An Approach to Lowering the In Situ Visualization Barrier

Thomas Fogal
University of Duisburg-Essen, &
Scientific Computing & Imaging (SCI) Institute
tfogal@sci.utah.edu

Jens Krüger
University of Duisburg-Essen, &
Scientific Computing & Imaging (SCI) Institute
Building LE, Lotharstraße 65
Duisburg, Germany
jens.krueger@uni-due.de

ABSTRACT

Coupling visualization and analysis software with simulation code is a resource-intensive task. As the usage of simulation-based science grows, we asked ourselves: what would it take to enable *in situ* visualization for *every* simulation in existence? This paper presents an alternative view focusing on the **approachability** of *in situ* visualization. Utilizing a number of techniques from the program analysis community and taking advantage of commonalities in scientific software, we find that we can vastly reduce the time investment required to achieve visualization-enabled simulations.

Keywords

in situ visualization, program verification

1. INTRODUCTION

This paper has been accepted, but is still undergoing edits. You are reading the author's personal copy. To obtain the official published version, please use the DOI at the bottom left.

In situ visualization has proven to be useful for simulation-based sciences. The majority of *in situ* visualization literature is focused on the performance story: the growing size of outputs from simulations makes the commonplace post-processing regime less attractive [5, 6, 20]. While these efforts push us in the right direction, the impetus is flawed. The post-processing approach is not inferior because it scales poorly—though it does indeed scale poorly—it is *intrinsically* inferior. The ability to visualize and understand a simulation's data as it is generated is *inherently useful*. The *in situ* approach has not been ignored until recent years because it was not useful. A more likely explanation is that the difficulty was prohibitive.

Let us redefine '*in situ* visualization' as 'interactive simulation'. Interactive simulation is simulation that can be controlled: sped up or slowed down, reinitialized with new parameters, visualized, selectively refined, or even have its underlying physics live-edited. This model of simulation is clearly superior to our present batch-oriented model. The cycle time from hypothesis to verification would be greatly reduced.

Hall et al. note [10] the importance of "program-analysis strategies to improve software construction, maintenance, and evolution." We introduce a methodology for "0 day" coupling of simulation and visualization code. We remove the need to link in *any* external code to the simulation. The simulation software often does not even need to be recompiled. The bulk of our contribution is in the form of program understanding: we demonstrate how to infer those data that are interesting *as well as* the parameterization of those data that enables visualization. This obviates the need for the simulation author to conform to or even learn external APIs.

ISAV 2015 Austin, TX

ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

2. TRIVIAL EXAMPLE

Consider the task of modifying an existing simulation program to interactively visualize its in-progress results. The developer must identify the primary loop that advances the state of the simulation. That loop modifies some memory of interest that the developer typically has some external knowledge about. The external knowledge generally revolves around data type and format: 'a point cloud of 64-bit floating point values', for example. This in turn helps the developer search for memory matching that organization. Once the location where the simulation advances its state is discovered, metadata is uncovered via a similar process, and a call to the *in situ* visualization library is inserted.

The code fragment in Listing 1 is exemplary of this task. The developer adding *in situ* visualization to a 2D simulation would be pleased to find these loops: the code is accessing and updating a 2-dimensional structure, 'data'. Next tasks would be to identify the type and source of the memory in 'data'. Should it align with the developer's ideas of the simulation's data model, visualization will be inserted after the loops and testing would be done.

The work presented here automates this exploratory search-and-insert-visualization process.

3. PROGRAM MODEL AND SIMULATION ANALYSIS

In this section we develop an abstract model of an executing simulation program. We will then use this general model to describe a collection of properties that code such as that in Listing 1 follows. This set of properties codifies the aforementioned developer process.

```
for(size_t j=0; j < dims[1]; ++j) {
  const size_t row = j*dims[0];
  for(size_t i=0; i < dims[0]; ++i) {
    data[row+i] = (S(x-1,y-1) + S(x-0,y-1) + S(x+1,y-1) +
                   S(x-1,y-0) + S(x-0,y-0) + S(x+1,y-0) +
                   S(x-1,y+1) + S(x-0,y+1) + S(x+1,y+1)) / 9.0
  }
}
```

