ORNL/TM-2023/3137

Initial OpenStudio Profiling Results

Jason W. DeGraw
Yeonjin Bae

September 2023

ORNL/TM-2023/3137

Buildings and Transportation Science Division

INITIAL OPENSTUDIO PROFILING RESULTS

Jason W. DeGraw
Yeonjin Bae

September 2023

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831
managed by
UT-BATTELLE LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

ABSTRACT

OpenStudio's performance has not historically been an area of much work, but as it has successfully replaced ad hoc model generation solutions, the performance of the software is more and more central to continuing success. This report describes an initial effort to profile OpenStudio, describes the problems encountered, the solutions to those problems, and some early recommendations for further work should funding become available. The approach taken here is to use special software, referred to as profilers, to assess the code and how it executes. This approach is more appropriate for this kind of software than the checkpoint-style timing that is often done with numerical codes. Profiling was most successful on the MacOS platform, where Apple's Instruments software was able to decipher the complexities of OpenStudio's command line execution of a workflow. Even with the limited exploration of performance done here, the team quickly ran into limitations imposed on the code by the stateless architecture, and the team recommends an evaluation of this architecture as a good next step to improve performance.

1. INTRODUCTION

The OpenStudio [1] middleware software development kit (SDK) provides programmatic access to a building energy model (hereafter referred to as the "Model" while "model" will refer to the abstraction that is represented in the Model) that uses the EnergyPlus building energy simulation engine [2] to assess the performance of the modeled buildings. The SDK implements a stateless object-oriented approach that utilizes accessor functions over a low-level data structure (the "Workspace") that somewhat resembles the EnergyPlus Input Data File (IDF) format.

The Workspace represents the model but does not build out any of the interconnections or store any of the information in the form used by the Model. The OpenStudio approach to the model is termed "stateless" because the objects that are used do not story any state associated with the model. An instructive example is the "contacts" app on most smart phones today. Since the data ultimately resides in the cloud, a simplistic implementation would store no data on the phone and would interact with a database in the cloud via the internet. Searching for a particular name would query the database for that name and act accordingly based on what is returned. In the same way, the Model stores a connection to the Workspace and then will extract information that is requested from the Workspace without storing (much) of anything in the Model object.

There are good reasons to implement this approach. It is flexible and resilient, and just as in the address book example, there's a separation of concerns that is attractive. However, there are also plenty of reasons to implement a more traditional object-oriented approach. Chief among these is that the performance impacts will be substantial. Note that in the above discussion of a smart phone contacts app, the term "simplistic" was applied to the storage of no local data. Consider the case when there is no internet connection available – should the user be unable to search for a contact phone number? It still might be prudent not to store all the data locally. Storing recent contacts or just names and phone numbers (and leaving information like physical and email addresses in the cloud) would allow offline usage. Similarly, if an object has to look up its properties before a calculation can be done, any calculation necessarily takes longer. Periodically, there have been calls to merge the EnergyPlus and OpenStudio models. The fundamental differences between the representation of HVAC would largely make such an endeavor quite challenging, but even if that issue was solved, the fundamental stateless nature of Model would make adoption by EnergyPlus a complete non-starter. Much has been made of EnergyPlus's tendency to do index-based array lookups to make references from one object to another but replacing that fast (if risky) lookup with the OpenStudio procedure would be a very bad idea.

The present effort is being undertaken to begin developing the capability of understanding the performance of OpenStudio more formally. Anecdotal reports and the authors' experience indicate that some OpenStudio measures take a very long time to run, resulting in workarounds that are suboptimal in many ways. Much of the power of the programmatic approach to energy modeling that OpenStudio enables is the introduction of standard, best practices for the programmatic creation of models, and when users are forced to step outside the framework much of that power is lost. Having to do text-based search-and-replace on model files because it is much faster than the equivalent OpenStudio measure is not a good situation, and an effort to better understand the challenges will benefit both the developers and the users. The approach taken here is a profiling-based approach, in which the OpenStudio command line interface is executed with particular inputs and the execution of the code is then examined with specialized tools.

