allocation continuity for the purpose of memory model construction.

The analysis of `WMTrace` files is simplified by their structure, the layout of frames in a binary file makes it very simple to `skip` through portions of the output to the desired section. For this reason analysis is performed in a multi-pass manner, each pass gaining more information about the trace. Figure 3 demonstrates how, with a three-pass analysis, it is possible to obtain different levels of detail.

One of the most important features of the separation of the trace and analysis phases is the portability of the analysis. By generating a complete trace file, the source binary only needs to be executed once, but the analysis can be performed multiple times, at different levels of granularity. Further to this, the, trace file is also portable, so whilst execution may require a supercomputer, analysis can be performed on a desktop server.

Although `WMTools` is not designed for memory debugging, certain debugging statistics can be obtained 'for free'. One of the most useful features in this regard is the identification of memory leaks, which can be detected and traced back to a source function and execution time.

4.1 High-Water Mark Analysis

High-Water Mark (HWM) analysis is the most fundamental component of `WMAnalysis`, as it establishes the high point of memory consumption through the source program's execution. The implementation of `WMTrace` ensures a single trace file is generated per process, thus a high-water mark analysis will return the high-water mark for that process; for a more complete picture we need to take aggregate statistics over all trace files from the execution.

Unlike other tools `WMTrace` does not perform any analysis during execution; all analysis is performed post-execution. For this reason even simplistic statistics like high-water mark must be obtained via post processing. The mechanism through which memory is allocated and de-allocated in POSIX systems means that whilst memory is allocated with a given size, it is impossible to accurately calculate the size of a variable when it is de-allocated. This means that a *tracker* must be created which persists through the lifetime of that variable, which links the two allocate and de-allocate statements together. This can be performed through a relatively simplistic linked-list data structure, but the downside to this approach is the increased memory consumption. `WMTrace` makes every effort to minimise the additional memory and runtime overheads during the execution of the source binary, and thus this process is delayed until the post processing phase in `WMAnalysis`.

4.2 Post Execution Analysis

As previously stated, no analysis is performed during the tracing phase, but often a user desires at least some basic memory statistics at the time of program

execution. To enable this `WMAnalysis` is made available as a library, which allows `WMTrace` to 'hook' into it and perform some basic analysis in parallel.

This analysis exploits the existing parallel MPI environment to inspect each trace file on a different process (i.e. the one it was generated on). Subsequent analysis consists of establishing the high-water mark of each process, and presenting aggregate statistics across all the processes. The structure of this post-execution analysis, with reference to the original trace, is illustrated in Figure 1, the actual analysis performed is represented by the 'First Pass' shown in Figure 3. The aggregate statistics focus on the memory high-water mark and the variance between the different processes, specifically: maximum high-water mark and process ID; minimum high-water mark and process ID; mean high-water mark and standard deviation.

This analysis is deliberately simple in order to reduce runtime, but provides initial statistics that then allow for a more focused secondary analysis phase, outside of the MPI processes.

4.3 Temporal Breakdown

One of the most interesting forms of trace analysis is exploiting the temporal data to replay the execution. From the order and distribution of memory allocations, we can graph the memory consumption over time. This visual representation provides useful insights into the memory profile of an application.

On a fundamental level we are interested in how much memory is used at peak memory consumption, to determine how much memory a job requires. Beyond this we are also interested in the duration of the peak consumption. This information can assist with code modification to reduce memory footprint. If peak consumption is sustained throughout a large portion of the execution, suggesting a large data set, then addressing the storage mechanism for the data set could help reduce the overall consumption. Whereas a short-lived peak might be indicative of a specific algorithm or technique, for example an out of place matrix transpose, which could be refined with a more memory centric approach.

The comparison of temporal graphs between different sized runs (either problem size or core count) can help identify a lack of continuity, which would otherwise be expected. In Section 5.4 we demonstrate this temporal analysis as part of a case study.

4.4 Functional Breakdown

Functional analysis extends the temporal analysis by analysing 'who' is actually consuming the memory. This stage of analysis requires stack tracing, as it looks at unique call paths in code which consumed the highest volume of memory. Whilst this analysis can be performed in addition to temporal analysis, in the form of a stack graph, we are often more interested in just the breakdown at the point of high-water mark. Again we demonstrate this analysis on a selection of our case study applications in Section 5.4.

4.5 Parallel Analysis

Parallel analysis is an in-depth approach to trace file comparison, looking at more than just high-water mark. This form of analysis takes two trace files as input, and establishes a relationship between them by performing a pointwise functional comparison at the time of the high-water mark.

The purpose of this mapping is to establish which functions allocate memory objects with dependencies on external factors, such as problem size or core count. Thus by taking a trace file from the high-water mark process from an 8-core run, and comparing it with a similar high-water mark process trace from a 16-core run, the parallel analysis will identify which live allocations, at the point of high-water mark on both processes, have different sizes, and by what factor they have changed by.

There are many applications to this form of analysis, it can aid the construction of memory models, pinpoint key growth functions and even identify continuity problems. We demonstrate the parallel analysis in Section 6 where we construct a memory model for an application and validate it against actual high-water mark results.

5 Case Studies

In the following section we illustrate the use of `WMTools` with nine different scientific applications/benchmarks, identified in Table 1. These codes, used either independently or as part of a larger workflow/application, represent significant proportions of the compute time at supercomputing sites around the world [24–30]. These codes are also interesting from a technical perspective because they represent the three principle implementation languages – `C`, `C++` and `Fortran 90` – which are used to write most modern parallel scientific applications. The ability to successfully trace each of these languages is critical if our tool is to be more generally applicable.

While memory consumption is technically non-deterministic, as a result of the temporal overlapping of buffers, experimentation has shown minimal variation in high-water mark. The peak memory consumption of a 32-core run of `Graph500` was seen to vary by around 580KB, 0.08%, over five runs, and we experience no variation for `miniFE`. In the case of `phdMesh`, we observed a 'best' and 'worst' case memory consumption, depending on the decomposition, we have reported the worst case for consistency. For this reason the results presented are for single executions, as opposed to averages.

