Predicting Dynamic Properties of Heap Allocations using Neural Networks Trained on Static Code

An Intellectual Abstract

Christian Navasca
University of California, Los Angeles
Los Angeles, USA

Martin Maas
Google
Mountain View, USA

Petros Maniatis
Google
Mountain View, USA

Hyeontaek Lim
Google
Mountain View, USA

Guoqing Harry Xu
University of California, Los Angeles
Los Angeles, USA

Abstract

Memory allocators and runtime systems can leverage dynamic properties of heap allocations – such as object lifetimes, hotness or access correlations – to improve performance and resource consumption. A significant amount of work has focused on approaches that collect this information in performance profiles and then use it in new memory allocator or runtime designs, both offline (e.g., in ahead-of-time compilers) and online (e.g., in JIT compilers). This is a special instance of profile-guided optimization.

This approach introduces significant challenges: 1) The profiling oftentimes introduces substantial overheads, which are prohibitive in many production scenarios, 2) Creating a representative profiling run adds significant engineering complexity and reduces deployment velocity, and 3) Profiles gathered ahead of time or during the warm-up phase of a server are often not representative of all workload behavior and may miss important corner cases.

In this paper, we investigate a fundamentally different approach. Instead of deriving heap allocation properties from profiles, we explore the ability of neural network models to predict them from the statically available code. As an intellectual abstract, we do not offer a conclusive answer but describe the trade-off space of this approach, investigate promising directions, motivate these directions with data analysis and experiments, and highlight challenges that future work needs to overcome.

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **Allocation / deallocation strategies**.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ISMM '23, June 18, 2023, Orlando, FL, USA

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0179-5/23/06.
https://doi.org/10.1145/3591195.3595275

Keywords: Machine Learning, Profile-guided Optimization, Lifetime Prediction, Memory Management

ACM Reference Format:
Christian Navasca, Martin Maas, Petros Maniatis, Hyeontaek Lim, and Guoqing Harry Xu. 2023. Predicting Dynamic Properties of Heap Allocations using Neural Networks Trained on Static Code: An Intellectual Abstract. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management (ISMM '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3591195.3595275

1 Introduction

Memory allocators and runtime systems often rely on predicted properties of heap allocations to maximize performance. For example, HALO [52] uses memory access profiling to identify related data accesses, which can be used for heap-layout optimizations. MaPHeA [45] places data based on allocations' memory access frequencies (*hotness*). The LLAMA C++ memory allocator [41] and the ROLP Java garbage collector [11] rely on predicted object lifetimes.

These approaches have parallels to profile-guided optimization (PGO), which in this paper, we take to refer to both offline (e.g., ahead-of-time) and online (e.g., JIT-compiled) approaches. For example, both static ahead-of-time compilers [14] and JIT compilers [24] can leverage branch profiles to optimize code. While branch profiles are cheap to collect [14], heap allocation properties such as object lifetimes, data hotness, or memory access correlations are often much more expensive to profile. For example, ROLP reports up to 6% runtime overheads for profiling even coarse-grained lifetimes, and DJXPerf [38] reports 8.5% overheads even with statistical sampling. While these overheads may not seem large, they are prohibitive in production deployments where even 1% performance degradation is substantial [36].

A common approach is to collect these profiles during a profiling phase: In ahead-of-time compiled languages such as C++, benchmark runs on an instrumented binary are used to collect a performance profile that is then used during the compilation of the final binary. Meanwhile, managed runtimes such as Java Virtual Machines spend a significant

amount of time on warm-up [39] during which they collect initial performance profiles and use them to JIT-compile code. These approaches introduce significant challenges:

1. **Deployment Velocity**: Profile collection introduces a long delay into the deployment process. In ahead-of-time compilation, this results in a large number of additional steps before a final binary can be produced. In JIT compilation, it reduces elasticity by requiring services to warm up for longer.
2. **Deployment Complexity**: Setting up PGO pipelines can be very complex, requiring the development of representative benchmarks and flows to run workloads automatically with instrumentation. For JIT compilers, ensuring that the initial load that (e.g.,) a server sees is representative is challenging as well.
3. **Non-representative Profiles**: If the workload used to generate the profile is not representative of the real workload, results will be suboptimal. It is very difficult to ensure that benchmarks are fully representative.
4. **Incomplete Profiles**: Individual workloads often do not exercise every corner case in the code, which means that profiles will often be incomplete.

In this paper, we investigate an experimental and radically different approach to solving these problems: Can we train a machine learning model that can predict heap allocation properties such as lifetimes or object hotness for future workloads from the statically available program code alone, without running an instrumented build or a warm-up phase?

As an intellectual abstract, this paper does not offer a conclusion to this question but instead lays out the trade-off space of such an approach, investigates promising directions that suggest such an approach is feasible, and highlights the challenges that need to be collectively overcome to enable it. Our specific contributions are as follows:

- We introduce a framework to reason about the design space for predicting heap allocation properties using machine learning.
- We gather and analyze a data set derived from the DaCapo benchmarks [7] that combines static code with dynamic heap allocation properties.
- We introduce a range of model architectures to predict heap allocation properties from code, and characterize their trade-off space.
- We provide a detailed discussion of challenges to making this approach work in practice, and highlight future research direction.

