Creating Concise and Efficient Dynamic Analyses with ALDA

Xiang Cheng
Georgia Institute of Technology
Atlanta, GA, USA
cxworks@gatech.edu

David Devecsery
Meta Platforms, Inc
Seattle, WA, USA
ddevec@fb.com

ABSTRACT
Dynamic program analyses are essential to creating safe, reliable, and productive computing environments. However, these analyses are challenging and time-consuming to construct due to the low-level optimization required to achieve acceptable performance. Consequently, many analyses are often never realized, or have inefficient implementations. In this work we argue that many analyses can and should be constructed with a high-level description language, leaving the burden of low-level optimizations to the analysis instrumentation system itself.

We propose a novel language for dynamic analysis called ALDA. ALDA leverages common structuring of dynamic analyses to provide a simple and high-level description for dynamic analyses. Through restricting the supported behaviors to only essential operations in dynamic analyses, an optimizing compiler for ALDA can create analysis implementations with performance on-par to hand-tuned analyses. To demonstrate ALDA’s universality and efficiency, we create an optimizing compiler for ALDA targeting the LLVM instrumentation system itself.

We construct 8 different dynamic analysis algorithms, including the popular MemorySanitizer analysis, and show their construction is succinct and simple. By comparing two of them (Eraser and MemorySanitizer) with their hand-tuned implementations, we show that ALDA’s optimized analyses are comparable to hand-tuned implementations.

CCS CONCEPTS
• Software and its engineering → Domain specific languages;
  Software testing and debugging.

KEYWORDS
Domain specific language, dynamic analysis, compiler optimization

1 INTRODUCTION
Dynamic program analyses are essential to productivity in today’s computing environments. They enable dynamic code safety [36], aid in debugging [39, 51, 58], support stronger security primitives [1, 4], and aid in the forensic analysis [17, 24]. However, these analyses’ prohibitive run-time overhead and difficulty in development can limit their use in practice.

Constructing efficient and correct dynamic analyses presents a significant challenge. Many analyses require large amounts of fine-grained instrumentation and introduce complex and unpredictable memory access patterns. Due to the fine-grained control that analysis designers need, state-of-the-art instrumentation frameworks usually operate at a low-level (often assembly, or an IR equivalent). This low-level of abstraction often forces the implementation of even simple algorithms to be complex and difficult to optimize, increasing the burden to build a usable analysis. Not only must the analysis designer work with low-level interfaces [26, 28, 38], but the designers perform a variety of low-level optimizations [37], and carefully construct data-structures [56]. Moreover, as many of these analyses are built with platform specific tools, most implementations are not portable. Once another language or compiler becomes available, the entire process must repeat.

In this work, we argue that this low-level process of building analyses is wasteful and unnecessary. Instead, we propose that analyses should be described algorithmically, at a high-level, while low-level optimizations are left to compilation. Constructing efficient dynamic analyses in high-level languages is appealing for several reasons. First, the descriptions of the analysis will be more succinct, more easily written and prototyped, and less prone to errors. Second, high-level descriptions are more friendly to novice analysis designers, enabling custom analyses for specific problems. Third, analyses constructed in high-level languages can be trivially combined. Today if a user wanted to combine two different dynamic analyses in one run, they would have to re-work each of the analyses to run together, often a daunting task.

Despite the benefits of using a high-level description language for dynamic analysis, such a tool is only practical if it produces analyses which are competitive with hand-tuned implementations. Therefore, such a system must 1) generate highly optimized dynamic analyses that are competitive with hand-optimized implementations, and 2) describe common dynamic analyses succinctly.

However, achieving both of these properties is non-trivial, and requires careful language construction. One particular challenging class of dynamic analyses are heavy-weight dynamic analyses [38]. These analyses are classified by their fine-grained instrumentation and challenging metadata access patterns. Because of these operations, heavy-weight analyses tend to have very high overhead and prove hard to optimize. They include some of the most
used analyses today, like taint tracking [9, 15, 22], data-race detection [18, 43, 45], memory-safety analyses [33, 34], and debugging tools [42].

Heavy-weight dynamic analyses require fine-grained analysis, adding relatively light-weight instrumentation for nearly every instruction in the program [18, 22, 42]. As instrumentation is frequent and the work-per-instrumentation is small, many dynamic analysis designers have found great benefit in low-level, micro-optimizations [37, 38, 56]. This presents both a great challenge and opportunity for a dynamic analysis optimization system. These optimizations require reasoning about behaviors which compilers traditionally struggle to analyze, such as memory access patterns and caching behaviors. However, if the compiler could automatically perform these micro-optimizations, it would free the analysis designer from the burden of repeating this tedious process.

In this work, we present A novel Language for Dynamic Analyses (ALDA). We have designed ALDA specifically for creating succinct and easily optimized descriptions of dynamic analyses. Analyses are written as programs in ALDA. When compiled, these analyses are automatically injected into the application to be analyzed. The workflow of an ALDA analysis is found in Figure 1.

ALDA leverages two insights to achieve our analysis goals of efficiency, succinct representation, and generality to many analyses. First, we recognize that many operations fundamental to dynamic analyses are easily described using two logical primitives: maps and sets. Maps create an association between a program-value and any analysis state associated with that value (or metadata). Additionally, many analyses naturally track sets of data, ensuring all operations operate on data within a valid set. For instance, a use-after-free analysis will ensure that any memory uses do not occur on the “set” of free memory. By making these primitives first-class citizens, ALDA can succinctly represent many dynamic analyses.

Second, we observe that the kernel meta-data updates used by dynamic analyses are often very simple, succinct operations. ALDA need not support many behaviors, such as indirect memory accesses through pointers, which greatly complicate our ability to statically reason about memory accesses. By reducing the core language to a few simple semantics, ALDA dramatically increases its ability to reason about and optimize implementations of these kernel functions, while still retaining the ability to express many of the most-used dynamic analyses.

