Scalable Analysis of Large Dynamic Dependence Graphs

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Shashank Singh, B.E.

Graduate Program in Computer Science and Engineering

The Ohio State University

2015

Master’s Examination Committee:

Dr P Sadayappan, Advisor

Dr Atanas Rountev, Co-Advisor
© Copyright by

Shashank Singh

2015
Abstract

Dynamic analysis is used for analyzing different properties of the runtime execution of a program. It helps in gaining useful insights into program’s behaviour for a given execution. The work in this thesis extends an existing dynamic analysis framework which has been used to develop dynamic analysis tools e.g. a tool to identify vectorization potential of existing programs, a tool responsible for characterizing and assessing the inherent data locality properties of a given computation. This existing framework is based on construction and analysis of the dynamic dependence graph for a given execution. The size of such graphs can easily grow to have millions or billions of nodes, even for simple programs and inputs.

This thesis addresses the task of enabling scalable analysis of large dynamic dependence graphs (DDGs). We develop an out-of-core API to handle large DDGs - allowing analysis to run on machines with limited available memory. The existing analysis frameworks has the fundamental limitation that the DDG must be small enough to fit in memory. With the framework’s extension for handling out-of-core DDGs, any future or existing tools built on the framework can make use of this new API, allowing them to scale well with programs having large dynamic dependency graphs. A client of this API can also use the out-of-core functionality to manage other useful metadata (e.g., iteration vectors) needed for the analysis because; usually metadata associated with large graphs will also run into scalability issues.

To validate effectiveness and efficiency of our implementation we run simple graph algorithms on dynamic graphs and measure performance. We also re-implement one of the dynamic analysis tool, making use of the new API to verify if this new version of the tool can analyse and handle large input graphs.
Dedicated to my parents Ravindra B. Singh and Meena Singh
Acknowledgments

I would like to thank my advisors Dr P Sadayappan and Dr Atanas Rountev for having me as their advisee and for guiding me through my Masters degree. I am really grateful to them for giving me an opportunity to be a part of their research group. I learnt a lot from them, through their courses and various discussions that we had. Their energy, commitment and work ethics are big inspirations for me and I will always cherish the time spent under their guidance.

Thank you Dr Feng Qin for advising and guiding me. I really enjoyed your courses, reading group sessions and being part of your research group.

A very special thanks to Venmugil and Prashant - the two most helpful and humble people I have ever met. I will remember all our procrastination sessions, and discussions would almost never involve any research discussions. Venmugil needs another shout out, for getting me up to speed on my research topics, helping me fix weirdest of the issues and for being my design discussion peer. I also thank my friends and fellow researchers at the lab, specially Wenlei, Nandan, Qingpeng and Shridhar. It was an enjoyable experience working with all of you.

Special thanks to my friends Abhinav and Tushar for fun times and memories that we shared together. Finally, I wouldn’t be what I am today without my parents who
are my greatest support and source of inspiration. I would also take this opportunity to thank my family and friends for their encouragement and support.
Vita

October 14, 1987 ....................... Born - Lucknow, U.P., INDIA

2006-2010 ....................... B.E. Computer Science and Engineering,
Manipal Institute of Technology,
Karnataka, INDIA

2010-2013 ....................... Senior Software Engineer,
Philips Healthcare,
Philips Electronics India Ltd.,
Bangalore, INDIA

2013-present ....................... Graduate Research Associate,
Department of Computer Science Engineering,
The Ohio State University, USA

Publications

Fields of Study

Major Field: Computer Science and Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problem</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Improvements Contributed by This Work</td>
<td>2</td>
</tr>
<tr>
<td>2. Background</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Dynamic Analysis Library Overview</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1 Phase 1: Instrumentation</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2 Phase 2: Trace Generation</td>
<td>8</td>
</tr>
<tr>
<td>2.1.3 Phase 3: Trace Analysis</td>
<td>8</td>
</tr>
<tr>
<td>2.1.4 Dynamic Dependency Graph</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Computation Directed Acyclic Graph (CDAG)</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Dynamic Analysis for Data Locality Potential</td>
<td>13</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Input Configuration Parameters</td>
</tr>
<tr>
<td>4.1</td>
<td>Graph Traversal Experiments</td>
</tr>
<tr>
<td>4.2</td>
<td>Jacobi 2D : Convex Partitioning Tool (new version)</td>
</tr>
<tr>
<td>4.3</td>
<td>Jacobi 2D : Convex Partitioning Tool (previous version)</td>
</tr>
<tr>
<td>4.4</td>
<td>Single Level Heuristic : Time taken to complete</td>
</tr>
<tr>
<td>4.5</td>
<td>Iteration Vector Heuristic : Time taken to complete</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Single-sweep two-point Gauss-Seidel code.</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Induction Variables Vector</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>Extending Induction Variables Vector</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Disk-Graph Construction</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Disk-Cache</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Jacobi2D : Single Level Heuristic</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>Jacobi2D : Single Level Heuristic</td>
<td>42</td>
</tr>
<tr>
<td>4.3</td>
<td>Jacobi2D : Single Level Heuristic</td>
<td>43</td>
</tr>
<tr>
<td>4.4</td>
<td>Jacobi2D : Single Level Heuristic</td>
<td>43</td>
</tr>
<tr>
<td>4.5</td>
<td>Jacobi2D : Iteration Vector Heuristics</td>
<td>44</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

Dynamic analysis is the testing and evaluation of properties of a running program over one or more executions. Its gives a precise analysis of a program since it involves real runtime values. We can use dynamic analysis to observe runtime dependencies of a given program and analyse it to learn more about a program’s behavior. We start with an existing dynamic analysis framework [11] that has been used to develop different dynamic analysis tools ([7, 4]). It comprises of components responsible for instrumenting a given program and for analysing generated runtime traces by constructing dynamic dependency graphs.

1.1 Problem

The existing dynamic analysis framework detailed in next chapter is not designed to handle large graphs. It implements a dynamic dependency graph (Section 2.1.4) which is a streaming graph and the complete graph is never really created. The framework lacks a graph API that can handle very large dynamic dependency graphs, scaling well for million or may be billion nodes. The size of the graph that can be
handled by existing implementations, built on top of the framework, are limited to size of the available memory.

The CDAGs that are generated for SPEC benchmarks, are typically very large graphs and, depending on how a graph node is represented, could easily occupy memory in GBs. As mentioned in Section 6 of [4], a CDAG of size 120GB was generated for one of the SPEC benchmark which was then reduced to 3GB by changing the input data set. Reducing the input data set may or may not be a feasible step for every benchmark being analyzed. We try to tackle in-memory limitations of existing dynamic analysis framework by developing an out-of-core graph API.