Our goal is to open up a new research direction for the ISMM community, combining research on ML models for code with research on data-driven memory allocators.

2 Background & Related Work

We provide an overview of approaches for predicting heap allocation properties and related work in this area.

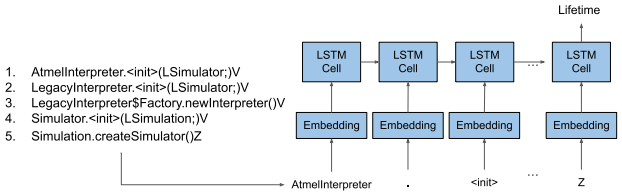

Figure 1. Predicting heap allocation properties from stack traces using the LLAMA [41] approach.

2.1 Predicting Heap Allocation Properties

Leveraging heap allocation properties to improve memory allocation has been a long-standing topic of interest. A classic example is pre-tenuring in managed runtimes such as JVMs, which relies on predicted object lifetimes [9]. Other examples include the prediction of object affinity [52], hotness [38, 45], and container sizes for presizing optimizations [19].

What these approaches have in common is that they need to make predictions at the time a heap allocation is performed. Such predictions are usually performed based on the *allocation context*, namely the program counter of the *allocation site* and the current *stack trace* (Figure 1). Note that the allocation site alone is often insufficient to uniquely identify an allocation context [6]. For example, an allocation performed in a string constructor does not provide much information about the allocation's lifetime, but the frame on the stack where the string is allocated might.

Once an allocation can be attributed to a particular allocation context, the second question is how to profile the relevant property. There are a range of different methods. For example, Harris [27] introduced an approach that samples objects and keeps information in a separate data structure. V8 uses "memento" objects that contain an additional pointer back to the allocation site [19], and when these objects are recycled by the garbage collector, it attributes any accumulated profiling information back to the allocation site. TCMalloc [32] samples a small fraction of allocations and collects profiling information just for these objects. This profiling may not always be active.

A significant amount of work in this space has focused on effective ways to summarize the stack trace at the time of allocation to enable such methods, including keeping track of the current calling context in a bit vector [52] or using stack hashing strategies [43]. There has also been work on trying to leverage data on the stack to make these predictions, rather than program counters [20]. However, such approaches are rare and not widely used in practice.

2.2 Profile Guided Optimization

Collecting profiles for each allocation context and using them for optimizing memory allocation is a special instance of profile-guided optimization (PGO). Here, we use this term

to describe two different types of setup[1]: 1) Collecting profile data from previous, specially instrumented, runs of an application and using this data to improve performance in future builds of this application. 2) The analogous approach in JIT compilers, where profile data may either stem from an instrumented run [8] or the same run [24] of the application. In the latter case, most JIT profiling data is collected at the beginning of the execution, resulting in a period when the JVM runs more slowly, known as *warmup* [39].

PGO has been used to great success in tasks such as cache miss reduction [52], receiver class prediction [25], and I/O partitioning [56]. However, collecting the profiling data required by PGO is often expensive and complex. Generating this profiling data typically requires running an instrumented (and hence, slower) version of the application. For example, techniques such as hot datastream profiling [16] or value profiling [12] can achieve speedups as high as 20%, but require profiling runs that can be tens to hundreds of times slower than the original program.

Our proposed approach takes the place of a PGO optimization pass, but instead of relying on profiles collected in a previous run, it tries to predict these properties from program code. This approach has similarities to LLAMA [41], a recently introduced approach for lifetime prediction in C++.

2.3 LLAMA

LLAMA is a memory allocator that uses a neural network to predict lifetime classes of objects using a symbolized allocation context. It assumes a scenario where a *partial* profile is available – e.g., from a previous version of the same program or where only a subset of allocation contexts was observed due to sampling. The authors show that profiling data (in their case, lifetimes of objects) transfers between similar binaries. For instance, LLAMA could accurately predict lifetimes of unseen stack traces even after a number of code changes or when changing compiler settings.

LLAMA performs these predictions by converting each stack trace into a list of symbols and treating this representation as natural language (Figure 1). It subdivides the stack trace into tokens which are then passed into a *Long Short-term Memory Network* (LSTM) [31]. This model is trained against known stack traces and associated object lifetimes. A caching mechanism is used to run this model only the first time a particular stack trace is encountered.

The key idea of LLAMA is to use ML to extract programmer intent by treating the symbols of the program as language. While this work takes a step towards generality, it is rather limited. For example, LLAMA cannot generalize to completely different programs, primarily because the calling contexts were represented only by function names. While LLAMA performs well on a single program, it is not applicable to a new program full of different function names, as the function names in the training set might not appear in this new program. Furthermore, the approach is not amenable to even the same application with all of the function names changed (e.g., due to refactoring or obfuscation), even if program behavior has not changed.