2. PROFILING OPENSTUDIO

2.1 PROFILING GUIDED PERFORMANCE IMPROVEMENT

The use of profiling to assess the performance of a computer code is one way to improve the performance of a computer code. This approach is distinct from timing-based approaches. Timing-based approaches often use strategic placement of checkpoints to assess the performance of particular parts of the code. This approach is very useful in more monolithic codebases where there's a particular section of that is known to be computationally expensive or if there is a well-defined, linear progression of whatever is being computed. The performance of many numerical simulation programs can be well understood with this approach because the most time-consuming operations (e.g., any number of matrix operations) are well known and swamp the resource usage of other parts of the code. OpenStudio is not monolithic, and it is not characterized by a linear progression of operations; rather, it is up to the user to determine what OpenStudio does. Placing timing checkpoints in the OpenStudio command line interface would be time-consuming and would not likely result in any insight into the operations of the program.

The use of profiling to assess performance is a more likely solution. Profiling is the act of using a special program or package, called a "profiler," to determine the resource use of the parts of the code as the code executes. Different profilers have different reporting options, but in general an expected output would be a list of all the functions invoked by the program being profiled and the time spent in each of these functions. This can help identify functions that are taking too long, but can also help find functions that should not have been called or are being called too many times.

Examples of profiling programs/packages include Intel's VTune, GNU gprof, and Apple's Instruments. AMD has also provided various profiling tools through the years for its CPUs and GPUs. For the purposes of this report, Intel's VTune and Apple's Instruments will be considered.

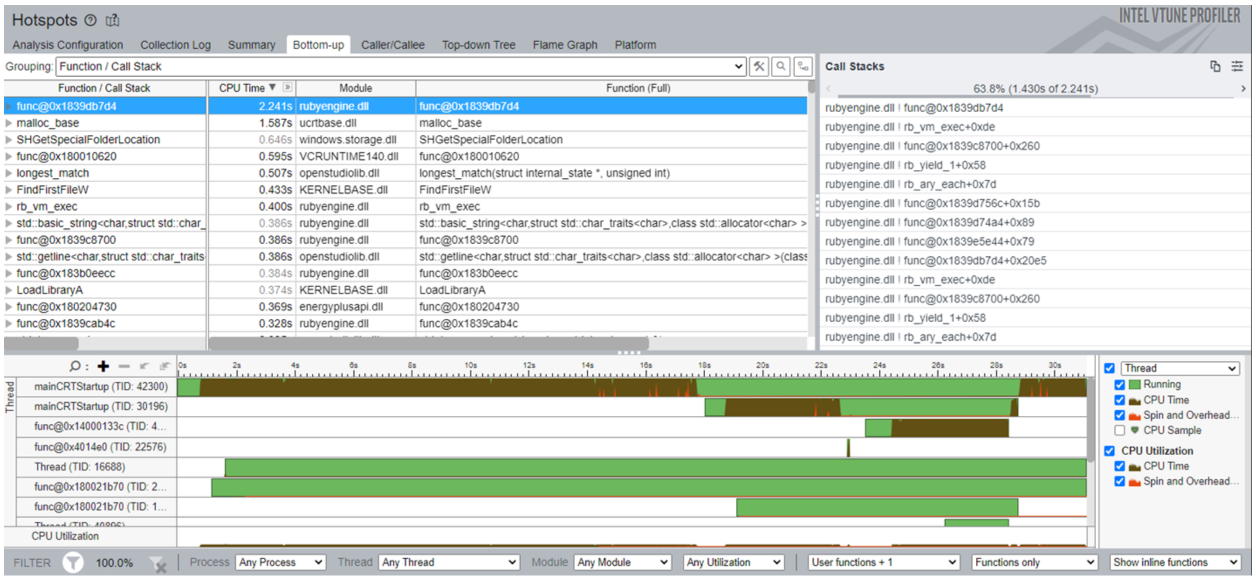
2.2 VTUNE PROFILING

2.2.1 Setup

The VTune profiler is part of Intel's performance suite and can be used in conjunction with Microsoft's Visual Studio development suite. The only barrier to use was the requirement that the OpenStudio build be done in "RelWithDebInfo" mode, in which debug symbols are included in the output binaries but otherwise the build is completed as a release build. This procedure allows for optimizations to take place while preserving enough information to track the execution of the code. After investigation, the ORNL team sought assistance from the NREL OpenStudio team to activate the "RelWithDebInfo" build, which the NREL team was quickly able to accomplish. Once OpenStudio was built in the required mode, the ORNL team was able to run VTune on multiple cases without issue.