1.2 Improvements Contributed by This Work

The overall dynamic analysis framework was developed by [7, 11], and the algorithms for measuring data locality potential, which is what we implement as a client for our API, were developed by [4]. The work done in this thesis extends the existing framework to handle large dynamic graphs and also try to implement an efficient, scalable version of the algorithms.

The Graph API developed as part of this thesis, presents an out-of-core Graph implementation that allows a client to run required analysis on any machine, with any problem size and not be restricted by size of the available memory. This allows dynamic analysis tools to run on programs producing a large execution trace (which in turn translates to large dynamic dependency graphs). This also reduces the overhead of analysing the benchmarks and figuring out a way to run them with reduced
problem sizes which may or may not involve significant effort. The API maintains a serialized disk graph which is streamed through on-demand and has an efficient implementation to handle a large number of nodes in a reasonable amount of time.

The API also provides a disk based “Cache” implementation, which, apart from being used by Disk Graph, is available to clients for handling any extra information they may need during the analysis, that otherwise would not fit in memory for large dynamic graphs. This is especially useful for analysis that require very frequent graph traversals e.g. computing neighbor information for every node in a graph.

Lastly, we re-implement Convex partitioning algorithms using the new graph API in an efficient way. We make use of the cache API to pre-compute frequently used information which greatly reduces the analysis time, in some cases bringing it down from over a day to an hour. Having an efficient implementation of these algorithms allows a client to analyze benchmarks with actual inputs and for different parameters of the heuristics which previously could never complete in a reasonable time.

In rest of this thesis we provide implementation details and evaluation results for our API. In Chapter 2 we give necessary overview of the dynamic analysis framework that our API is built on. It describes dynamic dependency graphs and their construction. We also describe Convex Partitioning algorithms that are re-implemented using our API. In Chapter 3, we describe how a disk based graph is constructed from a dynamic trace and how components of this API work together to expose necessary
interfaces for accessing this graph. In Chapter 4, we present evaluation results for our API and finally in Chapter 5 we summarize this thesis.
Chapter 2: Background

2.1 Dynamic Analysis Library Overview


1. Instrumentation phase identifies and marks interesting events or locations in the source program such as loops, basic blocks etc.

2. After getting an instrumented source program, the trace generator component is responsible for writing the trace files; outputting dynamic execution trace for the program. It’s a run time library that is linked and executed with the original program.

3. Once we have a trace file, the library can be used to read it back and for running through the program execution along with other user recorded run time events.

The tool is implemented on top of the LLVM compiler infrastructure [9]. LLVM can be used with different programming languages like C++, Fortran because of
several front ends [1, 3] that are available for compiling different languages to LLVM IR.

To summarize the whole process, a program is compiled using an appropriate front end to get a valid LLVM IR. The output LLVM IR is instrumented and linked with the runtime library to generate an executable. The resulting instrumented executable, when run, produces a trace file.

2.1.1 Phase 1: Instrumentation

An input source program is automatically instrumented using LLVM infrastructure by annotating points of interest using events. Events are represented as function calls in the instrumented program. These functions are then invoked on program execution. When called, they are responsible for writing useful information pertaining to the program state. For example, an instrumented loop would have a loop enter event before it begins and a loop exit event after it exits and these events would convey the current loop id on execution, similarly a "load" instruction would be instrumented using a load event, which on execution will be called with the memory address that was loaded. These parameters are either user defined entities like memory address or LLVM IR entities like instruction type, basic block, function etc.

LLVM IR entities are assigned ids, to uniquely identify them. Instructions, Basic Blocks, Functions and Loops are all the entities that are assigned a unique id. These ids are added to the instrumented code by storing them as a instruction metadata. LLVM provides the required infrastructure for adding metadata information to a
source program which can then be read back during the trace analysis for identifying currently executing instruction.

Following is a list of events that are instrumented in the source program:

1. **Basic Block Enter**: marks entry of a basic block. This runtime method is called with basic block id information.

2. **Load**: marks a load instruction. This runtime method is called with instruction id of the load statement and the memory address being read in during the execution.

3. **Store**: marks a store instruction. Similar to Load instruction, store runtime method is called with instruction id and the memory address being written to.

4. **Function Events**: marks function’s call, enter and return sites. The purpose of this event is to clearly marks all the statements that were executed within a function call.

5. **Loop Events**: marks a Loop Begin, Loop End, Loop Iteration Begin, Loop Iteration End and Loop Induction Variable change events. These events are called with loop id value and induction variable value for loop induction variable event.

The library allows a user to instrument just the interesting parts of the code by making use of “ddg_start_trace” and “ddg_stop_trace” methods. A user may also choose to interpret these calls differently.
Before the actual instrumentation, a few code transformations are done on the input program. These transformations are referred to as passes in LLVM terminology. For example, a loop-simplify pass attempts to canonicalize loops into standard form, and more importantly inserts a pre-header where some of the loop events are inserted.

2.1.2 Phase 2: Trace Generation

The instrumented LLVM IR, is then linked to a dynamic runtime library which records the events to a trace file on program execution.

The trace file is written in binary format and is zipped using gzip library since the file size could easily become very large. It contains a list of events that occurred during a program’s execution. Each event has a number associated with it that uniquely identifies it followed by the event parameter list. For example, on execution of a load instruction the trace file will be appended with the load event id and the memory address that is read.

2.1.3 Phase 3: Trace Analysis

Last section describes how library uses LLVM APIs to add event callback methods in the original program. The implementation of these methods are part of a runtime library, which writes them out to a trace file.

During trace analysis, the trace is read back; which essentially implies going through the program execution (basic block and function call events) and convey interesting events as they happen during the execution. A library client can extend
these callback interfaces exposed to handle specific events of interest. For example, we can extend a method to handle loop enter and exit event in the trace file and use the parameters to maintain meaningful information for, say, calculating the loop depth. The instrumented IR file is used to read the metadata information and for identifying different dynamic instances of an instruction.

2.1.4 Dynamic Dependency Graph

A dynamic dependency graph is an abstract way to represent program’s execution. To construct such a graph we can use APIs exposed by the trace analysis component by tracking all dynamic dependencies between instructions. While the trace is being analysed, every instruction is represented as a node which encapsulates all the needed metadata for that instruction. The purpose of a dynamic dependency graph is to provide necessary information about an instruction like predecessors of the instruction (statements on which the execution of this instruction depends), and/or the type and/or the address associated with this instruction.

The predecessor information for a instruction is evaluated by tracking dependencies via virtual registers or through actual memory addresses between instructions. For every virtual register and memory address, the library tracks the last writer information. Any dynamic instance of an instruction reading from an address/register will have the last writer of that address/register as its predecessor.

Note that the graph only represents flow dependencies. Anti-dependence and output dependence are not considered, since they do not represent essential constraints
of the computation, and could potentially be eliminated via transformations such as scalar/array expansion. Control dependencies are also not considered, since the focus was on data flow and optimization potential implied by it.