[1]The terminology differs and sometimes only refers to the first as PGO.

2.4 ML for Code and Programming Languages

There has been a large amount of work on ML for code in recent years. Allamanis et al. [2] provide a survey. A significant portion of this work looks at the problem from a software engineering perspective — such as neural code completion [22, 46] and finding bugs in code [49, 54].

There have been uses of ML for compiler optimizations [37], but they are less common. MLGO trains models to make inlining and register allocation decisions within LLVM [53] (the latter is a problem that sometimes uses PGO). Autophase [26] learns an ML policy for ordering compiler passes that generalizes to unseen programs, but makes decisions at the granularity of entire compiler passes, rather than individual objects or their profiles. Rotem and Cummins show that instead of relying on PGO profiles for branches, they can be predicted with learned decision trees [51], which resemble a complex, learned compiler heuristic for branches. None of this work looks at properties of heap allocations, which are more complex and difficult to predict. Source code could provide the necessary signal for these predictions to the model, even without profiling data for a calling context.

3 High-Level Overview

We now provide an overview of our approach. We start with a conceptual framework how to reason about the general problem of predicting heap allocation properties. We then describe the intuition behind our proposed ML approach.

3.1 Conceptual Framework

At a high level, prediction of heap allocation properties can be performed using a broad range of methods, including profile-guided optimizations, program analysis and heuristics. The lines between these methods can be blurry. We therefore start by introducing a conceptual framework to reason about these problems and the associated challenges.

We classify approaches addressing this problem along three dimensions: Data, Model, and Application. *Data* describes the input used to drive predictions (such as the instruction pointer of the allocation site, symbolized stack traces, or an abstract syntax tree of the code). *Model* describes the mechanism by which the data is used to determine a label, such as object lifetime – this could be any function and does not have to be an ML model. *Application* describes how the predictions made by the model are used (e.g., how code is compiled differently based on this prediction, or how a memory allocator may leverage it). Many different strategies map to this conceptual framework:

Profile-guided optimization. Depending on the specific optimization, the *data* the prediction is based on is code locations of a particular allocation site or on the call stack. The *model* is a lookup table that maps these call stacks to previously measured values. This approach may be *applied* offline or online, at compile time or at run time.

Static analysis. The *data* is typically some form or intermediate representation (e.g., LLVM IR). The *model* is an algorithm that processes this IR symbolically (e.g., by applying escape analysis) to make a prediction. The *application* of this prediction occurs usually within the compiler.

LLAMA. The *data* LLAMA uses are symbolized stack traces rather than the instruction pointers. The *model* is an LSTM neural network that maps these stack traces to lifetime classes. The *application* is to make these predictions online in the context of a custom lifetime-aware memory allocator.

The part of the design space explored in this paper is one where the data spans all code that is statically available without running the application, models capture a wide range of different machine learning methods, and applications contain a range of offline and online methods. We now motivate why we believe this is a promising design space to explore.

3.2 Our Approach

Instead of collecting profiling data through expensive instrumentation like PGO, we could predict it using machine learning. A strawman approach would be to train a model on profiling data from a set of binaries, and use this model to predict the profiling data for new, unseen binaries. If the accuracy is high enough, we could use these predictions in the same way as profiles, but without the overheads.

LLAMA took a first step in this direction, by training a model on a subset of allocation contexts in one binary and using this model to predict unseen allocation contexts in a potentially different version of the same program. LLAMA showed that symbolized calling contexts are sufficient to perform these predictions with high accuracy.

However, we find that the same approach does not work well *across* programs. We recreated a LLAMA model and attempted to predict object lifetimes on DaCapo [7] benchmarks. When predicting across benchmarks, we find that this model performs only one percentage point better than random prediction (we discuss this further in Section 5).

Intuitively, the function names contained in the symbolized calling context capture the behavior of the functions. A model can learn that when certain names appear in a certain order in a calling context, the corresponding object will be long or short-lived. However, since we want to predict on unseen binaries, it is likely that the model will encounter many unknown names, or a new and unusual combination of names, in unseen calling contexts.

To address these problems, we propose to move beyond just function names and instead consider the whole source code, which could capture program behavior from code structure, variable names, and even comments (although we do not currently explore the latter in this work).

3.3 Intuition

Source code defines the program behavior and could thus be used to predict heap allocation properties, which are determined by this behavior. There have been a number of works that predict many different properties of code, such as security vulnerabilities [15], performance [13], bugs [49, 54], types [28], summarization [23], and other static program properties [50]. There is also work on representing code in other ways, such as graph representations [3]. However, prediction of dynamic properties has seen less attention.

Listing 1a shows a strawman example of the intuition why code structure is predictive of heap allocation properties, such as object lifetimes. We can see two allocation sites, one defining a variable *x*, and another defining *y*. When looking at the source code, we can see that *y* will outlive *x*, as it exists throughout the execution of the *main* function. On the other hand, the variable *x* will live only as long as one iteration of the inner `for` loop. In this way, we can compare the lifetimes of the two variables, based on the source code, and conclude that *y* will have a longer lifetime than *x*. A model could learn a general pattern that variables defined in the inner-most part of a nested loop are likely short-lived, and variables defined in the main function are likely long-lived.