2.2.2 Results

While the VTune profiler was able to execute the program as part of the Visual Studio environment and collect the information that was required, the results were less than informative. See Figure 1 for a sample snapshot of the output from VTune for a relatively simple model creation workflow.



Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Module	Function (Full)
func@0x1839db7d4	2.241s	rubyengine.dll	func@0x1839db7d4
malloc_base	1.587s	ucrtbase.dll	malloc_base
SHGetSpecialFolderLocation	0.646s	windows.storage.dll	SHGetSpecialFolderLocation
func@0x180010620	0.595s	VCRUNTIME140.dll	func@0x180010620
longest_match	0.507s	openstudiolib.dll	longest_match(struct internal_state *, unsigned int)
FindFirstFileW	0.433s	KERNELBASE.dll	FindFirstFileW
rb_vm_exec	0.400s	rubyengine.dll	rb_vm_exec
std::basic_string<char,struct std::char_	0.386s	rubyengine.dll	std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >
func@0x1839c8700	0.386s	rubyengine.dll	func@0x1839c8700
std::getline<char,struct std::char_traits	0.386s	openstudiolib.dll	std::getline<char,struct std::char_traits<char>,class std::allocator<char> >(class
func@0x183b0eecc	0.384s	rubyengine.dll	func@0x183b0eecc
LoadLibraryA	0.374s	KERNELBASE.dll	LoadLibraryA
func@0x180204730	0.369s	energyplusapi.dll	func@0x180204730
func@0x1839cab4c	0.328s	rubyengine.dll	func@0x1839cab4c

```
Call Stacks
63.8% (1.430s of 2.241s)
rubyengine.dll ! func@0x1839db7d4
rubyengine.dll ! rb_vm_exec+0xde
rubyengine.dll ! func@0x1839c8700+0x260
rubyengine.dll ! rb_yield_1+0x58
rubyengine.dll ! rb_ary_each+0x7d
rubyengine.dll ! func@0x1839d756c+0x15b
rubyengine.dll ! func@0x1839d74a4+0x89
rubyengine.dll ! func@0x1839e5e44+0x79
rubyengine.dll ! func@0x1839db7d4+0x20e5
rubyengine.dll ! rb_vm_exec+0xde
rubyengine.dll ! func@0x1839c8700+0x260
rubyengine.dll ! rb_yield_1+0x58
rubyengine.dll ! rb_ary_each+0x7d
```

```
mainCRTStartup (TID: 42300)
mainCRTStartup (TID: 30196)
func@0x14000133c (TID: 4...
func@0x4014e0 (TID: 22576)
Thread (TID: 16688)
func@0x180021b70 (TID: 2...
func@0x180021b70 (TID: 1...
CPU Utilization
```

Figure 1: Snapshot of VTune Profile of OpenStudio

Note that most of the referenced items have names like "func@0x1839cab4c", which is not the name of any function within the OpenStudio codebase. This result was initially confusing, but under further investigation made sense. The program "openstudio" embeds a Ruby interpreter and much of the resulting functionality of the program (running Ruby scripts, executing OpenStudio workflows, etc.) is accomplished by the Ruby interpreter executing embedded and provided Ruby code (e.g., OpenStudio measures). VTune (as currently configured) is not able to fully connect the code that it sees is being executed to the name associated with that code because some of the code that is running within Ruby. The authors believes that with additional effort, this connection could be made, but progress made with another profiler (see below).

2.3 INSTRUMENTS PROFILING

2.3.1 Setup

Apple's Instruments profiling package rather lives up to Apple's older advertising slogans and works on the released version with minimal difficulty for a knowledgeable developer.