Based on these insights, we have implemented an optimizing compiler for ALDA in the LLVM compiler infrastructure. We show that our compiler, ALDAcc, can take simple analysis descriptions written in ALDA and produce an optimized analysis code competitive with state-of-the-art and hand-tuned solutions. In fact, our ALDAcc generates code that is comparable with LLVM’s hand-optimized Memory Sanitizer over our benchmark suites. We additionally demonstrate that ALDA is expressive, easily implementing eight different analyses. Finally, ALDA analysis descriptions are succinct, leading to a 83.1% reduction in size versus hand-tuned implementations. In fact, ALDA analyses are so simple and concise, that we argue it enables new targeted analyses which were previously impractical. We demonstrate two such analyses by creating library-specific sanitizers to check the usage of library functions in the SSL and ZLib libraries.

The contributions of the work are as follows:

- We present ALDA, a domain-specific language targeted at representing many analyses and enabling automatic optimization. We show that ALDA can succinctly and clearly represent many dynamic analyses.
- We present ALDAcc, the first optimizing compiler for ALDA targeted at the LLVM compiler framework [26]. We show that ALDAcc can convert the description of many commonly used dynamic analyses into highly-optimized executables, providing competitive speed with hand-tuned analyses at a fraction of the implementation cost.
- We describe several static optimizations for ALDA analyses and show its efficiency.
- We apply ALDA on complex libraries like OpenSSL and Zlib creating two new sanitizers: SSLSan and ZlibSan. We show that these sanitizers’ implementation is simple, yet the analyses still efficiently detect real-world bugs.

2 MOTIVATION

This section highlights the challenges of designing and constructing a heavy-weight dynamic analysis, discussing the design decisions, trade-offs, and complexities an analysis designer must consider when creating these analyses. We restrict our focus to heavy-weight dynamic analyses [38], or analyses which perform expensive and intensive analysis operations. We focus on these analyses as they are highly useful, yet often some of the most challenging analyses to design and implement.

We begin by explaining the standard construction of a heavy-weight dynamic analysis at a high level, then explore the challenges an implementer may encounter when building their analysis.

2.1 Logical Dynamic Analysis Construction

The purpose of a dynamic analysis is to identify a dynamic property of a program by observing the execution of that program. The analysis accomplishes this by adding extra analysis state (called metadata) and monitoring instructions into the program. Heavy-weight dynamic analyses are particularly challenging to write, as they are classified by their tendency to maintain large amounts of metadata for the program’s state, access that metadata in unpredictable patterns, and instrument a wide variety of instructions in different ways [38]. These heavy-weight analyses consist of three components: metadata association, propagation, and monitoring.

1ALDAcc is available at https://github.com/cxworks/ ALDA-analysis
Metadata association provides a mapping from values in the program to their corresponding metadata. This is different for each analysis. For instance, a data-race analysis will only care about per-thread and per-memory location metadata, while a taint-tracing analysis will maintain metadata for every register as well as memory locations.

Metadata propagation Each analysis has its own propagation policy, which defines how the metadata evolves as the program runs. Importantly, these propagation policies tend to be relatively simple when expressed in a high-order logic. For instance, on memory access a lockset-based data-race detector will simply intersect the thread’s lockset with that of the accessed memory location.

Metadata monitoring is the process by which an analysis monitors its metadata and raises a notification if the monitored property is violated. For instance, whenever a memory location’s metadata is updated, a lockset-based data-race detector will check for a race condition by seeing if that location’s lockset is empty.

Logically each of these components is simple. Metadata association can be done with a one of many map data-structures. Propagation is analysis specific, but generally simple, something like taking the union of two sets (taint tracking analysis), or advancing a finite-state-machine (FSM) (Eraser data-race detection). Monitoring is also typically simple, often just ensuring a set is non-empty, or a FSM has not reached some state. However, many challenges arise when we require efficient implementations of these algorithms, dramatically increasing the complexity, and time-cost of analysis construction.

2.2 Efficient Dynamic Analysis Implementation

Unfortunately, naive implementations of dynamic analyses are often an order-of-magnitude or more slower than their optimized counterparts, requiring developers carefully performance-tune their implementations. This is challenging, as these analyses often build upon propagation algorithms so simple that they often consist of a single instruction in the common case (e.g. a logical or [22]). For these applications, analysis costs are dominated by metadata organization and lookup, as well as the low-level caching effects. Optimizing these details are both challenging and time-consuming, requiring a great deal of programmer expertise, and often trial-and-error experimentation. When optimizing an analysis at a low-level, the designer must consider two critical factors:

Metadata Access Patterns: Whenever a propagation event occurs, the analysis will access and manipulate metadata according to the analysis’s propagation policy. However, not all data is always accessed and manipulated on every event. For example, FastTrack’s primary optimization is a summary-based analysis [18], which allows the analysis to access only a single clock-value (often 32-bits) in the common case, but a full vector-clock (100s of bits) in the uncommon. If a programmer naively implements their FastTrack algorithm to look up all metadata at once, then the cache costs of loading vector clocks will dominate the analysis, largely mitigating the improvement of the FastTrack optimization. Instead, for an efficient analysis implementation the programmer must reason at a memory-layout level about access patterns, determine the best layout that balances lookup-costs and cache efficiency, and then carefully access metadata to follow this pattern.

Metadata Storage and Structure Efficiency: Data-structure choice is critical in most applications. Nevertheless, due to the low-level nature of dynamic analysis optimization, these trade-offs become much harder to reason about. Two data-structures, even with equivalent asymptotic complexity may have vastly different performance results. For instance, the critical operation in taint-tracking is merging taints, so the decision of how to represent a taint set is vitally important. If a taint-set is small, it could be represented with a 32-bit bit vector (implemented as an int). This is an efficient data-structure, however it only applies when the taint domain is statically bounded to less than 32. Alternatively, the sparse bit-vector is a data-structure constructed as a linked-list of bit-vector chunks. This structure can represent and union taints of arbitrary sizes with the same asymptotic complexity of a bit vector. However, the sparse bit vector requires indirection, forcing the application to access a larger cache footprint per memory access, giving it far worse performance than the int-based solution. This forces the designer to trade-off flexibility for analysis efficiency. This is also why most efficient taint analyses only support very small taint-sets, and many data-race detectors only support small numbers of locks [45].

3 DESIGN

Our system accomplishes two primary tasks. First, it enables a high-level description of many dynamic analyses, providing the benefits of succinct and simple high-level analyses. Second, it enables the compilation of highly-efficient analysis implementations given these high-level descriptions. To achieve these outcomes, we require ALDA, a high-level analysis description language, and an optimizing compiler.

