Generating and Analyzing Program Call Graphs using Ontology

Ethan Dorta and Yonghong Yan
Department of Computer Science
University of North Carolina at Charlotte
Charlotte, North Carolina, USA
{edorta, yyan7}@uncc.edu

Chunhua Liao
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, California, USA
liao6@llnl.gov

***Abstract*—Call graph or caller-callee relationships have been used for various kinds of static program analysis, performance analysis and profiling, and for program safety or security analysis such as detecting anomalies of program execution or code injection attacks. However, different tools generate call graphs in different formats, which prevents efficient reuse of call graph results. In this paper, we present an approach of using ontology and resource description framework (RDF) to create knowledge graphs for specifying call graphs to facilitate the construction of full-fledged and complex call graphs of computer programs, realizing more interoperable and scalable program analyses than conventional approaches. We create a formal ontology-based specification of call graph information to capture concepts and properties of both static and dynamic call graphs so different tools can collaboratively contribute to more comprehensive analysis results. Our experiments show that ontology enables merging of call graphs generated from different tools and flexible queries using a standard query interface.**

***Index Terms*—Callgraph, ontology, knowledge graph, resource description framework, program analysis**

I. Introduction

Program call graphs [1], [2] have been used to represent the calling relationships between procedures or subroutines in a computer program. Each node of the graph represents a procedure and each edge indicates the caller and callee relationship of two procedures. A cycle in a graph indicates recursive procedure calls. Call graphs, also known as call multigraphs, are control flow graphs that can be used for program analysis tasks that rely on knowing the control flow of a program execution, e.g. inter-procedure data flow analysis and optimization. Call graphs themselves can be used for analysis such as finding procedures that are never called, detecting anomalies of program execution, or code injection attacks.

Call graphs can be created by different tools (e.g. compiler, profiling, or tracing tools), and contain different types of information (e.g., static or dynamic call graph). For program analysis and execution analysis, it is often required to construct and combine multiple call graphs and to query and analyze the call graphs. Existing graph languages such as DOT and GraphML that are used for call graph description focus on graph visualization and layout, but are not designed for composing multiple similar graphs as needed for call graphs. They are also not designed for graph query and searching. With these languages, when a call graph becomes large, it is hard to read to find useful information even after visualization.

In this paper, we present our approach of using ontology and Resource Description Framework (RDF) for specifying call graphs to facilitate the construction of full-fledged and complex call graphs of computer programs, and to support query and search of call graphs using tools. Leveraging the interoperability feature of ontology, call graphs specified as knowledge graphs would be naturally and highly composable, thus facilitating combining dynamic and static call graphs of a program into one graph. Ontology also provides its own semantic query language, SPARQL, for retrieving and manipulating data stored in RDF. The language and its tools would enable more advanced searching and analysis of the call knowledge graphs for caller-callee relationships, its contextual information, as well as whole program analysis. With RDF, the caller and callee (graph nodes) and the caller-callee relationships (graph edges) in conventional call graphs become more extensible than using DOT or GraphML when it needs to include additional attributes for program analysis.

The contributions of this paper include: 1) introducing the notion of knowledge graphs for program analysis starting with call graph construction; 2) designing a formal ontology-based specification of call graph using Resource Description Framework (RDF) aiming for interoperability and composability for constructing call graphs of complex software packages, and for query and searching knowledge call graph using tools; 3) implementation and demonstration of call graph construction from both static and dynamic call graphs. The evaluation is performed using the Quicksilver proxy application to show the advantage of using RDF-based call graphs.

For the rest of the paper, Section II provides the background and motivation. Section III describes the formal ontology-based specification of call graphs. Section IV includes our implementation details for creating static and dynamic call graphs. Section V provides evaluation study. In Section VII, we study the related work, and we conclude the paper in Section VIII.

II. Background and Motivation

Call graphs or caller-callee relationships have been used for various kinds of static program analysis, performance analysis

and profiling, and for program safety or security analysis such as detecting anomalies of program execution or code injection attacks. Depending on the purpose, different kinds of call graphs, e.g. static or dynamic call graphs, and with and without calling context information can be constructed. For example, static interprocedural program analysis relies on static call graphs, and often requires calling-context combined with data-flow analysis. Such analysis and optimizations include functions within or across compilation units, program partitioning and re-organization to improve the proximity of frequently called functions and usage of data, coalescing of global variables, unreachable code elimination, interprocedural constant or copy propagation and set propagation, interprocedural pointer alias analysis, interprocedural unreachable code and store elimination, etc.

A. Challenges of Generating and Using Call Graphs

Call graphs represent the high-level code structure of a program and can be used for information or knowledge extraction from software systems [3]. Constructing a full-fledged call graph can be a challenging task for several reasons. 1) Dynamic call graphs represent the caller-callee relationships obtained during the program execution, and it is often used for performance profiling. Static call graphs represent every possible caller-callee relationships of a program and it is often used for static program analysis. It is also often needed that these two types of graphs can be combined together for more advanced analysis and performance optimization. 2) Call graphs can also be enhanced to include calling context information, thus making a call graph to be context-sensitive for more advanced program analysis tasks. However, adding calling-context is often problematic since it requires more data-oriented tracing of program execution that could significantly slow down program execution and generate a large amount of tracing data. 3) With languages that feature dynamic dispatch of polymorphic class methods or functions to call at run time, such as in Java and C++, computing a static call graph precisely is complicated, e.g. requiring alias analysis [4]. 4) Constructing a complete call graph of a software package that depends on other packages and libraries could also be challenging as the call graphs from multiple packages need to be combined and composed. In summary, the complexity of call graphs often requires the merging of results generated by different tools and using different methodologies.