2.3.2 Results

Running a similar workflow through the OpenStudio command line interface generates similar outputs to the results obtained with VTune. Example results are shown in Figure 2, in which the text-based output from Instruments is shown. As expected, many of the calls are to functions associated with the Ruby interpreter. Processing eventually moves into library calls, though the reader should keep in mind that these are calls from Ruby code back into the OpenStudio SDK rather than native calls from the compiled executable.

```
27.70 s  100.0%    rb_vm_exec
27.70 s  100.0%     vm_exec_core
27.60 s   99.6%      vm_call_cfunc
26.16 s   94.4%       rb_ary_each_index
26.16 s   94.4%        rb_yield
26.16 s   94.4%         rb_vm_exec
26.16 s   94.4%          vm_exec_core
26.12 s   94.3%           vm_call_cfunc
16.87 s   60.8%            rb_ary_each
16.87 s   60.8%             rb_yield
16.79 s   60.6%              rb_vm_exec
16.79 s   60.6%               vm_exec_core
16.18 s   58.4%                vm_call_cfunc
 4.07 s   14.6%                 rb_class_s_new
 4.06 s   14.6%                  rb_funcallv_kw
 4.06 s   14.6%                   rb_call0
 4.06 s   14.6%                    vm_call0_body
 1.78 s    6.4%                     _wrap_new_AirLoopHVAC(…)
…
 1.84 s    6.6%                 _wrap_StraightComponent_addToNode(…)
 1.82 s    6.5%                 _wrap_PlantLoop_addDemandBranchForComponent(…)
 1.50 s    5.4%                 _wrap_IdfObject_setName(…)
 1.05 s    3.7%                 _wrap_WaterToAirComponent_addToNode(…)
…
```

Figure 2: Instruments Text Output

The team noted that the `IdfObject` `setName` method is taking as long as functions that are ostensibly doing much more, so an investigation of `setName` was chosen as a first step in the process.

2.4 INVESTIGATION OF SETNAME

The `setName` method of the `IdfObject` data structure is investigated to see what can be learned, and if there are opportunities to improve its performance. `IdfObject` is the parent class of all objects in the IDF file. The `IdfObject` class, like most other classes within OpenStudio, follows the PIMPL design pattern and is a thin wrapper around an implementation class named `IdfObject_Impl` that implements the functionality of the object. The complete function is only 60 lines of C++ code and is included in Appendix A. To preform ad hoc testing of the code's functionality, a small program was compiled against the OpenStudio SDK and then run in various ways to investigate the operation of `setName`.

The function starts off with an encoding of the input name, which is followed by a lengthy (lines 4 to 33) switch statement to modify the name for EMS objects. The encoding operation could probably be improved, but tests did not show that significant time was spend in that operation. The main portion of the function is contained in an "if" statement that checks if the object has a name field (lines 36 to 58). After some setup, lines 46 to 57 set the name and then return an appropriate version of the name to the calling code. Testing showed that a significant portion of the execution time was spent in getting the index of the name field.

Each `IdfObject` is linked to an `IddObject` that describes the object. Note that in line 36, the call that will hopefully return an index is a call to a member function of the linked `IddObject` (stored in `m_iddObject`). Tracking the call through the PIMPL chain shows that the `IddObject` implementation caches a return value after it is called, which does seem to make a difference with the ad hoc testing that was done. Ultimately, though, determining whether an object has a name can result in string comparisons (checking that the field is named "Name") and this is probably something that should be avoided since all EnergyPlus objects now have a name (courtesy of the epJSON format). For this function, it appears that the best fixes are somewhat out of the scope of the function itself.

3. SUMMARY AND RECOMMENDATIONS

3.1 SUMMARY

Profiling was set up and run on both Windows (using Intel's VTune) and MacOS (using Apple's Instruments), and initial results confirm the authors' expectations that the call structure is rather complex and intertwined with the way that Ruby is used by OpenStudio. The VTune profiler has some additional issues that will need to be resolved for it to be useful going forward, but the results from Instruments were sufficient for the work done this FY. The Instruments output suggested some initial places to look, and the `setName` method of the `IdfObject` class was chosen for exploration. Results of the investigation of `setName` point back to the general handling of the data (i.e., the Workspace object) as a place where work may be needed. In particular, recent developments within the EnergyPlus data model could reduce the work required for operations on the Workspace. With the discontinuation of this effort in FY24, no additional resources are available to continue the exploration begun here. It is hoped that this report will allow for any future effort in this area to be started quickly.