3.1 ALDA Language

ALDA is constructed with a focus on heavy-weight dynamic analyses. These analyses are far more challenging to construct and optimize than their lightweight counterparts. To support these analyses, ALDA must efficiently achieve all three aspects of heavy-weight analysis construction outlined in section 2.1 while also specifying the analysis in such a way that an optimizing compiler can take advantage of the high-level description to generate efficient low-level implementations. The key to ALDA is our observation that analyses are actually simple at a high-level. Here we give ALDA’s syntax in Figure 2 and will discuss how ALDA supports the different high-level abstractions.

3.1.1 High-Level Description. As outlined in section 2, there are three primary actions attributed to heavy-weight dynamic analyses: metadata association, metadata propagation, and metadata checking. ALDA is structured to facilitate these three operations.

Metadata Association – Each analysis needs to specify the type of metadata present in the analysis, and associate that metadata with program values. The very first section of the program is its type declaration, where programmers specify the types they will use as their metadata. This section of the program can be thought of as a series of typedefs (although with stronger typing), where the programmer makes explicit to the compiler what different types may exist within the analysis.
(program) ::= ⟨stmt⟩*
( stmt) ::= ⟨type-decl⟩ | ⟨meta-decl⟩ | ⟨func-decl⟩ | ⟨insert-decl⟩
(type-decl) ::= ⟨typename⟩ ;='(type) ';' ⟨sync? ';' ⟨number⟩)?
(type) ::= int8 | int16 | int32 | int64 | pointer | lockid | threadid
(meta-decl) ::= ⟨identifier⟩ '=' ⟨meta-type⟩
(meta-type) ::= ⟨specifier⟩ ⟨map-type⟩ | ⟨set-type⟩ | ⟨typename⟩
(set-type) ::= ⟨insert⟩ '(' ⟨typename⟩ ')'
(map-type) ::= map '(' ⟨typename⟩ ',' ⟨typename⟩ | ⟨meta-type⟩ ')
(specifier) ::= universe | bottom | ∈
(func-decl) ::= ⟨typename⟩? ⟨funcname⟩ '(' ⟨func-arg-list⟩? ')' '(' ⟨func-body⟩ ')
(func-arg) ::= ⟨(if-stmt)⟩ ⟨(return-stmt)⟩ ⟨(expression-stmt)⟩*
(insert-decl) ::= ⟨insert⟩ ⟨before | after ⟩ ⟨insert-point⟩
call ⟨func-name⟩ '(' ⟨call-arg-list⟩? ')' 
(insert-point) ::= ⟨func⟩ ⟨identifier⟩ | LoadInst | StoreInst | ...
(call-arg) ::= ⟨call-arg-base⟩ | sizeof '(' ⟨call-arg-base ⟩ ')
(call-arg-base) ::= $⟨number⟩ | Ss | Ss | St

Figure 2: ALDA’s Syntax in eBNF form. We omit some definitions similar to standard C syntax for brevity. A complete syntax can be found in the ALDAcc repository.

Once typing is done, ALDA’s metadata declaration statements specify the metadata in the analysis. ALDA uses map primitive, allowing programmers to declare maps associating metadata with their values. These metadata declarations are highly valuable to optimizing compiler, as the compiler can use these information to optimize memory layout and metadata association structures. For this reason, ALDA does not support compound data types (such as structures or tuples), instead preferring the programmer specify all associations explicitly, and allowing the compiler to later choose ideal compound data types.

Metadata Propagation – ALDA provides an event based metadata propagation policy. The programmers first define event handlers as function declarations and specify instrumentation declarations to connect the events with corresponding handlers. Event handlers have a function-like syntax, in which the arguments pass information about the event to the handler, such as any addresses accessed or any local metadata the operation depends upon and the optional return value is used to propagate local metadata. The body of the function describes the propagation policy, expressed in ALDA’s own syntax.

3.2 Optimizing Compiler

Our compiler takes in a ALDA analysis description, a program representation and compiles them into a program with instrumented analysis code. The compiler retains the original functionality of the program, only supplementing its behavior with the analysis described in ALDA. Furthermore, it is the compiler’s job to leverage the information and flexibility provided by ALDA to optimize the analysis code, memory accesses, and data-structure layout. To accomplish these tasks, the compiler operates in four phases.

3.2.1 Static Analysis. The primary goal of static analysis is to identify the metadata access behaviors of the analysis so later phases of compilation can optimize data-structure selection, metadata mapping layout, and memory accesses. Fortunately, as ALDA restricts memory indirection by disallowing pointers, references, local variables, and loops, this process is far simpler in ALDA than it would
be for a general purpose program. In ALDA the only way to access global metadata is through the program’s global metadata maps. Our analysis can trivially identify these sites by iterating the statements of the analysis body and identifying map look-ups.

We note that the analysis can still not reason perfectly about memory access patterns. If a memory access exists within a branch, that access may or may not occur. Currently the compiler has conservatively assumes all branches will occur. In cases where the branch is rarely or never taken, this may cause the compiler to falsely group together metadata. We are interested in exploring improving this behavior through profile-guided optimizations as future work.

3.2.2 Metadata Layout. Once the compiler has determined both the logical metadata mapping and the memory access patterns associated with those mappings, it is ready to layout the metadata in memory. This process consists of two primary components: First, the compiler determines a structure to organize the analysis metadata. Second, it determines the ideal way to map from program value to analysis metadata.

For primitive data-types, storage choice is trivial, the analysis must maintain enough bits to store the information the analysis requires. For example, if the analysis requires a lock identifier with a domain of 32 values, the compiler must assign at least 5 bits of storage to uniquely identify a lock. However, for non-primitive types (in particular sets) selecting the ideal data-structure to store the value in question is far more nuanced. If the domain of the set is finite and small, then the compiler will choose a highly efficient bit-vector. However, if the set’s domain is large, a less efficient dynamically sized set representation may be required.

Once the structure of all metadata is known, the compiler determines the best way to map that metadata to its associated program state. In particular, our compiler performs two optimizations here. First, it determines which sets of data should be co-located to reduce caching and metadata lookup costs. Second, it determines the best structure as an implementation of this map.