Existing approaches to generating call graphs and usage of call graphs have been focusing on analysis or visualization by humans, or to be combined with program analysis of the compiler. Different tools generate call graph results in ad-hoc formats. It is challenging to programmably use call graphs generated by existing tools. For example, the text-format DOT flat graph used for storing call graphs is not programming friendly, such that searching or querying for a caller-callee relationship requires parsing of text file. Embedding in the call graph both caller-callee relationships as graph edges and nodes, and call context information as attributes of edges and nodes makes the call graph complicated and hard to sort and query.

In many communities, standard guidelines and recommended best practices are being developed to make scientific data Findable, Accessible, Interoperable, and Reusable (FAIR) [5]. We believe that the same principles can be applicable for the domain of program analysis, including call graph analysis. Our approach explores the "*knowledge*" or "*ontology*" aspects of the software structure and its high-level information, and aims to design a more interoperable structure and interface of software call graphs.

B. Ontology and Knowledge Graph

An ontology [6], [7] is a formal specification for explicitly representing knowledge about types, properties, and interrelationships of the entities in a domain. Defining ontologies has many benefits, including providing a common vocabulary to represent and share domain concepts and enhancing interoperability and reusability of heterogeneous datasets.

An ontology can form a complex graph to model any kinds of relationships between entities. The resulting graphs are often called knowledge graphs. Each node in the knowledge graph represents a concept or instance, and each edge carries a property indicating the relations between the two nodes it connects. For example, there can be two nodes named “Animal" and “Cat" indicating two concepts (or classes). An “is-a" edge between them means that one is a subclass of the other. An edge labeled “rdf:type" denotes that a node is an instance of a node representing a class. A simple data model called Resource Description Framework (RDF) is used to store an ontology (or its equivalent knowledge graph) as triples, in a form of *(subject, predicate, object)*. For instance, *(“Cat", is-a, “Animal")* states that *Cat* is a subclass of *Animal*.

A knowledge graph can be queried using a standard RDF query language named SPARQL. An example query in Listing 1 can be used to find all individual cats located in a city called Dublin. *rdf:type* is a standard RDF property to link a resource as an instance to its class. *:locatedIn* is the property to link a resource to its location. ?<identifier>, e.g. ?s in the list, can be considered as a variable declaration or reference.

Listing 1: Example SPARQL Query

```
SELECT ?s
WHERE { ?s rdf:type :Cat .
        ?s :locatedIn :Dublin }
```

Many ontologies have been developed to enhance interoperability and reusability of data in different domains. For example, DBpedia [8], [9] is an ontology extracted from Wikipedia to connect datasets. Schema.org [10] is designed to improve the interoperability of web data. Yue et al. developed ontology-based program analysis [11], though the work is limited to representing language constructs. Liao et al. explored ontologies to enhance domain-specific language implementations [12]. Pattipati et al. [13] built an extensible framework for ontology-based advanced program analysis. Thus ontology and the existing tools in this domain provide

an innovative direction for software program analysis. In this paper, we experiment the use of this methodology for call graphs to evaluate its feasibility and explore its potential.

III. Design and Specification of Call Graphs Using Ontology

In many research communities, researchers are establishing standard guidelines and recommending best practices to make scientific data Findable, Accessible, Interoperable, and Reusable (FAIR) [5]. We believe that the same principles can be applicable for the domain of program analysis. Briefly, Findability means that data can be found online, typically through indexing in search engines. Accessibility indicates that data can be retrieved directly or via an approval process. Interoperability means that data follows standards. Finally, reusability denotes that the context of data generated (metadata) is documented so it can be compared to or integrated with other datasets.

Following the FAIR principles, we are designing a call graph ontology to enable interoperable and scalable call graph analysis results. This ontology is a supplemental component of the ongoing HPC Ontology [14] effort aimed to implement FAIR principles for training datasets and AI models in the high-performance computing (HPC) domain, with a focus on Machine Learning driven program analysis and optimizations. Currently, the HPC Ontology has a core component and a few supplemental components. The core component captures essential concepts and properties related to an experiment using some input data and generating output data. Supplemental components provide more details for associated software, hardware of the experiments, as well as different types of input and output data (such as performance profiling datasets and machine learning models).

A. Ontology for Call Graphs

We define a few concepts in the call graph ontology, including CallGraph, Function, MainFunction and CallSite. An instance of CallGraph may include a set of Function instances. MainFunction is a subclass of Function to represent the main entry functions. CallSite is the location within source code denoting a call to a function. The concepts and instances can be linked using a set of properties as shown in Table I.

TABLE I: Major Properties of the Call Graph Ontology

Property	Data-type	Description
Basic Properties		
hpc:wasDerivedFromSoftware	xsd:anyURI	URI of the software/benchmark
hpc:name	xsd:string	Name of the thing (e.g. function or call graph)
hpc:alternateName	xsd:string	An alias for this item
hpc:mangledName	xsd:string	The mangled name of a function
hpc:description	xsd:string	Short description
hpc:function	xsd:anyURI	link a call graph to its function instance
hpc:sourceFile	xsd:anyURI	link a function to its source file name
hpc:lineNumber	xsd:integer	link a function to its line number
hpc:isPartOf	xsd:anyURI	link a library function to its parent library
Pairs of Edges for Callsite		
hpc:callSite	xsd:anyURI	link a function to its call site
hpc:srcFunc	xsd:anyURI	link a call site to its source function (caller)
hpc:destFunc	xsd:anyURI	link a call site to destination function (callee)
hpc:upstreamCallSite	xsd:anyURI	link a callee to its call site
hpc:calls	xsd:anyURI	link a caller to a callee
hpc:calledBy	xsd:anyURI	link a callee to a caller
Provenance Information		
hpc:version	xsd:string	Version Number of a call graph
hpc:contributor	xsd:anyURI	Contributors of the call graph
hpc:submitter	xsd:anyURI	Who submits this piece of info.
hpc:submitDate	xsd:dateTime	Date of submission
Profiling information for dynamic call graph		
hpc:numOfCalls	xsd:integer	Number of times the function is called
hpc:inclusivePercentageTime	xsd:float	Execution time percentage inclusively
hpc:inclusiveTime	xsd:float	Execution time in s/ms/... inclusively
hpc:exclusivePercentageTime	xsd:float	Execution time percentage exclusively
hpc:exclusiveTime	xsd:float	Execution time in s/ms/... exclusively

