DepGraphAPI Programmer's Guide
Release 0.9b

Paradyn Parallel Performance Tools

May 26, 2009

Contents

1 Introduction

The DepGraphAPI is a multi-platform library for creating and analyzing dependence graph representations of binary code. A dependence graph is a representation of must-happen-before and must-happen-after relationships between program elements (e.g., instructions and basic blocks). A program may consist of several logically separate streams of execution that are interleaved by the compiler; these representations undo this interleaving. We represent these relationships in terms of a *graph*, a data structure that consists of nodes connected by directed edges. Nodes in the graph represent program elements and edges represent *dependences* (must-happen-before or must-happen-after) relationships between elements.

The DepGraphAPI currently provides four types of graph representation:

Data Dependence Graph (DDG) This graph represents the relations between instructions that define and instructions that use registers and memory. Nodes in this graph represent instructions, and edges connect definitions of a particular location to its uses.

Control Dependence Graph (CDG) This graph represents conditional execution of basic blocks in the program.

Program Dependence Graph (PDG) This graph is the union of the DDG and CDG.

Extended Program Dependence Graph (xPDG) This graph is the PDG augmented with additional nodes and edges necessary for forming an executable slice (defined below).

The main goal of this API is to provide the user with abstractions representing the logical dependencies between instructions in a program. An abstract interface provides two benefits: it simplifies the development of a tool since the complexity of a particular architecture is hidden, and it allows tools to easily be ported between platforms. Using a dependence graph representation of a program allows the user to focus on a particular aspect of program behavior and ignore program elements that do not affect that aspect of behavior.

An excellent example is the *slice*, a common use of program dependence. Intuitively, a slice of a program from a particular point is the sub-program that affects that point (a backwards slice) or is affected by that point (a forwards slice). Let i represent an instruction in the program and a some location that i defines (writes a value into). A forward slice from (i, a) are all instructions (and defined locations) that are affected by the definition of a by i. Similarly, the backwards slice from (i, a) are all instructions (and defined locations) that may affect the value written into a by i.

A future goal of this library is to allow users to improve these graph representations through the use of additional analyses. The included analyses used to generate these graph representations are conservative, and may overapproximate the actual dependences between instructions. A future release will provide an API for updating these graph structures, either with information known to the user directly or with the results of more sophisticated analysis.

The current beta of the DepGraphAPI depends on the InstructionAPI library and the DyninstAPI; future versions will depend only on the InstructionAPI and ParsingAPI libraries released as part of the DyninstAPI. Currently we support the IA-32 and AMD-64 architectures as these are the only architectures supported by the InstructionAPI. Future architecture support includes PPC, IA-64, and SPARC; the DepGraphAPI has no file format or operating system constraints.

2 Abstractions

DepGraphAPI provides a simple set of abstractions over complicated data structures to make it easy to use. The fundamental representation used by this library is the graph. A graph is a set of nodes connected by directed edges; each edge has a distinct (and unique) source and target. Nodes represent logical elements of the program. We define two classes of nodes: physical and virtual. Physical nodes that represent a particular instruction, basic block, or function. Virtual nodes represent a summary of the behavior of code that is not contained within the graph, such as a called function or the inputs to a function. We provide a set of methods for operating on Graphs, Nodes, and Edges; these methods provide a common interface to all four dependence graph types provided by the DepGraphAPI.

Figure 2 shows the inheritance hierarchy for the DepGraphAPI classes. All references to DepGraphAPI classes are internally reference counted; we do not require the user to perform any manual memory allocation or deletion.

2.1 Shared Abstractions

Graph The Graph represents a dependence graph for a particular function.

Node The Node represents an element within the graph. Nodes are connected by edges and are labelled with information.

Physical Node These Nodes represent an element (instruction, basic block, etc.) of the program. They are labelled with the starting address of that object.

Virtual Node These Nodes represent summaries of program behavior not contained within the graph. Virtual nodes do not have an address associated with them.

Edge Edges connect Nodes. Edges are directed and have a source and target.

Figure 1: Inheritance diagram for the DepGraphAPI. The Graph, Node, and Edge classes provide a common interface specification. The DDG, CDG, PDG, and xPDG graphs customize these three classes as necessary.

2.2 Data Dependence Graph

The data dependence graph adds several specialized node types.

OperationNode Each physical DDG node represents an operation (a definition of an abstract location by an instruction). We describe operations and our justification of this abstraction below.

FormalParameterNode These virtual nodes represent input parameters to a function. An input parameter is a definition of an abstract location before the function is executed.

FormalReturnNode These virtual nodes represent definitions that persist after the function returns.