5.1 Applications/Benchmarks

For each of these codes we have selected an appropriately sized input deck and used the `WMTrace` library to record memory behaviour over a variety of core configurations. To illustrate how the memory consumed alters as the applications scale, the problems are strong scaled over 16-, 32-, 64-, 128- and 256-core executions.

Table 1. Demonstrator applications and benchmarks used in our analysis

	Language	Description
miniFE	C++	Unstructured finite element solver
phdMesh	C++	Unstructured mesh contact search
DLPoly	Fortran 90	Molecular dynamics simulator
Lare3D	Fortran 90	Non-linear molecular hydrodynamics
AMG	C	Parallel algebraic multigrid solver
LAMMPS	C++	Classical molecular dynamics
LU	Fortran 90	LU PDE solver
FT	Fortran 90	FFT PDE solver
Graph500	C	Graph solver

The following experiments were performed 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 Infiniband. Each node provides 24GB of system memory (2GB per core). The runs presented all use the Intel 12.0.4 compiler toolkit with OpenMPI 1.4.3, compiled with the `-O3` optimisation flag and debugging symbols.

5.2 HWM Analysis

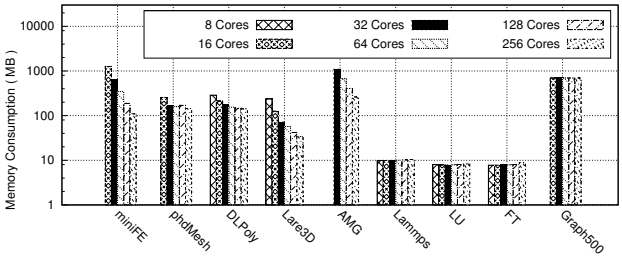

Fig. 4. Application memory consumption

In Figure 4 we can see the memory scalability of the different benchmark codes. We have selected these particular benchmark codes because they demonstrate variations in memory scalability. Some codes, `miniFE` and `Lare3D`, exhibit good memory scalability, as the memory consumption continues to drop as the core count is increased. Other codes, such as `phdMesh`, have a more complex memory behaviour, as the memory consumption can either go up or down depending on the decomposition. Finally the last class of code, `LAMMPS`, `LU`, `FT` and `Graph500`, do not demonstrate a decrease in memory consumption as the number of cores is scaled. We note that gaps exist in the data due to incompatible decompositions and insufficient memory.

To properly understand the complexities of a code with poor memory scalability we need to employ further analysis. By observing the per-core memory

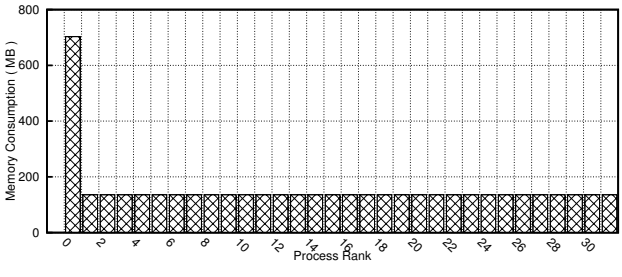

Fig. 5. `Graph500` memory consumption per process on 32 cores

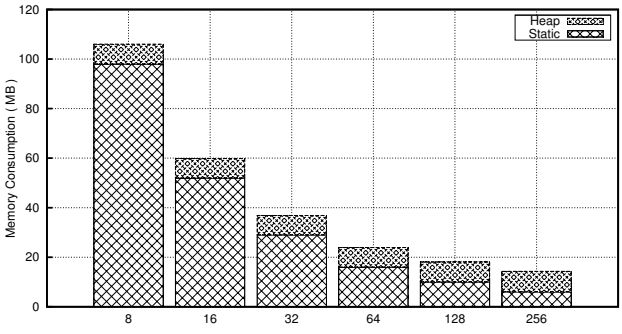

Fig. 6. `LU` static and dynamic memory consumption

consumption of a 32-core run of `Graph500`, Figure 5, we can see that the memory high-water mark is only occurring on process 0, and that there is a significant imbalance between this and the remaining processes.

5.3 Static Memory

From Figure 4 we observe that the heap memory consumption of `LU` and `FT` remains static at around 8 MB as the code is strong scaled. However, this is not the complete picture, and it is only by employing the static memory analysis component of `WMTrace` that we can expose the true memory requirements of these two benchmarks. During compilation memory is statically allocated within the binary to handle the specific decomposition of the problem.

Using the static memory analysis component of `WMTrace`, which uses `Libelf`, we can expose this memory consumption. We demonstrate how this static memory consumption varies as we strong scale both `LU`, Figure 6, and `FT`, Figure 7. From these figures it is evident that the memory profile of both codes is significantly different from that previously exposed through just heap-based analysis, as the dominant factor in consumption comes from static allocations.

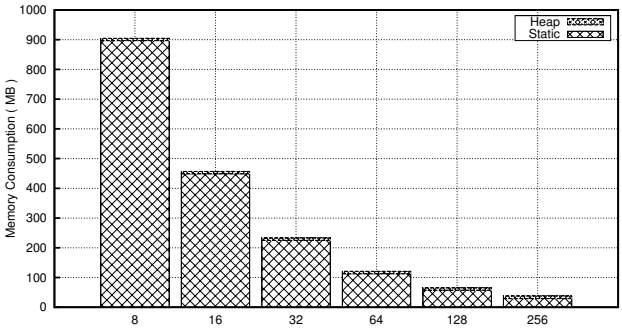

Fig. 7. `FT` static and dynamic memory consumption

5.4 Temporal and Functional Breakdown

Table 2. Per process memory HWM comparison for `phdMesh`

Cores	Min (MB)	Max (MB)	Mean (MB)	Std. Dev. (MB)
16	239.50	255.26	251.54	4.33
32	149.34	164.96	161.75	3.56
64	122.40	158.46	154.87	5.14
128	92.18	170.21	139.49	18.14

