Efficient utilization of secondary storage for scalable dynamic slicing

Ulf Kargén, Nahid Shahmehri
Department of Computer and Information Science
Linköping University
SE-58183 Linköping, Sweden
{ulf.kargen, nahid.shahmehri}@liu.se

Abstract—Dynamic program slicing is widely recognized as a powerful aid for e.g. program comprehension during debugging. However, its widespread use has been impeded in part by scalability issues that occur when constructing the dynamic dependence graph necessary to compute dynamic slices. A few seconds of execution time on a modern CPU can easily yield dynamic dependence graphs on the order of tens of gigabytes in size. Existing methods either produce imprecise slices, incur large time overheads during slice computation, or run out of memory for long program executions.

By carefully designing our method to take advantage of locality, we are able to efficiently use secondary storage for dynamic dependence graphs, thus allowing our method to scale to long program executions. Our prototype implementation runs directly on x86 executables, eliminating problems with e.g. binary-only libraries. We show in our experiments that graphs can be constructed for program runs with billions of executed instructions, at slowdowns ranging from 62x to 173x. Our optimized format also allows graphs to be traversed at speeds of several million dependence edges per second.

Keywords—dynamic slicing; dynamic dependence graph; debugging; x86; binary analysis;

I. INTRODUCTION

Dynamic program slicing was first proposed in 1988 by Korel and Laski [1]. Intuitively, a dynamic slice is the set of statements that, during one execution of a program, somehow influenced the computation at a specific point of interest, called the slicing criterion. To compute a dynamic slice, a dynamic dependence graph (dyDG) must first be created. The dyDG is a directed acyclic graph that represents all data and control dependencies between executed statement instances for a given program execution. Computing a dynamic slice simply implies traversing the dyDG backwards from the point of interest. All statements whose dynamic instances were visited at least once during traversal are part of the slice.

While the utility of dynamic slicing for program comprehension tasks, such as debugging, has been recognized in several works [2]–[4], the technique has seen little use in practice. The main impediment to its widespread use has been scalability issues when creating the dyDG for realistic programs. As an example, a few CPU-seconds of execution of a typical program on a modern computer easily yields a dyDG on the order of tens of gigabytes in size. Several methods for addressing scalability issues in dynamic slicing have been proposed in previous work. Some early works, such as Algorithm I and II in [5], rely on removing information from the dyDG, resulting in imprecise slices. The methods described in [6] and [7] work by retrieving the relevant data and control dependencies directly from a program trace, instead of explicitly constructing the dyDG. While large memory overheads are eliminated with these methods, computing slices instead becomes very time-consuming, since a sequential search of the trace must be performed to recover dependencies. Zhang and Gupta propose a method in [8] where the size of the graph is reduced by identifying static control and data dependencies in the program code. Such dependencies can be reproduced during traversal, eliminating the need for storing every instance of the dependencies explicitly. Their method is very effective in reducing the size of the dyDG, while still achieving efficient computation of slices, but relies on being able to fit compressed graphs entirely in main memory. In general, this cannot always be assumed to be possible, especially for long-running, complex software.

In order for dynamic slicing to scale for arbitrarily long program executions, secondary storage must be utilized for dynamic dependence graphs. While storing dyDGs to secondary media may seem like a trivial problem, retaining fast lookup and traversal of the graphs presents several challenges. In this paper, we propose a method to address these challenges. To the best of our knowledge, this problem has not been previously treated in the literature. Since previous work [8] has shown that the data dependencies typically amount to about 90% of the overall size of the dyDG, we focus on data dependencies in this work, i.e. the dynamic data dependence graph (dyDDG). However, our method is also readily extendible to treat control dependencies.

The main challenges faced when using secondary storage for the dyDDG are due to high random-access times. This mandates the use of a “flat” data format, which can be stored in long, continuous units. In order to facilitate fast lookup and traversal, however, the logical graph structure of the data must be retained. A program trace, for example, can be seen as a “flattened” representation of a dynamic dependence graph, and is ideally suited for secondary storage. While it is possible to record all information necessary to deduce dynamic data and control dependencies in a trace, the graph
structure is lost, and must be recovered during slice computations, introducing significant overhead. Efficient storage and traversal of the dyDDG requires a balanced tradeoff between flatness and structure.

The main contributions of our work are as follows:

1) We identify several requirements for efficient use of secondary storage when creating and traversing dyDDGs, and propose a method for achieving the aforementioned tradeoff between flatness and structure, satisfying these requirements.

2) We also present a prototype implementation of our method, working directly on unmodified x86 executables running in Linux. Performing analysis directly at the binary level has several advantages; similar to a regular debugger, our tool can work on any binary, regardless of the high-level language used. If symbolic debug information is available, it can be used to translate the binary-level slice into a slice of the original high-level program. Working directly on machine code also avoids incomplete dyDDGs for programs that use binary-only libraries, as well as enabling applications in security, such as malware analysis, or security auditing of closed-source programs. It should be noted, however, that our method as such is not limited to binary programs, but is equally applicable for any language with a notion of data flow.

