DySectAPI: Scalable Prescriptive Debugging

Jensen, Nicklas Bo; Karlsson, Sven ; Quarfot Nielsen, Niklas ; Lee, Gregory L.; Ahn, Dong H.; Legendre, Matthew ; Schulz, Martin

Publication date:
2014

Document Version
Publisher's PDF, also known as Version of record

DySectAPI: Scalable Prescriptive Debugging

Nicklas Bo Jensen and Sven Karlsson
Technical University of Denmark
{nboa,svea}@dtu.dk

Niklas Quarfot Nielsen
Mesosphere Inc
niklas@mesosphere.io

Gregory L. Lee, Dong H. Ahn,
Matthew Legendre and Martin Schulz
Lawrence Livermore National Laboratory
{lee218,ahn1,legendre1,schulzm}@llnl.gov

***Abstract*—We present the DySectAPI, a tool that allow users to construct probe trees for automatic, event-driven debugging at scale. The traditional, interactive debugging model, whereby users manually step through and inspect their application, does not scale well even for current supercomputers. While lightweight debugging models scale well, they can currently only debug a subset of bug classes. DySectAPI fills the gap between these two approaches with a novel user-guided approach. Using both experimental results and analytical modeling we show how DySectAPI scales and can run with a low overhead on current systems.**

I. Introduction

The current state-of-the-art debugging models are inadequate for the scale of today's largest supercomputers. Traditional debuggers do not scale and manually stepping through code and inspecting an application can quickly overwhelm the user with too much information. In contrast, lightweight debugging tools have the opposite problem – they scale well, but in some cases do not provide enough information to lead the user to the root cause. Future extreme-scale systems are predicted to push core counts even further, which will increase the load on debugging tools and developers. Debugging of these extreme-scale systems points towards non-interactive batch debugging due to the cost of interactively waiting on the debugger [1].

We present DySectAPI, the Dynamic Scalable Event Tracing API, based on a novel debugging model that strikes a balance between scalability and capability. DySectAPI allows users to install debugging probes that can gather data under user specified conditions, e.g. breakpoints. Probes can be linked into a probe tree, automating actions that the user would normally perform using a traditional interactive debugger.

A probe consists of an event, a condition, a domain and a set of actions. An event can be a breakpoint, a signal or a timeout. The condition is an expression that is evaluated locally on each backend, e.g. `x > 2`. A domain defines the set of tasks in which to install the probe. Probe actions can be formulated by the user as an aggregation or a reduction, for example, aggregated messages, min and max of a variable, or stack traces using the Stack Trace Analysis Tool, STAT [2].

Probes can be composed together into a tree. A probe installs its children after being triggered and if its condition is satisfied. Probe composition is used to reduce the task space, making the information presented to the programmer scalable.

DySectAPI leverages the existing infrastructure from STAT [2] and an efficient tree-based overlay network, for scalable tool communication and data processing, MRNet [3]. DySectAPI uses Dyninst and the ProcControlAPI to control and debug application processes [4].

To summarize, we make the following contributions:

- We propose a novel prescriptive debugging model that strikes a balance between scalability and capability
- DySectAPI, an implementation of this model
- We evaluate the performance and demonstrate the scalability of our implementation

II. Evaluation

The performance of a traditional interactive debugger is very hard to quantify as a human guides the debugging session interactively. However, this clearly does not scale well on a supercomputer due to the cost of waiting on human input. Another common strategy is debugging using printf. This quickly overwhelms the programmer, for example on the Sequoia machine at Lawrence Livermore National Laboratory where printing just one byte from each core gives the programmer 1.57 megabytes of data to analyze!

We evaluate DySectAPI's performance using a model that predicts the scalability beyond current machine resources. All experiments were conducted on the Cab Linux cluster at the Lawrence Livermore National Laboratory. This cluster consists of 1,296 nodes, each with 2 Intel Xeon E5-2670, 16 cores per node and a total of 20,736 cores.

A. Performance Modeling

During execution of a DySectAPI session, a reduction in the task search space naturally occurs as probes are dynamically enabled only when a specified condition is met, which leads to reductions in the amount of instrumentation and in the amount of debug information generated. We want to show the impact of different probe trees and our analytical model can represent any probe-tree shape. In our initial evaluation, we look at a flat tree consisting of four probes that are not linked and four chained probes in a deep tree.

In each probe, we include a pruning factor, which is the fraction of processes that a probe filters out. In our modeling and experiments, we assume that pruning of processes is spread out equally over all the backends.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-ABS-658446).

