GraphDTC: A Graph Processing System for Scalable and Precise Program Analysis

Department of Computer Science, University of California, Irvine

Abstract

In this technical report, we present a graph processing system, Graph-DTC. It is designed to efficiently compute dynamic transitive closure (DTC) on program graphs of large complex systems (e.g. Linux), which may be of the order of a billion vertices, using a single machine. DTC computation enables us to get precise software diagnostics such as bugs and data flow information. Thomas Reps first showed in his seminal work how graph reachability problems like the DTC problem could aid different program analysis techniques.

While the DTC problem has been studied for more than two decades, implementing it on large graphs has not been scalable. As a result, most existing program analysis and bug detection techniques have been deprived of detailed program information (e.g. precise aliasing information), which has limited their highest attainable precision. In order to leverage those techniques, we introduce a novel graph computation model, the edge-pair centric model, on which we build GraphDTC. GraphDTC enables those techniques to capture more detailed information about the software, and thus produce more precise diagnostics information.

1. Introduction

Static program analysis techniques such as alias/points-to analysis, flow/path-sensitive analysis, context-sensitive analysis, and object-sensitive analysis have been extensively used for revealing various diagnostic information of software programs [20]. For instance, alias analysis, a technique for determining pointers that point to the same location [2], has been used to discover the presence of certain kinds of faults in Linux and OpenBSD kernels [5, 14]. Another example is the work of Bugrara and Aiken [3] who checked for security vulnerabilities in Linux, using path- and context-sensitive analyses, two subtypes of alias analysis. They extracted unchecked user pointer dereferences in Linux to prevent the exposure of the kernel space. More recently, alias analysis have also been used to identify resource leaks [16].

Despite their success in various important applications, most of these static program analysis techniques had to grapple with one major challenge: how to balance the trade-off between (1) the precision of their outcomes (e.g. false negatives in bug detection) and (2) their efficiency.

The precision of these techniques depends on the comprehensiveness of the information used by them. For example, context-sensitive analysis uses information about the calling context of program method calls, whereas context-insensitive analysis does not [20]. Similarly, flow-sensitive analysis considers the flow of the program and thus yields separate solutions for each ordering of the program points, whereas flow-insensitive analysis ignores flow information and yields an approximated solution that holds for all orderings of program points in a program [7].

Unfortunately, greater precision comes with a cost on efficiency. As a result, the scalability of precise analysis techniques have been restricted. For instance, context-sensitive analysis based techniques have been shown to scale to hundreds of thousands of lines [6, 18], path-sensitive to tens of thousands of lines [6, 19]. Alternately, alias analyses that scale to a million lines of code have been found to give imprecise alias information with respect to the actual aliasing when the code is executed [6, 10, 12].

To address this precision-efficiency bottleneck, researchers have used program abstraction in order to keep only information that is relevant for proving properties of interest [21]. Such approaches have an overhead of computing the abstractions. Other researchers have addressed the scalability problem by simply ignoring intra-procedural analysis and by only analyzing procedures independently (e.g. [3]).

In this work, we tackle the problem head-on, by providing an infrastructure that can provide the large scale computation that precise program analysis techniques need for both intra-procedural and even inter-procedural analysis. We design a graph processing system that can be used to perform computations over massive program graphs (e.g. call graphs) which

are used by such techniques. Our inspiration comes from the insights of Thomas Reps [15].

Reps shows that program analysis problems could be transformed to graph reachability problems. In particular, he shows how program analysis problems could be reduced to a CFL (Context-Free Language)-reachability problem, which is a type of a *transitive closure (TC)* problem. The TC problem is to determine paths, or connected edges, in a directed graph. In the *CFL-reachability problem*, a path is considered to connect two nodes only if the concatenation of the labels on the edges of the path is a word in a particular context-free language [15]. In order to solve the CFL-reachability problem in a directed graph with labels, we need to iteratively add an edge each time we find such a path, such that the edge connects the end vertices of the path. Solving the CFL-reachability problem can thus be essentially reduced to computing the *dynamic transitive closure (DTC)* of a directed graph.

Our graph processing system, GraphDTC, can efficiently compute the DTC on graphs of the order of a billion edges, on a single machine. With such a system, we can facilitate the use of the aforementioned precise program analysis techniques and thus enable them to extract more precise diagnostics information. (An elaborate plan for gathering such information is given towards the end of this report.) Although there are other approaches for scaling program analysis, such as [17] and [7] our approach is significantly different and novel. We offer a systems solution to a bottleneck in software analysis, by addressing a fundamental problem in programming language design, the CFL-reachability problem.