ActualParameterNode These virtual nodes represent arguments to a called function.

ActualReturnNode These virtual nodes represent definitions made by a called function.

2.3 Control Dependence Graph

The control dependence graph adds one new specialized node type:

BlockNode We represent control dependence at the block level for efficiency. Therefore, each node in the CDG represents a block.

2.4 Program Dependence Graph

The Program Dependence Graph is the union of the DDG and CDG and is constructed from the same abstractions. The block-level information in the CDG is automatically converted to OperationNodes.

2.5 Extended Program Dependence Graph

Although users can use PDGs to slice programs, the slices obtained through PDGs are not always executable slices. An executable slice is a slice of a program that can be executed without any change in program behavior with respect to the given slicing criteria. Binary slices obtained through a PDG do not always include all necessary branch and return instructions that make the slice follow the flow of control of the original program due to fallthroughs. Therefore, an Extended Program Dependence Graph contains dependencies between an instruction i and all instructions that are required to satisfy correct control flow in case i appears in a slice. An xPDG also contains all nodes and edges that are parts of a regular PDG.

2.6 Annotations

The DepGraphAPI makes use of the Annotation infrastructure in the DyninstAPI classes. First, all DepGraphAPI abstractions are annotatable. The internal implementation of the annotation infrastructure is optimized for sparse annotation; that is, when the majority of abstractions are not annotated. Currently this cannot be tuned by the user. Second, the function abstractions are annotated with the completed graphs. Therefore, the user does not need to keep a reference to an analyzed graph. This annotation can be discarded by using the appropriate methods on the Graph class.

3 Examples

To illustrate the ideas in the API, this section presents short examples that demonstrate how the API can be used.

The first example demonstrates how to access the PDG for a particular function and take a slice from a known instruction (identified by address) and register that instruction defines. The code for this example is shown in Figure 2. Lines of interest are:

- Line 16 identifies all nodes with a particular address `insnAddr`. The set of these nodes is represented by the pair of iterators `nodeBegin` and `nodeEnd`.
- Line 19 determines whether there were nodes at the given address. If the iterators are equal the range is empty.
- Line 25 determines the set of nodes reachable from the given node (the *forward closure* from the node). The statement `*nodeBegin` returns the first node from the set identified in line 16. As before, the closure is represented by an iterator pair `sliceBegin, sliceEnd`.
- Line 28 shows how to iterate over the forward closure. The iterator `sliceBegin` will represent each such node; the sequence of the nodes is undefined. Each node can be accessed by dereferencing the iterator: `*sliceBegin`.

The second example shows how to determine which instructions in a basic block define themselves. This is one method to identify a loop iteration variable. The code for this example is shown in Figure 3. Lines of interest are:

- Line 15 shows how to access the instructions in a basic block. The `InsnInstance` typedef consists of an InstructionAPI instruction object and the address of the instruction. We use these addresses to identify nodes within the graph.
- Line 18 shows how to iterate over each instruction in the block.
- Line 23 shows how to find the set of nodes representing each instruction. Since the DDG may represent a single instruction as multiple operation nodes this set may have multiple elements.

```
using namespace Dyninst;
using namespace DepGraphAPI;

// Assume this represents a function of interest
BPatch_function *func;
// And an address of an instruction of interest
Address insnAddr;
// And a register defined by the previous instruction
InstructionAPI::RegisterAST::Ptr reg;

// Access the PDG for this function
PDG::Ptr pdg = PDG::analyze(func);

// Find the node of interest
NodeIterator nodeBegin, nodeEnd;
pdg->find(insnAddr, reg, nodeBegin, nodeEnd);

// Make sure we found a node...
if (nodeBegin == nodeEnd) {
    // Complain
}

// Create the forward slice from the node of interest
NodeIterator sliceBegin, sliceEnd;
pdg->forwardClosure(*nodeBegin, sliceBegin, sliceEnd);

// Iterate over each node in the closure and do something
for (; sliceBegin != sliceEnd; sliceBegin++) {
    // ...
}
```

Figure 2: Slicing example. This code fragment identifies the nodes reachable by following edges forward from the node with address `insnAddr`.

- Line 26 shows how to get the targets of a node. These targets can be represented either as a set of edges or a set of nodes, whichever is convenient.
- Line 28 identifies nodes that have edges to themselves; that is, nodes that define themselves.

4 Definitions and Basic Types

The DepGraphAPI supplies three types of dependence graphs. We define these forms of dependence here, along with definitions of the underlying concepts. The following definitions and basic types are referenced throughout the rest of the document.