DDG’s are constructed in a "lazy" fashion; which implies that a whole DDG graph is never held in memory. Thus from a client perspective only a window of dependency graph is available. During trace analysis, for each instruction a graph node is created. This graph node along with its predecessor information is streamed to clients. The predecessor information for a given node can be tracked by maintaining a table of last writer(instructions) for virtual registers and memory addresses and then use this table to look up last writer of its operands. A newly created node is then inserted in the table for the virtual register it writes to. At this point, the previous last writer for this address is overwritten and that information is lost. Thus the library never creates a whole graph in memory.

A client of the library, can extend Lazy Graph’s callback methods to construct a complete in-memory ddg. visitNode(), exposed by Lazy Graph is called for every dynamic instance found in the trace. Node’s information, node’s predecessors and the operands for this instruction are arguments of visitNode(). Thus a client can extend this method to get access to relevant information for a dynamic node and construct a complete in-memory dynamic dependency graph.

As a client of this framework, for Disk-Graph construction, we would extend the visitNode() callback method exposed by lazy dynamic dependence graph API. But
instead of maintaining an in-memory dynamic dependence graph we serialize node’s information to a file for creating a disk based dynamic dependency graph. Further, the type information associated with every dynamic instance can be utilized for identifying load and store nodes or computational nodes that is needed for building a CDAG (section 2.2).

2.2 Computation Directed Acyclic Graph (CDAG)

To model the range of valid scheduling orders for the operations of an algorithm, it is common to use a schedule-invariant abstraction referred to as the computational directed acyclic graph (CDAG), with a vertex for each instance of a computational operation, and edges from producer instances to consumer instances. Fig. 2.1(c) shows the CDAG for the codes in Fig. 2.1(a) and Fig. 2.1(b), for $N=6$. Although the relative order of operations is different between the tiled and untiled versions, the set of computation instances and the producer-consumer relationships for the flow of data are exactly the same (special “input” vertices in the CDAG, shown in black, represent values of elements of $A$ that are read before they are written in the nested loop).

More formally we define a CDAG as follows:

**Definition 1 (CDAG-HK [8])** A computational directed acyclic graph is a 4-tuple $C = (I, V, E, O)$ of finite sets such that: (1) $I \subset V$ is the input set and all its vertices have no incoming edges; (2) $E \subseteq V \times V$ is the set of edges; (3) $G = (V, E)$ is a
for(i=1; i<N-1; i++)
  for(j=1; j<N-1; j++)
(a) Untiled code

for(it=1; it<N-1; it+=T)
  for(jt=1; jt<N-1; jt+=T)
    for(i=it; i<min(it+T,N-1); i++)
      for(j=jt; j<min(jt+T,N-1); j++)
(b) Equivalent tiled code

cd dag

Figure 2.1: Single-sweep two-point Gauss-Seidel code.

directed acyclic graph; (4) $V \setminus I$ is called the operation set and all its vertices have one or more incoming edges; (5) $O \subseteq V$ is called the output set.

Like DDG, a CDAG captures instruction and their dependencies. But CDAGs are very different from DDGs. Unlike DDGs, they only have nodes for representing computations and input nodes that are read in before performing those computations. DDGs on the other hand would also have nodes for store instructions, loop iteration variables etc. A CDAG construction thus requires tracking of transitive dependencies between dynamic instances in the execution trace whereas a DDG is generated "as-is" from a given trace.
2.3 Dynamic Analysis for Data Locality Potential

To test the effectiveness of Disk-Graph API we re-implement algorithms proposed for measuring data locality potential of a given program using dynamic analysis, as introduced by Fauzia et al.[4]. Their work uses reuse distance analysis metric that models data locality [10, 2]. The reuse distance of a reference in a memory address trace is defined as the number of distinct memory references between two successive references to the same location.

The reuse distance analysis (RDA) for a given trace, gives data locality characterization for one execution order. Whereas [4] shows that there exists other valid execution orders that may help in achieving better data locality. The proposed heuristics tries to form a dynamic data dependence graph and re-order the operations by forming convex partitions. Then the final step is to evaluate if the generated reordering of the instructions results in an improved RDA profile.

A reuse distance profile generated from these algorithms is a result of performing a Convex Partitioning of a CDAG such that vertices within a partition are executed in the same relative order as the original execution. Convex partitioning attempts to create tiles by grouping points, in the iteration space of a loop, into smaller blocks such that it results in a better (reduced) reuse distance profile. A more formal definition of Convex Partitions for a graph is presented in [4]. While applying partitioning algorithms, on a given CDAG, the key requirement for adding a new vertex to a convex component is, that, if any path to that vertex exists from a vertex
in the component then all vertices in that path must also be included. Moreover, each vertex is assigned to a single convex component resulting in a acyclic convex components graph. Therefore ensuring that components can be executed in any topologically sorted order, and executing all the convex components results in a execution trace for the full computation. Basically, convex partitioning heuristics tries to perform tiling where the tiles can be of any arbitrary shape but convex and the final schedule must preserve the program dependence structure.

We re-implement two partitioning heuristics : Single Level Reuse Heuristics and Loop Induction variable Heuristics but using the new graph API. The new API now allows these algorithms to scale very well with the number of nodes in a input graph. With new implementation we can run convex partitioning algorithms on a graph with a million nodes, whereas the previous implementation could only handle small problem sizes (producing a graph with some hundred thousand nodes). We briefly explain these two heuristics in following subsections. The dynamic analysis for performing convex partitioning involves the following steps:

- Instrument and run the input source program to generate a execution trace
- Generate a CDAG from the execution trace.
- Write the original memory trace of the program using the generated CDAG and run reuse distance analysis for it.
• Perform Convex Partitioning using one of the heuristics to generate a valid schedule of operations and write a memory trace from this reordered schedule.

• Run standard reuse distance analysis on the reordered memory trace.

Chapter 3 give more details on implementation of these heuristics.

2.3.1 Convex Partitioning Single Level heuristic

This heuristic defines a notion of \textit{maxlive} which represents the maximum number of simultaneously live nodes for this schedule. A node can be either in \textit{Initial state} (not live), \textit{birth state} - right after its executed/fired, \textit{Resurrected state} - when not yet added to a convex component but used by another node of a component or \textit{death} - a dead node is a node right after its last successor is executed. The goal of this heuristic is to generate a scheduling such that \textit{maxlive} of each component doesn’t exceed the local memory capacity.