3.2 RECOMMENDATIONS

Based upon the findings here, the following recommendations are made:

1) Since Windows is the predominant platform for desktop computers, determining an approach to using the VTune profiler would be very valuable. To continue this effort, working this out would be a good starting point.

2) The Workspace approach should be reevaluated. Much of the effort in `setName` is unnecessary since every object in EnergyPlus must have a name, which means that most (as far as the authors know) OpenStudio objects should also have a name. The stateless nature does not help either, as the object does not know if it has a name or not. Transitioning away from the stateless approach might be very involved but may be the only way to make progress on performance.

4. REFERENCES

[1] NREL, *OpenStudio*, 2023, openstudio.net

[2] U.S. Department of Energy, *EnergyPlus*, 2023, energyplus.net

APPENDIX A. C++ SOURCE FOR SETNAME

APPENDIX A. C++ SOURCE FOR SETNAME

```cpp
boost::optional<std::string> IdfObject_Impl::setName(const std::string& _newName, bool) {
    std::string newName = encodeString(_newName);

    switch (m_iddObject.type().value()) {
      // get all EMS idd object types in both E+ and OS IDD
      case openstudio::IddObjectType::EnergyManagementSystem_Actuator:;
      case openstudio::IddObjectType::OS_EnergyManagementSystem_Actuator:;
      case openstudio::IddObjectType::EnergyManagementSystem_ConstructionIndexVariable:;
      case openstudio::IddObjectType::OS_EnergyManagementSystem_ConstructionIndexVariable:;
      case openstudio::IddObjectType::EnergyManagementSystem_CurveOrTableIndexVariable:;
      case openstudio::IddObjectType::OS_EnergyManagementSystem_CurveOrTableIndexVariable:;
      case openstudio::IddObjectType::EnergyManagementSystem_GlobalVariable:;
      case openstudio::IddObjectType::OS_EnergyManagementSystem_GlobalVariable:;
      case openstudio::IddObjectType::EnergyManagementSystem_InternalVariable:;
      case openstudio::IddObjectType::OS_EnergyManagementSystem_InternalVariable:;
      case openstudio::IddObjectType::EnergyManagementSystem_Program:;
      case openstudio::IddObjectType::OS_EnergyManagementSystem_Program:;
      case openstudio::IddObjectType::EnergyManagementSystem_Sensor:;
      case openstudio::IddObjectType::OS_EnergyManagementSystem_Sensor:;
      case openstudio::IddObjectType::EnergyManagementSystem_Subroutine:;
      case openstudio::IddObjectType::OS_EnergyManagementSystem_Subroutine:;
      case openstudio::IddObjectType::EnergyManagementSystem_TrendVariable:;
      case openstudio::IddObjectType::OS_EnergyManagementSystem_TrendVariable:;
        // replace “ ” with “_”
        std::replace(newName.begin(), newName.end(), ' ', '_');
        break;
      //case openstudio::IddObjectType::EnergyManagementSystem_OutputVariable:;
      //case openstudio::IddObjectType::OS_EnergyManagementSystem_OutputVariable:;
      //TODO Enforce Title-Case
      default:
        // no-op
        break;
    }

    // check Idd to see if this object has a name
    if (OptionalUnsigned index = m_iddObject.nameFieldIndex()) {
      // if so, change the name, or create it
      unsigned n = numFields();
      unsigned i = *index;
      OS_ASSERT(i < 2u);
      if (n == 0 && i == 1) {
        OS_ASSERT(!m_handle.isNull());
        m_fields.push_back(toString(m_handle));
        m_diffs.push_back(IdfObjectDiff(0u, boost::none, m_fields.back()));
      }
      n = numFields();
      if (i < n) {
        std::string oldName = m_fields[i];
        m_fields[i] = newName;
        m_diffs.push_back(IdfObjectDiff(i, oldName, newName));
      } else {
        m_fields.push_back(newName);
        m_diffs.push_back(IdfObjectDiff(i, boost::none, newName));
      }
      //return decoded string since we might have made changes to it if its an EMS object.
      newName = decodeString(newName);
      return newName;  // success!
    }
    return boost::none;  // no name
  }
```