The rest of this technical report is organized as follows: Section 2 presents the fundamental computation model of GraphDTC. Section 3 provides the design and implementation of our system. Then in Section 4, the evaluation plan for GraphDTC is presented. Finally, we conclude the report in Section 5.

2. GraphDTC Computation Model

In this section, we present the GraphDTC computation model. In 2.1, we present a novel edge-pair centric (EPC) model that can be used to address the DTC computation on a directed, labelled graph. Finally we present the edge computation model which is built on the EPC model in 2.2.

2.1 Edge-Pair Centric (EPC) Model

The *EPC model* allows us to (1) find whether there exists a transitive relation between the source and sink vertices in a path of two directed edges and (2) add an edge from the source vertex to the sink vertex if such a relation exists, while ensuring that the "new" edge does not already exist.

Fig. 1 demonstrates the use of the EPC model. It shows all the out-edges of two vertices a and b_x in an arbitrary graph. We want to find whether there exists a transitive relation from a to c_y, and add a directed edge from a to c_y if the relation does exist. (The reason why b_x's other out-edges have been dashed will be explained shortly). The bold dashed edge shows the candidate new edge. If the symbols represented by the edge values, v_{a_x}, v_{b_x} form a valid production, in the sequence shown, then a transitive relation exists between vertices a and c_y. In that case, all out-edges of a are scanned to check whether directed edge (a, c_y) already exists. If not, the edge is added.

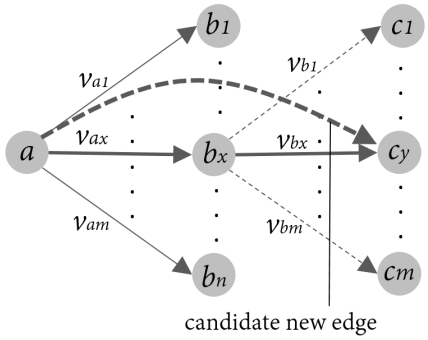

Figure 1. The Edge-Pair Centric (EPC) model.

As can be seen, to check for transitive closure between source a and sink c_y, we do not require to check out-edges of the mid-vertex b_x. These edges do not belong to the EPC model and have thus been shown as dashed.

2.2 Edge Computation Model

Fig. 2, shows the DTC computation model of GraphDTC. In preprocessing, which will be shown in 3.1, partitions from the input graph are generated by allocating vertices to intervals, creating a vertex-interval table, as shown in Fig. 2(a). The conditions for creating the partitions are also shown in the figure. Fig. 2(b) shows an example of two partitions on which DTC computation has been done in the memory. In this example, we have omitted edge values. The edges with dashed arrows are new edges generated as a result of the computation. For instance, edge $(4, 21)$ was generated as a result of the edges $(4, 1)$ and $(1, 21)$.

3. System Design and Implementation

In this section, we discuss the design and implementation of all the phases of GraphDTC. We have implemented GraphDTC in Java. The system has been written in approximately 4,000 lines of code (LOC).

Fig. 3 shows the preprocessing phase of Graph DTC. The remaining phases, and the coordination between them, are shown in Fig. 4. The mechanisms of each phase are explained in the following subsections.

3.1 Preprocessing (Partition Generation)

This phase generates partitions by scanning the input graph, which is in an edge list format. In particular, each line of the input graph file has the source vertex id, the destination

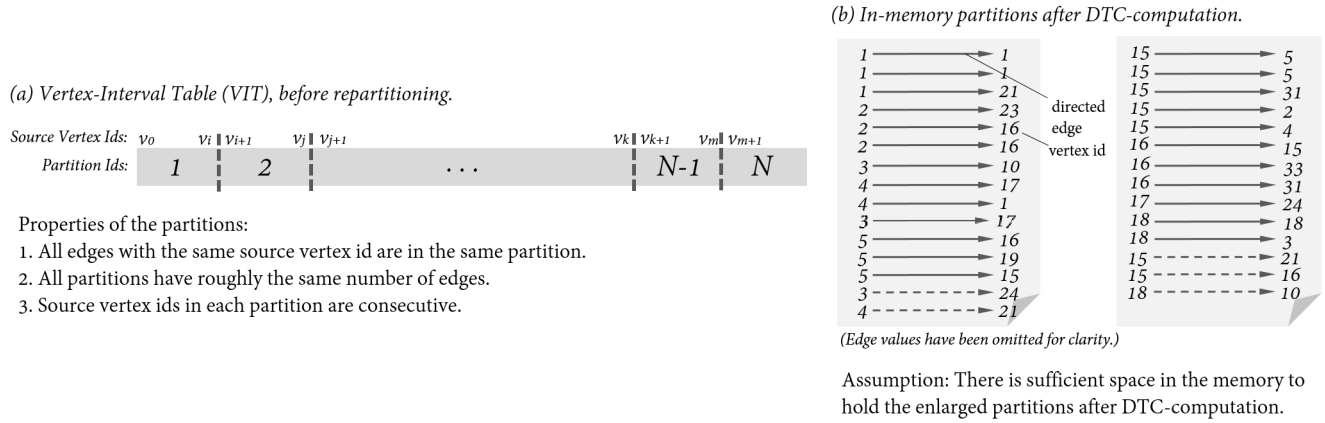

Figure 2. DTC computation model.