Listing 1: A code fragment representative of simulation software. A large array is smoothed using a set of nested loops. `S` is presumed to be a macro that samples `data` while properly accounting for edge cases.

$$
\begin{array}{rcl}
BaseType & := & Booleans \cup Integers \cup FP \cup Strings \\
Type & := & BaseType \cup Array \cup Pointer \\
Memory & := & Heap \cup Static \cup Local \cup Arguments \cup Text \\
IPtr & \in & Text \\
T & := & Memory \mapsto Type \\
BT & := & Memory \mapsto BaseType \\
Fqn & := & [begin \in Text, end \in Text] \mid \ begin < end \\
Where & := & Text \mapsto Fqn \\
Wr & := & (m \in Text) \mapsto (n \in (Memory \setminus Text)) \mid m \neq n \\
\textbf{class} & & CFG_{Node}\ address\ edges \\
CFG & := & \{n \mid n = CFG_{Node}\} \\
BB & := & Text \mapsto CFG \\
K & := & Text \mapsto CFG_{Node} \\
Hdr & := & CFG_{Node} \mapsto Boolean \\
Depth & := & CFG_{Node} \mapsto Integer
\end{array}
$$

Listing 2: Definitions for an abstract machine and analysis based on control flow properties.

We use the formalisms given in Listing 2. We consider an abstract machine described by an *instruction pointer* and the current state of *memory*. The instruction pointer advances automatically, and memory operations consist of reads and writes that map an address to a mutable memory location. Note that our definition denies self-modifying code. Memory is assumed to be *typed*, with a small set of available types. The T and BT mappings define mappings from memory locations to type information.

The running process is assumed to consist of a series of *functions* (Fqns), that are defined as the functions' upper and lower addresses. We make use of an inverse mapping $Where$ that allows us to identify a function from the current instruction pointer. We build local *control flow graph*s (CFGs) that describe the potential execution paths. These graphs are represented as a set of *nodes* that contain an entry *address* as well as a set of *edges*. We build these CFGs based on the function address range. We define a mapping K that allows us to identify a node in the control flow graph from an instruction address. We define two final mappings from a node in the control flow graph: 1) a predicate identifying *loop headers* (Hdr), and 2) a mapping for the calculated *loop depth* ($Depth$). In Listing 1, the basic blocks containing `j < dims[1]` and `i < dims[0]` would be the loop headers. Loop depth is the nesting level of the provided basic block. In Listing 1, the assignment to `row` has a depth of 1, whereas assignment to the element in `data` has a loop depth of 2.

Using this model of program execution, we consider the problem of automatically identifying memory regions that house data that a user would want to visualize. We model these as a set of constraints on type classes. An instance of the type class allows one to visualize data within a simulation.

We currently support searching for N-dimensional ("ND") data arrays. This type is parameterized by a `base` address, a `length` (in bytes), the number of dimensions `ndims`, an array of dimensions `dims`, and finally the type of the data.

$$
\begin{array}{rcll}
\textbf{class} & ND & base\ length\ ndims\ dims\ type & \\
& \wedge & \exists m \in Heap : base \rightarrow m & (1) \\
& \wedge & BT(base) = type & (2) \\
& \wedge & T(dims) \in Array \cup Pointer & (3) \\
& \wedge & BT(dims) \in Integers & (4) \\
& \wedge & T(ndims) \in Integers & (5) \\
& \wedge & ndims > 0 & (6) \\
& \wedge & Wr(IPtr) \in [base, base + length] & (7) \\
& \wedge & \exists b \in BB(Where(IPtr)) : & \\
& & \wedge\ IPtr \notin b & \\
& & \wedge\ Hdr(b) & \\
& & \wedge\ Depth(K(IPtr)) > Depth(b) & (8)
\end{array}
$$

We use the $\rightarrow$ notation to mean "points to"; the first constraint simply states that the data of interest live on the heap. As simulation data is large, it rarely fits on the stack or even in statically initialized memory. The second constraint conveys that the base type matches a parameter of our model, such as FP (floating point). The third and fourth constraints dictate that the dimensions are stored in a linear list of integers, and the fifth and sixth say that that the length of that dimension list is a positive integer.

