Temporal Classification of Allocations for Reduced Memory Usage

Kristi Belcher, David Beckingsale, Marty McFadden
Lawrence Livermore National Laboratory
Livermore, California, USA
{belcher6,beckingsale1,mcfadden8}@llnl.gov

Sam Schwartz
University of Oregon
Eugene, Oregon, USA
sam@cs.uoregon.edu

ABSTRACT

Umpire, a data and memory management API created at Lawrence Livermore National Laboratory (LLNL), provides memory pools which enable less expensive ways to allocate very large amounts of memory in HPC environments. In practice, memory pools commonly contain both allocations that persist for only a portion of the program (temporary) and those that persist for the entire program (permanent). However, too much of a mix of both temporary and permanent allocations can lead to pool fragmentation and cause the pool to perform poorly or even run out of memory. The Umpire team created a tool that uses a machine learning (ML) model to perform temporal classifications and categorize allocations as either temporary or permanent. We conducted several experiments using trace files from two LLNL applications to study how much memory can be saved when those allocations are separated into distinct pools. We found that our ML tool accurately classifies memory allocations and that separating these allocation types into distinct pools reduces overall memory usage significantly (for this study, up to 29.5%).

ACM Reference Format:
Kristi Belcher, David Beckingsale, Marty McFadden and Sam Schwartz. 2023. Temporal Classification of Allocations for Reduced Memory Usage. In *Proceedings of ACM Conference (SuperComputing '23).* ACM, New York, NY, USA, 3 pages.

1 INTRODUCTION AND BACKGROUND

Umpire [1] is an open source library created at LLNL that provides a unified, portable memory management API to accommodate modern HPC platforms with complex combinations of memory resources. Many of the large, multi-physics applications at LLNL use Umpire for memory management. To address the limited memory resources on HPC devices, Umpire provides memory pools which allow developers to allocate all needed memory at once instead of making multiple, smaller memory allocations which can become quite expensive, particularly with device specific APIs.

Sometimes, however, memory pool performance can suffer if the pool becomes fragmented. Fragmentation occurs when not all the available (i.e. empty) blocks of memory within a pool can be used, so a pool grows in size to accommodate a new allocation instead of reusing the available blocks already in the pool. Many of the multi-physics codes at LLNL have complex memory allocation patterns where some allocations persist for only a portion of the program (temporary allocations) yet other allocations persist for the entirety of the program (permanent allocations). Both of these types of memory allocations get placed in the same memory pool, making a mix of temporary and permanent allocations in the same pool and increasing the likelihood that fragmentation will occur.

Motivated to minimize the chances of fragmentation developing in a pool, the Umpire team created a tool that uses a machine learning model to temporally classify each allocation as temporary or permanent. With each allocation type categorized, the Umpire team then used Replay, a debugging tool included in the Umpire API that can track and trace memory allocations, to determine if separating those allocations into distinct memory pools would improve overall memory usage and memory pool performance.

2 METHODOLOGY AND APPROACH

For our experiments, we used SAMRAI [5], an open source library from LLNL for applied structured adaptive mesh refinement and BoBa [2], an applied mathematics library for tensor network algorithms. The Umpire team started by determining features to be used in the machine learning model. Table 1 shows the full list of features we created that describe our allocations in detail.

We determined there were 3 main categories of features and used Python and Scikit-Learn [4] to compute the feature importance of each.

Below is a summary of the most important types of features used in our model with the corresponding feature importance scores:

- Encoded Backtrace Information: Includes memory addresses and a stack trace for each allocation. **Feature Importance: 73%**
- Time-Related Information: Includes information about when an allocation was made and how long the allocation persisted. **Feature Importance: 15%**
- Allocator Frequency Information: Includes information about how active the Umpire allocator was which made the allocation. **Feature Importance: 6%**
- All Other Features: Includes any other feature we tried not already included above. **Feature Importance: 6%**

With our list of features, we performed a check to see which machine learning model produced the highest F1 score for a collection of test runs in both the SAMRAI and BoBa applications. After running each test multiple times, the highest F1 scoring model was then used to categorize our allocations. All of our tests produced a median F1 score of **0.998** or higher, giving us confidence that our classification model fit the data.

In our workflow, we first generated trace files with the Replay tool from running several SAMRAI and BoBa test problems. The

SuperComputing '23, November 2013, Denver, CO, USA
© 2023 Association for Computing Machinery.

Feature Name	Description
A	Time since Umpire manager created
B	Time since allocator created
C	Percentage of time elapsed after allocator was created
D	Percentage of time elapsed before allocator was created
E1	Allocator name includes the substring "temp"
E2	Allocator name includes the substring "perm"
E3	Allocator name and order index
E4	Allocator name and order is the same as the first created
E5	Allocator name and order is the same as the last allocation event
E6	Number of times this allocator has been invoked
E7	Allocator name and order is the same as the most commonly invoked allocator name and order
E8	Allocator type index
E9	Allocator type is same as first one created
E10	Allocator type is the same as the last allocation event
E11	Number of times this allocator type has been invoked
E12	Allocator type is the same as the most commonly invoked allocator type
F	Allocation size requested
G	Total number of frames
H	Backtrace information

Table 1: A table of features used in the ML model.

trace files include information about allocations throughout the program including backtraces, memory addresses, allocator metadata, and much more. We then used the trace files as input to our ML tool to create predictions on which allocations could be placed in a permanent memory pool. Next, we used the Replay tool again to interpret the output from the ML tool and produce an ULTRA file. We then visualized and analyzed these results with PyDv [3] to determine the performance savings after separating these allocations into distinct pools.