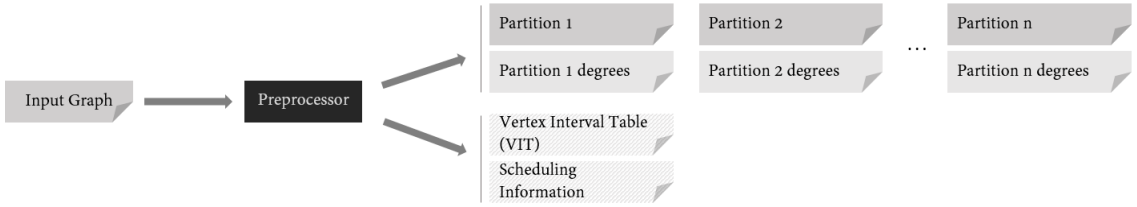

Figure 3. GraphDTC preprocessing phase.

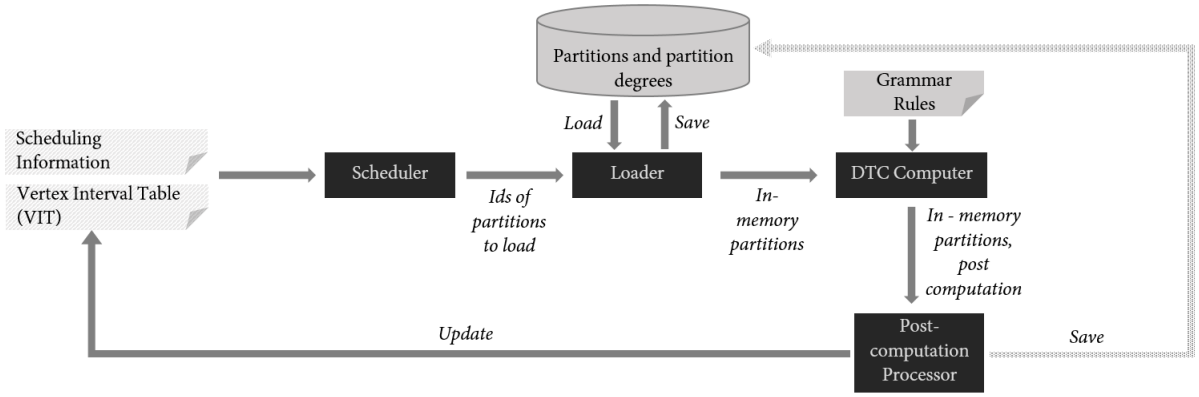

Figure 4. GraphDTC computation phases.

vertex id, and the edge value. The resultant partition file is stored in a format similar to an adjacency list, as shown in Fig. 5. All data are stored in binary in order to reduce the overall size of the partition files.

While generating each partition, the size of each partition should depend on the DRAM memory space for computation, such that at any time the memory can hold two partitions, with just enough space for other meta data structures and system resources. Following this constraint, we can ensure minimal unused memory space during each computation.

As shown in Fig. 3, other products of the preprocessing phase are (1) degrees file for each partition, which is used for setting up the array data structures of a partition when the partition is loaded in the memory, (2) the Vertex-Interval Table (VIT) which is used throughout all phases of GraphDTC to refer to partition information of edges, and (3) the scheduling information, which will be explained in Subsection 3.7.