Table 2 shows how the high-water mark values vary for each process in a run of `phdMesh` as we scale the number of cores. Increasing the core count reduces both the minimum and mean high-water marks, while the maximal high-water mark increases at 128 cores. The standard deviation between the high-water mark values identifies large discrepancies between memory consumption on different processes, particularly in the case of 128 cores. This is indicative of problems in the data decomposition. By studying the temporal high-water mark analysis of `phdMesh` on 128 cores (see Figures 8 and 9) we can examine the differences in memory consumption between the maximal and minimal high-water mark processes. Figure 8 shows the maximal high-water mark thread, with a peak memory consumption of 170MB; Figure 9 shows the minimal high-water mark process, with a peak memory consumption of 92MB. It is clear that both processes have a very similar temporal memory trace, despite the difference in peak memory consumption. We see a start up phase with significantly increased memory consumption during the first 15% of execution, followed by sustained consumption after this point. Despite the large variation in high-water mark values (an 85% increase from the minimal value) the sustained memory consumption is very similar, at around 20MB. At the end of the start-up phase in `phdMesh` a re-balance is performed – to ensure a consistent decomposition – this

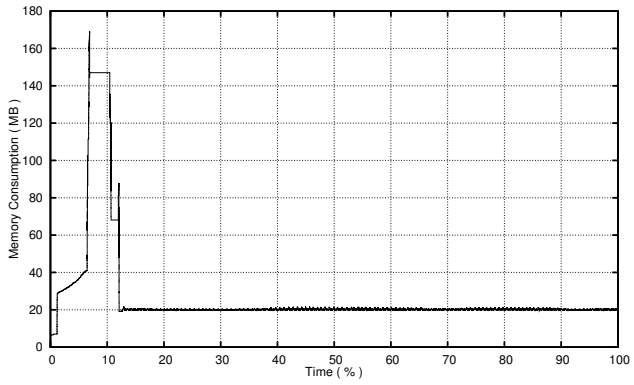

Fig. 8. Temporal memory trace for `phdMesh` maximal high-water mark process on 128 cores

coincides with the decrease in memory in our temporal analysis. This is suggestive of the application preloading data, which is then discarded/redistributed for the actual computation phase. It is highly likely that this operation could be arranged in a more efficient configuration which would significantly reduce the application's high-water mark, and the initial variation between processes.

To aid in this redesign, we analyse the functional breakdown of the high-water mark for both the maximal and minimal processes for each run of `phdMesh` (see Figure 10). From this breakdown, we see that despite the variations in memory consumption, the proportions of each function remain similar between the maximal and minimal high-water mark threads and between runs of different size. This indicates that the memory consumption is distributed across all of the primary functions, rather than just being limited to a single function involved in the start up. Although this suggests that it might be difficult to uncouple the data causing the high-water mark from the algorithm, it does confirm that the initial data distribution is the root cause of the high-water mark problem.

5.5 Overheads and Compression

Table 3 presents the slowdown incurred by memory tracing with `WMTools`. This slowdown represents the additional cost to capture the memory management data, store it to file, and perform an initial analysis of the results to produce a high-water mark result. These slowdowns represent a significant improvement over previously published results [23, **?**], due to both improvements in the tool suite and the testing environment. Due to the nature of the tracing, many context switches are incurred as execution is paused to perform tracing, thus a platform's ability to handle these context switches will have a significant impact on the overheads of the suite.

As a general rule, the proportional slowdown increases as the core count increases, this is primarily a result of decreased runtimes for similar data volumes.

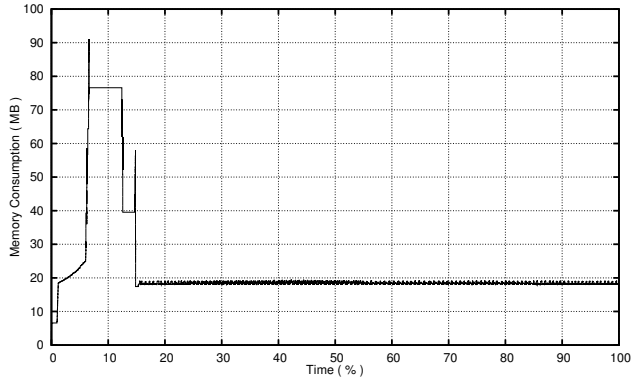

Fig. 9. Temporal memory trace for `phdMesh` minimal high-water mark process on 128 cores

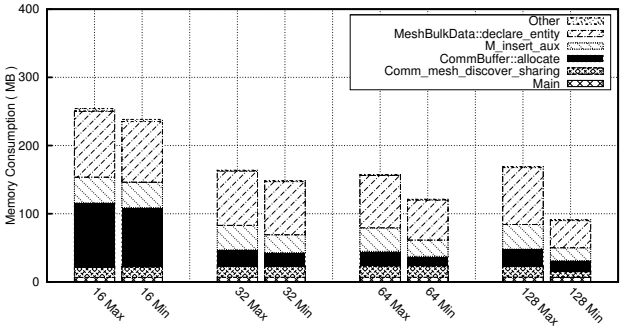

Fig. 10. Functional breakdown of `phdMesh` for minimal and maximal HWM

The strong scaling of a code often has little effect on the number of allocations performed, but will likely reduce the base runtime. Thus the tracing library must handle a similar number of allocations, generating a similar amount of data, but do so in a shorter time frame and thus with increased impact.

The volume of data generated by `WMTrace` is a key consideration. Although aspects of the runtime environment are configurable (the type of data to collect, the amount of data generated etc.) the amount of data generated can be problematic in some circumstances. Table 4 illustrates the total volume of data generated for specific runs, with full data collection enabled. The size of these trace files is proportional to the number of memory management calls made during execution, but due to the storage of stack traces their size has a minimal impact on the overall file size. As `WMTrace` generates one file per process, larger core count runs will generate more files and therefore, most likely, more data. Due to the existing compression on files, and the extracted repetition, there is little advantage to file amalgamation and re-compression.