Function calls are represented either by direct properties among functions or through an intermediate concept called call sites. The latter allows more precise modeling of caller-callee relationships. The hpc:calls property associates two functions by indicating one function calls the other. Likewise, hpc:calledBy inverts this relation by associating a callee function upstream with its caller. Functions can be similarly linked through call sites they are involved with as well using the hpc:callSite and hpc:upstreamCallSite properties, with hpc:callSite being used to indicate a function is the caller of a callsite and hpc:upstreamCallSite indicates a function is called by a call site. For call sites, the hpc:srcFunc and hpc:destFunc properties indicate the caller of the call site (the function in which the callee function is invoked) and the callee of the call site, respectively.

Functions and call sites may have both static and dynamic properties. The static properties describe information about the source code and program structure, such as source locations and other debug attributes. The dynamic properties are relating to information extracted from a timed run of the given program. The integer property hpc:numOfCalls simply lists the number of times each function is called or each call site is reached. For the dynamic properties hpc:inclusive(Percentage)Time and hpc:exclusive(Percentage)Time, the inclusive and exclusive refer to whether or not the times spent in callee functions also are factored into the measurement. For the performance tool gprof, inclusive and exclusive response to cumulative and self times/percentages, respectively.

For functions defined in the C/C++ standard libraries, the call graph ontology can be modified to provide instances for both the functions and their associated libraries. We have generated nodes for the functions in the C standard library and imported them into the ontology. For example, math functions like hpc:sqrt and hpc:abs are provided, as well as their corresponding library hpc:cmath. We also provide a concept of hpc:StandardLibrary so hpc:cmath is specified to be an instance of hpc:StandardLibrary. These entities can be helpful when creating queries skipping standard library functions used in user codes.

Figure 1 shows an example call graph using the proposed call graph ontology. It includes information from both static

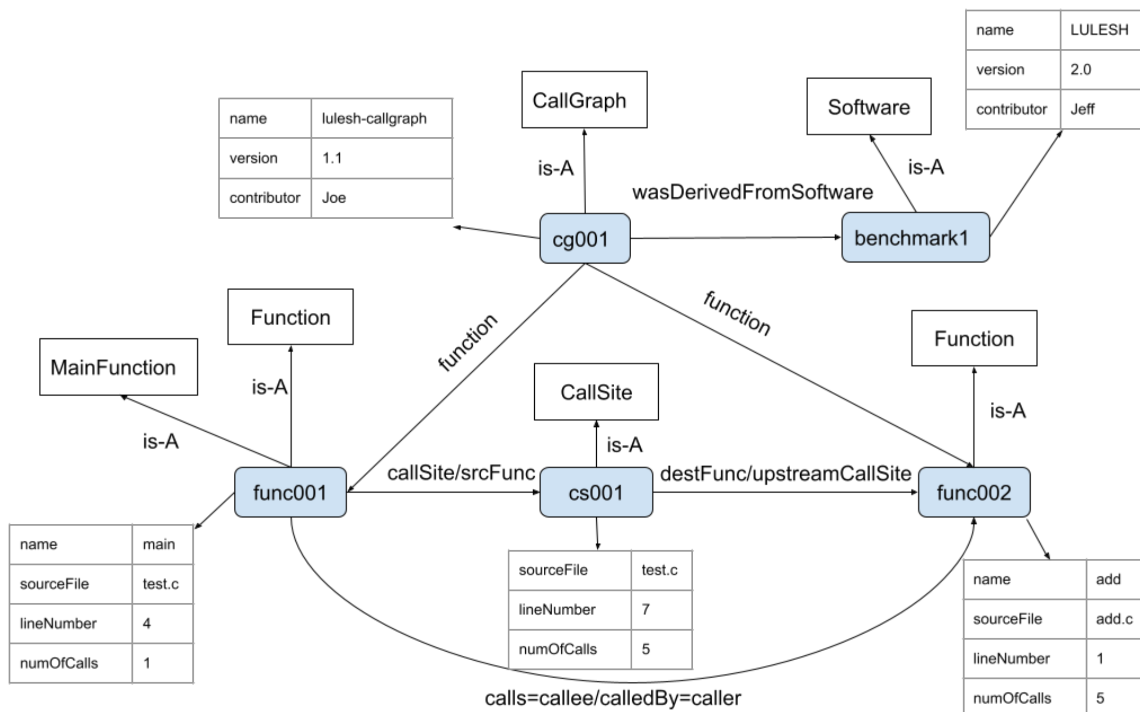

Fig. 1: An Example Call Graph Visualization

and dynamic analysis. The callgraph instance (cg001), derived from a benchmark software, provides links to two functions: func001 [main] and func002 [add]. The two functions are linked together by a Callsite node that indicates the caller/-callee relationship between them. Specifically, there is an outgoing hpc:callSite link from func001 to the call site node (cs001) and an outgoing link (hpc:destFunc) from the call site node to func002, indicating that func001 calls func002 through a call site. The hpc:numOfCalls property, derived through dynamic analysis, also indicates that func002 (and cs001) was called five times. This callgraph in RDF forms provides detailed information about caller-callee relationships, function attributes, call attributes, etc, thus allowing for more advanced analyses.