Different algorithms are pieced together to generate valid schedules using this heuristic that are detailed in the paper [4]. A convex component is grown in size, adding new nodes based on a user specified priority until the maxlive value for a component is reached. A node can only be added to a component if any path to that node exists from a node in the component, then all vertices in that path must also be included. This condition is met by constraining added nodes to be those that have all predecessors either in current or some previous convex component.
A node is considered *ready* to be added in a component if all its predecessors are fired. From this set of ready nodes, we pick the next best node by choosing a ready neighbor or a ready successor for the last executed node. Favoring nodes of the ready successor list would favor growing in depth-first in the CDAG, and choosing a neighbor favors growing in breadth-first strategy. If the heuristic is run with equal number of neighbors and successors then its called equal-priority, if more neighbors then breadth-priority or depth-priority otherwise.

### 2.3.2 Convex Partitioning Using Loop Induction Variable

A loop induction variable is defined as a variable that is incremented or decremented by a constant value in a loop. With this heuristics, loop induction variables are used to generate convex partitions. From a given trace we generate a CDAG, in which now every node also has a loop induction variables vector created. For a statement at loop depth $d$ there will be $2d+1$ induction variables in loop iteration vector. Figure 2.2 shows a example of how induction variable vectors for statements are generated. The example has three nested loops $L_i$, $L_j$ and $L_k$ and thus a perfectly nested statement will have a 7 element induction variable vector.

The aim is to form tiles to allow reuse in multiple directions and such that each tile can be executed atomically. While growing partitions, the heuristics try to achieve components close to size $T^d$ where $T$ is tile size which is a input parameter to the algorithm and $d$ is the maximum loop depth for the instrumented code. It should be noted for easy comparison of the iteration vector, there is pre-processing.
step involved which normalizes the iteration vectors for the maximum loop depth in a program by extending the vectors and filling them with trailing zeros. This step does not change the information conveyed by the original induction variable vector because 0 implies no execution of that loop for this statement. Figure 2.3 presents an extended version of these vectors for the previous example.

Similar to Single Level heuristic, the algorithm for this heuristic grows by adding nodes to a convex component. A ready node is picked up and a tile of size $T^d$ is created by gathering all the nodes that fall within a valid range. This process can add nodes that are not yet ready (i.e. have unprocessed predecessors), and thus to maintain convex component property, the algorithm adds that does not belong to
the partition yet but has a path to one or more nodes in convex component. Thus, the partitioning algorithm starting from current nodes in the partition grows as long as there are no unprocessed node that is a direct or indirect predecessor of current nodes in the partition.
Chapter 3: Implementation

This chapter describes design and implementation details of the Graph API built on top of the dynamic analysis library described in Chapter 2.

3.1 Overview

Disk-Graph API has two major components:

- The first component builds upon the existing dynamic library’s graph builder. It’s responsible for handling each dynamic instance of instrumented statements and incrementally build a graph out of it, which is then written to a file as a “disk-graph”.

- The second component is Disk Cache which is responsible for reading through the generated disk-graph file - reading graph nodes in specified sizes. It consists of slots that can hold a certain amount of data - both these values are user configurable. A graph node is cached in on-demand and remains in the cache until evicted (depending on the user specified eviction policy e.g. Least Recently Used(LRU), Most Recently Used (MRU) etc.).
3.1.1 Input Configuration File

The API reads an input configuration file with user specified options. Table 3.1 lists different input parameters read by the API.

3.2 Disk-Graph

The API is capable of building both DDGs and CDAGs. Building these graphs from a generated trace can be summarized as a three step process:

1. Make a first pass over the generated trace to get some insight into the graph we are trying to construct: Number of nodes and successor count for every node, and address associated with each node.

2. Make another pass over the generated trace to write a temporary graph with Node Id, Static Id, Node Type and Predecessor List. Also, simultaneously write successor information for every node in another temporary file.

3. Last step is to write a final disk-graph i.e. merge temporary graph with temporary successor information and address information. Once a final graph is created, the temporary files used during the process are deleted (or are not deleted depending on the input configuration flag value).

Thus, as described, a final disk graph is obtained by making two passes over a generated trace. First pass is made for pre-calculating some useful counts for the graph and a second pass writes all the graph nodes to the file. This section further
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM_SLOTS</td>
<td>Number of slots in the Disk Cache</td>
<td>256</td>
</tr>
<tr>
<td>BLOCK_SIZE</td>
<td>The size of data (in KB) that should be read in or written from a slot</td>
<td>4</td>
</tr>
<tr>
<td>CREATE_GRAPH</td>
<td>A flag marking whether a disk-graph should be created by parsing the trace file or it should be read from an already created disk-graph file. If marked as 1(True) the graph is created from the trace file. If marked as 0(False), then the value specified for the parameter DISK_GRAPH_FN is the disk-graph file name.</td>
<td>1</td>
</tr>
<tr>
<td>DISK_GRAPH_FN</td>
<td>If CREATE_GRAPH flag was set then the string value of this parameter is the suffix for the disk-graph file name, else it is the complete file name for the already created disk-graph file.</td>
<td>&quot;diskgraph&quot;</td>
</tr>
<tr>
<td>PRINT_GRAPH.ASCII</td>
<td>A flag specifying if the user wants to print a human readable version of the graph. If set to 1(True), a file named &quot;graphInAscii.txt&quot; will be written to the current directory.</td>
<td>0</td>
</tr>
<tr>
<td>PRINT_GRAPH.DOT</td>
<td>A flag specifying if graph should be written in .dot format. If set to 1(True), a file named &quot;diskgraph.dot&quot; is written to the current directory.</td>
<td>0</td>
</tr>
<tr>
<td>CLEAN_UP_TEMPFILES</td>
<td>A flag specifying if all the temporary files used for writing the disk graph should be deleted. If set to 1(True) all the temporary files will be cleaned up.</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.1: Input Configuration Parameters
describes graph node representation, provide details on why we choose the three step approach, describes trace passes and lists other optional graph information like iteration vectors for nodes, that can also be generated if needed.

### 3.2.1 Graph Representation

One of the problem that is synonymous with scalability is available space (memory and disk). We had to choose a way to minimize the amount of data needed for representing a graph. Since we are implementing a graph API our default representation should allow a client to represent any graph and not specifically cater to either dense or sparse graphs. Although, with dynamic/static analysis we almost never expect to encounter a dense graph, such that every node (which represents dynamic/static instance of statement) is connected to many other nodes in the graph; the API provides flexibility of representing a sparse/dense graph by choosing a custom Graph Node representation.

The two standard representations are Adjacency List and Adjacency Matrix representation. The API by default chooses Adjacency List representation where a typical graph node has following attributes:

- **Node ID**: stores the global (dynamic) id of the node
- **Node Address**: stores the address for the node. For a LOAD node and STORE node, these values correspond to the actual memory addresses read from or written to, while for other nodes these are same as Static ID of the node
• Node Type: identifies the LLVM type associated with the node e.g. LOAD, STORE etc.