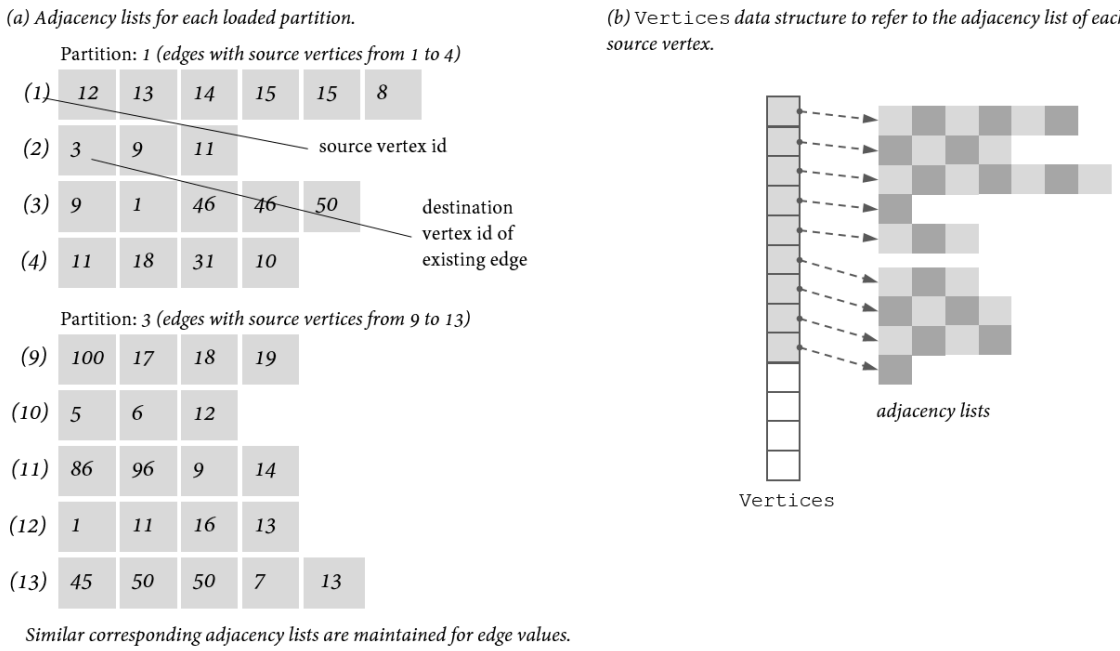

Figure 6. Loading design.

```
[Source Vertex Id]
[Count]
[Destination Vertex Id]
[Edge Value]
[Destination Vertex Id]
[Edge Value]
[Destination Vertex Id]
[Edge Value]
[Source Vertex Id]
[Count]
[Destination Vertex Id]
[Edge Value]
.
.
.
```

Figure 5. Partition data format on disk.

3.2 Loading Partitions

In order to load the partitions in the memory, we use adjacency list data structures via multi-dimensional arrays as shown in Fig. 6(a). A data structure, `Vertices`, is used to refer to the row of each loaded source vertex (Fig. 6(b)). `Vertices` will be useful later when we need to replace only partition in the memory.

Fig. 7 shows another data structure, `LoadedVertex Interval` that is set-up during loading. As can be seen, it stores the corresponding indexes of the rows of the loaded source vertices in the `Vertices` data structure. The memory consumption of `LoadedVertex Interval` is negligible as the total number of partitions is small, and also at most two partitions are loaded in the memory at any time.

```
LoadedVertexInterval {
int firstVertex;
int lastVertex;
int indexStart;
int indexEnd;
int partitionId;
boolean isNewEdgeAdded;
}
```

Figure 7. LoadedVertexInterval data structure for keeping information about partitions loaded in DRAM.

3.3 Edge Computation

Fig. 8 shows how the newly computed edges are stored in the memory. As can be seen, the new edges are stored in a linked list, where each node of the linked list is an array. In order to refer to the linked lists we have used the `NewEdge-Lists` data structure as shown in Fig. 9.

We have used this storage design for new edges in order to obviate the need to declare new memory space every time a new edge is created, which is expensive considering the number of computations we will be dealing with. Instead we declare a new node only if an existing node is filled up. The drawback of this approach is that there may be some unused slots in these nodes.

Fig. 10, shows how the actual computations are carried out. As can be seen, we have used a multi-threaded design for the computation, where new edge computations for each source vertex row are done in-parallel.

The process is carried out iteratively, wherein during each round, new edge computations are done based on the edges