Additionally, the variable names themselves can provide useful information [5, 34]. For example, variable names may contain short descriptive atoms such as *tmp*, or suggest a relative lifetime ordering between variables. Consider, for instance, *requestBatch*, which hints at an object that encompasses multiple requests, versus *requestStatus*, which hints at an object with a lifetime shorter than that of a request.

A more concrete example is shown in Listing 1b. In this example, taken from the FOP benchmark in DaCapo, we can see an array named *tmp* that is used only for constructing the String and nothing else. Here, the variable name signals intent that it is not used in other places.

Another useful variable naming pattern is that short-lived objects often have short names. An example is shown in Listing 1c. This method is found in the source code for ANTLR[2], a parser generator written in Java. In this example, we can see a number of variables with short names: *f, fr, br*. Each of them is used locally and does not outlive the method call.

Finally, when available, code comments, literals, and logging statements can provide hints about lifetimes of objects.

Some of these properties could be exploited without learning. In fact, we could manually construct a very large number of such rules, which is similar to how compiler heuristics

[2]https://github.com/antlr/antlr4

```
void doWork() {
    for (int i = 0; i < ... ; ++i) {
        for (int j = 0; j < ... ; ++j) {
            Bar x = new Bar();
            doTask();
        }
    }
}

void main() {
     Foo y = new Foo();
     doWork();
}
```

(a) Strawman: Code structure provides hints about object lifetime.

```
public final String readTTFString()
    throws IOException {
    int i = current;
    while (file[i++] != 0) {
        ...
    }
    byte[] tmp = new byte[...];
    System.arraycopy(file, current,
                     tmp, 0,...);
    return new String(tmp, ...);
}
```

(b) Apache FOP: Variable names provide hints about lifetime.

```
public Grammar getRootGrammar(...)
    throws IOException {
    ...
    File f = null;

    if (haveInputDir)
        f = new File(...);
    else
        f = new File(...);

    fr = new FileReader(f);
    br = new BufferedReader(fr);
    grammar.parseAndBuildAST(br)
    ...
    br.close();
    fr.close();
    return grammar;
}
```

(c) ANTLR: Short-named variables have short lifetime.

Listing 1. Motivating code examples.

are often designed. However, such rules would be brittle and shift over time. Machine learning provides a way to "generate these rules automatically", by having a model learn them instead of deriving and encoding them by hand.

In the following sections, we discuss the potential design space, challenges, and potential solutions associated with such a machine-learning based approach. To support our exploration, we developed an end-to-end implementation of the approach for object lifetimes. While our experiments

Figure 2. An example of a stack trace with associated profiling data and source code

show evidence that an ML approach is able to learn heap allocation properties, we also find that its current accuracy is limited. In each of the following sections, we describe the trade-offs and challenges that contribute to these limitations, and the research problems that we believe need to be solved to further increase accuracy. Following our framework, we will discuss the underlying *Data* (including training data collection), *Model* (including a range of different designs) and *Application*. We focus on Java, but our approach is generally applicable to other languages such as C++.

4 Part 1: Data

Training a heap allocation property model requires two types of information: static information and dynamic information. The static information can be found in code, while the dynamic information must be found through profiling. Related work has mostly looked at either one or the other. Code models look at large amounts of code, but do not connect them to object lifetimes or other dynamic properties. PGO collects such dynamic properties, but only minimally connects them to code, usually via stack traces.

Similar to LLAMA, we collect stack traces that represent *allocation contexts*, the calling context under which particular objects are allocated. Each stack frame of the stack trace represents a certain function, centered around a callsite or an allocation site (in the case of the topmost frame). Each stack trace is associated with the distribution of object lifetimes. In contrast to LLAMA, we also collect the appropriate source code corresponding to each stack frame. An example of a single stack trace is shown in Figure 2. We collected this stack trace from the avrora benchmark in the DaCapo benchmark suite [7]. On the left side, we can see a sequence of function calls. Each stack frame is qualified with the full classname of the function (although some are omitted here for the sake of clarity). The stack frames are also associated with a particular file and line number. For example, the first stack frame, representing an allocation site, corresponds to line 326 of the file AtmelInterpreter.java. Note that some stack frames are missing source code locations. We discuss this problem in Section 4.4. Lastly, the entire stack trace is associated with

our object lifetime profiling data. We now describe one way of collecting this dataset, and challenges that we encounter.

4.1 Collecting Dynamic Properties of Objects

While lifetime predictions are a language-agnostic problem, in this work we collected a Java dataset. To this end, we modify OpenJDK11 to perform fine-grained profiling of object lifetimes. While OpenJDK does not have an easy way of collecting fine-grained object lifetimes, there are several approaches in the literature on how to collect object lifetimes in Java. Naively, it would be possible to measure lifetime as the number of garbage collections that an object survived. However, object lifetime granularity would be determined by the frequency of GC passes. Since a full garbage collection can take milliseconds or even minutes, this is much more coarse-grained than the lifetimes that we wish to capture.