To determine co-location of data-sets, the compiler relies on its static memory access pattern analysis. If it detects that two metadata values are accessed together with the same key, then the compiler will attempt to group those metadata elements on the same cache line, ensuring that look-ups to the data are fast. Once this grouping has been done, the compiler now knows the value types (both data-structure representation, and co-located groupings) in each of its maps.

Finally, the compiler determines the correct type of map to use. This mapping is critical to performance [37]. However, the best choice of mapping is both data and language-specific. In a type-unsafe language like C, type-safety is not guaranteed and types cannot reliably be statically determined. Therefore, the dynamic analysis cannot know how a memory region will be accessed and must generate a generic mapping of metadata to memory addresses. However, for type-safe languages, such as Java, it is practical to embed the metadata within the object itself, dramatically reducing lookup costs.

Besides the language, the structure of metadata can also affect this map. Imagine creating global mappings for a type-unsafe language. If a map’s key-domain is large (e.g. pointers), it would not make sense to use a hash map because the structure would consume valuable memory resources and provide poor cache locality. A compiler should instead choose a pagetable based map for better performance. ALDA provides tools to aid the compiler in these operations.

ALDA compilers can leverage its type declaration infer a type’s domain size. This allows the compiler to determine when a map key’s domain is small and allocate data-structures accordingly.

3.2.3 Event Handler Generation. Once the metadata layout and mapping have been decided, the compiler is ready to generate the event handler code. The majority of this process is a standard compilation of the handler’s body, however, our compiler implements one notable optimization: metadata lookup minimization. Through its static analysis, the compiler has appropriate knowledge of what metadata maps will be accessed by the analysis at different points in an event-handler body. To minimize lookup costs and memory pressure, the compiler coalesces these into a single metadata lookup when possible (e.g. two keys are logically equivalent, and the maps are co-located).

3.2.4 Event Handler Insertion. Finally, the compiler inserts the event handlers and metadata into the original program. This process is fairly straightforward and target dependant (e.g. Java code will be instrumented in a different manner than an C code). After the event handlers have been inserted, the resulting binary will contain not only the original code, but also the optimized analysis code and is ready for execution.

3.3 Limitations of ALDA

The goal of the ALDA language is to allow expression of many common dynamic analyses, while shifting the burden of analysis optimization from the programmer to the compiler. ALDA provides a fundamental trade-off, in that the language is intentionally not as expressive as general-purpose languages, particularly around indirect memory accesses and loops. However, these restrictions dramatically increase an ALDA compiler’s ability to reason about memory access patterns, improving the compiler’s optimization abilities. Anecdotally, we found this trade-off to be minimally intrusive for all of the common dynamic analyses we attempted to construct in ALDA.

We have designed ALDA to succinctly represent most commonly-used dynamic analyses, however, some behaviors cannot be expressed naturally in ALDA. For these behaviors we have provided escape hatches, such as external function call support. These escape hatches allow programmers to express analyses which do not naturally fit within ALDA at the potential cost of (1) optimization, and (2) loss of portability.

Finally, ALDA, is designed to allow construction of an optimizing compiler on top of it. However, care must be taken to construct an optimizing compiler for each target language ALDA wishes to support. We discuss the design of one such compiler in section 5.

4 ALDA LANGUAGE SPECIFICATION

As mentioned in section 3.1.1, ALDA programs can be thought of as having roughly four components: (1) type declarations, which indicate what types of data will be used in the analysis; (2) metadata
address := pointer : sync

tid := threadid : 4

status := int8

thread2WLock = universe::map(tid, set(lid))

thread2Lock = universe::map(tid, set(lid))

addr2Lock = universe::map(address, universe::set(lid))

addr2Thread = universe::map(address, set(tid))

addr2Status = universe::map(address, status)

onLoad(address addr, tid t) {
    if(!addr2Thread[addr].find(t)
        & & addr2Status[addr] != VIRGIN){
        if(addr2Status[addr] == EXCLUSIVE)
            { addr2Status[addr] = SHARED; }
        addr2Thread[addr].add(t);
    }
    if(addr2Status[addr] > EXCLUSIVE){
        addr2Lock[addr] = addr2Lock[addr] &
        thread2Lock[t];
    }
}

onStore(address addr, tid t) {
    if(!addr2Thread[addr].find(t){
        addr2Thread[addr].add(t);
        if(addr2Status[addr] == SHARED)
            { addr2Status[addr] = SHARED_MODIFIED; }
        if(addr2Status[addr] == EXCLUSIVE)
            { addr2Status[addr] = SHARED_MODIFIED; }
        if(addr2Status[addr] == VIRGIN)
            { addr2Status[addr] = EXCLUSIVE; }
    } else {
        if(addr2Status[addr] == SHARED)
            { addr2Status[addr] = SHARED_MODIFIED; }
    } if(addr2Status[addr] > EXCLUSIVE){
        addr2Lock[addr] = addr2Lock[addr] &
        thread2WLock[t];
    }
}

Listing 1: State machine transformation of Eraser algorithm.

Virgin, Exclusive, Shared and Shared-Modified are 4 states defined in Eraser’s algorithm.

4.1 Type Declaration

ALDA currently supports six primitive types, shown as type in Figure 2. Four of these types are integer types with different lengths. Pointer types are used to indicate pointer values in ALDA programs, as pointer sizes vary across architectures. Likewise, to handle language-specific lock/thread implementations, ALDA defines special lockid and threadid. Since most dynamic analysis do not regularly use float types, ALDA does not yet support them, although it could in the future.

Additionally, ALDA supports two specifiers for primitive types: sync and number. The sync keyword is used to indicate a value should be locked whenever it, or a map or set containing it is accessed. There is an inherent cost of providing the isolation associated with locks. The use of sync can be seen in Listing 1 line 1, when the address is marked as sync, all the metadata that using it as a key (like addr2Lock, addr2Thread) will be protected by a lock. The number specifier allows a programmer to specify that a value has a limited range, somewhat like a bitwise width in a C structure. By limiting the range of some values, ALDA can better optimize the memory layout of some structures, particularly when limiting the domain of map keys or set values. For example, in Listing 1 line 4, the domain size of locks in the program is limited to 256. With this information, ALDAcc can use a fixed-size data structure, such as a bit-vector, to compress the memory usage and improve cache locality.