• Static ID: static id for the node which corresponds to the statement id in the source program.

• Predecessors List: list storing all predecessors of the node

• Successors List: list storing all successors of the node

A client can also extend the existing graph node to represent a graph using adjacency matrix representation. The utility methods are already added to the library to represent each node in the graph by a bit. Thus a byte can be used to represent 8 nodes. This probably is the most compact data structure we can have, that can be used to maintaining predecessors and successors list for a node of a dense graph. With this representation one row of the adjacency matrix for a graph with 1 million nodes will occupy about 122 KB in memory.

3.2.2 Graph Construction

For a given node we need both its successors list and predecessors list. The trace analysis module after processing the current dynamic instance can provide a list of its predecessor statements and we make one more pass over the trace file to get successor information for a node. The first pass calculates the number of nodes in the graph, successor count and the node address information for a node, while second
pass writes a temporary graph with predecessors and also a file having successor information for all the nodes. This graph creation process is depicted in Figure 3.1

![Diagram of Disk-Graph Construction]

**Figure 3.1: Disk-Graph Construction**

Some design choices that were considered before choosing the two pass scheme are listed below:
• **Why do we need successor information for nodes when we can construct successors list on-the-fly by using predecessor’s list of node?**

Almost all analysis performed on graphs require successor information for every node in the graph. Calculating successor information using predecessor’s list of nodes (and for different nodes) will lead to an in-efficient and slow analysis of the graph where a good chunk of time is spent in extracting needed information from the disk-graph.

• **Why to write successor information to a separate file first and then merge it with the actual graph? Why not update the graph file directly?** Simply because there is no easy way to perform update in the middle of a file. The only possible way of achieving that is to seek to the correct file location, extract remaining file data, perform the update and re-write the extracted data back. This will lead to an inefficient implementation and will also increase chances of a file corruption. It’s a good practice to write a new file or append contents at the end, rather than editing a file from middle.

• **Why don’t we preallocate ”some” space for successors in the graph file for all the nodes of a graph, which would allow an easy successor update?** Because there is no tight upper bound on successor count, unlike predecessor count, that will allow us to preallocate some space for every node of a graph. An extremely weak upper bound is the total number of nodes in a graph, but this is not a good idea especially when we have a million nodes.
• Why is successor count calculated in first pass when we can directly write the successor information in that pass? This could have saved an extra pass? Pre-calculating successor count for nodes allows us to pre-allocate the required space in the temporary successor file which will allow extremely efficient and easy updates to the file, without any issues of file corruption.

• Why do we need need a temporary successor file when we can simply preallocate space in disk-graph file and update it? Especially since we calculate successor count in the first pass? This was done for improving efficiency. A disk-graph file can be huge, could easily be double in size than the temporary successor file, say its in GBs and seeking around in the file for that size was not resulting in an efficient implementation. Imagine a case where the 50 millionth node is a successor of the node numbered ten thousand, and thus now we need to update the successor information for this node. Seeking in temporary successor file will be way more efficient than seeking in the disk-graph file, especially for scale of 10 million graph nodes or more.

At the end of a graph construction process, the API will write a disk graph file in binary format and initialize Disk-Cache (detailed in following section) which is responsible for reading the graph file to fetch needed information.
3.2.3 Other Graph Information

The Disk-Graph implementation can easily be extended to store extra information in graph nodes. For instance, Convex Partitioning’s loop iteration heuristics requires iteration vectors for all graph nodes. This information can be obtained by extending dynamic analysis framework’s graph builder and handling callbacks for loop iteration methods and then extending the default Graph Node class to a custom graph node class that can store the iteration vectors.

3.3 Disk-Cache

There are a lot of I/O operations being performed by this API. The I/O is typically bounded by the disk transfer rate and by location of data on the physical disk. Once a disk-graph file is written, seeking around in the file to fetch information, on demand, will be naturally slow and costly, especially when compared to an in-memory graph. The dynamic analysis being performed on these graphs are inherently slow and considering easy scalability we needed a cache mechanism for reading contiguous chunks of disk content and cache it in memory for an increased performance.

Disk-Cache provides an in-memory cache implementation for allowing fast retrieval of information from a disk file. It comprises of at least one or more SLOTs. Each slot reads-in and writes-out data in BLOCK SIZE units. A slot is evicted (invalidated) depending on a eviction policy. By default, the API uses Least Recently
Used (LRU) policy for cache management, but it also provides an option of using a user-defined policy.

Disk-Cache also supports a “write-back” feature i.e. if a disk-cache is initialized with a “write-back” enabled flag then it will flush out contents of a slot to the file before evicting it. But this feature has one important caveat associated with it: the cache should be interacting with a file that has pre-allocated space for the data i.e. the modified data should be of the same size as the data that was read in. This feature is still in its preliminary stage and for now write-back, if enabled, happens for every slot irrespective of whether the data in the slot was modified or not. Thus it may be a costly feature to enable especially when dealing with memory-intensive analysis.

The initialization of Disk-Cache starts by preallocating various cache data structures. For pre-allocation we evaluate maximum number of data instances that can be held in a cache slot, which may be well over the actual amount of data being read in. But this a trade-off that we are willing to make to gain some good performance because pre-allocation results in an improved data locality. This is followed by a data indexing step where the input data file is scanned through and divided in different blocks of user-specified size. Figure 3.2 depicts the Disk-Cache and the data structures involved in building it. As seen from the figure, the Disk-Cache is using typical cache data structures: a doubly linked list to maintain least recently and most recently used slots used for implementing a cache eviction policy, a map
for mapping slots to the data it contains and also for maintaining a pointer to the eviction list.

![Disk-Cache Diagram](image-url)

Figure 3.2: Disk-Cache

It is important to note that Disk-Graph and Disk-Cache are two separate components that happen to interface with one another i.e. Disk-Graph is using Disk-Cache to gain some performance. A Disk-Cache can interface with any ‘Data’ class that
defines and implements a set of required methods. It provides a generic and well
abstracted implementation which can index and read through a disk file that don’t
break three implicit assumptions that Disk-Cache currently operates on: 1) Each
data item stored in cache has a unique positive id associated with it, 2) Data items are
in a contiguous range, 3) Data items are stored in a sorted order (increasing/decreasing).
The last assumption is actually a design decision we made (plus was something
that happens for dynamic dependence graphs at no extra cost), to have an efficient
look-up implementation, and can be easily changed to accommodate more clients, if
needed.

A disk-cache client can simply “setup” a Disk-Cache for files that doesn’t break
the design assumptions, and implementation for such data instances should provide
an interface for Disk Cache to read through the file for which it is setup. Following is
a list of methods that should be implemented by a class interfacing with Disk-Cache:

- **getId:** Fetches the unique ID associated with this particular instance of Data.