4.1 Definitions

Instruction An instruction represents a single machine instruction with a unique starting offset. Instruction instances are identified by this offset.

Abstract Location An abstract location represents a machine register, memory location, or set of memory locations. Registers are referred to by their InstructionAPI representation. Memory locations consist of a region and an optional offset within that region. Regions include the stack, the heap, and global memory. Stack locations are assumed to be relative from the top of the stack at the beginning of the function. Our current implementation assumes a single heap location; this may change in future releases. Finally, offsets into global memory represent absolute addresses.

Operation An operation is a pair of an instruction and an abstract location defined by that instruction. An instruction may define more than one abstract location, particularly on CISC architectures. If we represent data dependence at the instruction level we may overapproximate dependences between instructions; we describe an example of this occurrence below. Instead we represent dependences at the operation level.

Data Dependence In general instruction j is data dependent on i if i defines an abstract that j uses and there is an execution path from i to j along which a is not redefined. We use a more precise definition that uses operations as nodes. Let m = (i,a) be an operation representing the definition of a by i, and similarly for n = (j, b). Then n is data dependent on m if i defines a, j uses a to define b, and there exists a path as above.

Control Dependence - An instruction j is control dependent on i if i has multiple successors and j is executed along at least one, but not all, possible execution paths from i.

```
using namespace Dyninst;
using namespace DepGraphAPI;

// Assume these represent a function and block of interest
BPatch_function *func;
BPatch_basicBlock *block;

// Access the DDG
DDG::Ptr ddg = DDG::analyze(func);

// Get the list of instructions (and their addresses) from the block

typedef std::pair<InstructionAPI::Instruction, Address> InsnInstance;
std::vector<InsnInstance> insnInstances;
block->getInstructions(insnInstances);

// For each instruction, look up the DDG node and see if it has itself as a target
for (std::vector<InsnInstance>::iterator iter = insnInstances.begin();
     iter != insnInstances.end(); iter++) {
  Address addr = iter->second;

  NodeIterator nodeBegin, nodeEnd;
  ddg->find(addr, nodeBegin, nodeEnd);
  for (; nodeBegin != nodeEnd; nodeBegin++) {
    NodeIterator targetBegin, targetEnd;
    (*nodeBegin)->getTargets(targetBegin, targetEnd);
    for (; targetBegin != targetEnd; targetBegin++) {
      if (*targetBegin == *nodeBegin) {
        // Found a node that has itself as a target
        actOnSelfDefiningNode(*nodeBegin);
      }
    }
  }
}
```

Figure 3: DDG traversal example. This code fragment identifies nodes within a basic block that define themselves.

4.2 Basic Types

`typedef unsigned long Address`

An integer value that represents a unique location in memory.

Smart/shared pointers

All objects returned to users are transparently wrapped with a reference counted pointer implementation. This smart pointer automatically handles deallocation and garbage collection. These pointers are referred to by the `::Ptr` suffix (e.g., `Graph::Ptr`, `Node::Ptr`, etc.). Our implementation is derived from the Boost `shared_ptr` implementation; for more information, please visit www.boost.org. Shared pointers have some limitations when compared with standard pointers. In particular, `dynamic_cast` (as well as other casting operators) are not defined on shared pointers. Performing such a cast must be done with the `dynamic_pointer_cast` method. For example:

```
VirtualNode::Ptr virt = dynamic_pointer_cast<VirtualNode>(nodePtr);
```

Iterators

The DepGraphAPI uses an iterator-based interface in favor of a collection-based interface. This is done to reduce copying and improve efficiency. Any method that returns a range of types (e.g., `Graph::allNodes`) takes as arguments two iterators that are updated to point to the beginning and end of the range. The user can then use standard iterator methods (e.g., a for loop) to examine each element in the range. We define two types of iterators: `NodeIterator` and `EdgeIterator`.

4.3 Operation-Level Data Dependence

The conventional definition of the DDG (in which nodes represent instructions) may over-approximate the data dependencies within a binary. This occurs when an instruction defines multiple abstract locations and uses different sets of abstract locations for each definition. For example, consider the IA-32 instruction xchng which exchanges the contents of two registers. From an instruction perspective, this instruction uses and defines two registers. However, there is no dependence between the use and definition of the same registers. To avoid this over-approximation, each node in the DepGraphAPI DDG consists of an operation; an (instruction, abstract location) pair. We show an example of the use of instructions and operations in Figure 4.3.