4.2 Metadata Declaration

The metadata declaration section indicates the type of metadata presented in the analysis and associates that metadata with program values. The key primitives that ALDA uses to allow succinct and highly-efficient analysis implementations are the set and map primitives. Maps are used to store the mapping between the original program data and metadata while sets are a commonly used structure within analyses, that ALDA supports natively for optimization purposes.

Additionally, ALDA supports two different types of initial states for metadata, universe and bottom or e. The universe qualifier means that the collections initially contain all the entities in the domain of the collection, just like the definition of universe set U in mathematics. This is useful in many analyses, for example Eraser algorithm assumes each memory address holds all the locks initially. Bottom or e initialized collections to be empty. ALDA additionally provides a fairly standard set of operators on set and map abstractions listed Table 1. These interfaces are straightforward and provide succinct and commonly-used interfaces for dynamic analysis to interact with metadata.

4.3 Event Handler Declaration

This part of ALDA program defines the logic for how analysis handles an exact event with its metadata. The analysis may also raise any warnings or errors if appropriate. ALDA allows this declaration through a function syntax, very similar to C functions. The syntax
First, is the event handler declaration as a function. Functions’ arguments are used to indicate the time to instrument. (e.g. A data race analysis needs to instrument after a thread grants a lock, but before it releases it). The insert-point is used to specify the location to instrument: either a program function call such as malloc, or an instruction (e.g. load, or add). Finally, insert-decl describes how arguments are passed from the program to functions. ALDA provides syntax (noted as call-arg) to address this connection, details are shown in Table 2. For example, the onMalloc function’s insertion point is found on line 32 of Listing 2. We define the arguments passed to onMalloc’s insertion declaration as $r$ and $1$. $r$ indicates the first argument to the malloc function (allocation size), and $1$ indicates malloc’s return value.

4.4 Insertion Point Declaration

The last part of an ALDA program tells the compiler how to instrument the event handlers into programs. Insertion point declarations answer a question: when and where to instrument which event handler function. The syntax definition of this part can be found in Figure 2 at the insert-decl statement. The keywords before and after are used to indicate the time to instrument. (e.g. A data race analysis needs to instrument after a thread grants a lock, but before it releases it). The insert-point is used to specify the location to instrument: either a program function call such as malloc, or an instruction (e.g. load, or add). Finally, insert-decl describes how arguments are passed from the program to functions. ALDA provides syntax (noted as call-arg) to address this connection, details are shown in Table 2. For example, the onMalloc function’s insertion point is found on line 32 of Listing 2. We define the arguments passed to onMalloc’s insertion declaration as $r$ and $1$. $r$ indicates the first argument to the malloc function (allocation size), and $1$ indicates malloc’s return value.

Second, ALDA restricts the function’s body to only a small subset of C-like statements: if statements, return statements and expression statements. This restriction has several notable impacts. On the one hand, ALDA has no support for indirection outside of its set and map primitives, disallowing pointers and reference types. On the other hand, ALDA does not support any type of looping. We have found that very few dynamic analyses actually need or heavily use these behaviors. However, because of these restrictions, an intelligent ALDA compiler can reason very precisely about data access patterns within a metadata, allowing for very powerful optimizations.

Finally, we note that as a part of ALDA’s expression statement, ALDA allows external function calls to C code within function metadata bodies. We allow this in ALDA for two reasons. First, there are some rare instances when analyses require looping or indirection, we allow limited support through external calls. Second, ALDA has no support for indirection outside of its set and map statements.

### Table 1: Builtin interfaces and functions in ALDA

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>Refer to the i-th parameter of the original function or operand of the instruction</td>
</tr>
<tr>
<td>$p$</td>
<td>All the elements in the insert point (i in range [1, ..., n])</td>
</tr>
<tr>
<td>$r$</td>
<td>Return value of the instrumented function</td>
</tr>
<tr>
<td>$t$</td>
<td>Current thread id</td>
</tr>
<tr>
<td>$X.m$</td>
<td>The local metadata value for a given argument</td>
</tr>
<tr>
<td>sized($X$)</td>
<td>Memory size in bytes of the argument in original program</td>
</tr>
</tbody>
</table>

### Table 2: call-arg syntax breakdown.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>Refer to the i-th parameter of the original function or operand of the instruction</td>
</tr>
<tr>
<td>$p$</td>
<td>All the elements in the insert point (i in range [1, ..., n])</td>
</tr>
<tr>
<td>$r$</td>
<td>Return value of the instrumented function</td>
</tr>
<tr>
<td>$t$</td>
<td>Current thread id</td>
</tr>
<tr>
<td>$X.m$</td>
<td>The local metadata value for a given argument</td>
</tr>
<tr>
<td>sized($X$)</td>
<td>Memory size in bytes of the argument in original program</td>
</tr>
</tbody>
</table>

of these function declarations can be found in Figure 2 as func-decl elements. Listing 2 has examples of these functions in onMalloc, onFree, and onBranch. There are three interesting aspects of the function bodies we outline here.

First, is the event handler declaration as a function. Functions’ arguments are used to receive data from the underlying program. This can include the operands of the instrumented operation, metadata associated with any local values used by the instrumented operation, or any needed evolving global state, such as the current thread’s identifier. Meanwhile, functions’ return value is optional. The definition of how program state is translated into these arguments and return value is propagated are discussed in section 4.4.
5 IMPLEMENTATION

We have constructed an optimizing compiler for ALDA using the LLVM 6.0.1 backend [26], we call this implementation ALDAcc. ALDAcc takes a ALDA analysis description, converts it into an abstract syntax tree, performs optimizations discussed in section 3.2. Once these optimizations have taken place, the compiler instruments the analyzed program, and outputs an instrumented binary. In this section, we talk about ALDAcc’s implementation details and limitations.

5.1 Analysis Granularity

Currently ALDAcc defaults to word-based (8 bytes on 64bits machine) metadata granularity. This means it will create 1 metadata unit representing 8 bytes of address-space in the program. Any accesses to sub-word granularity data (e.g. chars in C) will coalesce their access into the word representing their metadata. Word-based metadata tracking is common as it provides a trade-off between accuracy and performance. ALDAcc can also be configured to handle metadata at byte, quarter-word and half-word granularity.