- **readNodeFromFile:** Provides the implementation for reading the contents
  into a Data instance.

- **writeNodeToFile:** Defines and implements the format in which this instance
  can be serialized to a file.

Enabling a generic Disk-Cache implementation allows us to read any file that
can be indexed (follows disk-cache design assumptions). For instance, the Convex
Partitioning single level heuristics uses a ready neighbor list to pick next ready node for adding to a partition. Now, its pretty straight forward to evaluate neighbors for a node given a graph. But in this case, we needed to optimize the heuristics, to allow it to complete in reasonable time. Thus the neighbor info for nodes was written to a file and we could use disk-cache to retrieve it when needed. The advantage of pre-computing neighbor list for every node was very evident with the analysis time reducing from over 24 hours to an hour in case of matrix multiplication with problem size of 100.

3.4 API methods

This section lists API methods that are exposed by Disk-Graph API. Also, Appendix A lists out a simple example usage of this API. The example implements a topological sort algorithm.

The different methods exposed by this API are as follows:

1. **generateGraph**: Generates the actual CDAG or DDG by performing a trace traversal.

2. **getNumNodes**: Gets count of nodes for the generated graph.

3. **getNode**: Returns a pointer to the GraphNode data structure for a specified node id value.

4. **getMaxLoopDepth**: Returns a maximum loop depth value for a graph built with iteration vectors.
5. **getNumSlots**: Gets number of slots with which Disk-Cache for the graph is initialized with.

6. **getBlockSize**: Gets block size with which Disk-Cache is initialized with.

7. **initNeighborCache**: Prepares neighbor information for all nodes in the graph.

8. **getNeighbor**: Fetches neighbor information if graph was initialized to create a neighbor cache otherwise returns a null pointer.

9. **printDiskGraph**: Writes graph in human readable form to a specified stream.

### 3.5 Convex Partitioning Algorithms

We re-implement Convex Partitioning algorithms as a client of Disk-Graph API. More specifically we implement two heuristics of Convex Partitioning: Single Level Partitioning and Partitioning using Loop Induction variables.

One of the optimization that we make using this API involved pre-computing neighbor information for all the nodes before we start the partitioning algorithm. This gave a big boost in performance because finding neighbors for a given node on fly requires reading in all its predecessors into the Disk-Cache. The predecessors for any node, may or may not have good locality in Disk-Cache and thus may cause slot evictions, resulting in a degraded performance. In the context of the algorithm, neighbor information for nodes is needed every time the next best node is selected by the heuristic and thus is a very frequent step of partitioning algorithm. Avoiding
reading in of extra nodes in disk-cache will result in a reduced I/O cost, especially in this case because the required information can be pre-calculated for the analysis. In fact, neighbor information for any node could be something that quite a few clients need and thus maintaining neighbor information for a graph is implemented as a disk-graph optional feature: its not computed by default and would need to be explicitly initialized by a client.

Another optimization that was made for both the heuristics was to maintain bit sets for marking and unmarking nodes during the analysis. Both analysis use node markers to check if a node is ready or not, or whether a node is processed (added to some convex partition) or not. A typical implementation style is to create a boolean array or a dictionary to mark/unmark nodes. But to allow scalability, Disk-Graph API also provide bit sets utility methods. Since ids of graph nodes for dynamic dependency graphs are sequential in a range, we can use bits to represent each node and a bit vector as a marker for all the nodes. Thus a complete bit set initialized with number of nodes in the graph can be used for this purpose. Although, this won’t give a significant performance boost if we are dealing with graphs having a few million nodes in which case the number of marker sets we need for the analysis are very few. But creating a boolean array for a graph with billion nodes would occupy close to 1 GB of memory (assuming 1 byte for boolean), and for such cases using a bit set implementation will occupy 0.1 MB of memory avoiding scalability issues for any analysis.
Chapter 4: Evaluation

To illustrate the usage of this API, SPEC CPU 2006 benchmark suite’s floating point benchmark, the SPEC CFP 2006 [6], was chosen.

This chapter has three sections. First section, details the setup used for running experiments. Second section presents results of applying simple graph algorithms on SPEC benchmarks, using the disk graph API. While the last section compares scalability of new Convex Partitioning implementation with the existing implementation, using Single level and Iteration Vector heuristics on Jacobi 2D.

4.1 Experimental Setup

The dynamic analysis library is built using LLVM version 3.1 [9] and GCC compiler version 4.6.1 [5]. The dynamic analysis library is now responsible for instrumenting the source program and generating traces. This was developed for performing dynamic analysis to assess vectorization potential in applications [Holewinski et al. 2012]. The convex partitioning algorithms developed by Fauzia et al. 2014, were implemented using the new Graph API. Finally, for the reuse distance analysis of
the reordered address trace generated after Convex Partitioning, we make use of paralleled reuse distance analyzer PARDA developed by Niu et al. 2012.

All experiments were performed on an Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz, using a single core. The machine has 16GB of memory.

4.2 Graph Traversal Experiments

The aim of these experiments is to apply simple graph algorithms on the CDAG’s making use of the graph API, and collect the time taken for these algorithms to finish. Typically, dynamic analysis is very time consuming, and performing graph traversals experiments will give an insight on APIs performance. For example, if a single pass over the graph is taking a lot of time then we can be sure that it will only get worse for any analysis we perform on the graph. Using the traversal tool, a user of the API can also try and find an appropriate values for Disk Graph configuration parameters before moving on to the actual analysis.

Namely we perform Breadth First Search (BFS), Depth First Traversal (DFS), Topological Sort (With Stack) and Topological Sort (With Queue) on the CDAGs generated for some randomly chosen benchmarks. These benchmarks were profiled to determine “hot” loops. The benchmarks were compiled using Intel icc compiler, with optimization level set to -O3. Hpctoolkit (version 5.2.1) was used to profile the cycles spent and the percentage of floating point operations.

We choose one of the “hot” loops to instrument and generate a CDAG for it. We could have chosen to instrument the whole program, which in case SPEC benchmarks
would have resulted in huge trace files and very large CDAG’s (with billion of nodes) but typically we would never perform dynamic analysis for the whole program or on a graph with billion nodes, plus performing traversal experiments on such graphs would have taken a lot of time with the amount of memory that was available for running these experiments. Although, it should be noted that the tool, with appropriate values of NUM_SLOTS and BLOCK_SIZE, can easily handle graphs with billion nodes. For these reasons, some of the benchmarks that were generating billion or more nodes for a CDAG were not used for these experiments.