3) Finally, we use the prototype implementation to empirically demonstrate the effectiveness of our proposed method.

II. EFFICIENT CONSTRUCTION OF THE DYDDG

In this section we first present the requirements that have guided the design of our method. We then lay the foundations for the remainder of the paper by describing our logical and physical graph representation. By logically grouping data dependencies according to basic block, our representation significantly reduces the space overhead, while still allowing rapid graph traversal. Grouping dependencies in this manner, however, requires maintaining separate buffers for each basic block, so that dependence edges can be stored temporarily before being committed to secondary storage. In section II-C we show how our buffer allocation algorithm exploits locality to improve performance. We also prove that our algorithm can optimally adjust buffer sizes to minimize the impact of random-access delays. We then show how dynamic data dependence graphs are constructed in an on-line fashion. The two final subsections describe optimizations and how slices are computed, respectively.

A. Requirements for efficient use of secondary storage

We have identified several requirements for efficient utilization of secondary storage when creating the dyDDG. These requirements reflect different aspects of the flatness and structure-preservation attributes discussed earlier, and have guided the design of the method described in this paper. The four requirements are as follows:

- **R1: Retain structure.** The key to efficient lookup and traversal of the dyDDG is using a storage format that, to the greatest extent possible, retains the graph structure.
- **R2: Avoid indirection.** Due to high random-access times, indirection when performing lookups in the graph will impact performance considerably more when secondary storage is used.
- **R3: On-line creation.** The high random-access times, together with the large data volumes, will also make scalability difficult for methods that rely on post-processing the graph to e.g. sort or re-structure elements. The graph should therefore be created in its final format in an on-line fashion.
- **R4: Aggregate stores.** Elements of the graph should, to the greatest extent possible, be bundled, so that they can be written to secondary storage in continuous units.

In the following subsections, we show how our method aims to satisfy these requirements. We refer to requirements in the text with the notation Rn.

B. Graph representation

In this section we describe how dyDDGs are represented, starting with the logical graph representation. We then briefly outline our physical representation, and discuss how graphs are traversed.

**Logical graph representation.** Let I and J be two instructions in a program, and i, j be execution instances of I and J, respectively. i has a data dependency on j if i uses data defined by j.

In the classical graph representation used in early works, such as [5], instruction instances are represented by vertices, and data or control dependencies are represented by outgoing edges from vertices. Vertices are labeled with an instruction identifier, e.g. an address, to allow computation of dynamic slices. While simple and intuitive, the classical representation leads to graphs with large amounts of redundant information, because a vertex with an instruction identifier needs to be replicated for each instruction instance. We instead use a graph representation similar to the one proposed by Zhang and Gupta in [8]. In their graphs, each static instruction is represented by a vertex, and data or control dependencies are represented by adding labeled edges between vertices. The number of vertices is thus bounded by the size of the program, while the number of edges per vertex is determined by the length of execution.

The label of an edge identifies the instruction instances involved in the corresponding dependency. For example, a data dependency of i on j is represented by adding an edge from I to J, labeled with instance(i) and instance(j) i.e. the use-instance of I and the define-instance of J.
Zhang and Gupta use a global timestamp, incremented on every basic block execution, to identify instruction instances. We instead use a per-basic-block execution count. Both approaches suffice equally well for identifying instances, but our subtly different approach has several benefits; while the total number of executed basic blocks may be very large, the execution counts of individual basic blocks are typically much smaller. We can therefore use a smaller data type for representing instruction instances, which is important in order to reduce space and time overhead. The use of execution counts instead of a global timestamp also allows use-instances to become implicit in our physical graph representation, as shown below.

**Physical graph representation.** A dependence edge in our dyDDG representation consists of four pieces of information; the addresses and execution count of the defining instruction, and the address and execution count of the using instruction. The defining instruction is explicitly identified by a tuple (address, instance), specifying respectively the address and execution count of the instruction. Every basic block in the analyzed program has an associated logical array (Figure 2a). A dependence edge is represented by adding an (address, instance) tuple to the logical array belonging to the basic block of the using instruction. In order to identify the using instruction in a dependency, stored tuples are grouped into frames. Each input operand of each instruction in the basic block has a slot at a specific offset in the frame. By keeping a record of which slots belong to which instruction, the address of the using instruction is unambiguously specified. This record, henceforth referred to as the frame layout record, is inferred statically from the program code, for each basic block. Figure 1 shows an example of an x86 basic block, and the layout of its frame. The first instruction copies register ebx to eax, and therefore gets one slot. The next instruction has two input operands (eax and ecx), and therefore occupies slots two and three, and so on.