Table 3. Application slowdown with `WMTrace`

	8	16	32	64	128	256
miniFE	-	1.1	1.1	1.4	1.8	2.9
phdMesh	10.5	6.0	8.0	10.4	9.2	11.8
DLPoly	1.1	1.2	1.1	1.0	1.0	1.0
Lare3D	1.0	1.00	1.0	1.0	1.0	1.1
AMG	-	-	1.1	1.2	1.2	1.7
LAMMPS	1.3	1.4	1.9	1.4	1.5	1.2
LU	1.0	1.2	1.1	1.0	1.0	1.3
FT	1.0	1.1	1.0	1.0	1.1	1.0
Graph500	-	1.0	1.0	1.0	1.2	1.2

Table 4. Trace file size in MB

	8	16	32	64	128	256
miniFE	10	37	71	230	456	904
phdMesh	2503	1433	2662	4812	6758	8089
DLPoly	219	258	353	384	494	770
Lare3D	4	11	19	41	84	155
AMG	-	-	453	465	481	518
LAMMPS	4	8	17	33	65	129
LU	1	2	4	8	17	33
FT	1	3	5	9	21	33
Graph500	-	24	25	49	97	65

In some circumstances the user may only be interested in specific artefacts, for example the peak high-water mark thread, or discontinuity between two particular processes. In this case the remaining files may be removed, with no impact on the analytics of the remaining files.

5.6 Comparison with other tools

Table 5. Application slowdown with alternative tools on 64 cores

	memP	Massif	Memcheck
miniFE	4.1	5.1	12.9
phdMesh	28.3	13.9	15.5
DLPoly	-	6.3	18.1
Lare3D	1.0	-	-
AMG	-	13.9	21.2
LAMMPS	1.0	5.3	16.5
LU	1.0	4.1	14.6
FT	1.0	9.5	18.4
Graph500	1.1	2.5	8.0

We compare the runtime overheads of `WMTools` against three alternative heap profiling tools, `memP`, `Massif` and `Memcheck`. Each tool performs different levels of analysis, but nonetheless provides a baseline for comparison. Table 5 illustrates the overhead of the three tools for a 64-core execution of the nine benchmark codes. These can be compared with the `WMTools` overheads presented in Table 3; we believe that this demonstrates the competitive performance of `WMTools` against existing tools.

memP is a similar memory tracing tool aimed at HPC codes, yet unlike WMTrace it performs all of its analysis during execution, and outputs a final result at the end of execution. Whilst memP does support call-stack traversals, again through Libunwind, we have not enabled this during testing. We can see from a comparison of these two tables that the overheads of the two tools are comparable, both exhibiting minimal slowdowns on the majority of codes. For miniFE and phdMesh we see significantly more slowdown when using memP compared with WMTrace, despite the lack of I/O overhead in writing trace files to disk.

We note that the memory consumption result from memP is consistently about 7MB less than that shown by WMTrace, Massif and Memcheck. We were unable to obtain results for both AMG and DLPoly using memP.

Massif, part of the Valgrind suite, is a heap profiling tool that is specifically designed to measure memory usage in the heap and stack. This is achieved through snapshots, measuring memory usage at the malloc'd block level. For our comparison we have not enabled stack memory profiling. Table 5 shows how the overheads of Massif are approximately inline with those incurred by both WMTools and memP.

Memcheck is another tool from the Valgrind suite, this time aimed at the identification of memory errors. Although this is significantly different to heap profiling, the methodology employed is applicable. Memcheck calculates the peak memory consumption by sandboxing memory and recoding metadata about accesses. This means that it can very accurately record the memory consumed, and illustrates the high overhead of the technique, as illustrated in Table 5.

Due to difficulties executing Lare3D with the Valgrind suite we have excluded the results from this evaluation.

In general we observe significantly higher overheads on Massif and Memcheck than previously reported for WMTools. The overheads incurred on phdMesh, are significant on all four heap profilers, ranging from 10.4× for WMTools to 28.3× for memP. Overall we observe an average reduction in overheads for WMTools of 1.09× over memP, 3.41× over Massif and 6.47× over Memcheck.

6 Building a Memory Model

Using the parallel analysis feature of WMAnalysis we perform a pointwise comparison of two memory traces from different runs of the same code, but at increased scale. From this comparison we establish a relationship between the size of memory allocations, the high-water mark, and the problem size and core count.

When strong scaling a code, using an increased core count to solve the same problem size, it is often expected that the memory consumption will scale proportionally with the core count. This assumption is based on a simple reduction of the data set per core, which is the fundamental basis of strong scaling. The expectation of a halving of memory consumption for a doubling of core counts, is both naïve and unrealistic; this complexity of memory scaling is one of the

prime motivations for `WMTools`. The composition of the memory consumption of an application will be based on three components: constant memory consumption, variable memory consumption based on problem size and variable memory consumption based on core count. The constant memory consumption, consisting of variables, will not change as the problem size or core count changes. The variable memory consumption based on problem size is often the primary consideration as it is commonly the dominant factor at small scale. The variable consumption based on core count is often overlooked, this is the memory dedicated to internal communication buffers and the root cause of many memory scaling issues, as at large scale it can quickly become the dominant factor.

Take as an example, a 1D processor decomposition of a 3D cubic problem space, decomposed in the z dimension. Let each processor store a regular portion of the cube, a slice, as well as two faces from neighbouring processors. These two faces will be in the x and y dimension. For a cube $m \times m \times m$ decomposed over n processors, each processor will store a slice of the cube of size $m \times m \times \frac{m}{n}$, and two faces of size $m \times m$. The total size of the stored elements will be $\frac{m^3}{n} + 2m^2$.

When strong scaled to $8n$ processors, the size of the slice per processor is halved, becoming $m \times m \times \frac{m}{8n}$. However, the size of the faces remains the same, so the total size stored per core is $\frac{m^3}{8n} + 2m^2$. Thus, even for a problem-size dependent allocation, there is a constant memory consumption when changing the number of processors.

If, on the other hand, we weak scale the global problem, now $p \times p \times p$ such that $p = \frac{m}{2}$, then for an 8-times increase in core count, the total size stored per processor will be $\frac{m^3}{8n} = \frac{p^3}{n}$. Since we have changed the total problem size rather than the per core problem size, the total memory consumed in the first instance is $\frac{m^3}{8n} + 2m^2$ but in the second instance $\frac{p^3}{n} + 2p^2$. Here the slice sizes are the same, but the size of the faces has changed. Substituting p in terms of m shows the face size for the weak scaled case is $\frac{m^2}{2}$, which is $4\times$ smaller than the strong scaled case, $2m^2$. This highlights the impact of the per core problem size reduction on per processor memory consumption.