B. Universal URI Design

In order for data to be highly accessible for universal analysis and comparison, it is important that each entity of the RDF-based callgraph to be uniquely identified as a single resource, namely the need for a universal resource identifiers (URI). URI's point a retrievable (resolvable) resource if possible and follow a distinct pattern to allow readability for node identifiers. Although this is difficult to do when working with local files, code bases that are reflected in a remote (and web accessible) repository can use the repository URL to create semantic linked data. The designed static pass allows for providing a base URI to the graph generator that then generates URI's for each resource, utilizing the file path (assuming it is congruent in the remote codebase as well), a fragment with either the function name or a compound name indicating a call site between two functions.

For consistency with function names, the Itanium C++ ABI[1] is used as the standard for mangling. The reasons are two-fold; cross-referencing composable data, such as information between two different compiler tool-chains, requires a consistent identification scheme. As both LLVM/Clang and GCC adhere to this scheme, mangled function names can be used to reliably merge data produced from either compiler. Secondly, qualified (unmangled) names can introduce characters such as asterisks, ampersands, colons, and whitespace that are not valid in a URI fragment [15]. A partial mangling of the name would therefore still be required. Itanium mangling does not support singular unqualified identifiers that contain non-alphanumeric characters (excluding underscores) and all other C++ constructs are mangled to conform to this as well.

The design of URI has to consider the toolchain involved in generating call graphs. For example, the LLVM linker does not allow symbols to be defined more than once in a bitcode unit, so the fragment defining the name of a function must be unique for every URI. As this name fragment is the end of the URI for functions, it is guaranteed to be unique and clashes will not occur. These function names form a *natural key* in the URI, allowing for an established pattern [16] for creating URI's using concatenation. It is suggested that for version control repositories, the chosen base URI should be reflecting a given commit or version of a codebase rather than a default link to the latest version, as this can change between

[1]http://itanium-cxx-abi.github.io/cxx-abi/abi.html

updates and reduce the underlying integrity of the knowledge graph.

As an example, we would like to refer to the `updateTrajectory` function within the `LLNL/Quicksilver` repository on GitHub, defined in the src/CollisionEvent.cc file. As linking to a specific commit is desired, `ad9bf04` will be used. The base URI for this example would then be https://github.com/LLNL/Quicksilver/tree/ad9bf04/. Mangled, the updateTrajectory function becomes `_Z16updateTrajectoryddR11MC_Particle`. Combining the file path as part of the URL and the mangled function name as a URI fragment gives a final URI of https://github.com/LLNL/Quicksilver/tree/ad9bf04/src/CollisionEvent.cc#_Z16updateTrajectoryddR11MC_Particle.

IV. Implementation

This section describes how the proposed call graph ontology can be used in practice. Our implementation leverages existing call graph generating tools and enhances them to collect more program information and generate RDF-based call graphs. For static call graph generation, we improve an LLVM call graph analysis pass to produce a static call graph in RDF format. For dynamic call graph generation, we modify the `gprof2dot` profiling analysis tool to produce a dynamic call graph, leveraging the capability of gprof2dot to analyze profiling information from profiling tools such as gprof and perf. We use the Blazegraph database [2] for storing and merging dynamic and static call graphs. This database also supports query and analysis of call graphs using SPARQL.

A. Static Call Graph

The Clang/LLVM compiler provides a callgraph compiler pass to produce call graphs in the DOT format via its `opt` command. However, this pass does not provide enough information to build a knowledge graph: the DOT format is unsuitable for storing structured data, and more detailed information about the functions and the corresponding call sites (such as file location and line numbers) is not present, making it unusable for more detailed querying.

We developed a custom LLVM pass that iterates over each defined function in the provided bitcode file, identifying call instructions within the function bodies, and extracting debug information from each call that can then be used to build an RDF knowledge graph. This process is run by compiling the individual files of the collective program into LLVM bitcode files with debug information, linking all files together with `llvm-link`, and passing the linked file to LLVM `opt` to run a custom-built pass. The extracted information is serialized into RDF format by the Redland `librdf` C library[3].

B. Dynamic Call Graph

For dynamic call graphs, we leverage GNU gprof [4] to produce text-formatted call graphs. We modified the Python library `gprof2dot` [5] to produce the RDF-formatted dynamic graph of a program. `gprof2dot` accepts many popular program profiling formats including gprof, callgrind, and perf, which it internally parses into a uniform structure to generate DOT graphs. The script was modified to extract this information in order to build a knowledge graph using the `rdflib` Python library.

As the chosen profiling software (gprof) does not provide complete call site information, we instead opt to use a different type of node called an AggregateCallsite. This node type does not guarantee all location information will be supplied (such as line, column, or file number) but that there is information describing performance statistics of a given call between two functions.

C. Merge Dynamic and Static Call Graph

Merging the two produced knowledge graphs allows for a more complete data set and enables new queries to be performed that can take into account both timing and source information. As programs compiled with GCC and Clang contain slightly different symbol tables and debug information, merging the two graphs requires logic to identify items in the different graphs that refer to the same entity. The merging process consists of a series of SPARQL queries run against both graphs, which are then processed by a Python script to be properly merged and written back to disk.

As LLVM and GCC use the same mangling scheme, identifying functions across the knowledge graphs can be trivial. For call sites, a more sophisticated merging approach is necessary since the generated dynamic callgraph does not have complete call site information. A merge candidate query is run that attempts to match each aggregate call site with a precise call site (obtained from static analysis) by checking 1. whether or not this caller-callee pair only occurs once within the program or 2. whether or not the pair occurs once within the given line. On the occasion that there has been shifting in line numbers due to differences between pre-processing, the corresponding call sites will be sorted and paired up by line number ordering rather than direct line number.