Since frames in one logical array are all of the same size, the offset of a frame within its array precisely determines the corresponding basic block execution count. Both the address and the instance of the using instruction in a data dependency is thus implicitly determined by the position of the (address, instance) tuple within the logical array, while the contents of that tuple specify the defining instruction. In the context of the logical representation of the dyDDG described above, a basic block, and its associated logical array, correspond to a group of vertices. Adding a frame to the array corresponds to adding labeled edges for all instructions in the basic block.

In order to store logical arrays to file, they must be split into smaller chunks, so that chunks from different logical arrays can be stored interleaved (Figure 2b). To facilitate fast retrieval of chunks during traversal, a chunk map is maintained for each basic block, associating offsets in the logical array with file offsets of chunks. Clearly, chunk maps must be small enough to fit in RAM, to avoid expensive indirection through data structures on file (R2). In order to keep the number of entries in chunk maps low, sufficiently large chunks must be used. It is also clear that larger chunks are favorable, in order to reduce the impact of random-access delays during storage (R4). However, using larger chunks requires some kind of buffering scheme, where frames are stored to temporary buffers before being committed to secondary storage. In section II-C we address the problem of optimal allocation of such buffers.

**Graph traversal.** Figure 3 shows the process of following a dependence edge in the form of an (address, instance) tuple. The containing basic block of the defining instruction is first retrieved from the address. An efficient mapping, associating addresses to basic blocks, is maintained for this purpose. Once the basic block has been identified, the instance is used to look up the correct chunk in the chunk map. Since frames are of fixed size, the offset of the correct frame within the chunk can easily be calculated from the instance. Finally, the frame layout record is used to find all slots belonging to the defining instruction, so that traversal can continue.

Note that using execution counts to perform lookups has the additional benefit of making data-dependence information more readily available. Our method allows answering queries such as “Compute the slice of the nth instance of instruction X”, whereas the traditional graph representation used in e.g. [5] would only allow answering queries similar to “Compute a slice starting at vertex m”.

**C. Optimal buffer allocation**

In this section, we address the problem of maintaining buffers, where chunks can be built in memory before being
committed to secondary storage.

In order to maximize performance, we want buffers to be as large as possible (R4). However, allocating one buffer for each basic block in a program would require unnecessarily constrained buffer sizes. To optimize buffer allocations, we take advantage of the well known fact that during any (small) time window of execution, typically only a small subset of all instructions in a program are actively executing. For the remainder of the paper we will refer to this subset as the working set.

By only allocating buffers for the basic blocks in the current working set, each buffer can be larger, leading to greater efficiency. That, however, requires a mechanism to decide which basic blocks should be considered part of the current working set. As the size of the working set differs greatly between different programs, and also changes during execution, a method to dynamically determine the working set size, and adjust buffer sizes accordingly, is required. In this section we present algorithms for that purpose.

We first present a simplified algorithm that uses fixed-size buffers, and then show how this algorithm can be extended to dynamically adjust buffer sizes. We also show that the latter algorithm achieves optimal distribution of buffer sizes given a fixed memory limit.

**Fixed buffer-size algorithm.** As part of the instrumentation required to record the dyDDG (see section III) a basic block preamble is inserted before the first instruction of every basic block. Figure 4 shows pseudocode for the basic block preamble used in our fixed-size buffer algorithm. Each basic block has an associated record, denoted Bbl in Figure 4, which holds a pointer to its current buffer, and other information specific to the basic block. The preamble first checks if the basic block has no allocated buffer (line 1), which means that the basic block is currently not considered part of the working set. We henceforth refer to basic blocks in the working set as active, and all other basic blocks as inactive. If the basic block is found to be inactive, a new buffer is allocated (lines 2 and 3). At that point, the basic block is considered active. Each basic block also has a frame_pointer that points to the current frame within the buffer (line 4).

If the basic block is active, but the buffer is full (line 5), the buffer is committed for storage to file, and subsequently freed. These steps are represented by the call to commit at line 6. A new buffer is then allocated and the frame_pointer is updated as in the previous case (lines 7 and 8). Note that, since the buffer size is seldom perfectly aligned to the frame size, and because buffers may be committed prematurely (see below), buffers will rarely be completely filled. Only the part of the buffer that is actually used is stored to file, becoming a chunk, in our terminology.

If neither of the above two cases apply, the frame_pointer is simply advanced one frame size (line 10). The basic block then proceeds to execute, while the inserted instrumentation for each instruction records data dependencies into the frame pointed to by frame_pointer. Note that, while allocating and committing buffers may be fairly expensive, in the vast majority of cases only the else-branch (line 10) needs to be executed, assuming buffers are appropriately sized.