5.2 Map Coalescing

With the information gathered by the static analysis, ALDAcc optimizes the analysis’s metadata layout. Specifically, ALDAcc performs two optimizations: metadata coalescing and data structure selection. Metadata coalescing aims to group metadata together to reduce map look-ups, improve data locality, and optimize cache-hit rates.

ALDAcc bases its coalescing of maps on the key-type of the map, merging multiple maps with equivalent keys into a single map. This key-type based lookup is more aggressive than, say using access-patterns to determine which maps are accessed together. However, we find this is effective for ALDAcc, as the data-space savings from reducing the per-map structure overhead often outweigh the cache-savings in the rare case combined values are not all used. To achieve this merge, ALDAcc goes through the global metadata maps declared by users and groups the maps based on their key types. Then, for each new map our compiler updates the event handlers to access this new map.

5.3 Shadow Memory Selection

After metadata coalescing, all the metadata types are fixed and ALDAcc selects data structures for the metadata. Currently, ALDAcc selects map and set data-structures based on the following factors: the domain size of a map or set, synchronization status, actual size of its value (for maps), and the granularity of the analysis. For sets, ALDAcc prefers a bit-vector if the set is small (less than 512 bytes), and fixed. We found that when a set is not of fixed size, it is rarely critical for performance, so, ALDAcc defaults to a tree-based set as they are the most flexible. For maps, ALDAcc prefers an array for maps of limited domain size. For address-space sized maps ALDAcc selects between a virtual-memory based shadow-memory and a page-table like data-structure, depending on the size of each map element and analysis’s granularity.

Offset based shadow memory has better performance but higher memory usage than a page-table like data structure [37, 57]. ALDAcc seeks out a balance between performance and memory utilization, allowing analyses with large metadata requirements to

run, while also highly optimizing those with smaller metadata. To achieve this, ALDAcc chooses its structure based on an individual metadata map’s value size and analysis granularity. More specifically ALDAcc defines a value known as the shadow factor, which is the amount of metadata that a single byte of program address space will consume, after accounting for the analysis’s metadata granularity. If this factor is greater than some threshold (3 by default), then ALDAcc uses a page-table based solution for memory efficiency. If the factor is less than that value, ALDAcc chooses performance and the virtual-memory based solution. For example, in Listing 2, the shadow factor for the map is 1 because the size of the value is 1 byte and the analysis granularity is 1 byte. So in this case, ALDAcc will choose a shadow memory based map implementation.

5.4 Metadata Lookup Reduction

Once metadata is laid out, ALDAcc determines optimal lookup patterns to minimize metadata lookup costs. Poorly optimized metadata lookup patterns can slow the program through wasted computation and cache misses. ALDAcc optimizes using common sub-expression elimination (CSE). Specifically, ALDAcc searches for common sub-expressions in map lookup statements, by iterating through the statements to find redundant look-ups. The compiler’s internal representation then creates a local variable storing the reference of the map lookup result and replaces all common lookups with that reference.

5.5 Event Handler Instrumentation

Finally, ALDAcc is tasked with instrumenting the specified analysis code into the analysis binary. ALDAcc uses the LLVM framework to instrument the analysis into the analyzed program. First, ALDAcc compiles the event handler bodies into C++ functions. Then it inserts these handlers into the analyzed program at the appropriate location (either before or after each analyzed event depending on the event description in ALDA), inlining when appropriate.

Additionally, ALDAcc adds function-local and language-specific tracking when appropriate. This includes adding metadata tracking for dynamic, local metadata structures (e.g. LLVM’s register variables and variadic arguments), as well as identifying C standard library calls and adding the appropriate calls for them. Once ALDAcc has finished instrumenting the program, it compiles the analysis functions, and links them together with the newly instrumented application binary.

5.6 Limitations

While ALDAcc seeks to create an optimizing compiler for ALDA, our implementation does currently have some limitations.

5.6.1 LLVM Backend. ALDAcc is implemented on LLVM, a static instrumentation framework. Therefore, ALDAcc cannot instrument dynamically loaded libraries. This limitation can be solved by either porting ALDA to a dynamic instrumentation framework, such as Pin[28] or provide handlers for all used external functions. Currently ALDA picks the later one.

Additionally, ALDAcc does not yet support precise metadata tracking for LLVM’s vector operations, since the vector type in LLVM IR doesn’t map to a type in C. To resolve this inconsistency, we move the compiler’s vectorization optimization after ALDAcc’s
instrumentation only when running with ALDA to ensure ALDAcc instruments event handlers with the correct granularity.

5.6.2 External Function Calls. ALDA provides built-in handlers for commonly used libc functions. However, there are some functions, such as printf and scanf, that ALDA cannot naturally handle due to its limited syntax. ALDA supports using external C function calls to handle these complex behaviors. External functions may either translate their logic into load and store operations that ALDAcc will optimize, or they may use a slow, optimized metadata reading and writing interface.

6 EVALUATION
In this section, we seek to answer following questions:

- How well can ALDAcc optimize code, and can it compare to highly-optimized hand-tuned analyses?
- Can ALDA represent common dynamic analyses?
- Can ALDA construct new analyses quickly and efficiently?

6.1 Experiment Setup
To evaluate ALDA, we create a series of analyses, compile them using ALDAcc, and then test them across our benchmark suites. Our benchmark suites include Splash2 and SPECint 2006, as well as four real-world programs: nginx, memcached, sort and ffmpeg.

We use SPECint to evaluate computationally intensive single-threaded workloads and Splash2 for multi-threaded performance. All SPEC experiments are run with the reference inputs. We modified our Splash2 inputs to increase runtime to a few seconds under two threads as the default Splash2 inputs are too short for meaningful benchmarking. We exclude ocean_nc and water_spatial since we could not find a reasonable input to evaluate. For nginx we benchmark with Apache benchmark launching 16 threads to make 1 million requests. For memcached, we use the memtier_benchmark developed by redis labs. Sort is a Linux program in GNU coreutils program, using multi-threading sort algorithm. We generate a 121MB random number text file as our test data. For ffmpeg we pick a 121MB H264 video from its official samples. We use SPECint to evaluate computational intensive single-threaded workloads and Splash2 for multi-threaded performance. All SPEC experiments are run with the reference inputs. We modified our Splash2 inputs to increase runtime to a few seconds under two threads as the default Splash2 inputs are too short for meaningful benchmarking. We exclude ocean_nc and water_spatial since we could not find a reasonable input to evaluate. For nginx we benchmark with Apache benchmark launching 16 threads to make 1 million requests. For memcached, we use the memtier_benchmark developed by redis labs. Sort is a Linux program in GNU coreutils program, using multi-threading sort algorithm. We generate a 112MB random number text file as our test data. For ffmpeg we pick a 121MB H264 video from its official samples to test encoding performance. We run these applications with 4 threads/processes.