Alternative approaches include algorithms such as Merlin [30] and Resurrector [57]. The general approach in these cases is to track incoming references to objects and determine the time at which an object dies, either retroactively during GCs (Merlin) or during execution (Resurrector). We add a simpler version of the latter approach to OpenJDK, by adding reference counts to each object, to detect when it becomes unreachable. We modify the object header to include a counter and modify relevant instructions (aload, astore, areturn, new, athrow) to update these reference counts. Additionally, when an object is allocated, we walk the stack to find the object's allocation context. Like Resurrector, our approach cannot detect cycles and ignores them.

For each object, our instrumentation must collect a lifetime. We defer the definition of our units of lifetime to Section 4.4.2 but for clarity in this section, we track lifetime as a *logical duration*, i.e., a difference between values of *logical time*. We maintain a global, monotonically increasing logical clock. Each object tracks its initial allocation time (*i.e.*, the logical clock at the time of its allocation). When an object's reference count reaches zero, we find its logical lifetime by subtracting the current logical clock by the object's logical allocation time. Because we care about allocation context (rather than individual objects), we aggregate this lifetime information by allocation stack trace.

It would be prohibitively expensive to store the individual lifetimes of each object at full granularity. Instead, we classify each object into lifetime classes separated by orders of magnitude: < 10 time units, 100 time units, 1,000 time units, etc. This mirrors LLAMA's bucketing of object lifetime, however in our case the bucketing is based on logical lifetime, rather than wall clock time. For each observed allocation site, we then store only ~ 10 integers: a histogram of the number of observed objects with each lifetime class. While this does lose some specific per-object information, it is a good tradeoff between collecting lifetimes and saving memory. Because we eventually classify stack traces based on this bucketing scheme, this granularity of data collection is sufficient.

4.2 Collecting Source Code

In order to use source code as input for our models, we need to collect the source code of our target application and any third party libraries it uses. We need all of this code, as some stack frames will be calls to these third party libraries.

Depending on the code base, this may be difficult to do automatically. We target the DaCapo benchmarks, which are built using tools such as Ant and Maven. We manually inspect the Ant build file and record the necessary third party dependencies that are downloaded. It is important to point out that not every dependency has a readily available source code download, even if many are hosted on the central Maven repository. This is particularly challenging as we must also use the exact same version of the source code that the compiled dependency uses, as using the wrong version of the source code could degrade the quality of the dataset.

We then manually create a source code repository containing all of the application code, Java standard library code, and third party dependency code. We organize the files by package, as Java code is typically organized. For example, the source code for the class `java.lang.String` will be in the file `java/lang/String.java`. This approach works well for Java code, as each source file will contain a single outer-level class and and we can find the source file for `java.lang.String` within the folder `java/lang`.

4.3 Contextualizing Stack Traces

After collecting the necessary source code files, we need to connect them to our stack traces. We call this process *contextualizing* the stack traces, as it connects the source code context to each stack frame of the stack trace. Each stack frame contains a line number (representing a call or allocation site) as well as a fully qualified method name. Using the fully qualified method name, we can find the appropriate file in the aforementioned source code repository, as the method name (with periods replaced with slashes) will point to a particular file. We then associate every stack frame of every stack trace with the entire source code of the appropriate method. This is a very large number of tokens for a model to handle. We discuss this challenge in Section 5.

4.4 Challenges

We now describe important research questions that need to be addressed to improve the approach.

4.4.1 Finding All Source Code. In many scenarios, not all source code is available. At least in Java code bases, it is rare for all third party library source code to be in the same repository as the application code. Additionally, because of the multitude of Java build tools (*e.g.*, Gradle, Maven, Ant), it would be difficult to gather code on a large scale (*e.g.*, on all of GitHub) since the process would be different for every repository.

In this work, we give a special "..." token to such stack frames (instead of source code) as a way to inform the model that the source code was missing. Although missing source code of a stack frame robs the data representation of code-specific nuance, at least a partial stack trace is visible to the model, and missing frames are represented the same way (albeit with a different function signature), akin to the way that ML vocabularies represent an "unknown" token.

Future work could look at more sophisticated techniques, such as imputing unknown code, decompiling byte code, or approximately adapting prior code versions to fill gaps.

4.4.2 The Right Logical Clock. Traditionally, allocation lifetimes in memory managers are measured in terms of a logical clock, usually the total number of *allocated bytes* [6]. The reason is that such a clock is more stable than wall clock time in light of performance variations (e.g,. due to sharing a machine between different workloads). Given that instrumentation is often expensive, logical time may sometimes be the only option – e.g., we observed more than two orders of magnitude slow-down in our instrumentation, and Merlin reports up to 300× slowdown [30]. Wall clock times would be meaningless in such a scenario and we therefore use *allocated bytes* as logical time.