We first consider the number of probe installations for a single probe:

$$installs(probe) = \begin{cases} \text{is root}, N_{backends} \times N_{procs\_per\_backend} \\ \text{otherwise}, invocs(ancs(probe)) \end{cases} \quad (1)$$

Where $N_{backends}$ is the number of nodes in the system, and $N_{procs\_per\_backend}$ is the number of processes per node. $ancs$ refer to the ancestor of a probe. The number of invocations for a single probe:

$$invocs(probe) = (1 - ratio(probe)) \times installs(probe) \quad (2)$$

Installation of a probe happens across processes on a backend, but occurs in parallel across the backends.

$$cost_{install}(probe) = cost_{install} \times \frac{installs(probe)}{N_{backends}} \quad (3)$$

The cost of invoking the probe has a sequential part and a communication part. The latter assumes that the depth of the MRNet topology increases as the number of processes increases, making network cost a logarithmic function as it depends on the MRNet tree depth.

$$cost_{invoc}(probe) = cost_{invoc} \times \frac{installs(probe)}{N_{backends}} + cost_{network} \cdot log(N_{processes}) \quad (4)$$

Where $N_{processes}$ is the number of processes involved in the communication. The total cost of all probes in the tree is the sum of the costs for installing and invoking each probe. The overall scaling of the model is $\mathcal{O}(logN_{processes})$. This is the desired scaling, as it allows us to scale to large systems.

The probes used in this evaluation each collect a single message aggregated across all processes that satisfy the probe condition (i.e., processes that are not pruned). Using microbenchmarking of DySectAPI we have obtained the following costs $cost_{invoc} = 0.72ms$, $cost_{install} = 0.28ms$ and $cost_{network} = 4.6ms$.

Figure 1 shows the actual and modeled overhead of a flat and deep probe tree with a pruning factor of 50% in each probe. We see that the modeled execution time for the deep probe tree closely resembles the actual execution time. For the flat tree there is a small difference. This is due to overlapping communication that results in a smaller communication overhead than modeled. Both probe trees scales logarithmically with respect to the number of cores in the system. More importantly, the filtering of processes also eliminates the amount of information presented to the programmer. In the chained deep tree consisting of four probe 87.5% of the original processes are filtered out.

B. Case Study

We have evaluated DySectAPI on a MPI bug that only manifested itself at or above 3,456 MPI processes with BoomerAMG, a high-performance preconditioner library developed at Lawrence Livermore National Laboratory. We were able to quickly create several debugging probe trees and diagnose the problem as an issue in a recently upgraded MVAPICH 1.2.7 MPI library in the initialization of a communicator structure.

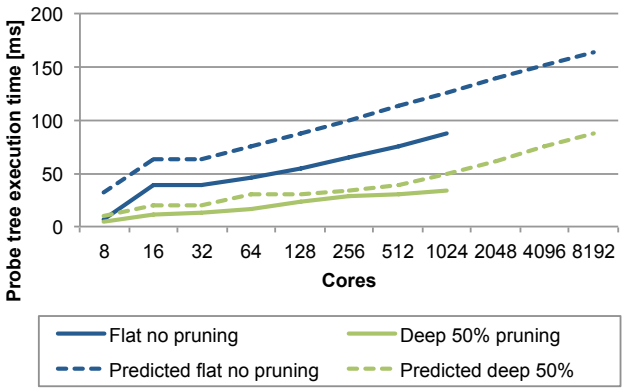

Fig. 1. Actual and modeled execution time on Cab with 16 cores per node for a flat and a deep probe tree.

III. Related Work

Other tracing techniques include systemtap, which has been used as a Linux trace/probe tool [5]. systemtap has been extended with user-space probe capabilities using Dyninst [4]. The main difference between traditional tracing tools and DySectAPI is the dynamic nature of DySectAPI's probe-tree model and that DySectAPI is targeted at parallel systems.

Other approaches are the automatic and semi-automatic debugging models, which include relative debugging. For example Dinh. et. al. introduce scalable relative debugging [6], which requires running two versions simultaneously, one working and one failing, thus solving a special case and may not scale to extreme scales. User-guided debugging like DySectAPI can complement automatic approaches, as automatic tools are only applicable for a subset of bugs.

IV. Conclusions and Future Work

We propose the novel prescriptive debugging model that can scale without sacrificing key debugging information presented to programmers, filling the gap between traditional and lightweight debuggers. Using both experimental results and analytical modeling we show that DySectAPI scales well and achieve the desired logarithmic scalability.

References

[1] B. Feldman, "Debugging exascale: To heck with the complexity, full speed ahead!" September 2010.

[2] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack trace analysis for large scale debugging," in *IPDPS*, 2007.

[3] P. Roth, D. Arnold, and B. Miller, "Mrnet: A software-based multicast/reduction network for scalable tools," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2003.

[4] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *Int. J. High Perform. Comput. Appl.*, 2000.

[5] V. P. Ibm, F. Ch, E. R. Hat, and J. K. Ibm, "Locating system problems using dynamic instrumentation," 2005.

[6] M. N. Dinh, D. Abramson, and C. Jin, "Scalable relative debugging," *IEEE Trans. Parallel Distrib. Syst.*, 2014.