A merge cannot occur if either: 1) the call sites occur multiple times per line such as a = callY()+callY (); and 2) the dynamic analysis contains a function or function invocation that was not present in the static analysis (such as a compiler generated function). The remaining unmerged nodes are collected via multiple queries to ensure that there is no duplication.

As an example, let <F(A, static info), F(B, static info), CS(A→B, 10, 13), CS(A→B, 11, 13), CS(A→B, 11, 17)> describe a static analysis result in which F(A, static info) represents a function named "A" with static info and CS(A→B,

[2]https://blazegraph.com/
[3]https://librdf.org
[4]https://sourceware.org/binutils/docs/gprof/
[5]https://github.com/jrfonseca/gprof2dot

11, 13) represents a static call site where A calls B on line 11 and column 13.

Then, let <F(A, dynamic info), F(B, dynamic info), ACS(A→B, 10), ACS(A→B, 11)> represent the dynamic analysis result (note the loss of column numbers due to gprof results), with ACS being an aggregate call site.

In merging, the static call site occurring on line 10 can be merged with its corresponding dynamic "aggregate" call site as there is a pairing between them. However, the merge fails for the two call sites on line 11, which do not pair with the single aggregate call site returned by gprof for line 11. Thus, there is no merge for these callsites. The resulting final merged graph is then <F(A, static + dynamic info), F(B, static + dynamic info), CS(A→B, 10, 13), CS(A→B, 11, 13), CS(A→B, 11, 17), ACS(A→B, 11)>.

Note that the merged node at line 10 becomes a Callsite after merging Callsite and AggregateCallsite. To maintain all original call graph information, we don't merge call sites at line 11, although they could be merged into a single AggregateCallsite with static information lost. This is a design choice with trade-offs so we can easily enable the merge with information loss, if users prefer.

V. Evaluation

We evaluate our implementation using the Quicksilver [6] proxy application. Quicksilver is developed at the Lawrence Livermore National Laboratory that comprises a large toolchain for performing particle simulations. Overall, Quicksilver comprises over 9000 lines of code and 841 defined functions. The static graph results in a densely populated call graph, making it an ideal candidate for testing queries against the program. We evaluate in two aspects: 1) how the call graph ontology enables merging of both dynamic and static call graphs into one, and 2) how our implementation enables scaling-down large call graphs using SPARQL queries. For hardware, the machine used is an AMD Ryzen 5 2500U CPU running at 2 Ghz, using Xubuntu Linux with 16 GB of RAM.

A. Integrating Heterogeneous Call Graph Information

1) Static call graph: The Quicksilver source code is analyzed using the given call site analyzer by using LLVM bitcode and linking into a single file. This bitcode file is then fed to the static call graph generator to produce the RDF call graph. In total, the resulting RDF knowledge graph of Quicksilver has just over 100,000 triples (edges or attributes). These triples provide function attributes such as source locations of definitions as well as relations between functions. More specifically, the resultant graph models all possible calls any function within the program can make, regardless of whether or not the call would actually happen in an execution.

2) Dynamic call graph for a sequential execution: The dynamic call graph is produced by first compiling the whole program to produce the binary executable with gprof profiling option enabled. This building step can be done together with the LLVM compilation step for generating static call graphs, or completed separately. The Quicksilver program was then executed with `-nSteps=1` to do a single full run-through. On the aforementioned machine, this run took 49.8 seconds. The execution produces gprof profiling output and we then run our enhanced gprof2dot script to process the gprof output and produce the RDF-based dynamic call graph, which contains just over 54,000 triples.

3) Merged (heterogeneous) call graph: For the final merge query, the two separate static and dynamic call graphs were loaded into Blazegraph, where the merge took place from a script that accessed Blazegraph via the SPARQL endpoint exposed by the software. The triples were then stored back onto disk using the `rdflib`. The resultant merged call graph has almost 125,000 triples On the machine listed above, the merge query took 2 minutes and 24 seconds for Blazegraph. As a demonstration of the merging mechanism, the function `cycleTracking` in src/main.cc calls `MC_Particle_Buffer::Receive_Particle_Buffers` on line 273, which has a corresponding node in the static call graph. The dynamic call graph has one and only one node describing the same interaction, and the two call sites are able to be merged. On the other hand, the function `Tallies::CycleFinalize` calls the overloaded operator `qs_vector<CellTallyTask>::operator[](int)` twice on line 80 of src/Tallies.cc as shown below (broken into multiple lines for clarity)

```
_cellTallyDomain[domainIndex]._task[0].Add(
    _cellTallyDomain[domainIndex]._task[
        replication_index
    ]
);
```

The two related static call sites are not able to merge with the dynamic call site describing the line, so both have to be included separately in the final graph.

B. Flexible and Scalable Analysis and Query of Call Graphs

An ontology can be evaluated based on its ability to answer common questions asked within the domain it intends to describe. These questions are often called competency questions. Several competency questions that arise in performance analysis that are related to function call and call graph analysis are listed as follows:

- Q1: Which functions take more than a certain amount of time to run?
- Q2: Which functions are called at least a certain number of times, or called from at least a certain number of places?

TABLE II: Properties of Generated Call Graphs

Graph	Nodes	Functions	Edges	Size (KB)	Exe. Time
Static	11,949	3,454	101,553	11,476	5.632 sec
Dynamic	7,268	2,563	54,089	4,968	15.237 sec
Merged	13,830	3,807	124,620	15,988	2 min 24.422 sec

[6]https://github.com/LLNL/Quicksilver

- Q3: Which functions are not from a standard or specific library?
- Q4: Which functions are directly or indirectly called by a given function, traversing the call graph down to a given depth?
- Q5: What are all possible callpaths to a given function?