Figure 5 shows pseudocode for the alloc_buffer function used in the basic block preamble. References to all currently active basic blocks are held in global queue, referred to as queue in the pseudocode. The alloc_buffer function uses the queue to implement a least-recently-used strategy to de-activate basic blocks when no more memory is available.

Before allocating a new buffer, the memory utilization is checked (line 3). If memory use has surpassed a set limit, the basic block at the end of the queue is moved to the inactive state by popping it off of the queue, committing and freeing its buffer, and setting its buffer pointer to NULL (lines 4–7).

After having allocated a new buffer for the requested basic block (line 9), that basic block is moved to the head of the queue. (Or it is inserted, if it was previously inactive.) This scheme ensures that a basic block is always moved to the head of the queue when it transitions from the inactive to the active state, or its buffer becomes full. The least recently used basic blocks move towards the end of the queue, until they are eventually evicted. The queue thus
represents the basic blocks that are currently considered part of the working set, and the algorithm implemented by the preamble and `alloc_buffer` ensures that the queue is dynamically updated to reflect this property.

While this simple scheme may seem sufficient at first glance, it has some serious drawbacks. Since the size of the working set may vary greatly between different programs, as well as during execution of a single program, fixed-size buffers will often result in poor performance. If a buffer size is chosen that is too small, memory will be underutilized, leading to unnecessarily reduced performance. If, on the other hand, overly large buffers are used, a “thrashing” phenomenon will occur, since not enough memory will be available to hold buffers for the entire working set. Basic blocks will be continuously evicted and re-activated, leading to extremely poor performance. We now show how our algorithm can be extended to avoid such problems.

**Dynamic buffer-size algorithm.** Execution frequencies of instructions in a program typically follow a power-law distribution, so that a few instructions contribute the vast majority of the execution time. Any buffer allocation algorithm that use uniform buffer sizes (such as the static-size algorithm above) will therefore be suboptimal, since buffers of frequent basic blocks will need to be committed too often, while buffers of infrequent basic blocks will take up an unnecessary amount of space. Our algorithm must thus (i) be able to accurately estimate the current working set and (ii) dynamically adjust individual buffer sizes to minimize the overhead, while keeping memory use below the limit. Moreover, to achieve (ii), the working set size and the individual basic block execution frequencies (i.e. how fast buffers fill up) must also be taken into account. To determine the optimal buffer allocation strategy, we must first model the overhead of committing buffers to secondary storage.

Let $d_i^t$ represent the amount of data (i.e. dynamic dependence edges) received into the buffer of basic block $i$ during a time interval $t$. Furthermore, let $b_i$ represent the fraction of the total memory $M$ reserved for the buffer belonging to $i$, so that $\sum_{i=1}^{n} b_i = 1$, where $n$ is the total number of basic blocks in the current working set. The number of times the buffer of $i$ must be committed during the time $t$ is then given by

$$\frac{d_i}{b_i M}$$

(1)

The total overhead from committing buffers during $t$ can be modeled as:

$$c = \sum_{i=1}^{n} \left( \frac{d_i}{b_i M} D + \frac{d_i}{T} \right)$$

(2)

where $D$ represents the constant delay due to the random-access time of the storage device, the overhead of the file system, performing system calls, etc., and $T$ is the write-throughput of the storage device. To find the optimal values of $b_i$, we need to solve $\frac{\partial c}{\partial b_i} = 0, \forall i$ taking the constraint $\sum_{i=1}^{n} b_i = 1$ into account. Since the constant term disappears in the derivation, the optimal buffer allocation is found by solving

$$\frac{\partial}{\partial b_i} \sum_{i=1}^{n} \frac{d_i}{b_i} = 0, \forall i$$

(3)

This is a recurring optimization problem within e.g. the field of network content delivery [9], [10]. It can be shown that the optimal buffer allocation is achieved when

$$b_i \propto \sqrt{d_i}, \forall i$$

(4)

We omit the proof for brevity here, but instead refer the reader to e.g. [9] for a full proof.

Since the total amount of data received into each buffer during execution cannot be known a priori, we need an algorithm that continuously adjusts buffer allocations towards the optimum in an on-line fashion. While conceptually possible, methods that try to estimate the fill-rates for buffers are problematic, since we don’t have a clear notion of global time in our system. Furthermore, the effectiveness of such methods is dependent on the sampling rate used for measurements. Instead, we chose an approach that does not rely on any explicit measurements.