6.1 Pointwise Comparison

The first step towards a memory model is establishing how each allocation at the high-water mark changes as you scale the problem. For this we must take two traces from runs at different scale and perform a pointwise comparison between these allocations. From this we must look at how each allocation has changed, with relation to the change in problem size and core count.

Using `WMAnalysis` we can evaluate the pointwise differences in the high-water mark of two trace files, we can then take this information and map each allocation to a ratio proportional to a change in the per core problem size. `WMAnalysis` can then aggregate this information and produce a general formula to express the changes in memory consumption as the problem is scaled.

$$F(P, N) = C_1 \frac{P}{N} + C_2 N + C_3 \tag{1}$$

Equation 1 represents the three components of the memory model for problem size P and core count N, the first constant, C_1, represents the portion of memory that shrinks when the problem size per core is reduced. The second, C_2, represents the portion that increases with the core count, for example communication buffers. The third, C_3, remains constant as both the core count and the problem size per core changes as the core count is scaled.

Each of these terms hides the complexity regarding the underlying relationship between the allocations. With the C_1 term we can represent a doubling of per core problem size resulting in a doubling of the allocation size, but any form of relationship will be captured by this term. If you take the case of a completely square 2D decomposition of an $X \times X$ problem size over N processors, then each processor will get $\frac{X}{\sqrt{N}} \times \frac{X}{\sqrt{N}}$ cells. By doubling the core count, to $2N$, the size per processor is now $\frac{X}{\sqrt{2N}} \times \frac{X}{\sqrt{2N}}$. Thus a one row (or column) buffer would change in size from $\frac{X}{\sqrt{N}}$ to $\frac{X}{\sqrt{2N}}$, representing an allocation size decrease of ratio $\sqrt{2}$. Any allocations with this, or similar, behaviour have the same scaling factor as the C_1 term but represent a subtle difference in the underlying allocation relationship.

To establish these three constants `WMAnalysis` must first map the stack dictionaries, from each trace, to each other. This enables equivalent allocation sites to be compared, in the knowledge that they share a call-stack. From this we establish a ratio of the change in size of allocation, and compare it to the ratio of the change in problem size per core, and change in core counts. A decrease in allocation size, proportional to the decrease in problem size per core, is assigned to C_1. An increase in proportion to the increase in core count is assigned to C_2, and finally allocations of the same size are assigned to C_3.

This grouping makes many assumptions about the source of the allocations, and their variable dependencies, primarily due to the lack of data. Evaluating two traces, with one difference, in this case core count, means that any other artefacts of allocation size will not show. For example this method assumes a decomposition in a single dimension. When more than one dimension is decomposed, by strong scaling, this equation will have more terms which would be based on the size and shape of the original problem size. To properly capture the behaviour of these additional components would require the analysis of more than two trace files, and more information regarding the decomposition of each trace. This is currently beyond the scope of the automated model generation in `WMAnalysis`.

6.2 Automated Model Generation

In this section we build a memory model, based on the memory profile of `Lare2D`, a 2D variant of `Lare3D` used in Section 5. From the results presented we see that `Lare3D` has a reasonably simple memory profile; although it does not scale linearly, it still scales well.

We first trace the execution of a 2- and 4-core run of `Lare2D` on a 4096^2 problem size, the result of these executions are shown in Table 6. We then feed the trace files from these executions into `WMAnalysis`, which will generate the

Table 6. High-water mark results for `Lare2D` 4096^2

Cores	High-Water Mark (MB)
2	2259.91
4	1138.13

constants for Equation 1, from this we can then make projections for the memory consumption of runs at larger scale.

$$F(P,N) = 280.7\,\frac{P}{N} + 1016\,N + 15257783 \tag{2}$$

The automated model generation, as a result of analysing two trace files, produced Equation 2 to describe the memory scaling of `Lare2D`, in Bytes. As we can see `Lare2D` has very good memory scalability, only a very small component increases with core count, and has relatively low constant consumption, 14.5MB.

6.3 Validation

Table 7. Model prediction results for `Lare2D` 4096^2

Cores	Prediction (MB)	Actual (MB)	Error (%)
1	4505.22	4495.02	0.22
2*	2259.88	2259.91	-0.00
4*	1137.22	1138.13	-0.08
8	575.89	577.37	-0.26
16	295.23	296.47	-0.42
32	154.91	149.36	3.7
64	84.78	85.67	-1.04
128	49.75	58.98	-15.64
256	32.34	42.32	-23.59

To test the prediction of the model we generate estimated memory high-water marks for increasing core counts, and validate them against actual recorded values. The results are displayed in Table 7, where it is clear that the predictions are very accurate at low core counts, under predicting by less than 1%. The model is less accurate at core counts above 64, where we under predict for 128 and 256 cores by 16% and 24% respectively. The actual runs used to generate the model are highlighted with an asterisk, and are naturally the most accurate predictions.

6.4 Model Progression

To understand this failure in the model we need to look at the temporal traces of the runs, shown in Figures 11, 12, 13 and 14. From these graphs we see how the predicted memory consumption maps to the actual memory trace of the run, and how the accuracy of the prediction drops at higher core counts. It is also clear