Most importantly, the ontology must provide intuitive query design; queries designed with the ontology should be human-readable and understandable by individuals familiar with the domain. The ability for our ontology to successfully answer these questions, as well as provide scalable call graph analysis, are evaluated in a series of queries against the Quicksilver software using Blazegraph.

Function A (bound as `?functionA`) refers to the class method `ParticleVaultContainer::collapseProcessing` defined on line 198 of src/ParticleVaultContainer.cc, which is only called 12 times throughout the program and sits near the top of the call graph. *Function B* (bound as `?functionB`) refers to `rngSample`, a psuedo-random number generator defined on line 23 of src/MC_RNG_State.hh, and is called 9,856,095 times during the sample run.

1) Query for the functions that take more than a certain ratio of the total execution time: Timing function execution is one of the most basic tasks required by a profiling tool. The query in Listing 2 shows an example to achieve this. The query first finds the total time of the main function. Then it finds all functions (the main function included) that takes more than a given ratio of the total time. This query essentially returns a list of hot functions and their execution times.

Listing 2: Find Hot Functions

```
SELECT ?function ?time WHERE {
  ?main hpc:name "main" .
  ?main hpc:inclusiveTime ?totaltime .

  ?function rdf:type hpc:Function .
  ?function hpc:inclusiveTime ?time .
  FILTER ( ?time/?totaltime >= ?ratio )
}
```

The query is constructed by initially finding the overall execution time of the program and binding it to a variable named `?totalTime`. This is done by searching for entities that have the name (specifically using the ontology property `hpc:name` "main", and then extracting the runtime attached to the `hpc:inclusiveTime` property. All other functions within the graph are then retrieved by matching their node type and their running times are similarly extracted. To retrieve functions that satisfy the condition, the SPARQL `FILTER` construct is used that accepts more complex expressions, including supporting arithmetic. The ratio between the total execution time and function execution time is computed and only functions that exceed the bound variable `?ratio` are returned. The values of `?ratio` are shown along with the results in Table III, with Functions and Runtime referring to the number of results, specifically functions that satisfy the query, and the execution time of the query, respectively.

2) Query for functions that are called more than 100 times, or called in more than 10 sites: The following query accesses data merged from two call graphs, leveraging both dynamic (call count) and static (callsite count) information together. Heterogeneous call graph information, such as the merged dynamic and static call graphs of Quicksilver, is able to be accessed intuitively through the ontology, demonstrating the support for interoperability provided by the ontology.

Listing 3: Find functions called either over 100 times or in 10 different places

```
SELECT DISTINCT ?func WHERE {
  {
    ?func rdf:type hpc:Function .
    ?func hpc:numOfCalls ?num .
    FILTER ( ?num >= 100 )
    } UNION {
      {
        SELECT (COUNT(?callsite) AS ?
            callsites) ?f WHERE {
          ?callsite rdf:type hpc:Callsite .
          ?callsite hpc:destFunc ?func .
        } GROUP BY ?func
      }
      FILTER ( ?callsites >= 10 )
    }
}
```

Table IV shows the result of the query. To retrieve functions that satisfy either of the conditions, a SPARQL `UNION` statement nests two queries and returns results if at least one subquery does. The number of calls against a function can be accessed simply by matching a triple that contains a function and a `hpc:numOfCalls` predicate and retrieving the value, and a `FILTER` statement is used to only get those with values above 100. Retrieving the number of callsites that invoke a function is done by running a subquery that gets all callsites that invoke the function, aggregating the results using the SPARQL `COUNT` function and the `GROUP BY` construct. The resulting value is passed to the parent query, and a `FILTER` construct is used again to only give functions who are involved in more than ten callsites.

3) Query for functions that are called over 100 times and are not a part of the standard libraries: Filtering results based on them being a part of the current program can help

TABLE III: Query 1 Results

?ratio	Functions	Runtime
0.1	14	127 msec
0.05	17	110 msec
0.01	68	134 msec

TABLE IV: Query 2 Results

Functions	Runtime
777	418 msec

pare down data to be more relevant to a given task and is a necessary component of providing scalable analyses. This query can be extended in different ways to identify and isolate groups of functions or callsites for more in-depth analyses.

Listing 4: Find functions called over 100 times that are not from standard library functions

```
SELECT ?function WHERE {
   ?function rdf:type hpc:Function .
   MINUS {
       ?function hpc:isPartOf ?lib .
       ?lib rdf:type hpc:StandardLibrary .
   }

   ?function hpc:numOfCalls ?num .
   FILTER (?num > 100)
}
```

This simple query utilizes the ontology explicitly in how it encodes standard libraries, such as the `hpc:isPartOf` property for being a member of a library and `hpc:StandardLibrary` for containing standard library functions. The SPARQL `MINUS` construct takes embedded queries and continues only if the internal query is not satisfied, specifically the condition that the function is a standard library function in the case of the above query. Issues with how gprof profiles software prevents this query from returning meaningful results, which is discussed in section VI.

4) Create a partial call graph from root with depth D: The SPARQL `CONSTRUCT` functionality provides the most direct support of scalability by allowing for queries to generate new graphs from pre-existing data. This can be used to integrate new analyses within a knowledge graph through a query engine or produce scaled-down call graphs that only look at given aspects of a program, which can then be used for more in-depth analysis.