First, we note that the optimal buffer sizes can be expressed as

$$b_i(t) \propto \sqrt{r_i t}$$

(5)

where $r_i$ is the arrival rate of data into the buffer $i$. For simplicity, let us first consider a case where the memory limit $M$ can grow arbitrarily large, while still keeping the optimal ratios $b_i$ for buffer allocations according to (4). The rate at which buffer sizes need to change in order to keep optimal ratios is then given by

$$b_i'(t) \propto \frac{r_i}{\sqrt{r_i t}} = \frac{r_i}{b_i(t)}$$

(6)

Note that the RHS of equation (6) is proportional to the rate at which buffers become full and need to be committed (cf. (1)). Optimal buffer allocations are thus maintained when
buffer sizes increase proportionally to the commit rate. By incrementing the size of a buffer with a constant $B_{inc}$ every time it becomes full, and periodically “normalizing” the sizes of all buffers to keep within a finite memory limit $M$, we can adjust buffer sizes towards the optimum in an on-line fashion. We now show how this strategy is implemented in our improved algorithm.

Figure 6 shows the basic block preamble of our improved algorithm. Each Bbl record now has a size field, holding the current buffer size of the basic block. We also define a minimum size for buffers, $B_{min}$, to prevent buffers from becoming arbitrarily small. (A buffer must e.g. be guaranteed to be able to hold at least one frame.) A maximum size $B_{max}$ is also defined. We use $B_{min} = 4$ kB and $B_{max} = 2$ MB in our current implementation.

The code for activating a basic block (lines 2–5) is similar to the fixed-size case, with the main exception being that the size field of the Bbl record is used for updating the total memory use. When a buffer is full, the size field of its basic block is incremented with the constant $B_{inc}$ (line 10), as stipulated by equation (6). (Provided that $B_{max}$ has not been reached.) The total memory use is updated with the difference between the new and old size (line 12), and a new buffer is allocated. The case where the buffer is not full (line 18) remains unchanged. For $B_{inc}$, we also use the value 4 kB.

Figure 7 shows our updated alloc_buffer function. It is here that the “normalization” of buffer sizes, to keep within the memory limit, takes place. If the memory usage has reached the hard limit (line 3), our algorithm adjusts buffer sizes by iterating over all active basic blocks (lines 10–12), uniformly decreasing their size field by a constant shrinkage factor $B_{shrink}$, with $0 < B_{shrink} < 1$. Shrinking the actual buffers in-place or, alternatively, committing and re-allocating all buffers, would be prohibitively expensive.

Instead, note that we merely change the size field of each basic block, and lazily change the buffer size upon the next commit (line 15 in Figure 6). Also note that the update to total_memory_use at line 12 in the preamble may therefore be negative.

Since the memory usage is not updated immediately, we need to make some room for buffers to settle in to their new sizes. We achieve this by first evicting the least recently used basic blocks (lines 4 – 9), until the memory utilization is below a soft limit (EVICT_LIMIT in the figure). By penalizing the least frequently used basic blocks, we assure that the impact on performance is minimal. We currently define the soft limit as 90% of the hard limit.

The rest of alloc_buffer remain unchanged, except for the size field being used for allocation at line 14, instead of a fixed size.

Note that our algorithm implements a weighted moving average of sorts, where a value of $B_{shrink}$ close to 1 assigns higher weight to previous updates to the buffer-size ratios $b_i$, whereas a lower value of $B_{shrink}$ allows the ratios to be updated faster to reflect changes in the working set. Lower values of $B_{shrink}$ will, however, lead to lower average memory utilization. We use $B_{shrink} = 0.9$ in our implementation, which appears to be a good tradeoff in practice.

### D. On-line graph generation

On-line dyDDG generation, i.e. constructing the graph continuously during runtime, not only avoids scalability problems (R3), but also improves the usefulness of dynamic slicing. In contrast to systems that generate the dynamic dependence graph after execution has finished, our system can, for example, be integrated with a debugger, allowing developers to interactively compute slices of variables during runtime, when e.g. a breakpoint is hit.

Like most systems that track dynamic data flow, we need to maintain a shadow state [11]. The shadow state mirrors the state of the analyzed program, and contains metadata
about the actual data used during execution. In our case, we need to maintain shadows of all registers and memory locations. A shadow location stores the \( (\text{address}, \text{instance}) \) tuple of the instruction that last defined the corresponding real location. When an instruction \( i \) uses data in a register or memory location \( s \), the defining instruction is fetched from \( \text{shadow}(s) \), and is stored in the current frame. When e.g. instruction 1 in the previous example (Figure 1) is executed, the tuple identifying the defining instruction of \( ebx \) is fetched from \( \text{shadow}(ebx) \), and is stored in the first slot of the frame. Since instruction 1 also writes to \( eax \), a tuple with the \( \text{address} \) and \( \text{instance} \) of instruction 1 is written to \( \text{shadow}(eax) \). When another instruction reads \( eax \), that tuple is in turn fetched. By propagating references to defining instructions in this manner, the dyDDG is successively built as the program executes.