5 Namespaces

The classes described below are defined in the `Dyninst::` and `Dyninst::DepGraphAPI::` namespaces. The Graph, Node, and Edge classes are contained in the `Dyninst::` namespace. To access them a user should refer to them using the `Dyninst::` prefix (e.g., `Dyninst::Graph` or `Dyninst::DepGraphAPI::DDG`). Alternatively, a user can add the C++ `using` keyword above any reference to such objects (e.g., "`using namespace Dyninst;`"). All other classes are defined under the `Dyninst::DepGraphAPI` namespace and should be referred to appropriately.

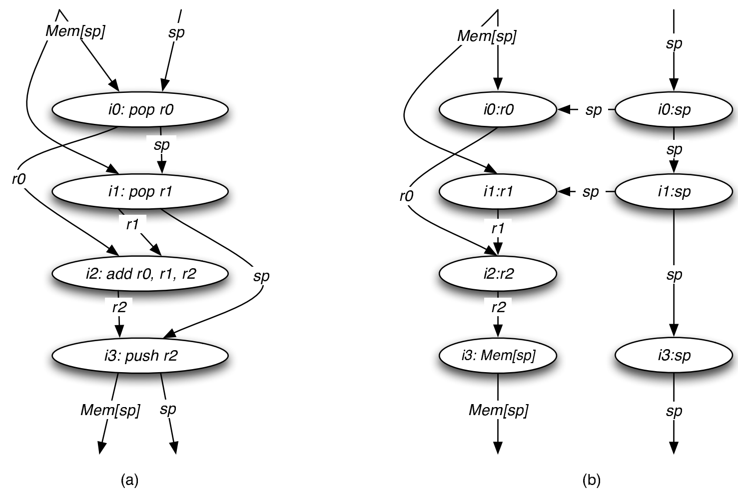

Figure 4: Example of DDG operations. Two data dependence graphs. Figure a) provides an example of the problems of representing instructions as single nodes. In this graph it is possible for paths to "cross" definitions; for example, there is a path from the definition of r_0 by i_0 to the definition of *sp* (the stack pointer) by i_3, when in the actual program there is no such dependence. The DDG shown in figure b) makes the intra-instruction data dependencies explicit and thus removes the possibility of erroneous paths.

6 API Reference

This section describes the interface of the DepGraphAPI. Each of the subsections represents a different interface.

6.1 Shared Classes

The `Graph`, `Node`, and `Edge` classes are written to be generic and shareable between DyninstAPI components. We include the API for these classes here.

6.1.1 Graph

void entryNodes(NodeIterator &begin, NodeIterator &end) This method returns a range of nodes (defined by `begin` and `end`) such that 1) all nodes in the graph are reachable from the nodes in this range by traversing out-edges and 2) the range is minimal. The nodes included in this range may be virtual.

void exitNodes(NodeIterator &begin, NodeIterator &end) This method returns a range of nodes (defined by `begin` and `end`) such that 1) all nodes in the graph are reachable from the nodes in this range by traversing in-edges and 2) the range is minimal. The nodes included in this range may be virtual.

void allNodes (NodeIterator &begin, NodeIterator &end) This method returns the range of all nodes in the graph.

void printDOT(std::string fileName) This method generates a representation of the graph in DOT format.

bool find(Address addr, NodeIterator &begin, NodeIterator &end) This method sets `begin` and `end` to point to a range representing the nodes with a particular address. It returns true if the range is non-empty.

void removeAnnotation() This method removes the graph from internal storage. Once all user handles to the graph are discarded the graph will be destroyed.

6.1.2 Node

bool hasInEdges() This method returns true if the node has at least one in edge.

void ins(EdgeIterator &begin, EdgeIterator &end) This method returns the range of in edges to the node (edges that have the node as a target).

void ins(NodeIterator &begin, NodeIterator &end) This method is similar to the previous, but automatically traverses the edges and returns a range of source nodes.

bool hasOutEdges() This method returns true if the node has at least one out edge.

void outs(EdgeIterator &begin, EdgeIterator &end) This method returns the range of out edges from the node (edges that have the node as a source).

void outs(NodeIterator &begin, NodeIterator &end) This method is similar to the previous, but automatically traverses the edges and returns a range of target nodes.

void forwardClosure(NodeIterator &begin, NodeIterator &end) This method returns all nodes reachable from this node in the forward direction (by traversing out-edges).

void backwardsClosure(NodeIterator &begin, NodeIterator &end) This method returns all nodes reachable from this node by traversing in-edges.

std::string format() This method returns a textual representation of the node.

bool isVirtual() This method returns true if a node is virtual.

6.1.3 PhysicalNode : Node

Address addr() This method returns the starting offset of the code object (basic block, instruction, or operation) the node represents.

bool isVirtual() This method returns false for physical nodes.