Listing 5: Construct Call Subgraph from a Specified Function as the Root with Specified Depth (depth=3 or going through 3 callsites)

```
CONSTRUCT {
  ?function1 hpc:callSite ?callsite1 .
  ?callsite1 hpc:destFunc ?function2 .
  ?function2 hpc:callSite ?callsite2 .
  ?callsite2 hpc:destFunc ?function3 .
  ?function3 hpc:callSite ?callsite3 .
  ?callsite3 hpc:destFunc ?function4 .
} WHERE {
  ?function1 hpc:callSite ?callsite1 .
  ?callsite1 a hpc:Callsite .
  ?callsite1 hpc:destFunc ?function2 .
  OPTIONAL {
   ?function2 hpc:callSite ?callsite2 .
   ?callsite2 a hpc:Callsite .
   ?callsite2 hpc:destFunc ?function3 .
    OPTIONAL {
      ?function3 hpc:callSite ?callsite3 .
      ?callsite3 a hpc:Callsite .
      ?callsite3 hpc:destFunc ?function4 .
    }
  }
}
```

The SPARQL `CONSTRUCT` statement works by accepting a query on the bottom half of the statement (within the `WHERE` clause) and a general graph outline on the top. The query on the bottom returns bound variables that satisfy conditions similar to a `SELECT` query, which are then passed to the upper part of the statement to generate triples described in the statement for each result as part of a new knowledge graph. This graph generation contrasts the row results returned from a `SELECT` query.

Query 4 works by manually matching steps in the overall call graph by going downwards through each function and call site. The SPARQL construct `OPTIONAL` is used within the inner queries to allow for the possibility of a function that does not have any downstream call sites. Otherwise, functions that were not a given depth away from the root node and did not have any child calls would not be matched, resulting in an incomplete call graph. Table V shows the results of the query as the depth is increased (by nesting `OPTIONAL` statements). The number of triples (specifically edges linking together call sites and functions as they are discovered) are provided and increase as the depth does, along with the runtime of each query.

5) Generate a partial call graph showing all callpaths to a given function: This query utilizes property paths introduced in SPARQL 1.1, which allows for matching arbitrary-length paths across nodes rather than individual triples [17].

Listing 6: Construct Call Subgraph from a Specified Function Upwards For All Callpaths

```
CONSTRUCT {
 ?incomingNode ?link ?pathNode .
} WHERE {
  ?incomingNode ?link ?pathNode .
  FILTER (?link = hpc:destFunc || ?link =
      hpc:callSite)
  ?pathNode (hpc:destFunc|hpc:callSite)* ?
      targetFunction .
}
```

The above query uses arbitrary length property paths to identify all possible paths consisting of call sites and functions to a given target function, specifically by testing edges whose destination nodes must be either of the two. The discovery is ran recursively, successively identifying endpoints of these paths and reconstructing links upwards to the top of the call graph. This is achieved by first identifying two nodes that are attached by either an `hpc:destFunc` or `hpc:callSite` edge. The destination node of this edge

TABLE V: Query 4 Results

Depth	Triples	Runtime
1	22	72 msec
2	42	75 msec
3	68	114 msec
4	76	118 msec

is then tested to see whether or not there is a path containing only `hpc:destFunc` and `hpc:callSite` edges between the destination node and the intended target function, with the asterisk operator indicating the path can be of possibly zero length (allowing the target function to be within the graph). The initial triple containing the original source and destination functions is added to a knowledge graph by using a `CONSTRUCT` query. As both of the initial source and destination nodes have a path to the target function, the combination of all triples generated by these possibilities will result in a call graph that contains all possible call paths to the target function. Table VI lists attributes of the resultant graphs with target functions `ParticleVaultContainer::collapseProcessing` and `rngSample`, providing the number of functions, call sites, and overall edges in the generated graphs, alongside the running time for each query.

VI. Discussion

The HPC Ontology for call graphs provides a flexible interface for realizing complex queries over a call graph, allowing for sophisticated analysis. Specifically, these queries are written in such a way that an individual with understanding of performance analysis and basic SPARQL could reliably understand the goal of a given query. Moreover, the merged data of the dynamic and static call graphs can be exploited to create more comprehensive queries, as shown with Query 2 utilizing both callsite frequency from static analysis and call count from dynamic analysis.

Query 4 currently requires manual nesting of `OPTIONAL` statements to create a subgraph of a specified depth. Although SPARQL 1.1 introduced the property paths feature to match paths along nodes rather than individual triples, these paths are of arbitrary length (a demonstration of arbitrary length path matching via property paths can be seen in Query 5). Graph databases may provide custom SPARQL extensions to implement this functionality, such as Blazegraph providing a custom SPARQL `SERVICE`[7], but these were not explored to keep the queries implementation-agnostic.

For Query 5, there is not a strict relation between function and callsite counts as a function may invoke the same function numerous times. This causes a diamond pattern in the call graph where multiple callsites are generated by a single function, resulting in more than one callsite for a caller-callee pair.

As gprof profiles via statistical sampling, functions with smaller runtime have unpredictable timing results and are often incorrect. gprof also cannot profile functions that come from shared libraries (including standard library functions), restricting dynamic information to user-defined functions only. This renders an incomplete dynamic call graph, which can be problematic in applications that span multiple libraries. Query 3 was not able to have meaningful results due to this restriction, as `libc` is dynamically linked.

HPC software often employs parallel and distributed models within its applications. Further research into this area includes identifying software candidates that can instrument non-sequential applications by producing a consumable call graph, as well as adapting the HPC Ontology to adequately describe non-serial execution. Profilers such as totalView[8] are proprietary, while stand-alone parsers that can interface with other, more in-depth call graph formats are required to build knowledge graphs.

With the wide range of different languages used in HPC applications, support for more languages is crucial for adoption of this profiling method. Preliminary support for static call graph generation of languages other than C/C++ that can be compiled to LLVM is present, but proper generation of both static and dynamic call graphs are to be explored in a future paper. As such, completion of the profiling tools and ontology are ongoing research.