\section*{E. Optimizations}

\textbf{Graph compression.} In [8], Zhang and Gupta identify several types of data and control dependencies that can be inferred statically in program code, and do not need to be stored explicitly in the dynamic dependence graph. They present several optimizations, based on identifying such dependencies, that reduce the size of the dyDDG. The optimization that contributed most to the size reduction was identifying static data dependencies between instructions in the same basic block. Consider the basic block in Figure 1. Instruction 2 adds the registers \( eax \) and \( ecx \), and stores the result in \( eax \). We can statically infer that, when instruction 2 executes, \( eax \) is always defined by the most recent instance of instruction 1. Therefore, edges corresponding to that dependency do not need to be stored explicitly in the graph (i.e. we can remove the second slot in the frame).

We have implemented identification of static dependencies for registers only. Identifying static dependencies on memory is challenging, due to e.g. pointer aliasing. Zhang and Gupta solved the problem by allowing exceptions to static dependencies, which complicates their graph representation, but is feasible if all required data structures can be kept in main memory. In our case, however, the increased complexity would lead to considerably higher overhead, since additional indirection via secondary storage would be needed.

\textbf{Liveness analysis.} As a further optimization, for all register definitions in a basic block, we identify those definitions that are killed before the end of the basic block. Since all dependencies on such definitions must be static, we can avoid updating the shadow state in these cases. For example, the definition of \( eax \) at instruction 1 in Figure 1 is killed by instruction 2. There is thus no need to update the shadow of \( eax \) when instruction 1 executes.

\section*{F. Computing slices}

The dynamic data dependence graph is traversed as described in section II-B in a depth-first manner. To avoid re-visiting already visited parts of the graph, a spanning tree is used. Removing or "tagging" followed edges in the graph on file is not feasible, since that would require destructively re-writing large parts of the graph. Instead, we use per-chunk bitmaps, with one bit per dependence edge in the chunk, to keep track of already followed edges. Bitmaps are allocated on demand the first time a chunk is used, to minimize memory usage.

Due to the strong locality of reference that most programs exhibit, many dependence edges from the same chunk tend to be referenced in quick succession when the dyDDG is traversed. We exploit this phenomenon by maintaining a \textit{cache} of the most recently used chunks during traversal. In section IV we show that our approach is very effective at reducing the number of file system accesses during slice computations.

\section*{III. Implementation}

Our prototype implementation is based on the dynamic binary instrumentation (DBI) framework Pin [12]. Like all DBI systems, Pin performs on-the-fly instrumentation on binary programs as they execute. Instrumentation is performed on single-entry multiple-exit superblocks. The first time a superblock is about to execute, it is passed to an instrumentation callback registered via the Pin API. After the callback has finished inspecting the block, possibly inserting instrumentation, the block is recompiled by Pin and stored in a code cache. Since instrumentation typically only needs to be performed once per superblock, the instrumentation cost is amortized over many executions of the block.

We use Pin's instrumentation facilities to insert the aforementioned basic block preamble, as well as per-instruction instrumentation to record dependence edges and maintain shadow state. Shadow memory is implemented using a two-level table, similar to e.g. [13]. Our implementation currently does not instrument special-purpose instructions, such as SIMD and floating point.

It is also during instrumentation that we determine the size and layout of frames, as well as identifying static dependencies and performing liveness analysis on registers. Note that, despite this analysis being repeated on every execution, it is only performed once per static code block, and is therefore effectively static analysis. In section IV, we show that the cost of our static analysis is negligible.

Some additional effort is also required to correctly record dyDDGs for multithreaded programs. Since updates to the shadow state and real state do not happen atomically, there can be discrepancies between the recorded and real data flow, in cases where several threads access the same memory location. Similarly to e.g. Valgrind [11], we serialize thread execution to avoid this problem. To ensure that only one thread at a time can execute, we apply additional instrumentation, so that a global lock is acquired when entering a superblock, and later released when leaving the block.
To minimize the overhead of storing dependence edges to file, committed buffers are written to secondary storage in a separate thread, using a producer-consumer setup.

IV. EXPERIMENTAL RESULTS

In this section we evaluate the performance of our implementation when recording dynamic dependence graphs, as well as the time taken to compute dynamic slices from these graphs. We performed our experiments on a workstation with an Intel Xeon E3-1245 CPU at 3.3 GHz and 16 GB of RAM, running 32-bit Ubuntu 12.04. The storage device used was a Samsung 840 Pro 512 GB solid-state drive. For the evaluation, we have used the benchmark executions in Table I. We have chosen programs that are representative of various types of software; two compression utilities (gzip and bzip2), a compiler (gcc), a media encoder (libx264\textsuperscript{1}), and a document parser/converter (pdftops). We have selected suitable inputs in order to achieve execution times of a few seconds.