We launch all the experiments on a virtual machine with four Intel(R) Xeon(R) CPU E5-2630 v4 cores and 32GB memory under Ubuntu 18.04 Desktop LTS. We run each test six times and report the geometric mean of the last five executions as our final result. Then we normalize the result to the original program execution.

6.2 Performance
To evaluate the performance of ALDAcc, we first conduct two experiments comparing ALDAcc with heavily optimized, hand-tuned programs. We then evaluate the impact of ALDAcc’s optimizations.

MSan. First, we compare ALDAcc to MemorySanitizer [47], a widely used and highly optimized memory safety analysis in LLVM. We reproduced MSan’s algorithm in ALDA. To verify the correctness of our re-implementation, we ran MSan’s unit tests on our ALDA MSan and verified the outputs were correct. Besides, we manually checked the errors reported by LLVM MSan and ALDA MSan finding they are equivalent. These errors are listed in Table 3 and we exclude these error programs (gcc in SPECint and barnes, ocean_c, volrend in Splash2) from our benchmark as they have uninitialized memory use errors, which cause MSan to stop the test prematurely.

Our results are found in Figure 3. Overall, these results are positive, with ALDAcc showing similar performance with MSan among all the benchmarks (on average 2.21x for ALDAcc and 2.29x for LLVM). We additionally analyzed the generated code and found MSan and ALDAcc have very similar generated LLVM IR code after compiler optimization, with only slight deviations in metadata memory layout. This divergence causes MSan to experience several cache misses in critical portions of libquantum that ALDAcc does not. The speedup in ALDAcc comes from slightly more efficient cache behavior on our target machine, but the two systems would likely perform similarly if run across a wide variety of machines. We also measured the memory overhead for MSan and ALDAcc, and found the two to have roughly equivalent memory footprints.

Eraser. The second implementation we evaluate is our hand-optimized version of the classic Eraser data-race detection algorithm [43]. We optimized Eraser with hash-based locking operations, static tables to represent state transformations, and careful data-structure selection. We then built the same algorithm with ALDA and compiled it using ALDAcc. We compare the two analyses across the Splash2 benchmark suite.

The results can be found in Figure 4, runtimes are normalized to baseline overhead and all results are statistically significant at 95% confidence. ALDAcc achieves 24.79x overhead, which is comparable with our hand-tuned version with an average of 25.12x overhead. We attribute this difference to the inline function calls and metadata layout. The metadata memory overhead of ALDAcc is also nearly identical between the two implementations.

Table 3: Error report validation of MemorySanitizer

<table>
<thead>
<tr>
<th>Program</th>
<th>Location</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fmm</td>
<td>imm.c:313</td>
<td>LLVM MSan doesn’t intercept “gets” function, leads to false positive.</td>
</tr>
<tr>
<td>Barnes</td>
<td>gelparam.c:35</td>
<td></td>
</tr>
<tr>
<td>Ocean_C</td>
<td>multi.c:261</td>
<td>Uninitialized memory use reported by both ALDA and LLVM MSan.</td>
</tr>
<tr>
<td>Volrend</td>
<td>main.c:503</td>
<td></td>
</tr>
<tr>
<td>Gcc</td>
<td>sbitmap.c:349</td>
<td></td>
</tr>
</tbody>
</table>

8We run LLVM 6.0.1’s MSan with the flags disabling handle-icmp, check-access-address, handle-ioctl and enabling strict-icmp, strict-memcmp.
Table 4: ALDA is able to represent multiple analyses with few lines of code.

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Name</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eraser</td>
<td>70</td>
<td>MSan</td>
<td>192</td>
</tr>
<tr>
<td>UseAfterFree</td>
<td>35</td>
<td>StrictAliasCheck</td>
<td>12</td>
</tr>
<tr>
<td>FastTrack</td>
<td>69</td>
<td>TaintTracking</td>
<td>33</td>
</tr>
</tbody>
</table>

Metadata Layout Optimization. We additionally evaluate the efficiency of our metadata optimizations. For this evaluation, we turn off our map coalescing and metadata lookup reduction (CSE) optimizations and re-ran the Eraser analysis on SPLASH2 benchmark. We choose the Eraser benchmark over MSan as MSan only uses a single metadata map, making map coalescing irrelevant. The result is shown in Figure 4 as *ds-only* bars and the average speed up for our metadata layout optimizations are 26.9%. Unfortunately, we could not separate the CSE and map coalescing optimizations as their implementation in ALDAcc is intertwined. We also could not quantitatively evaluate the impact of our data-structure selection optimization, as non-trivial benchmarks ran out-of-memory when run without the optimization.

6.3 Generality
The second aspect we evaluate is ALDA’s ability to represent a variety of dynamic analyses. We look at both the breadth of analyses we can represent in ALDA, and how succinctly it can represent these analyses. For this, we use ALDA to implement 6 different dynamic analysis algorithms and approximate their complexity via lines of code (LOC). These algorithms can be found in Table 4. However, since we can’t find reasonable implementations of these algorithms except for Eraser and MSan, we only compare our LoC with these two algorithms. Specifically, ALDAcc’s implementation only requires a fraction of the effort compared to LLVM MSan, which takes 8146 LOC and to hand-tuned Eraser, which requires 690 LOC.