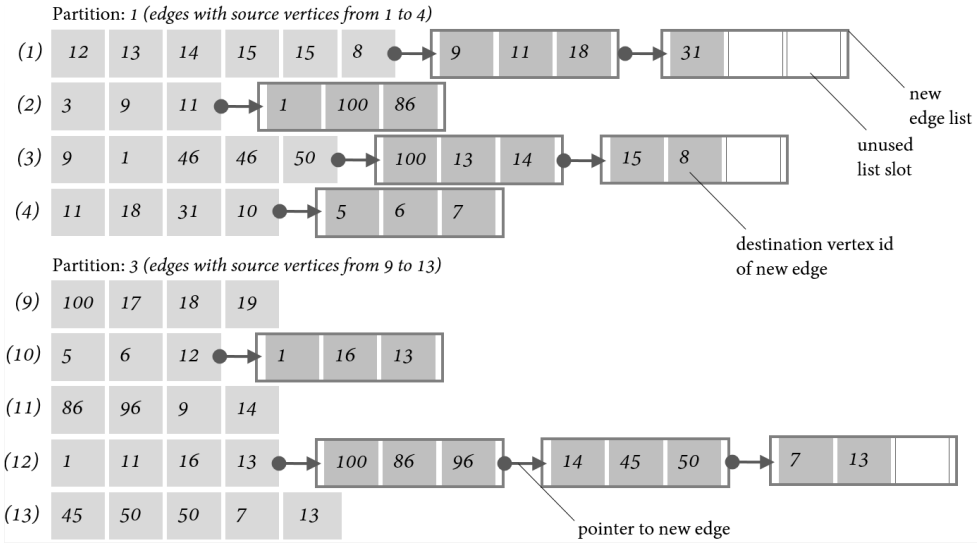

Figure 8. Adjacency lists linked to new edge array lists of fixed sizes.

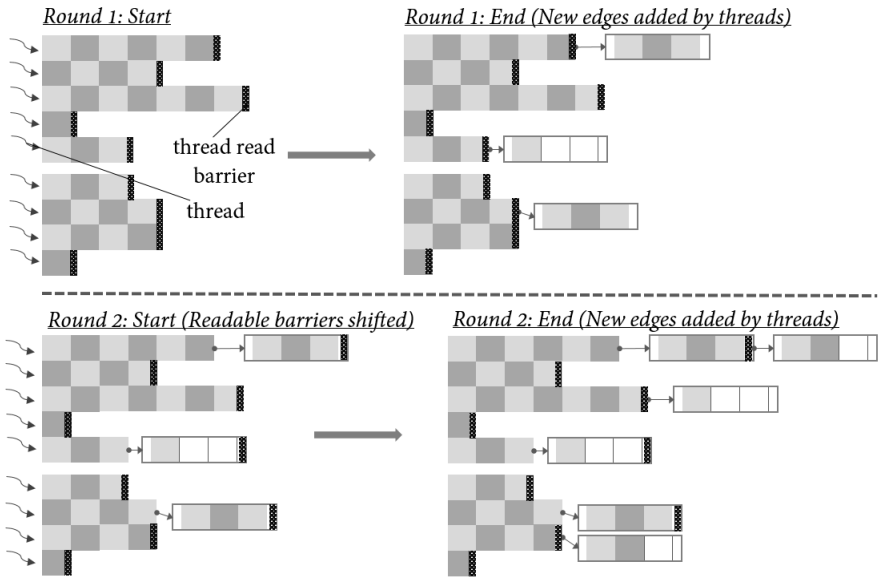

Figure 10. Multi-threaded edge computation design.

of the previous round. To facilitate this, we used readable barriers for each source vertex row.

These barriers limit the extent of a source vertex row that can be read by a thread during a round. This consequently prevents a thread from reading an array slot that is being written on by another thread, which prevents atomicity issues. In order to know when a round has ended, we used an atomic integer variable.

The atomic integer variable keeps a count of the number of active threads. Once it is 0, the computation proceeds to the next round, shifting the readable barriers ahead. The process repeats until no new edges have been generated in a particular round.

Alternate design for edge computation. An alternate design for edge computation is shown in Fig. 11. In this design, new edges are stored in a fixed-size multidimensional array, the size of which is declared at the beginning of the computation process. This approach eliminates the need to declare space for new edges completely, as it does so at the beginning. New edges are written in this array, and a table is used to record the correspondence between new edges and the loaded source vertex. The color coded cells in Fig. 11 illustrate this concept.