In applicable cases, we have disabled support for SIMD-instructions during compilation, since our implementation currently does not instrument such instructions. We have also chosen to use only single-threaded programs\textsuperscript{2} in our evaluation. The rationale for this decision is that, since our tool serializes execution, using multithreaded programs would complicate measuring the instruction-for-instruction slowdown. We have, however, left the serialization of execution active, to more accurately represent how our implementation would perform in a multithreaded setting.

Columns 4 and 5 of Table I show the size of dynamic dependence graphs, with and without graph compression, respectively. All graphs are tens of gigabytes in size, but graph compression reduces the size of dyDDGs by between 30\% and 63\%.

Graph generation overhead. Figure 8 shows the overhead of creating dyDDGs, with and without optimizations. Graph compression significantly reduces the overhead, both because the amount of added instrumentation is reduced, and because the data volumes that need to be stored to file decrease. The effect of graph compression varies for different programs, with libx264 benefiting the most, and pdftops the least. We also see that the effect of liveness analysis is very modest for most programs. With both optimizations enabled, the slowdown ranges from 62x to 173x in our experiments.

Buffer allocation. Figure 9 compares the overhead of our tool when using the fixed-size and dynamic-size buffer allocation algorithms. 512 MB of memory was used for buffers in the experiments, and both optimizations were enabled. The results clearly demonstrate the need for dynamic buffer adjustments. Less complex software, such as gzip and bzip2, benefit from having very large buffers (1 MB or more), whereas programs with larger working sets benefit from smaller buffer sizes. Performance rapidly degrades with increasing buffer sizes for these programs, since thrashing starts to occur. The tool runs out of memory and crashes when very large fixed-size buffers are used for the gcc and pdftops benchmarks. This is due to severe thrashing, which causes the system to degenerate to storing one frame at a time to file. As a consequence, the chunk map will receive one new entry for every executed basic block, rapidly exhausting memory.

Sources of overhead. The relative contribution to the overhead from different parts of analysis is shown in Figure 10. For these measurements, we wanted to isolate the computational overhead of our tool from the overhead of storing edges to file. We therefore specified /dev/null, which discards all data written to it, as the output file in our

\begin{table}[ht]
\centering
\caption{Benchmark Executions Used in the Experiments.}
\begin{tabular}{|l|c|c|c|c|}
\hline
Program & Execution time (s) & Executed instructions & dyDDG size (GB) & Compressed
\hline
gzip & 1.652 & 7,290,523,400 & 39.720 & 65.987
\hline
bzip2 & 2.308 & 9,937,025,934 & 63.928 & 91.130
\hline
gcc & 1.182 & 4,394,599,917 & 20.983 & 33.322
\hline
libx264 & 1.925 & 12,442,355,696 & 47.474 & 128.850
\hline
pdftops & 2.350 & 15,010,881,410 & 72.349 & 110.083
\hline
\end{tabular}
\end{table}
experiments.

In the figure, results labeled Pin denote the overhead imposed by a minimal Pin-tool, which doesn’t perform any instrumentation. The results labeled Static denote the additional overhead of performing static analysis for graph compression and liveness analysis. Serialize denotes the increase in slowdown caused by serializing execution, and Buffers denotes the added overhead of our dynamic-size buffer allocation algorithm. Finally, Dataflow denotes the increased overhead of actually storing edges into buffers and maintaining shadow state. The figure shows the contribution of each part as the percentage of the total overhead.

The per-instruction instrumentation for storing edges and maintaining shadow state is the main contributor to the overhead for most programs, but the cost of serialization is also high, contributing about 33% of the total overhead on average. Disabling serialization for single-threaded programs could thus improve performance considerably. The results also show that the cost of our static analysis is negligible in most cases.

Buffer management contributes a moderate 14% to the computational overhead on average, showing that our buffer allocation algorithm can adequately perform on-line adjustment of buffer sizes at low cost.

Slicing. To evaluate the performance when traversing dynamic dependence graphs, we computed 1000 data slices at random execution points for each of the five benchmark runs. We used 512 MB memory for caching recently used chunks. Columns 2–4 in Table II show the average slice size, the average number of unique instruction instances per slice, and the average slice computation time. It should be noted, however, that the arithmetic mean may not be the best way to summarize the results, as both the slice size and the number of dynamic instances per slice appear to be distributed according to a power-law distribution; a small number of slices are very large, whereas the vast majority of slices are small. For very small slices, the cost of e.g. allocating data structures dominates over the actual traversal time. The rightmost column shows the average traversal rate (edges per second) for slices with more than 1,000,000 dynamic instances. Due to our optimized graph representation, together with chunk caching, we achieve very high traversal rates of between 4.8 and 8.9 million edges per second.

The cumulative cache miss ratios for all 1000 slices are shown in column 5. All ratios are well below 1%, showing that our caching mechanism is very effective in reducing the access to secondary storage while traversing graphs.