The 7th and 8th constraints are complex and intertwined. First, the application must write into the relevant memory block. Secondly, the basic block where the data are written must be deeper than another basic block that contains a loop header. That is to say that the data write occurs within a loop.

The formulation gives rise to a pattern matching problem. The `class`es of interest are the patterns, and the space to match within is the running process' `Memory`. In Listing 1, the parameter bindings are: `data` for `base`, the size of the allocation (not shown, but assumed to be `dims[0]` $\times$

`dims[1]` × `sizeof(float)`) for `length`, `dims` for `dims`, and 2 for `ndims`.

We do not claim our set of properties is perfect, though they have proven remarkably effective for our uses thus far. There are a number of promising approaches for discovering new invariants automatically [14, 12, 17], as well as low-hanging fruit (e.g. 'the memory is written to a file'). In the future, we wish to allow simulation authors to specify these constraints at runtime. The penalty for a lax specification would be too many visualization windows popping up; the penalty for too strict a specification would be too few windows popping up. The author would see either error almost immediately.

4. IMPLEMENTATION

We seek to realize the aforementioned 'search' for a given simulation process. At first glance static analysis is the best tool for this task, however it runs into difficulties proving some of the properties. The pernicious problem is aliasing in C-based languages. A statement as trivial as '`x = &y;`' creates two names for the same memory; thereafter, proving that a write to '`*x`' changes or does not change '`y`' can be anywhere from unambiguous to undecidable.

A lesser reason to shy away from static analysis is the computational expense, of which the largest for us is building control flow graphs. Especially for languages that enjoy methods for exponential code expansion (e.g. C++ templates), building CFGs for the entire program would be prohibitively expensive. A more targeted mechanism is desirable.

For this and other reasons we utilize static analysis techniques judiciously sourced by dynamic analysis. To receive a notification when a particular memory location is written, we use page protection to detect writes to it. Address `0xdeadbeef` is address `0xdeadbeef`, regardless of the variable name used to access it, and thus there is no need to solve the aliasing problem. Control flow graph construction and analysis are still expensive, but we isolate their construction to the functions of interest. Of particular note and perhaps surprise to the scientific visualization community is that binary analysis need not be lossy as compared to source analysis [16].

Not all of the 8 aforementioned properties are worthy of exposition; variable type information, for example, is straightforwardly sourced from the binary's debug information. In the next subsections, we focus on three of the larger issues: efficiently tracking memory, control flow graph analysis, and teasing out the dimensions of an array from the instruction stream of the code accessing it.

4.1 Memory tracking

Any heap-allocated memory might potentially be of interest to us. We utilize a `ptrace(2)`-based supervisor on the target program, and model each allocation using the finite state machine in Figure 1[1]. Memory regions begin in the 'null' state and change state based on events observed in the simulation process. This event tracking induces overhead, but in the absence of events simulation execution proceeds at native speeds.

[1]We note that `ptrace` was explicitly chosen for portability. Previous work relied on `LD_PRELOAD` [8], and that created issues porting to some supercomputers.

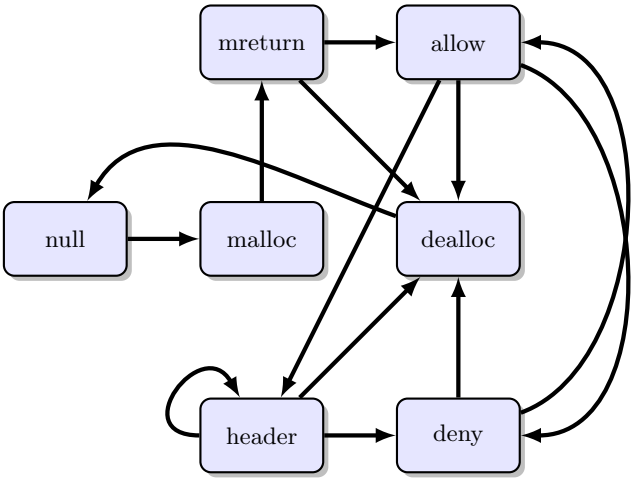

Figure 1: Finite state machine governing memory regions of interest. Regions transition between the states based on events observed in the observed simulation process. Basic information is obtained in the *malloc* and *mreturn* states. The *allow* state initializes parameters for visualization and enables unfettered access to the memory. *header* states build up the dimensions of the data. The *deny* state reenables access detection.