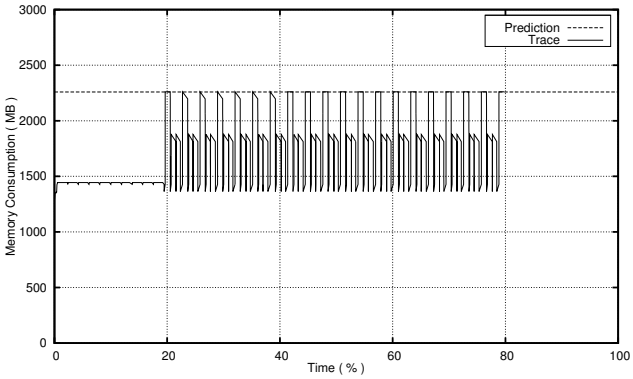

Fig. 11. `Lare2D` memory consumption predictions and real on 2 cores

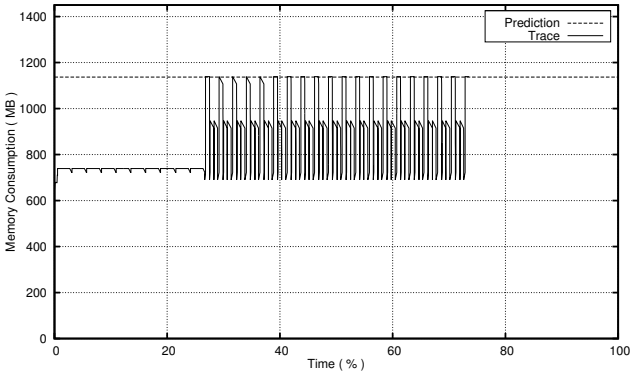

Fig. 12. `Lare2D` memory consumption predictions and real on 4 cores

why the accuracy drops at higher core counts, as the high-water mark changes. The prediction made from the 2- and 4-core runs predicts the high-water mark from the middle of the execution phase, whereas we see on the high core count runs the high-water mark now occurs at the start, and end, of the execution. We can also see that the prediction for the high-water mark continues to track this middle-phase memory consumption, thus our model is accurate for this portion of execution, but not accurate overall.

$$\begin{aligned} F(P,N) &= 180.1\,\frac{P}{N} + 94.5\,\frac{P}{\sqrt{2}N} + 566.5N + 24939843 \\ &= 246.9\,\frac{P}{N} + 566.5N + 24939843 \end{aligned} \tag{3}$$

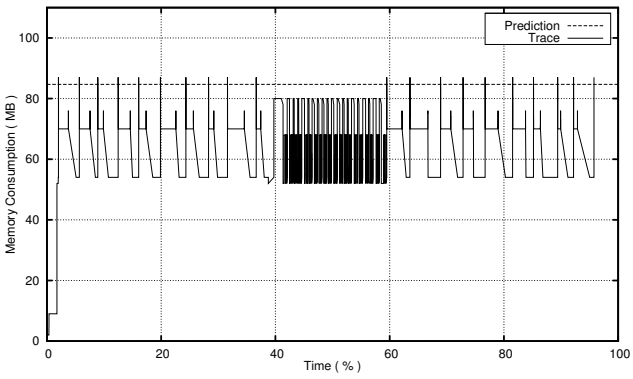

Fig. 13. `Lare2D` memory consumption predictions and real on 64 cores

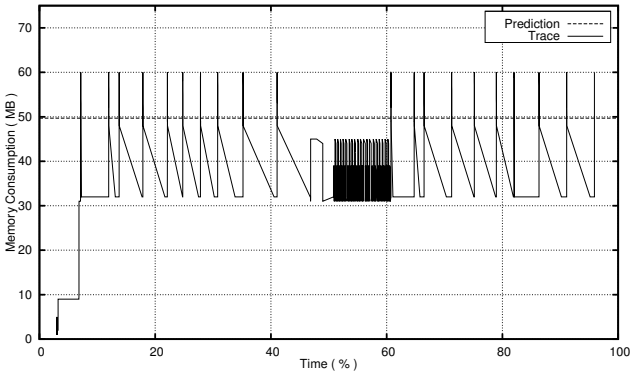

Fig. 14. `Lare2D` memory consumption predictions and real on 128 cores

By performing the analysis on the high core count traces we can establish a second equation, to better represent this new high-water mark, shown in Equation 3. As discussed in Section 6.1 the $\frac{P}{\sqrt{2N}}$ term introduced in Equation 3, represents a more complex relationship in allocation sizes than the $\frac{P}{N}$ term but, fundamentally, has the same scaling factor and can be represented in the same form.

We can see that the constants for Equation 3 are significantly different to those in Equation 2. As the C_1 term has reduced, from 280.7 to 246.9, we will see this new high-water mark reducing at a much slower rate than before, signifying a reduction in memory scalability for `Lare2D` above a certain point; for this problem size this is at 64 cores. These results are validated in Table 8, and exhibit a significant improvement in accuracy.

Table 8. Advanced model prediction results for `Lare2D`

Cores	Prediction (MB)	Actual (MB)	Error (%)
64*	85.55	85.67	-0.14
128*	54.71	58.98	-7.23
256	39.35	42.32	-7.02

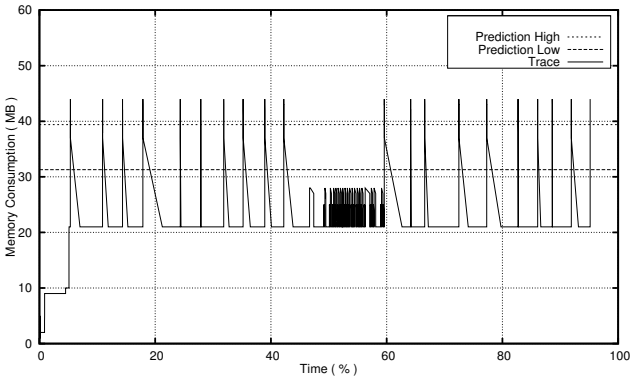

Fig. 15. `Lare2D` 256 cores model predictions, maximal high-water mark process

To form a combined model we would need to take the point where these two functions cross, represented by the maximum of the result of the two equations.

It is worth noting that this transition to Equation 3 is to predict the peak memory consumption across all the processors in a run. Here we can see the impact of the I/O phase of execution, which is handled by MPI-I/O. By inspecting the high-water mark of the other processes we see that there is a large disparity between the memory on the highest consuming process and the lowest consuming process, due to the layout of the parallel I/O.

Figures 15 and 16 illustrate the differences in memory profile between the highest consuming process, number 120, and the lowest, number 99. The impact of the I/O phase on the high consuming process is apparent, and the minimal effect on the low consuming process. For the low consuming process we see that the computation phase is still the dominant memory factor, this was captured by Equation 2, which over-predicts by 22%, based on a prediction for the high-water mark process.

6.5 Further Model Predictions

Further interesting applications of the memory model come when applying Equations 2 and 3 to different problem sizes. This enables us to predict both smaller and larger problems at scale using no further analysis.