6.1.4 VirtualNode : Node

bool isVirtual() This method always returns true for virtual nodes.

6.1.5 Edge

Node::Ptr source() This method returns the source node of an edge.

Node::Ptr target() This method returns the target node of an edge.

6.2 Data Dependence Graph

6.2.1 DDG

DDG::Ptr analyze(BPatch_function *func) This method creates and returns a DDG for the provided function.

void formalParameterNodes(NodeIterator &begin, NodeIterator &end) This method returns the range of all formal parameters to the function.

void formalReturnNodes(NodeIterator &begin, NodeIterator &end) This method returns the range of all formal returns from the function.

void actualParameterNodes(Address callAddr, NodeIterator &begin, NodeIterator &end) This method returns the range of all actual parameters for the call instruction at the given address.

void actualReturnNodes(Address callAddr, NodeIterator &begin, NodeIterator &end) This method returns the range of all actual returns for the call instruction at the given address.

bool find(Address addr, Absloc::Ptr absloc, NodeIterator &begin, NodeIterator &end) This method returns the range of nodes that fit the specific address and absloc requirements. This range will contain at most one element. It returns true if the range is non-empty, and false otherwise.

6.2.2 Absloc

std:string format() This method returns a textual representation of the abstract location. This representation is guaranteed to be unique for unique abstract locations.

void getAliases(AbslocIterator &begin, AbslocIterator &end) If more than one Absloc may refer to the same abstract location (e.g., a particular stack slot and the representation of the entire stack) return any such aliases.

bool isPrecise() This method returns true if the absloc does not contain any others; that is, if any aliases are more general than this one.

6.2.3 OperationNode : PhysicalNode

Absloc::Ptr absloc() This method returns the abstract location represented by this node.

6.2.4 FormalParameterNode : VirtualNode

Absloc::Ptr absloc() This method returns the abstract location represented by this node.

6.2.5 FormalReturnNode : VirtualNode

Absloc::Ptr absloc() This method returns the abstract location represented by this node.

6.2.6 ActualParameterNode : VirtualNode

Absloc::Ptr absloc() This method returns the abstract location represented by this node.

BPatch_function *callee() This method returns the callee function whose argument is represented by this node.

6.2.7 ActualReturnNode : VirtualNode

Absloc::Ptr absloc() This method returns the abstract location represented by this node.

BPatch_function *callee() This method returns the callee function whose return is represented by this node.

6.3 Control Dependence Graph

6.3.1 CDG

CDG::Ptr analyze(BPatch_function *func) This method creates and returns a CDG for the provided function. ¡¡¡¡¡¡¡ HEAD:depGraphAPI/doc/depGraphAPI.tex

bool find(BPatch_basicBlock *block, NodeIterator &begin, NodeIterator &end) This method returns the range of nodes representing the provided block. This range will have at most one element. It returns true if the range is non-empty and false otherwise.

bool find(Address addr, NodeIterator &begin, NodeIterator &end) This method returns the range of nodes containing the provided address. It returns true if the range is non-empty and false otherwise.

6.3.2 BlockNode : Node

BPatch_basicBlock *block() This method returns the basic block represented by this node.

6.4 Program Dependence Graph

6.4.1 PDG

PDG::Ptr analyze(BPatch_function *func) Creates and returns a PDG for the provided function.

find(Address addr, Absloc::Ptr absloc, NodeIterator &begin, NodeIterator &end) This method returns the set of nodes that fit the specific address and absloc requirements. This node will be singular.

6.5 Extended Program Dependence Graph

6.5.1 xPDG

xPDG::Ptr analyze(BPatch_function *func) Creates and returns an xPDG for the provided function.

find(Address addr, Absloc::Ptr absloc, NodeIterator &begin, NodeIterator &end) This method returns the set of nodes that fit the specific address and absloc requirements. This node will be singular.

7 Implementation Status

This release of the DepGraphAPI is a public beta and has limited platform support and implementation features. These limitations are as follows:

- Platforms: the DepGraphAPI is implemented for IA-32 and x86-64. This is primarily due to a dependence on the InstructionAPI.
- Limited operator node support. We currently do not identify intra-instruction dependences between used and defined abstract locations. Instead, there is a complete interconnection between these.

8 Building DepGraphAPI

This appendix describes how to build DepGraphAPI from source code, which can be downloaded from http://www.paradyn.org or http://www.dyninst.org.

8.1 Building on Unix

The beta of the DepGraphAPI depends on the DyninstAPI. It is currently packaged with the DyninstAPI source tree. It can be built using the DepGraphAPI make target once the DyninstAPI has been built and installed.