Allocations cause us to begin tracking a memory region. We implement this event notification using a breakpoint on `malloc` calls. By examining the stack and return address, we can create a map of the heap-allocated memory in the process. Overhead for this operation is predominantly context switching between the simulation process and our supervisor.

The `allow` and `deny` states solve the access detection problem. As mentioned earlier, solving the aliasing problem would be prohibitively expensive. Inserting checks at every instruction that modifies memory is another alternative, but Antoniu and Hatcher previously demonstrated this to be too expensive for our needs [2]. Instead, we rewrite allocations of interest, enabling write protection on the returned memory. This causes the simulation to trap when altering the data of interest, notifying our supervisor. To avoid the performance issue of a notification on *every* access, we catch only the first per function.

4.2 Control flow

The memory tracking described above enables our supervisor to track most of the events it needs. To pinpoint the remaining events we use analysis based on the local control flow. When a region is accessed, we build the local control flow graph for the currently-executing function. Our supervisor computes common compiler analysis information and uses the results to define per-node depth as well as perform loop header identification, as shown in Figure 2.

Our loop header identification relies on the common definitions of *reachability* and *dominance* [18]. Our current algorithm is known to be fallible in the presence of harmful `goto`s, but we have found it is reliable in practice and cheap to compute. We deem a basic block to be a loop header when the basic block:

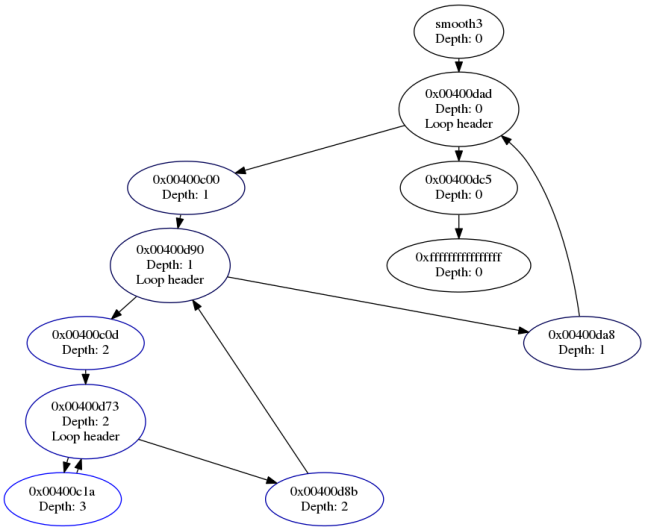

Figure 2: Simplified control flow graph for a small function that smooths a 3D array (the 3D analog of Listing 1). Analysis identifies loop headers and the nesting level ('Depth') of each basic block. On access, the loop tree is traversed to determine the dimensionality of the array.

1. has exactly 2 in-edges,
2. has exactly 2 out-edges,
3. is reachable from one or both out-edges,
4. is dominated by exactly one in-edge, and
5. the dominating in-edge does not directly-dominate the other in-edge

In the future, we hope to simplify our flow graphs into loop trees [18]. This will resolve some of the possible ambiguities and modestly improve memory consumption.

We currently use Dyninst's ParseAPI [9] to compute the initial graph, and then perform the analysis with custom code. Other tools in this domain are DynamoRIO [3], Pin [11], and Valgrind [13]. All of these tools are capable of sophisticated binary transformations. However, our needs are modest and Dyninst presently represents the majority of our overhead. In the future we hope to replace this with custom graph construction code that can more effectively limit computation to the region of interest.

4.3 Symbolic execution

As described in the **class** ND of Section 3, we assume a relation between loop headers and the basic blocks that are contained within those loops and accessing memory. The loop variable must be involved: if it were not, the access would be loop-invariant and hoisted out of the loop, either explicitly by the programmer or implicitly by the compiler. We assume a stronger relation, however: that the loop conditions imply the dimensionality of the memory regions accessed therein.