6.4 Newly Enabled Analysis
Finally, we demonstrate new use-cases for dynamic analyses enabled by ALDA.

6.4.1 Library-Specific Sanitizers. The first use-case we consider, is the creation of library-specific sanitizers. Currently sanitizers require too much developer effort build for niche use-cases. However, as we have shown, ALDA enables simple, concise, and relatively low effort sanitizer construction. To demonstrate this, we consider the potential impact of custom sanitizers for complex library interfaces. In particular, we have built SSLSan, targeting the OpenSSL library [50], and ZlibSan, targeting ZLib [20]. These sanitizers are constructed to identify memory-leak, uninitialized-memory-access, use-after-free and misuses stemming from the libraries.
To demonstrate the utility of SSLSan, we run our SSLSan on memcached and nginx, validating two bugs [27, 49] in memcached and one in nginx [31]. The first bug was a memory leak caused by improper SSL interface usage. However, because the bug occurred within the SSL library, general-purpose dynamic analysis tools, such as valgrind, failed to identify the existence of the bug. Instead, we constructed SSLSan in 177 lines of ALDA code and found it can easily detect this bug. We additionally detected two misuse bugs resulting from improper closing of SSL ports.

Besides, we ran ZlibSan on ffmpeg, validating one uninitialized memory access bug[32]. These tools show that ALDA allows simple construction of powerful sanitizers, capable of identifying bugs and interface misuses in mature codebases.

6.4.2 Combined Analysis. The second scenario enabled by ALDA is that of combining multiple analyses together, to run them all on a single execution. Users often wish to run multiple different analyses on the same execution during debugging or testing. However, different types of analyses generally do not compose cleanly together (in clang, it is impossible to combine any two of the TSan, ASan, or MSan at the same time), so programmers must instead run each analysis individually. However, ALDA easily supports the composition of analyses, and due to its automatically optimizing nature, can actually implement combined analyses more efficient than any independent analysis. To demonstrate this capability, we combine two data race detectors (Eraser and FastTrack) along with a taint-tracking analysis and a use-after-free checker into a single analysis. This combination is as simple as concatenating our 4 ALDA analysis source files into a single file.

Our evaluation uses the Splash2 benchmark and three real world programs (we exclude SPEC and Nginx as they are not multi-thread). The results can be found in Figure 5. For each program, we run Eraser, FastTrack, use after free (UAF), and index taint tracking (IndexTT) individually. Then we use ALDA to construct a combined analysis, running all four at once (Combined). Remarkably, we see that ALDAcc cannot only trivially combine and run these analyses, but by optimizing the analyses together, ALDAcc reduces the overall cost of running the combined analyses vs running each individually. This ultimately results in a 44.9% speedup on average.

7 RELATED WORK

ALDA is the first high-level language designed specifically for general-purpose dynamic analysis construction, with data-structure layout and metadata access optimization in mind.

Prior works have proposed several frameworks for dynamic analysis construction[13, 14, 30, 46, 54]. However, none of these works provide automatic optimization, relying on the programmer to build any needed low-level optimizations. GemS[14] is a generic source code level instrumentation framework aiming to provide a generic model for cross language, platform instrumentation. Its IR transformer does not explicitly analyze or optimize on instrumented code. There also exist many static and dynamic binary instrumentation toolkits [6-8, 10, 19, 26, 35, 48, 52, 55], but these tools are primarily focused on low-level instrumentation efficiency and ease, not concisely describing or optimizing high-level dynamic analyses. For instance, the CSI framework [44] defines the APIs like ALDA’s insertion point to instrument dynamic analysis into the program. However, these frameworks do not consider the performance or data-layout of any complex analysis behaviors, pushing the burden of low-level optimization onto the programmer. By contrast, ALDA leverages its restricted language-level syntax to not only create metadata mappings, but also automatically optimize the instrumented code.

Several frameworks have attempted to optimize dynamic analyses by including heavy-weight static optimizations [9, 16, 19]. However, these analyses are focused on removing dynamic instrumentation calls through heavy-weight static analysis. For example, MDL[19] is a language framework focusing on collecting data during program runtime. Its online instrumentation only collects the data requested by user and does not contain the analysis logic. ALDAcc instead optimizes each instrumentation call individually, however does not focus on reducing the number of instrumentation sites in the program.

Many other works have focused on dynamic analysis optimization [2, 21, 23, 25, 37, 57]. These works typically propose manual or profile-guided transformations that can aid a developer in constructing a more efficient analysis. Umbr[a][57] and Shadow[37] provide a detailed study on the efficiency and performance of shadow memory on both 32-bit and 64-bit machines. However, they mainly focusing on the use of shadow memory, ALDAcc automatically optimizes metadata layout, including use of shadow memory if appropriate.
High-level languages, such as Datalog, have been used to enable static analyses [3, 29, 40, 53]. However, dynamic analyses often use features such as finite-state machines which cannot trivially be represented in Datalog. Additionally, the optimization of dynamic analyses and their static counterparts are very different.

Finally, some tools have focused on generating dynamic analyses from high-level, prose-like descriptions [5, 11, 12]. For example, MOP is a domain-specific language targeting to automatically generate runtime monitors while program is running. However, MOP is focused on constructing analyses from prose-like descriptions, not on the metadata layout (map and set primitives) and computational optimizations that ALDA considers. For instance, JavaMOP, achieves efficient metadata performance by relying on Java’s strong typing, allowing it to co-locate data and metadata. When applying MOP to weakly-typed languages like C/ C++, these optimizations would not be possible, and MOP would have to tackle the metadata layout challenges addressed by ALDA.

8 CONCLUSION

This paper has argued for the use of high-level descriptions to construct many dynamic analyses. We have described ALDA, a language designed specifically to enable high-level descriptions of dynamic analyses, and ALDAcc, our compiler for ALDA built with language designed specifically to enable high-level descriptions of dynamic analyses, and ALDAcc, our compiler for ALDA built with LLVM. We have shown that ALDAcc is successful in building a wide array of useful dynamic analyses, with performance comparable to state-of-the-art implementations. We look forward to improving and advancing ALDA and ALDAcc.

ACKNOWLEDGMENTS

We thank our shepherd, Michael Bond, and other anonymous reviewers for their constructive comments and feedback. We thank our anonymous artifact reviewers for their patience and suggestions. ALDA’s artifact is publicly available at https://doi.org/10.5281/zenodo.5748338.

REFERENCES

[29] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to flix: comprehensive runtime code manipulation and optimization tool to implement adaptable systems. Adaptive and extensible component systems. IEEE, 249–256.

Creating Concise and Efficient Dynamic Analyses with ALDA

ASPLOS ’22, February 28 – March 4, 2022, Lausanne, Switzerland