Table 4.1 presents results obtained from running graph algorithms on some of the SPEC benchmarks. First column, Loop, lists out benchmark names and the source code file where the instrumented loop exists. Second column has disk cache configuration parameter values. The Disk Cache configuration parameters NUM_SLOTS and BLOCK_SIZE cannot have same values for every CDAG we analyze. The block size is typically determined by the maximum node size for a graph i.e. a slot should be able to hold at least one node. For e.g. in case of 410.bwaves, the maximum node size was 18 MB and hence we assigned a block size of 20 MB, whereas 433.milc worked well with 6 MB block size. Thus there is no single setting of these parameters that will work for all the benchmarks or CDAGs that are analyzed. Choosing an appropriate setting for a benchmark will definitely help with better analysis run times and are mostly dependent on graph properties of the CDAGs (e.g., low fan-out vs. high fan-out). Third column, lists the number of nodes in the generated
CDAG. Rest of the columns lists time taken (in minutes) for graph creation, BFS, DFS, Topological Sort (With Queue)(TS(WQ)) and Topological Sort (With Stack) (TS (WS)) respectively. Last column lists the peak memory usage for the program execution which is calculated using `/usr/bin/time` (version GNU time 1.7) program that comes pre-packaged with Linux distributions.
Table 4.1: Graph Traversal Experiments

<table>
<thead>
<tr>
<th>Loop</th>
<th>Disk Cache Configuration</th>
<th>Node Count Build Graph</th>
<th>BFS</th>
<th>DFS</th>
<th>TS (WQ)</th>
<th>TS (WS)</th>
<th>Peak Mem Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>410.bwaves</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>block_solver.f : 55</td>
<td>32 slots 20 MB</td>
<td>10,018,809</td>
<td>2.30</td>
<td>1.1</td>
<td>0.43</td>
<td>4.49</td>
<td>0.67 7 GB</td>
</tr>
<tr>
<td></td>
<td>64 slots 20 MB</td>
<td>10,018,809</td>
<td>2.39</td>
<td>1.09</td>
<td>0.34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>433.milc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>quark_stuff.c : 960</td>
<td>32 slots 6 MB</td>
<td>10,590,039</td>
<td>3.98</td>
<td>63.66</td>
<td>69.69</td>
<td>1.87</td>
<td>74.72 2.1 GB</td>
</tr>
<tr>
<td></td>
<td>64 slots 6 MB</td>
<td>10,590,039</td>
<td>3.22</td>
<td>19.57</td>
<td>21.12</td>
<td>1.07</td>
<td>20.65 3.5 GB</td>
</tr>
<tr>
<td></td>
<td>512 slots 6 MB</td>
<td>10,590,039</td>
<td>3.26</td>
<td>0.19</td>
<td>0.08</td>
<td>0.32</td>
<td>0.18 14 GB</td>
</tr>
<tr>
<td>436.cactusADM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StaggeredLeapFrog2.f : 366</td>
<td>64 slots 6 MB</td>
<td>33,684</td>
<td>0.08</td>
<td>5E-04</td>
<td>3E-04</td>
<td>0.0003</td>
<td>0.0003 690 MB</td>
</tr>
<tr>
<td></td>
<td>512 slots 6 MB</td>
<td>33,684</td>
<td>0.08</td>
<td>5E-04</td>
<td>3E-04</td>
<td>0.0003</td>
<td>0.0003 690 MB</td>
</tr>
<tr>
<td>437.leslie3d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tml.f : 3569</td>
<td>64 slots 6 MB</td>
<td>194,439</td>
<td>0.001</td>
<td>0.05</td>
<td>0.003</td>
<td>0.001</td>
<td>0.002 1.20 GB</td>
</tr>
<tr>
<td></td>
<td>512 slots 6 MB</td>
<td>194,439</td>
<td>0.001</td>
<td>0.05</td>
<td>0.003</td>
<td>0.001</td>
<td>0.002 1.20 GB</td>
</tr>
<tr>
<td>453.povray</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lighting.cp : 1373</td>
<td>64 slots 6 MB</td>
<td>3365</td>
<td>0.001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0001 240MB</td>
</tr>
<tr>
<td>459.GemsFDTD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NFT.F90 : 11659</td>
<td>64 slots 20 MB</td>
<td>33,473,462</td>
<td>11.3</td>
<td>3.45</td>
<td>2.95</td>
<td>104.43</td>
<td>5.46 12 GB</td>
</tr>
<tr>
<td>465.tonto</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mol.F90 : 11659</td>
<td>64 slots 20 MB</td>
<td>91,803</td>
<td>0.023</td>
<td>0.001</td>
<td>5E-04</td>
<td>0.001</td>
<td>0.0008 0.8 GB</td>
</tr>
<tr>
<td>470.lbm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lbm.c : 186</td>
<td>48 slots 20 MB</td>
<td>199,136,960</td>
<td>39.59</td>
<td>2.85</td>
<td>2.83</td>
<td>53.73</td>
<td>6.03 15.12 GB</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>approx_cont_mgau.c : 279</td>
<td>64 slots 20 MB</td>
<td>149,314</td>
<td>0.042</td>
<td>0.002</td>
<td>5E-04</td>
<td>0.001</td>
<td>0.001 0.25 GB</td>
</tr>
</tbody>
</table>

With above results we can conclude that choosing right configuration parameters is essential for an efficient analysis. For example, in case of 433.milc we observe that with 512 slots there is a significant improvement in time for each graph algorithm.
when compared to 32 slots. Also, it should be noted that picking 512 comes with a visible trade-off with memory. Thus for 433.milc we can say that a configuration with 64 slots might be the best choice for performing a memory intensive analysis, otherwise 512 slots is our best bet on a system with 15 GB of available memory.

4.3 Convex Partitioning on Jacobi 2D

We apply new implementation of Convex Partitioning algorithms, that is re-written using the Disk-Graph API, on Jacobi stencil on a regular 2-dimensional grid of size 128 and 120 time iterations. When compared to the previous version of the tool (Section 4.2.1 [4]), we could easily perform the analysis by scaling the input by 4 times. Table 4.2 lists disk-graph configuration and general information Number of Graph nodes for Jacobi 2D CDAG, time taken to build the graph etc. for the new version of convex partitioning tool. In comparison, Table 4.3 lists input and graph sizes for the Jacobi2D CDAG that was used for analysis using the previous version of the tool.

We apply two heuristics: Single Level Reuse and Iteration Vector Heuristics. Single level convex partitioning heuristics takes three parameters: Maxlive value, Neighbor Count and Successor Count. Together Neighbor Count and Successor Count defines what’s referred as priority in Algorithm 4 [4]. Maxlive sets a limit on the maximum number of live vertexes allowed while forming a convex component.