V. DISCUSSION

The problem we address in this paper is fundamentally a design problem, with a potentially infinite number of possible solutions. We believe that the solution presented here achieves a good tradeoff between “flatness” and structure-preservation, as discussed in the introduction. Solutions that did not preserve the graph structure, such as execution traces, were ruled out due to R1 (Retain structure). Adopting the alternate graph representation described in section II-B required using separate buffers for each basic block, in order to satisfy R4 (Aggregate stores), but helped reduce the time and space overhead of our method by allowing a more compact graph format.

To avoid unnecessary indirection (R2: Avoid indirection), we have strive to keep the graph representation as simple as possible. We therefore group dependence edges into frames with fixed size, and use execution counts instead of a global timestamp. These design choices simplify lookups during traversal, reduce the physical size of graphs, and make data-dependence information more readily available. Finally, adhering to R3 (On-line creation) avoided expensive post-processing and enabled future integration with an interactive debugger (section II-D), but required the implementation of a shadow-state tool, to track data-flow during runtime.

<table>
<thead>
<tr>
<th>Program</th>
<th>Slice size</th>
<th>Dynamic instances</th>
<th>Slicing time (s)</th>
<th>Cumulative cache miss ratio</th>
<th>Edges / second</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>18.9</td>
<td>4,225,609.5</td>
<td>0.4667</td>
<td>0.0024%</td>
<td>8,855,272.4</td>
</tr>
<tr>
<td>bzip2</td>
<td>300.2</td>
<td>1,775,472.0</td>
<td>0.1881</td>
<td>0.0058%</td>
<td>8,441,185.6</td>
</tr>
<tr>
<td>gcc</td>
<td>2591.3</td>
<td>3,499,708.1</td>
<td>0.7958</td>
<td>0.2874%</td>
<td>4,831,937.7</td>
</tr>
<tr>
<td>libx264</td>
<td>1118.6</td>
<td>7,466,786.4</td>
<td>1.9262</td>
<td>0.1337%</td>
<td>4,773,879.7</td>
</tr>
<tr>
<td>pdftops</td>
<td>549.7</td>
<td>49,852,315.5</td>
<td>10.0435</td>
<td>0.0146%</td>
<td>5,612,687.5</td>
</tr>
</tbody>
</table>

Figure 10. Relative contribution to the computational overhead from different parts of the analysis.
VI. RELATED WORK

In contrast to our method, which is explicitly designed to use secondary storage for dynamic dependence graphs, Zhang and Gupta’s method [8] aims to compress the graph enough to fit in RAM. While they achieve an impressive 94% reduction in graph size on average, the amount of system RAM still limits the length of executions for which dyDGs can be recorded. Another drawback of their method is that it requires a lengthy postprocessing step to construct the dyDG, which took upwards of an hour in their experiments. A direct performance comparison with our method is difficult, since no slowdown figures are presented in the paper. It can be noted, however, that the number of executed statements in their benchmarks was about two orders of magnitude smaller than the number of executed instructions in our experiments, indicating that our method is at least one or two orders of magnitude faster, taking hardware evolution into account. The authors briefly suggest that their system could be extended to utilize secondary storage, but it is unclear if their graph format would be suitable with regards to the requirements in section II-A.

While Zhang and Gupta’s method works on C programs, Nagarajan et al. [14] suggested a dynamic slicing system for debugging that, like our system, works directly on binary programs. They achieve scalability by only keeping the dyDG edges for the last approximately 20 million executed instructions, which they claim is often sufficient for debugging. Our system stores the entire graph, and is therefore more suitable for applications that require precise dynamic slices.

Slicing techniques for binary programs have received relatively little attention in the literature. Apart from the work by Nagarajan et al., Akgul et al. performed dynamic slicing at the assembly level by means of reverse execution [15]. Since runtime information necessary for computing slices must be reconstructed during the reverse execution, their method has performance limitations similar to methods that e.g., compute dynamic slices from execution traces. Kiss et al. [16] explored static slicing of binaries. However, the size of their static slices turned out to be too large to be practical for most applications.

VII. CONCLUSION

In this paper we have presented a method for storing dynamic data dependence graphs suitable for dynamic slicing of long program executions. We achieve scalability by carefully designing our method to allow storing graphs to file in an on-line fashion. We outline four requirements for efficient utilization of secondary storage when constructing the dynamic data dependence graph, and describe how our method has been designed to fulfill these requirements. In our experiments we show that the method can easily handle cases with billions of executed instructions, resulting in dynamic data dependence graphs that are tens of gigabytes in size, at slowdowns ranging from 62x to 173x. By optimizing our graph representation, and by employing caching to exploit locality, graphs can be traversed at speeds between 4.8 and 8.9 million edges per second.

REFERENCES