Table 9 demonstrates the ability to predict the memory consumption for a 8192^2 problem, $4\times$ the analysed problem size, on different core counts. We can see that the model remains proportionally accurate throughout the test, over

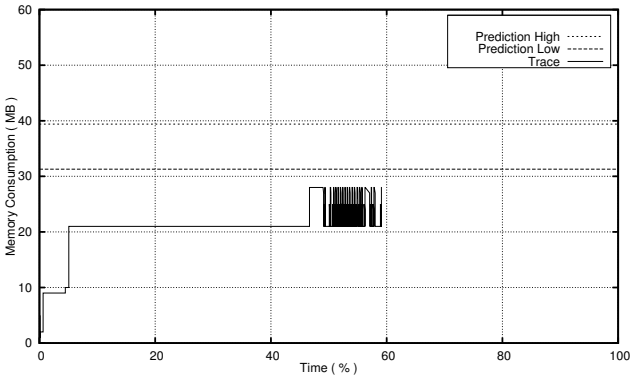

Fig. 16. `Lare2D` 256 cores model predictions, minimal high-water mark process

Table 9. Advanced model prediction results for `Lare2D` 8192^2

Cores	Prediction (MB)	Actual (MB)	Error (%)
1	17977.43	17942.23	0.20
2	8995.99	8978.81	0.19
4	4505.27	4495.13	0.23
8	2259.91	2253.38	0.29
16	1137.24	1131.62	0.50
32	575.92	570.83	0.90
64	295.28	289.97	1.83
128	155.01	149.64	3.59
256	85.65	86.18	-0.61

predicting for up to and including 128 cores, and then under predicting at 256 cores when Equation 3 begins to dominate. Throughout the validation the model demonstrates accuracies exceeding 96%.

7 Conclusion

The diverging gap between processor and memory performance has been well documented. As the designers of processors utilise multi-core chips to improve compute performance still further, a series of additional concerns in architecture design have arisen. First, the slow rate of improvement in memory capacity has resulted in a reduction in the memory available per-core, and second, the increase in core density has resulted in higher levels of contention for memory channels. When these two factors are combined with the increasing scale of contemporary supercomputers, the efficient utilisation of memory at runtime is rapidly becoming a significant concern for the design of scalable applications.

In this paper we present `WMTools`, a parallel application memory utilisation analysis framework, comprised of a memory allocation tracing tool `WMTrace` and a supporting analysis tool `WMAnalysis`. These tools enable users to trace calls to POSIX memory handling functions in distributed MPI codes without modifica-

tion or recompilation of the application. The second half of this paper describes a case study in which we apply `WMTools` to tracing the memory allocation behaviour of nine applications and scientific benchmarks. We then describe the automatic generation of models by `WMAnalysis`, and validate a generated model using the `Lare2D` application. The results of this study demonstrate the use of `WMTools` in:

- Analysing the allocation and freeing of memory during application execution. The ability to track memory use over time represents a clear advantage over existing tools which report only aggregated statistics such as high-water mark and, more importantly in the context of diminishing memory per-core, the opportunity to investigate isolated points during execution where memory usage spikes;
- The study of varying scientific codes, identifying memory high-water mark as the codes are strong scaled. Additionally we demonstrate further analysis with `phdMesh` - an unstructured mesh engineering benchmark. We highlight the ability to perform temporal and functional breakdowns - illustrating a large memory imbalance as the code is scaled.
- We present a comprehensive comparison of the overheads of `WMTools` with three other similar heap profiling tools on the nine benchmark codes at scale. From this we demonstrate the lightweight nature of `WMTools`, demonstrating between 1.1× and 6.5× lower overheads than the other heap profiling tools.
- Generating memory models that allow us to predict the memory high-water mark of an application on an arbitrary number of cores. The memory model was validated using the `Lare2D` application with between -7% and 4% error, for a range of core counts. We further extend the mode to capture a separate growth term in execution. We demonstrate the application of the extended model to predict memory consumption on a 4× larger problem set, at scale, with between -1% and 4% error.

Future Work

We plan to apply the modelling process to more complex scientific benchmarks, and to use this analysis to work with code engineers to reduce the memory footprints of the codes, and improve scalability. A further target of the model is to capture the memory consumption disparity between processes within the same job. Additionally we plan to continue exploring new techniques and methodologies to reduce tracing overheads, allowing the tool to scale to tera- and peta-scale executions of production workloads.

Acknowledgements

This work is supported in part by The Royal Society through their Industry Fellowship Scheme (IF090020/AM) and by the UK Atomic Weapons Establishment

under grants CDK0660 (The Production of Predictive Models for Future Computing Requirements) and CDK0724 (AWE Technical Outreach Programme). The performance modelling research is also supported jointly by AWE and the TSB Knowledge Transfer Partnership grant number KTP006740.

The analysis of the `Lare` codes is supported through the EPSRC project EP/I029117/1 (A Radiation-hydrodynamic ALE Code for Laser Fusion Energy).

Sandia National Laboratories is a multiprogram laboratory managed and operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04- 94AL85000.

References

1. Wulf, W. A. and McKee, S. A. (1995) Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, **23**, 20–24.
2. Murphy, R. (2007) On the Effects of Memory Latency and Bandwidth on Supercomputer Application Performance. *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, Washington, DC, USA IISWC '07, pp. 35–43. IEEE Computer Society.
3. Koop, M. J., Sur, S., Gao, Q., and Panda, D. K. (2007) High Performance MPI Design using Unreliable Datagram for Ultra-scale InfiniBand Clusters. *Proceedings of the 2007 IEEE/ACM International Conference on Supercomputing*, New York, NY, USA ICS '07, pp. 180–189. ACM.
4. Liu, J. et al. (2003) Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. *Proceedings of the 2003 ACM/IEEE International Conference on Supercomputing*, New York, NY, USA SC '03, pp. 58–71. ACM.
5. Feitelson, D. G. (1997) Memory Usage in the LANL CM-5 Workload. *Proceedings of the Job Scheduling Strategies for Parallel Processing*, London, UK IPPS '97, pp. 78–94. Springer-Verlag.
6. Chambreau, C. (2010). 'memP: Parallel Heap Profiling'. Sourceforge. `http://sourceforge.net/projects/memp/ (2 Febuary 2012)`.
7. Zorn, B. G. and Hilfinger, P. N. (1988) A Memory Allocation Profiler for C and Lisp Programs. Technical report., Berkeley, CA, USA.
8. Budanur, S., Mueller, F., and Gamblin, T. (2011) Memory Trace Compression and Replay for SPMD Systems using Extended PRSDs. *SIGMETRICS Perform. Eval. Rev.*, **38**, 30–36.
9. Burtscher, M. (2004) VPC3: A Fast and Effective Trace-Compression Algorithm. *SIGMETRICS Perform. Eval. Rev.*, **32**, 167–176.
10. Nethercote, N. and Seward, J. (2007) Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. *Proceedings of the 2007 ACM SIGPLAN Conference on Programming language design and implementation*, New York, NY, USA PLDI '07, pp. 89–100. ACM.
11. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005) Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *Proceedings of the 2005 ACM SIGPLAN Conference on Programming language design and implementation*, New York, NY, USA PLDI '05, pp. 190–200. ACM.