For single level reuse, we try different values of Max live (500, 1000, 5000, 10000) and for each max live value we try different priorities: equal, depth and breadth.
Equal priority implies same number of neighbor and successor count; Depth Priority implies higher successor count; and Breadth Priority implies higher neighbor count. Similarly for Iteration Vector heuristics we try experimenting with different values of ’T’ (tile size diameter). Trying different parameters for the heuristics and measuring the time taken for them to complete, which includes performing convex partitioning
and writing the memory traces to a file, gives us an idea of API performance because
varying parameters will likely

Eventually we plot reuse distance profile for different maxlive values and priority, and similarly for different T (tile diameter) values (figures 4.1, 4.2, 4.3, 4.4, 4.5). A lower curve implies better reuse profile and lower I/O cost.

The time taken to run each of these heuristics is listed in Table 4.4 and Table 4.5.

<table>
<thead>
<tr>
<th>Max Live</th>
<th>Neighbor Count</th>
<th>Successor Count</th>
<th>Time (in mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>10</td>
<td>10</td>
<td>7.93</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>10</td>
<td>7.86</td>
</tr>
<tr>
<td>5000</td>
<td>10</td>
<td>10</td>
<td>10.45</td>
</tr>
<tr>
<td>10000</td>
<td>10</td>
<td>10</td>
<td>24.57</td>
</tr>
<tr>
<td>500</td>
<td>10</td>
<td>20</td>
<td>9.59</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>20</td>
<td>8.71</td>
</tr>
<tr>
<td>5000</td>
<td>10</td>
<td>20</td>
<td>14.54</td>
</tr>
<tr>
<td>10000</td>
<td>10</td>
<td>20</td>
<td>20.98</td>
</tr>
<tr>
<td>500</td>
<td>20</td>
<td>10</td>
<td>6.85</td>
</tr>
<tr>
<td>1000</td>
<td>20</td>
<td>10</td>
<td>7.46</td>
</tr>
<tr>
<td>5000</td>
<td>20</td>
<td>10</td>
<td>10.62</td>
</tr>
<tr>
<td>10000</td>
<td>20</td>
<td>10</td>
<td>20.06</td>
</tr>
</tbody>
</table>

Table 4.4: Single Level Heuristic: Time taken to complete
Figure 4.1: Jacobi2D : Single Level Heuristic

Figure 4.2: Jacobi2D : Single Level Heuristic
Figure 4.3: Jacobi2D : Single Level Heuristic

Figure 4.4: Jacobi2D : Single Level Heuristic
Jacobi2d(T=120,N=128) : Iteration Vector Heuristic

Figure 4.5: Jacobi2D : Iteration Vector Heuristics

<table>
<thead>
<tr>
<th>$T$</th>
<th>Time (in mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7.68</td>
</tr>
<tr>
<td>8</td>
<td>7.15</td>
</tr>
<tr>
<td>16</td>
<td>6.59</td>
</tr>
<tr>
<td>32</td>
<td>6.57</td>
</tr>
</tbody>
</table>

Table 4.5: Iteration Vector Heuristic : Time taken to complete
Chapter 5: Conclusion

The thesis has described the motivation for having a scalable out-of-core graph API, that can be used while performing dynamic analysis on arbitrary programs whose execution is represented as a dynamic dependency graph. We give a brief overview of the dynamic analysis framework that serves as a basis for this API. The two main components of this API are Disk-Graph and Disk-Cache that work together to generate and traverse a dynamic dependency graph. We detail on how we write a disk based graph efficiently by making two passes over a given trace file. We explain how Disk-Cache component interacts with disk graph file by reading it in user specified block sizes. We re-implement Convex Partitioning heuristics as a client of our scalable API, detailing on how we used our API to implement it efficiently.

We then evaluate the API by generating disk based CDAGs for SPEC CFP 2006 benchmark suites and running simple graph algorithms on them. We also evaluate how well we scale by running Convex Partitioning heuristics on a Jacobi stencil with a large input size.
Appendix A: Example Usage of the API

```c
int main(int argc, char **argv)
{

    /*** LLVM Initialization
         The initialization should give the
data structure "id", that can be used
build a graph from trace
    ***/

    // Generate a graph and get a pointer to the graph

    /*
     * Points to note:
     * - A DiskCDAG is a templated class that takes a GraphNode
class as its template argument. This way a user can
implement its own GraphNode class to build a graph with
each node storing custom information.

     * - DiskCDAGBuilder derives from Library's graph builder
class and implements the visitNode() method that is
called for every instruction in the trace file.
     */
    DiskCDAG<GraphNode> *cdag = DiskCDAG<GraphNode>::generateGraph<GraphNode, DiskCDAGBuilder<GraphNode>>(ids, programName);

    // Check if we have a valid pointer
    if(cdag)
```

46
void performTopoSort(string outFile, bool useStack)
{
    Id numNodes = cdag->getNumNodes();
    boost::unordered_map<Id, Id> nodeIdToUnprocSuccsCountMap;
    deque<Id> readyNodeQ;
    Id readyNodeCount = 0;
    Id processedNodeCount = 0;

    // INITIAL SETUP

    // Traverse over the graph once to mark all the nodes as ready
    // that have 0 preds i.e. all input vertices
    for(Id i=0; i<numNodes; ++i)
    {
        CDAGNode *node = cdag->getNode(i);
        if(node->predsList.size() == 0)
        {
            readyNodeQ.push_back(i);
            ++readyNodeCount;
        }
        nodeIdToUnprocSuccsCountMap[i] = node->predsList.size();
    }

    // PERFORM TOPOLOGICAL SORT
    intbitsetIndex = 0;
}

// Use the graph pointer to perform a Topological Sort
// and write the new order to a specified file.
performTopoSort("topological_order_out", false);

// Delete graph pointer after use
delete cdag;
cdag = 0;
} else
{
    std::cerr << "Fatal: Failed to generate graph\n";
}
while(readyNodeQ.size() > 0)
{
    CDAGNode *node = 0;
    // Check if we are performing topological sort
    // using a stack or queue
    if(useStack)
    {
        node = cdag->getNode(readyNodeQ.back());
        readyNodeQ.pop_back();
    }
    else
    {
        node = cdag->getNode(readyNodeQ.front());
        readyNodeQ.pop_front();
    }
    ++processedNodeCount;

    Id numSucc = node->succsList.size();
    for(Id i=0; i<numSucc; ++i)
    {
        Id succId = node->succsList[i];
        boost::unordered_map<Id, Id >::iterator it =
            nodeIdToUnprocSuccsCountMap.find(succId);
        if(it != nodeIdToUnprocSuccsCountMap.end())
        {
            --nodeIdToUnprocSuccsCountMap[succId];
            if(nodeIdToUnprocSuccsCountMap[succId] == 0)
            {
                readyNodeQ.push_back(succId);
                nodeIdToUnprocSuccsCountMap.erase(it);
            }
        }
    }

    if(processedNodeCount != numNodes)
    {
        cout << "\nError: Cycles found in the graph!";
    }
}
Bibliography