However, a logical time base is specific to a particular workload. For example, an image processing workload with MB-sized allocations may have entirely different allocation sizes than a text processor with KB-sized allocations, and the same library code behaving in the exact same way in both applications may result in orders of magnitude of difference in logical lifetimes. When used in the context of training a model across workloads, this means that these lifetimes may not be comparable and thus not learnable – an effect we observed in our own experiments.

This creates a dilemma: While wall clock time may be more stable during learning and enables better transfer across workloads, it is more sensitive to local performance variations and instrumentation effects. Meanwhile, logical time is more stable within an individual workload, but does not transfer well across workloads.

We believe there are several directions of addressing this problem. For example, LLAMA enables the use of wall clock time by introducing a cheap, sampling-based approach for C++, at the cost of not capturing all allocation contexts and sensitivity to performance variations due to compiler settings [41]. Another approach may be to develop new logical time bases that are consistent across workloads.

4.4.3 Missing Context. Even with wall clock time and in the absence of performance variations, lifetimes may not be stable across different binaries – or even the same binary with different inputs. For example, imagine two server workloads that use the same server framework to process requests, and where the lifetime of an object is identical to that of its request. If the timescale of work within each request is very different (e.g., microseconds vs. seconds), the lifetime cannot be statically predicted without analyzing the unrelated code that performs the actual operation within the request. This code may not be anywhere near the allocation site and is thus not included in our contextualization approach.

As in the previous section, logical time bases that are less dependent on such variations may alleviate the problem. Another approach may be to define a *context* and express lifetimes with respect to this context rather than in absolute terms (e.g., that the lifetime of an object does not exceed a partcular request or subportion of a program). A third option may be to expand the scope of the data set to include code that is not involved in the allocation itself.

4.4.4 Data-Dependent Lifetimes. Objects with the same allocation site may have different profiling data, but are represented by the same stack trace. For example, the lifetime of an allocation may be determined by a dynamic input parameter and is thus different for every input. Because we want to assign a single label to each allocation context, we must choose a single value from the distribution of profiling data. In our dataset, 72% of the stack traces observed objects with more than one lifetime class.

Choosing the right label in such cases is an important challenge. We currently assign a label to each input example based on the most common lifetime class found among objects allocated at the stack trace. For example, if a stack trace observed objects of every lifetime class, but had mostly lifetimes in the range 0 to 9 bytes, then it is assigned a label of 0. However, in some cases it may be preferable to pick the extreme labels (*i.e.*, the max observed or min observed lifetime), but this is best determined by the downstream task.

Additionally, different inputs might result in entirely different stack traces, so it is important to collect a representative dataset during profiling.

4.4.5 Ambiguous Allocation Sites. Each allocation context represents a different number of objects and thus lifetimes. Some represent a single object, while others may represent millions of objects. In our dataset, on average, an allocation site represents about 2,000 objects. This may affect the best labelling strategy, as the most common label may be incorrect for thousands of objects, even if it is the most common label for the stack trace.

There are a number of potential strategies to address this issue. One approach would be to introduce additional features into the model to facilitate differentiation between allocation sites. Another approach is to define lifetime classes in such a way that allocation sites are more likely to only have one label. Finally, Zhou and Maas investigated a similar problem in storage systems and proposed predicting a distribution of lifetimes rather than a single value [58].

5 Part 2: Model

We now turn to our exploration of learning the lifetime of objects. In all cases, we attempt to learn a function f that predicts a lifetime label: $predicted\_label = f(stack\_trace)$.

In this exploration, we are probing the ability to generalize a lifetime model beyond the binary it was trained on. We compare LLAMA as a baseline (Section 5.2), to techniques that enhance generalization of function-signature models (Section 5.3), as well as techniques that utilize more of a stack frame, such as code tokens (Section 5.4) and code structure (Section 5.5) or their combination (Section 5.6). We see that model generalization improves, but our results are only a hint that more work in this space is desirable. Section 5.7 suggests future directions.

5.1 Training Details

We convert our lifetime class profiling data into a binary classification task, where logical lifetimes $< 100K$ Bytes are "short" and higher logical lifetimes are "long".

We split our DaCapo dataset by benchmark, and arbitrarily choose three sets of benchmarks: 1 benchmark for validation (fop), 1 for testing (h2), and the rest for training. We perform the split this way as it most closely aligns with our goal of predicting across binaries. The alternative is to mix all stack traces into one dataset, then create splits. However, this would be more akin to a single-binary prediction task (with incomplete profiles) rather than cross-binary prediction.