Loop conditionals do not definitively describe the format of the data. We have however found them to be remarkably accurate, and the loop nesting to be practically infallible. Still, we allow the user to override these discovered bounds, at which point our tool degrades to only identifying *where* visualization should be performed. More work is needed in this area.

Our general approach is to differentiate the loop bound from the induction variable in the loop conditional. Listing 3 gives the basic block for a real-world loop conditional (what might be implemented for `i < dims[0]`):

```
MOV %rdx, [%rip+0x20507]
MOV %rax, [%rpb-0x60]
CMP %rdx, %rax
JB -0x275
```

Listing 3: Instructions within a sample loop header. The induction variable and the loop bound appear as arguments to the `CMP` instruction.

We would like to know which of `%rdx` and `%rax` in Listing 3 is the loop bound. Unfortunately a myopic view of the `CMP` instruction is insufficient for operand classification: the *source* of the values is in the two `MOV` instructions. We use Algorithm 1 to track the source of operands by interpreting each instruction in the basic block. A heuristic that the induction variable is a local variable is used to differentiate the loop bound from the induction variable.

Algorithm 1 Tracking virtual register sets to identify the source of data. The algorithm begins at a loop header basic block and symbolically executes each instruction. The resultant data structure can be used to query the source of an instruction operand's value.

```
register[*] := UNKNOWN
instruction := bb_addr   ▷ first instruction in loop header
repeat                   ▷ foreach instruction in the basic block
    if instruction.Opcode = MOV then
        mov := (MovInstruction)instruction
        if mov.source ∈ register then
            register[mov.target] := register[mov.source]
        else if mov.source ∈ Memory then
            register[mov.target] := mov.source +
                                    memdiff[mov.source]
        end if
    end if
    if instruction.Opcode = ADD then  ▷ track Δaddr
        add := (AddInstruction)instruction
        if add.dest ∈ register ∧ register[add.dest] ≠ UNKNOWN then
            register[add.dest] += add.source
        end if
        if add.dest ∈ Memory then
            memdiff[add.dest] += add.source
        end if
    end if
        ▷ ... SUB, MUL, etc. cases omitted for brevity
    instruction := next(instruction)
until instruction.Opcode = CMP
```

It is not strictly true that induction variables must be local variables. However, we have only seen this assumption violated in artifically-constructed test programs. Data dependency information and def/use sets [18] should make this more robust in the future.

Each iteration of this process gives a single loop bound. By following the state machine in Figure 1 and setting breakpoints up the chain of the loop tree, we derive the full set

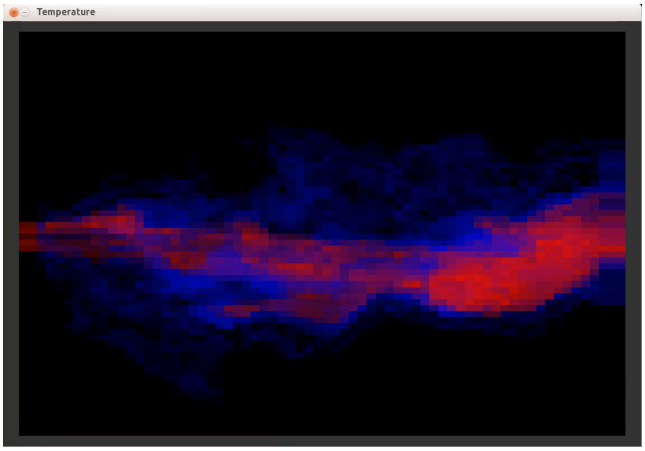

Figure 3: Volume rendering of the temperature field from a *PsiPhi* simulation [15]. Array shape information and data were pulled from the running simulation and given to an *ad hoc* simple volume renderer. Instrumentation and rendering time is on the order of milliseconds whereas a timestep can take seconds. User interaction was limited to transfer function design.