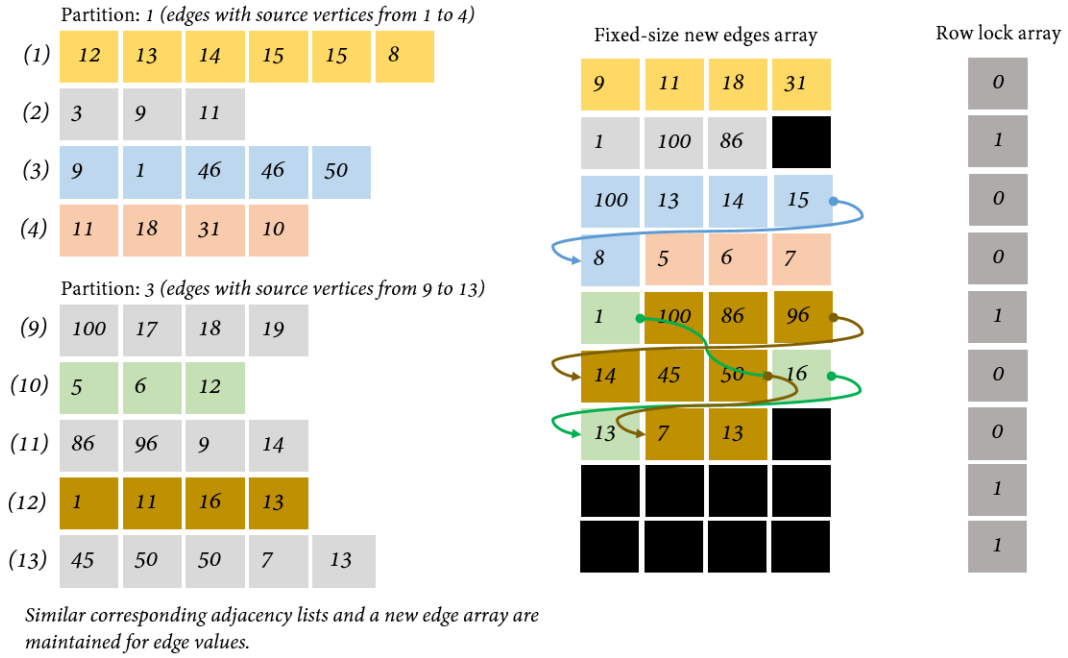

Figure 11. Alternate design for edge computation.

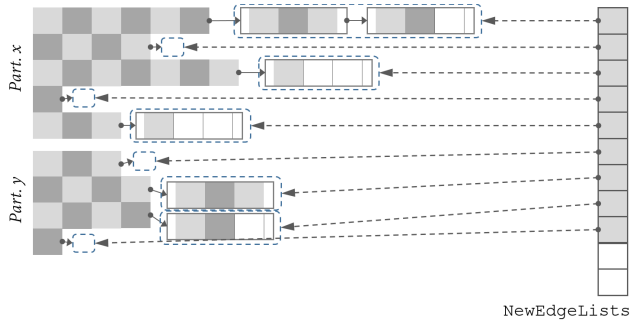

Figure 9. NewEdgeLists data structure to refer to the newly computed edge lists of each loaded source vertex.

For parallelism, we use threads such that each thread accesses a synchronized row lock array. A thread begins to write on a row only if it has acquired a lock for that row.

The problem with this technique is the maintenance of links. This is programmatically error prone as it requires consistent updating of "new edges array" indexes (first position, intermediate link positions, and last position) for each loaded source vertex.

3.4 Post Computation Processing

The post-computation processing phase entails updating the vertex interval table, the degrees information, and the scheduling information, based on the new edges added. Although the process is inexpensive, it could be expensive if repartitioning is invoked on the partitions. We explain repartitioning next.

3.5 Repartitioning

Repartitioning is a sub-phase of post-computation processing which is invoked if the sizes of any of the loaded partitions go beyond a threshold limit. (Typically it is around a small percentage larger than the partition size). Like the post-computation processing phase, the repartitioning also involves updating the same in-memory data structures. However, the recalculation of scheduling information could bear the most cost if the scheduling metric mentioned in part 3.7 is used for it would require scanning all the partitions once again. It is for this reason we want to keep the number of repartitionings in the entire execution of the graph to be minimum. For more details on scheduling cost see 3.7.

3.6 Reloading

Although the main functionality of the reloading phase is the same as that of the loading phase, which is loading partitions, reloading also requires to preserve any of the loaded partitions in the memory if the partition is among the requested set of partitions for the next computation load.

There could be a number of strategies for reloading. One such strategy is demonstrated in Fig. 12. With this strategy, partitions that have been repartitioned, and the new partitions that have been generated as a result, are saved directly in the disk and cleared from the memory, regardless of whether or not they are requested in the next round. An alternate and better strategy would be to hold all the partitions in the memory till the next load request is received from the scheduler.