3 RESULTS

After analyzing many SAMRAI test runs, we discovered that all of the SAMRAI tests did not use enough permanent memory to really benefit from our ML tool. The ML tool classified very few permanent allocations, if any, so no separate pool was needed. We used Umpire's Replay tool to confirm these results.

On the other hand, after running our ML tool on the BoBa application trace collected from Replay, we determined there was up to 624kB of memory saved out of the total 2.11MB memory used (about 29.5%). Figure 1 shows the analysis the Umpire team performed on this BoBa test trace file. By separating the permanent memory (purple dashed line) predicted from the ML tool into its own distinct pool, the BoBa memory pool (silver line) shrinks considerably compared to before (blue line).

Additionally, Figure 2 zooms in on the first 1000 events of the same BoBa test run. As the program starts up and allocates permanent memory, we can better see the impact of separating permanent

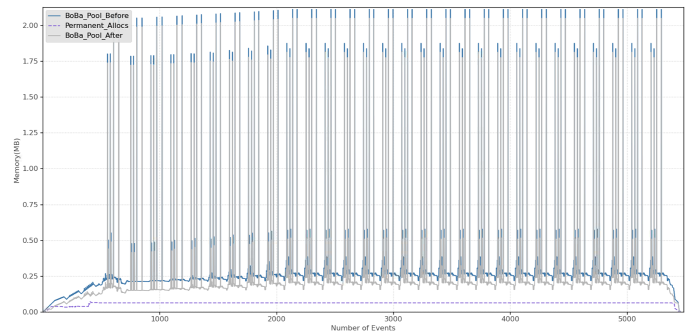

Figure 1: Comparison between the BoBa memory pool before (blue line) versus after (silver line) the permanent memory allocations (purple dashed line) were separated into a separate pool with Replay.

allocations. These results provide a proof of concept that this strategy can work for larger applications. Although we believe these results correlate to less fragmentation within the pool, further study is needed to measure to what degree fragmentation is reduced.

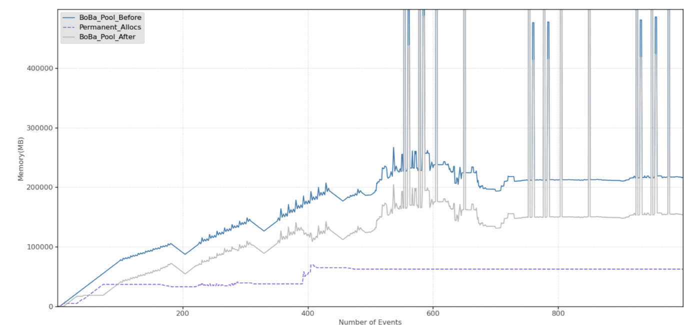

Figure 2: Zooming in on the first 1000 events of the BoBa test run to better illustrate what is happening.

We have demonstrated that our ML tool works as an accurate classifier of permanent vs. temporary allocations. To achieve better memory savings, we will use our ML classifier tool along with Replay with more production-level multi-physics codes at LLNL which have more complex allocation patterns and that allocate GBs of data in total.

4 CONCLUSIONS AND FUTURE WORK

After applying our machine learning model to our two applications, we found that the model was able to accurately predict permanent vs. temporary allocations with F1 scores of over 0.998. After using the predictions from our model, the BoBa test code was able to reduce peak memory consumption by up to 624kB (about 29.5% of total memory). On the other hand, since the SAMRAI test runs only used a very small amount of permanent memory, if any, we were not able to see similar reductions in memory usage. The next step in our study will be to use our ML classification tool with this workflow on applications with a more complex mix of temporary and permanent allocations. Additionally, we will conduct a more targeted study to determine the degree to which this kind of approach can reduce fragmentation within the memory pool.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC. LLNL-ABS-852653

REFERENCES

[1] D. A. Beckingsale, M. J. McFadden, J. P. S. Dahm, R. Pankajakshan, and R. D. Hornung. 2020. Umpire: Application-focused management and coordination of complex hierarchical memory. *IBM Journal of Research and Development* 64, 3/4 (2020), 00:1–00:10. https://doi.org/10.1147/JRD.2019.2954403

[2] P. Guthrey, J. Burmark, and W. Schill. 2022. *BoBa: HPC Implementations of Tensor Train Discretizations.* Technical Report. Zürich, Germany.

[3] M. Kwiat, D. Miller, K. Griffin, and E. Rusu. 2011. *PyDv: the Python Data Visualizer.* https://github.com/LLNL/PyDV

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[5] Andrew M. Wissink, Richard D. Hornung, Scott R. Kohn, Steve S. Smith, and Noah Elliott. 2001. Large Scale Parallel Structured AMR Calculations Using the SAMRAI Framework. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (Denver, Colorado) *(SC '01)*. Association for Computing Machinery, New York, NY, USA, 6. https://doi.org/10.1145/582034.582040

A REPRODUCIBILITY INITIATIVE

In order to reproduce these results, follow along the step-by-step guide found on Github here or copy and paste https://github.com/kab163/SC23-Reproducibility into your browser. In order to build and run the replay tool, you can follow the guides here and here. Or you can copy and paste these URLs into your browser: https://umpire.readthedocs.io/en/develop/sphinx/tutorial/replay.html and https://umpire.readthedocs.io/en/develop/sphinx/features/logging_and_replay.html. You can find more about the PyDv tool for visualizing and plotting data here or copy and paste https://pydv.readthedocs.io/en/latest/index.html into your browser.