We focus on classification accuracy as our target metric. Our datasets have high skew between the short and long lifetime classes. Roughly 90% of stack traces have the short label, while the remaining 10% have the long label. So, if a model simply predicted "short" for every example, it would achieve a vacuous 90% accuracy. We combat this skew in two ways: (a) we subsample the majority class to have a similar size to the minority class during training, and (b) we use Mean Per-Class Accuracy (MPCA) on the (non-subsampled) validation and test datasets. In the pathological example above, if a model predicted "short" for every stack trace, it would only achieve 50% MPCA (100% accuracy on "short" and 0% accuracy on "long"). We do not subsample the test split, because we wish to control for a task with skewed labels in practice, although during training it is important to teach the model with enough emphasis on both labels.

We implement these models using TensorFlow and Keras [1, 18]. We train our models using TPUv2s and TPUv3s on Google Cloud Platform. For the Transformer-based models, we use the Transformer implementation found in the BERT repository [21]. We perform a hyperparameter search for each model. For the simple LSTM models, we vary learning rate, sequence length, embedding size, and LSTM cell size. During training, we utilize recurrent dropout in the LSTM cell. For the Transformer-based models, we vary learning rate, hidden size, and number of layers. Due to memory constraints, we use only the top 32 frames of a stack trace in the Transformer models. A single configuration for the LLAMA-like models takes roughly an hour to train, while a single configuration for our Transformer-based models finishes training in roughly 22 hours.

We train by minimizing the binary cross-entropy loss on *predicted_label* compared to the ground truth we collected (Section 4). Specifically, we try to minimize function $L = -\frac{1}{N}\sum_{i=1}^{N}[t_i \log(p_i) + (1-t_i)\log(1-p_i)]$, where N is the number of examples, t_i is the ground-truth of the i-th example (0 for "short" and 1 for "long"), and p_i is the predicted probability that example i is "long". For each hyperparameter configuration, we select the checkpoint that achieves the highest MPCA. For each model type, we report the MPCA of the best checkpoint of the best configuration.

5.2 Baseline: LLAMA

LLAMA treats each stack frame in the stack trace as a string and tokenizes function signatures on special characters such as `,` and `::`. It then separates the tokenized stack frames with a special `@` token. These tokens are then encoded using a vocabulary to map each token to a specific vocabulary ID. The tokens of the entire stack trace are then fed into an embedding layer, then this sequence of embeddings is fed into an LSTM recurrent neural network, resulting in an embedding of the entire stack trace. The final embedding is then used to predict a lifetime label for the stack trace. This relatively straightforward model performs very well in their task of predictions on a similar binary to the training data.

LLAMA used only function signature tokens in its representation. However, this information is not enough to generalize. To demonstrate this experimentally, we recreate a LLAMA-like model and try to predict object lifetimes on our DaCapo dataset. Since our dataset is in Java, we tokenize our stack traces in similar ways to LLAMA's C++ tokenization.

We train this model in two different scenarios. First, we train on the entirety of our DaCapo dataset and test on the entirety of the dataset. This scenario parallels perfect coverage in LLAMA's data collection (LLAMA needs to use predictions, as not all stack traces can be covered by their sampling-based data collection). In this experiment, this LLAMA-like model for Java achieves 92% MPCA, meaning that this LSTM model could mostly lookup this previous profiling data.

The more interesting case is when we attempt this across benchmarks. When we train on only the training set, the model is not able to predict well on the test set. It achieves an MPCA of 51% on the held out test dataset (recall that random prediction would be 50%).

There are a few potential explanations for this result. First, it could be that out-of-vocabulary words can have a big impact. While our vocabulary of 5,000 tokens covers more than 99% of tokens (*i.e.*, less than 1% of tokens must be encoded as

a special out-of-vocabulary token), it might be the case that these tokens are very important for generalization. Second, function names may just not be representative of object lifetime across benchmarks. One potential solution is subword tokenization [10], which we address next.

5.3 Subtokenization of Function Names

Subword tokenization is a middle ground between word-based and character-based tokenization. Common words will be included in the subword vocabulary, while rare words can be losslessly encoded using subword tokens. This removes the out-of-vocabulary problem, as in the worst case, even unseen function names can be encoded (as single characters).

Another potential benefit is that some subword tokens in class or method names might be helpful for the learning task. Simpler token encoding would consider an entire name at once. This means that three method names called *get*, *getBestPlanItem*, and *getErrorListener* would each receive an unrelated, different ID. However, their names all contain *get*. With subtokenization, these methods could all share a *get* subtoken, followed by a *BestPlanItem* or *ErrorListener* subtoken if needed. Stack traces that share a descriptive subtoken might behave in a similar way. For example, in our DaCapo dataset, stack traces that contain a *get* subtoken in the top-most frame observe long-lived objects 39% of the time. Stack traces without such a subtoken observe long lived objects 8% of the time, which is much closer to the distribution of lifetimes as a whole.

However, this still is not enough signal for the model. We again create a LLAMA-like model, but this time we use subword tokenization. We use the CuBERT [35] Java tokenizer to tokenize our function signatures, ignoring whitespace tokens. We then encode these tokens using CuBERT's Java subword vocabulary and the Tensor2Tensor library [55] to produce a sequence of IDs. Using the same model architecture as in section 5.2, we embed the IDs and produce a prediction. On the same DaCapo train/test split, this model achieves an MPCA of 53%, which is only 2 percentage points higher than the non-subtokenization model.