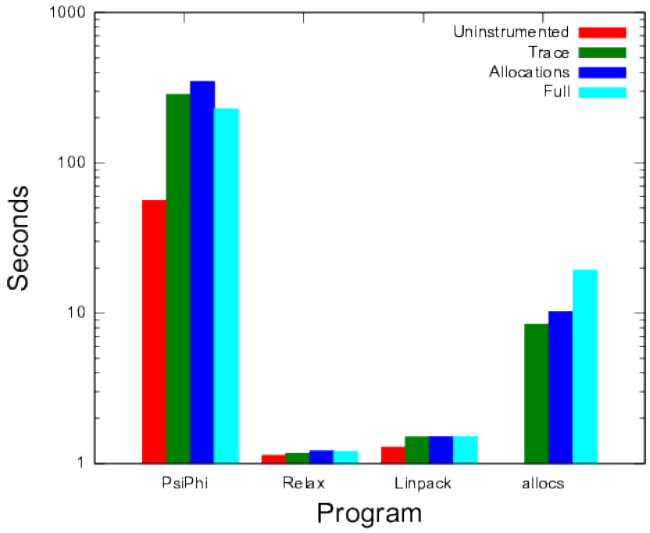

Figure 4: Performance of our evaluation programs under different instrumentation scenarios. Note logarithmic scale. 'Uninstrumented' is the runtime of the simulation without our interference. 'Trace' interrupts for allocations; 'Allocations' *reports* allocations as well, which requires reading more data from the instrumented program. 'Full' does allocation tracking, access detection, analysis, and visualization of the data.

of bounds. At the function boundary, we enter the 'deny' state and re-enable memory protection for that region.

4.4 Visualization

Our contribution is in the approach to program understanding, though rendering is required to demonstrate these aspects. We have implemented a simple GLSL-based volume renderer and a python-based `yt` [19] backend thus far. Figure 3 shows the former with data sourced from a combustion simulation. In the future, we hope to incorporate backends using established visualization tools such as ImageVis3D, VisIt, and/or ParaView [7, 4, 1].

4.5 Performance

Performance is a cause for concern, as our instrumentation's theoretical upper bound is on par with `valgrind`-level instrumentation [13]. Fortunately, in practice we have found the slowdown to be approximately 4x for real-world programs. There is much work still to be done in this regard: the largest limitation is that we currently visualize *every* timestep's results, at tremendous overheads.

Figure 4 looks at multiple aspects of performance across this set of programs. The red 'Uninstrumented' bar represents an upper bound on performance. 'Trace' inserts breakpoints at '`malloc`', '`free`', and their return addresses, measuring what it costs to start and stop the execution of the simulation program. Simulations that utilize more regions of dynamic memory will see higher overheads due to this aspect. However, the graph does not capture the phased nature of these processes: generally, our instrumentation is heavy for allocation-heavy startup routines and lightweight thereafter.

Figure 4's 'Relax' and 'allocs' are artificial programs constructed to illustrate overheads. The main component of 'Relax' is Listing 1. 'allocs' does nothing but allocate memory, the worst case for our instrumentation. We note that real-world programs experience considerably lower overheads, with the popular Linpack seeing a modest 15% slowdown.

5. CONCLUSIONS

We have elucidated a method and demonstrated a prototype that eliminates the surface area between simulation code and visualization tool. By recovering the loop structure of a target binary and carefully instrumenting memory accesses, one can automatically insert visualization at appropriate places in a running simulation.

5.1 Future work

The most glaring present omission is the lack of support for data types beyond regular N-dimensional grids. An obvious next target is related data types such as adaptive mesh refinement data. Curvilinear grids may prove simple as well, and meshes or point clouds would certainly be of interest. An area of uncertainty is in data decomposition in distributed memory simulations.

We do not seek to replicate the full functionality of tools like VisIt or ParaView. We must therefore couple with one of these tools; doing so would immediately increase the utility of our prototype implementation.

We make a number of assumptions that are practically but not strictly true. Each requires more investigation, and aspects such as the specification used in our search require more user control than we presently have made available.

While some of these issues involve significant engineering efforts, the work presented here demonstrates that there is no need to modify simulation code to inject *in situ* visualization. We hope this encourages others to pursue 0-modification approaches to *in situ* visualization.