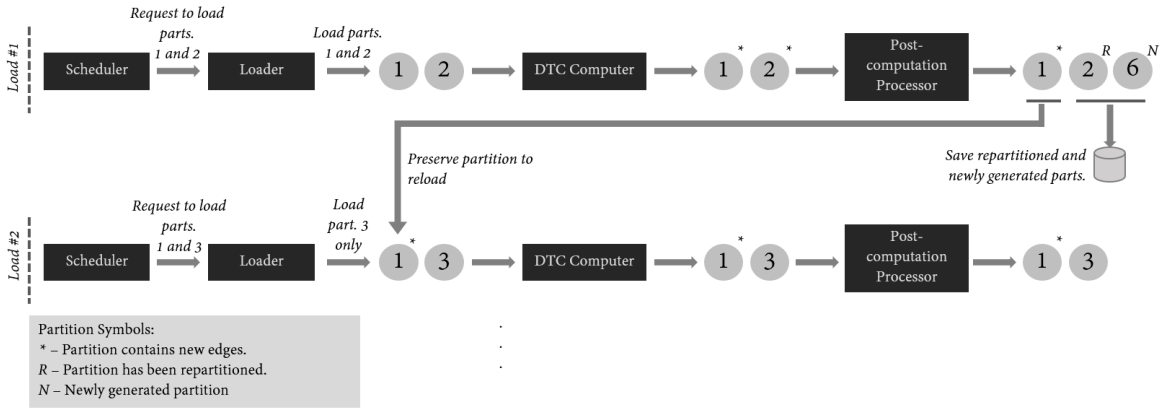

Figure 12. Example demonstrating in-memory partition management during GraphDTC execution.

3.7 Scheduling

The design of the scheduling phase of GraphDTC is summed up in Fig. 13. Scheduling in GraphDTC has two main objectives: (1) Maximize the number of edge pairs that form a path, and thus, that return a candidate new edge for checking, in a single load. (2) Favor the reuse of memory allocated for new edges during the previous load.

For objective (1), the scheduler uses a metric known as the edge-destination count ratio, X, which is calculated for all possible pairs of partitions. So say, for partitions P_a and P_b, $X_{(P_a,P_b)}$ is the fraction of edges in partition P_a that point to a vertex present in partition P_b as a source vertex. A high X value thus fulfills this objective.

The scheduler uses a map of edge-destination count ratios and favors to pick the partition pair that has the highest value, during each iteration. Calculating this map requires scanning the entire graph (which is expensive), which was done during preprocessing. In order to update this map, we only need to scan the partitions to which new edges were added. However, scanning the entire graph becomes necessary if we want to update this map after repartitioning.

For fulfilling objective (2), the scheduler only needs to know which partitions are currently loaded in the memory.

Finally, apart from fulfilling these objectives, the scheduler also keeps track of the pairs of partitions that have been computed using the termination map. The purpose of this map is to avoid loading a partition-pair that has already been computed. It is important to note that during the computation of a partition-pair, if new edges were added, the computations of the other partitions with these partitions would again need to be performed. The termination map is updated according to this principle.

The scheduler continues to request the loader to load partitions at the conclusion of each computation iteration, until the termination map shows all existing partition pairs to be computed.

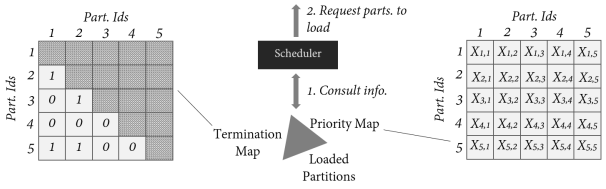

Figure 13. Scheduler design for loading two partitions.

4. Evaluation Plan

In this section, we outline our plan for evaluating GraphDTC. We expect to perform the deployment required to fulfill these plans in a workstation supplanted with an Intel i5-4570 Processor, and 8GB of RAM.

The plan will be primarily driven by three objectives: (1) investigate the performance of GraphDTC when processing very large graphs (of the order of a billion vertices) (Subsection 4.1), (2) implement DTC with an existing graph processing system GraphChi [8] and compare its results with ours (Subsection 4.2), and finally (3) obtain the level of improvement that can be obtained when GraphDTC is used to leverage existing program analysis tools in terms of the reduction in false alarm rates (Subsection 4.3).

Based on the outcomes of our evaluation, we shall also intend to locate components of our system that are amenable for optimizations.

4.1 Performance with Large Graphs

We look to stress test our system with existing large graphs: the Yahoo Web Graph (6,636,600,779 edges, 67GB disk storage size) and the Twitter Social Graph (1,468,365,182 edges, 25GB disk storage size). In practice, these graphs are not the type of graphs that our system is meant to deal with. In fact, our system targets program graphs, which are expected to be large but much smaller than these graphs. Nevertheless, as our system is agnostic of the application domain of the graphs, doing these tests allows us to see how

the system handles the pressure of performing computations on massive graphs. In addition, we expect these graphs to have varying structures. Therefore, processing them with our system allows us to verify more corner cases, and thus gain more confidence in our system's correctness.

The only configuration parameter we shall need to change in our system for running it on the aforementioned graphs is to ignore edge labels, i.e. the grammar, and thus only compute basic transitive closure.

4.2 Comparison with GraphChi

We intend to perform the DTC computation using a popular graph processing system, GraphChi [8] and compare GraphDTC's performance with GraphChi's.