12. Nethercote, N. and Seward, J. (2007) How to Shadow Every Byte of Memory used by a Program. *Proceedings of the 3rd International Conference on Virtual Execution Environments*, New York, NY, USA VEE '07, pp. 65–74. ACM.
13. Payne, B. (2011). 'Virtual Machine Introspection (VMI) Tools'. Sandia National Laboratories. `http://vmitools.sandia.gov/` (2 Febuary 2012).
14. Laguna, I., Gamblin, T., de Supinski, B. R., Bagchi, S., Bronevetsky, G., Anh, D. H., Schulz, M., and Rountree, B. (2011) Large Scale Debugging of Parallel Tasks with AutomaDeD. *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA SC '11, pp. 50:1–50:10. ACM.
15. Ahn, D. H., de Supinski, B. R., Laguna, I., Lee, G. L., Liblit, B., Miller, B. P., and Schulz, M. (2009) Scalable Temporal Order Analysis for Large Scale Debugging. *Proceedings of the 2009 International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA SC '09, pp. 44:1–44:11. ACM.
16. Szebenyi, Z., Gamblin, T., Schulz, M., Supinski, B. R. d., Wolf, F., and Wylie, B. J. N. (2011) Reconciling Sampling and Direct Instrumentation for Unintrusive Call-Path Profiling of MPI Programs. *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, Washington, DC, USA IPDPS '11, pp. 640–651. IEEE Computer Society.
17. Tallent, N. R., Mellor-Crummey, J., Franco, M., Landrum, R., and Adhianto, L. (2011) Scalable Fine-grained Call Path Tracing. *Proceedings of the 2011 IEEE/ACM International Conference on Supercomputing*, New York, NY, USA ICS '11, pp. 63–74. ACM.
18. Knüpfer, A., Brendel, R., Brunst, H., Mix, H., and Nagel, W. (2006) Introducing the Open Trace Format (OTF). *Computational Science ICCS 2006*, Berlin, Germany, LNCS, **3992**, pp. 526–533. Springer-Verlag.
19. Woodside, C. M. and Schramm, C. (1994) Complex Performance Measurements with NICE (Notation for Interval Combinations and Events). *Softw. Pract. Exper.*, **24**, 1121–1144.
20. Noeth, M., Ratn, P., Mueller, F., Schulz, M., and de Supinski, B. R. (2009) ScalaTrace: Scalable Compression and Replay of Communication Traces for High-performance Computing. *J. Parallel Distrib. Comput.*, **69**, 696–710.
21. Knupfer, A. (2005) Construction and Compression of Complete Call Graphs for Post-Mortem Program Trace Analysis. *Proceedings of the 2005 International Conference on Parallel Processing*, Washington, DC, USA ICPP '05, pp. 165–172. IEEE Computer Society.
22. Hammond, S. D., Mudalige, G. R., Smith, J. A., Jarvis, S. A., Herdman, J. A., and Vadgama, A. (2009) WARPP: A Toolkit for Simulating High-performance Parallel Scientific Codes. *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, ICST, Brussels, Belgium, Belgium Simutools '09, pp. 19:1–19:10. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).
23. Perks, O., Hammond, S., Pennycook, S., and Jarvis, S. (2011) WMTrace - A Lightweight Memory Allocation Tracker and Analysis Framework. *Proceedings of the UK Performance Engineering Workshop (UKPEW'11)*, Bradford, UK, pp. 6–24. Inprint and Design.
24. Heroux, M. A., Doerer, D. W., Crozier, P. S., Willenbring, J. M., Edwards, H. C., Williams, A., Rajan, M., Keiter, E. R., Thornquist, H. K., and Numrich, R. W.

(2009) Improving Performance via Mini-applications (Mantevo Overview). Technical Report SAND2009-5574. Sandia National Laboratories, Albuquerque, NM, USA.
25. Todorov, I. T., Forester, T., and Smith, W. (2010). 'The DL_POLY Molecular Simulation Package'. STFC. `http://www.ccp5.ac.uk/DL_POLY/` (2 Febuary 2012).
26. Arber, T. D., Longbottom, A. W., Gerrard, C. L., and Milne, A. M. (2001) A Staggered Grid, Lagrangian-Eulerian Remap code for 3-D MHD Simulations. *J. Comput. Phys.*, **171**, 151–181.
27. Henson, V. E. and Yang, U. M. (2002) BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner. *Applied Numerical Mathematics*, **41**, 155 – 177.
28. Plimpton, S. (1995) Fast Parallel Algorithms for Short-range Molecular Dynamics. *J. Comput. Phys.*, **117**, 1–19.
29. Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1991) The NAS Parallel Benchmarks - Summary and Preliminary Results. *Proceedings of the 1991 ACM/IEEE International Conference on Supercomputing*, New York, NY, USA SC '91, pp. 158–165. ACM.
30. Bader, D. A., Berry, J., Kahan, S., Murphy, R., Riedy, J., and Willcock, J. (2010). 'The Graph 500 List: Graph 500 Reference Implementations'. Graph500. `http://www.graph500.org/reference.html` (2 Febuary 2012).