6. ACKNOWLEDGMENTS

We thank Fabian Proch for help configuring a sample case in PsiPhi, Matt Might for fruitful insight into program analysis topics, Chris Johnson & Chuck Hansen for early review of the ideas here, and Ethan Burns for implementation review.

This research was made possible in part by the Intel Visual Computing Institute; the NIH/NCRR Center for Integrative Biomedical Computing, P41-RR12553-10; and by Award Number R01EB007688 from the National Institute of Biomedical Imaging and Bioengineering. The content is the sole responsibility of the authors.

7. REFERENCES

[1] J. Ahrens, B. Geveci, and C. Law. ParaView: An end-user tool for large data visualization. *The Visualization Handbook*, 717:731, 2005.

[2] G. Antoniu and P. Hatcher. Remote Object Detection in Cluster-Based Java. Research Report RR-4101, 2001.

[3] D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 133–144, New York, NY, USA, 2012. ACM.

[4] H. Childs, E. Brugger, B. Whitlock, J. S. Meredith, S. Ahern, K. Bonnell, M. Miller, G. Weber, C. Harrison, D. Pugmire, T. Fogal, C. Garth, A. Sanderson, E. W. Bethel, M. Durant, D. Camp, J. M. Favre, O. Ruebel, P. Navratil, M. Wheeler, P. Selby, and F. Vivodtzev. *VisIt: An End-User Tool for Visualizing AND Analyzing Very Large Data*, pages 357–372. CRC Press, October 2012.

[5] M. Dorier, R. R. Sisneros, T. Peterka, G. Antoniu, and D. B. Semeraro. Damaris/Viz: a nonintrusive, adaptable and user-friendly in situ visualization framework. In *Large Data Analysis and Visualization*, October 2013.

[6] N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion, B. Geveci, M. Rasquin, and K. E. Jansen. The ParaView Coprocessing library: A scalable, general purpose *In Situ* visualization library. In *Large Data Analysis and Visualization*, pages 89–96. IEEE, 2011.

[7] T. Fogal and J. Krüger. Tuvok, an Architecture for Large Scale Volume Rendering. In *Proceedings of the 15th International Workshop on Vision, Modeling, and Visualization*, November 2010.

[8] T. Fogal, F. Proch, A. Schiewe, O. Hasemann, A. Kempf, and J. Krüger. *Free*processing: Transparent in situ visualization via data interception. In *Proceedings of the 14th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV, Wales, 2014. Eurographics Association.

[9] B. P. M. Giridhar Ravipati, Andrew Bernat and J. K. Hollingsworth. Towards the deconstruction of dyninst. Technical report, UW Madison, June 2007.

[10] M. Hall, D. Padua, and K. Pingali. Compiler research: The next 50 years. *Commun. ACM*, 52(2):60–67, Feb. 2009.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[12] B. McCloskey, T. Reps, and M. Sagiv. Statically inferring complex heap, array, and numeric invariants. In *Proceedings of the 17th International Conference on Static Analysis*, SAS'10, pages 71–99, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[14] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 683–693, Piscataway, NJ, USA, 2012. IEEE Press.

[15] F. Proch and A. M. Kempf. Numerical analysis of the cambridge stratified flame series using artificial thickened flame les with tabulated premixed flame chemistry. In *Combustion and Flame*, volume 161, pages 2627–2646, 2014.

[16] T. Reps, J. Lim, A. Thakur, G. Balakrishnan, and A. Lal. There's plenty of room at the bottom: Analyzing and verifying machine code. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 41–56, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages #38; Applications*, OOPSLA '13, pages 391–406, New York, NY, USA, 2013. ACM.

[18] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[19] M. J. Turk, B. D. Smith, J. S. Oishi, S. Skory, S. W. Skillman, T. Abel, and M. L. Norman. A multi-code analysis toolkit for astrophysical simulation data. *Astrophysical Journal Supplement Series*, 192, 2011.

[20] B. Whitlock, J. M. Favre, and J. S. Meredith. Parallel in situ coupling of simulation with a fully featured visualization system. In *Proceedings of the 11th Eurographics conference on Parallel Graphics and Visualization*, pages 101–109. Eurographics Association, 2011.