[7]https://github.com/blazegraph/database/wiki/PropertyPaths

TABLE VI: Query 5 Results

Function	Functions	Callsites	Triples	Runtime
Function A	6	8	16	137 msec
Function B	17	44	80	185 msec

VII. Related Work

SARIF (Static Analysis Results Interchange Format) is a structured JSON schema that also aims to encode output from analysis tools and provide a uniform format for storage. This format is useful for being utilized in debugging tools and reporting results to users but the JSON format used does not naturally induce a graph from the data, preventing useful queries from being made against the output. Likewise, the main focus on static analysis excludes dynamic analyses (in this case, run time call graphs), preventing proper composition and merging. Our approach defines a formal knowledge representation of call graphs using ontology. The results can be easily analyzed using standard SPARQL queries for advanced analysis.

There are some prior efforts of leveraging ontology techniques in program analyses. Yu et al. [18] converted JAVA program AST into an ontology format in order to find bugs represented using SWRL (semantic Web rule language) rules. PATO [11] is a framework used to explore declarative program analysis using Prolog programs operating on ontology triples. However, it only encodes input programs into RDF triple formats into a C programming ontology and does not define how the analysis output results should be stored. OPAL [13] is an extensible framework for ontology-based program analysis leveraging external knowledge for libraries and domains. SPARQL queries are used to implement a set of analyses including stream safety analysis and divide-by-zero analysis. However, the call graph ontology used by OPAL is a primitive and static one since it does not encode callsites nor dynamic information.

[8]https://totalview.io/

VIII. Conclusion

Interoperable and scalable program analyses are widely needed to enable a wide range of applications in performance analysis and optimizations. In this paper, we have proposed an ontology-based approach to enable interoperable and scalable call graph analysis. We defined a set of standardized terms to capture concepts and properties of both static and dynamic call graphs so different tools can collaboratively contribute to more comprehensive analysis results. Our experiments also show that ontology enables flexible queries of the call graph information using a standard query interface.

In the future, we plan to extend the HPC ontology to support more types of program analyses such as alias analysis, side effect analysis, and dependence analysis. The existing call graph ontology can also be extended to support multi-threaded programs and more programming languages. Finally, we are interested in using deep learning to enable natural language queries over RDF knowledge graphs.

Acknowledgements

This work is supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Program. Prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-CONF-838827). This material is also based upon work supported by the National Science Foundation under Grant No. 1833332 and 2015254.

References

[1] D. Callahan, A. Carle, M. W. Hall, and K. Kennedy, "Constructing the procedure call multigraph," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 483–487, 1990.

[2] A. Lakhotia, "Constructing call multigraphs using dependence graphs," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, 1993, pp. 273–284.

[3] K. Sartipi and H. Safyallah, "Dynamic knowledge extraction from software systems using sequential pattern mining," *International Journal of Software Engineering and Knowledge Engineering*, vol. 20, no. 06, pp. 761–782, 2010.

[4] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," in *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1997, pp. 108–124.

[5] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne *et al.*, "The fair guiding principles for scientific data management and stewardship," *Scientific data*, vol. 3, no. 1, pp. 1–9, 2016.

[6] S. Staab and R. Studer, *Handbook on ontologies*. Springer Science & Business Media, 2013.

[7] M. Uschold and M. Gruninger, "Ontologies: Principles, methods and applications," *The knowledge engineering review*, vol. 11, no. 2, pp. 93–136, 1996.

[8] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "Dbpedia: A nucleus for a web of open data," in *The semantic web*. Springer, 2007, pp. 722–735.

[9] P. N. Mendes, M. Jakob, and C. Bizer, "Dbpedia: A multilingual cross-domain knowledge base." in *LREC*. Citeseer, 2012, pp. 1813–1817.

[10] R. V. Guha, D. Brickley, and S. Macbeth, "Schema.org: evolution of structured data on the web," *Communications of the ACM*, vol. 59, no. 2, pp. 44–51, 2016.

[11] Y. Zhao, G. Chen, C. Liao, and X. Shen, "Towards ontology-based program analysis," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[12] C. Liao, P.-H. Lin, D. J. Quinlan, Y. Zhao, and X. Shen, "Enhancing domain specific language implementations through ontology," in *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, 2015, pp. 1–9.

[13] D. K. Pattipati, R. Nasre, and S. K. Puligundla, "Opal: An extensible framework for ontology-based program analysis," *Software: Practice and Experience*, vol. 50, no. 8, pp. 1425–1462, 2020.

[14] C. Liao, P.-H. Lin, G. Verma, T. Vanderbruggen, M. Emani, Z. Nan, and X. Shen, "Hpc ontology: Towards a unified ontology for managing training datasets and ai models for high-performance computing," in *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*. IEEE, 2021, pp. 69–80.

[15] T. Berners-Lee, R. T. Fielding, and L. Masinter, "Uniform resource identifier (uri): Generic syntax," Internet Requests for Comments, RFC Editor, STD 66, January 2005, http://www.rfc-editor.org/rfc/rfc3986.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3986.txt

[16] I. D. Leigh Dodds, *Linked Data Patterns: A pattern catalogue for modelling, publishing, and consuming Linked Data*, 2012.

[17] S. Harris, A. Seaborne, and E. Prud'hommeaux, "Sparql 1.1 query language," *W3C Recommendation*, vol. 21, 2013.

[18] L. Yu, J. Zhou, Y. Yi, P. Li, and Q. Wang, "Ontology model-based static analysis on java programs," in *2008 32nd Annual IEEE International Computer Software and Applications Conference*. IEEE, 2008, pp. 92–99.