Most distributed/parallel graph processing systems like Pregel [13], Apache Giraph [1], GraphLab [11], Kineograph [4], and also GraphChi, are built on the vertex centric model, where the user adopts the *think-like-a-vertex (TLV)* approach to write applications on the system. Although this approach has been useful for writing algorithms involving propagation of vertex values (e.g. page-rank algorithm), we believe such an approach will increase the complexity and difficulty for writing applications for DTC computation. In particular, the edge matching process, which is easily achievable using the EPC model (shown in 2.1), would be cumbersome to write using TLV. We also expect it to incur a significant performance overhead on GraphChi.

4.3 Leveraging Current Program Analysis Tools

Towards leveraging program analysis tools we shall primarily target Coccinelle [14]. Coccinelle is a program matching and transformation tool that has been used to find faults in the kernel, drivers, and file systems of Linux. Our goal will be to increase the number of revealed true faults, and decrease the number of revealed false alarms, in Coccinelle. In order to do that, we shall first generate a call graph of Linux 2.6's kernel using LLVM [9]. Next, we plan to transform the graph to the prescribed input format of GraphDTC and perform the DTC operation on it. The resultant graph will carry more precise information, such as calling-context information, which we believe can benefit Coccinelle.

5. Conclusion

In this technical report, we have presented a graph processing system, GraphDTC, that can efficiently compute dynamic transitive closure (DTC) on program graphs of large complex systems, using a single machine. Consequently, the system can benefit static program analysis tools, that are today extensively used for revealing various diagnostic information of software programs. The novelty of GraphDTC lies in the fact that it provides a systems solution to the precision-efficiency bottleneck of software analysis, by addressing a fundamental problem in programming language design, the CFL-reachability problem. The technical report also elaborates upon an evaluation plan for the system.

References

[1] Apache giraph. . http://giraph.apache.org.

[2] APPEL, A. W. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, UK, 1998.

[3] BUGRARA, S., AND AIKEN, A. Verifying the safety of user pointer dereferences. In *In Proceedings of the 29th IEEE Symposium on Security and Privacy* (2008).

[4] CHENG, R., HONG, J., KYROLA, A., MIAO, Y., WENG, X., WU, M., YANG, F., ZHOU, L., ZHAO, F., AND CHEN, E. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys '12, pp. 85–98.

[5] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (2001), SOSP '01, pp. 73–88.

[6] HACKETT, B., AND AIKEN, A. How is aliasing used in systems software? In *SIGSOFT '06/FSE-14 Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering* (2006), pp. 69–80.

[7] HARDEKOPF, B., AND LIN, C. Flow-sensitive pointer analysis for millions of lines of code. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on* (2011), pp. 289–298.

[8] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. Graphchi: large-scale graph computation on just a pc. In *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (2012), pp. 31–46.

[9] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (2004), CGO '04, pp. 75–.

[10] LIANG, D., PENNINGS, M., AND HARROLD, M. Evaluating the precision of static reference analysis using profiling. In *In Proc. of the International Symposium on Software Testing and Analysis* (2002), pp. 22–32.

[11] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 716–727.

[12] M. MOCK, M. DAS, C. C., AND EGGERS, S. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *In Proc. of the Workshop on Program Analysis for Software Tools and Engineering* (2001), pp. 66–72.

[13] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (2010), SIGMOD '10, pp. 135–146.

[14] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., LAWALL, J., AND MULLER, G. Faults in linux: Ten years later. In *Proceedings of the Sixteenth International Conference on Archi-*

tectural Support for Programming Languages and Operating Systems (2011), ASPLOS XVI, pp. 305–318.

[15] REPS, T. Program analysis via graph reachability. *Information and Software Technology 40*, 11-12 (1998), 701–726.

[16] SRIDHARAN, M., CHANDRA, S., DOLBY, J., FINK, S., AND YAHAV, E. *Alias Analysis for Object-Oriented Programs*, vol. 7850 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013.

[17] WEISS, C., RUBIO-GONZÁLEZ, C., AND LIBLIT, B. Database-backed program analysis for scalable error propagation. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (2015), ICSE '15, pp. 586–597.

[18] WHALEY, J., AND LAM, M. S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *SIGPLAN Not. 39*, 6 (June 2004), 131–144.

[19] WILSON, R., AND LAM, M. Effcient context-sensitive pointer analysis for c programs. In *In Proc. of the Conference on Programming Language Design and Implementation* (1995), pp. 1–12.

[20] XU, G., ROUNTEV, A., AND SRIDHARAN, M. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming* (2009), pp. 98–122.

[21] ZHANG, X., MANGAL, R., GRIGORE, R., NAIK, M., AND YANG, H. On abstraction refinement for program analyses in datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), PLDI '14, pp. 239–248.