One possible explanation could be that subtokenization adds more tokens, but perhaps not enough extra signal. While subword tokens might be useful information, since we can only process a limited number of tokens, adding more tokens might reduce useful signal as some tokens must be pushed out. It could also be the case that these "helpful" subword tokens are simply not enough signal.

From these experiments, it is possible to conclude that function names, even when encoded in different ways, do not seem to provide much signal to the model for this lifetime prediction. We therefore turn to a different feature: code.

5.4 Featurizing Stack Traces with Code

Code may offer the signal that we need for this prediction. As mentioned in Section 3, code defines program behavior, and

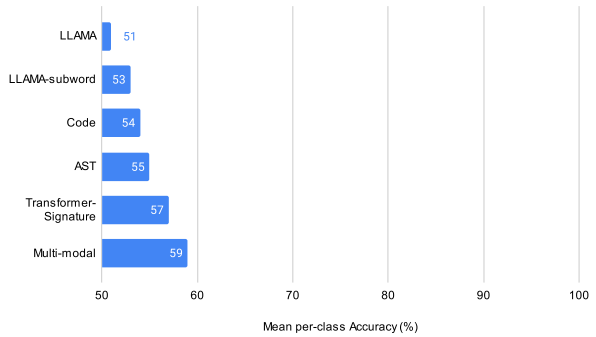

Figure 3. Mean per-class accuracy on the DaCapo holdout test set. Note that random predictions are 50% MPCA, so the difference between 51 and 59% is larger than it may appear

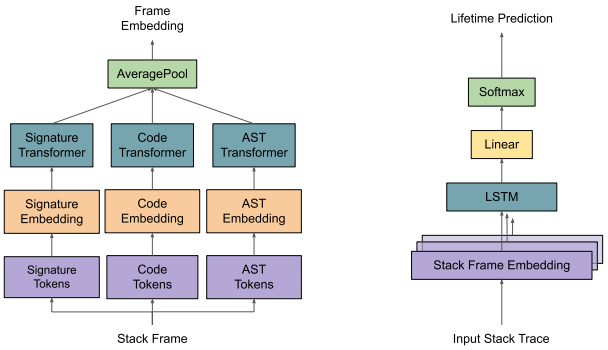

Figure 4. Multi-modal stack frame embedding (left). Lifetime prediction model with several frame embeddings (right).

should be more useful for generalization than function names across binaries. We thus apply code embedding models.

As described in Section 4, we associate every stack frame with the source code of the appropriate function. Additionally, each stack frame has a line number, representing an allocation site (in the case of the topmost frame) or a call site. One major challenge is selecting what code to embed for each stack frame. It is difficult to train long sequence lengths on traditional Transformer-based language models. For example, pre-trained BERT models are often limited to sequence lengths of 512 tokens. However, even a single function in our dataset could have thousands of tokens.

When combined with the fact that our stack traces have tens or even hundreds of stack frames, we can easily run out of token budget. This problem is exacerbated by the fact that many code-embedding models focus on embedding a single code snippet or context. However, in our case, we need to look at many snippets at once, which may even be from separate code bases when considering third-party libraries. Given these uniquely difficult code embedding challenges, we believe that there is room to improve code featurization.

We must select a subset of code tokens for each frame in order to satisfy our token budget. In this work, we take a simple approach: in each stack frame, take a window of

tokens around line number of the call or allocation site. Starting at the center token of the given line, we simply add one token from the left of the current window, then one from the right, and so forth until the per-frame budget is reached. If one side reaches the beginning or end of the function, we gather tokens from the non-exhausted side until we reach the per-frame budget, or the entirety of the function is selected. We are then left with a (smaller) sequence of code tokens for each stack frame. The tokens are then encoded using a subword vocabulary, leaving us with a sequence of subword token IDs for each frame.

In addition to limiting budget per-frame, we must also pick certain frames to include. These are related budgets: increasing one causes us to decrease the other to maintain a memory budget. In our experiments, we choose to keep the top 32 frames, and set a per-frame token budget of 256 tokens. We then embed these tokens using a Transformer-based model. Using a Transformer (of max sequence length 256 tokens), we embed each stack frame's code tokens to produce 32 Transformer embeddings, one for each frame. Next, we pass these Transformer embeddings through an LSTM to produce a single embedding of the entire stack trace. Finally, we apply a last Dense layer and a softmax to produce a lifetime label of the entire stack trace.

We train and test this code-only model with the same DaCapo training/test split. It slightly outperforms our simple LLAMA-like models, and achieved 54% MPCA on the test dataset, only 3 percentage points higher than the LLAMA-like model and 1 percentage point higher than the subtokenized LLAMA-like model. Note that this representation does not cleanly supersede that of Section 5.3: the function signature does not always fit in the per-frame token budget.

One potential validity sanity-check is that it is not the code that is causing improvement, but the model size. This is a valid concern, as the 32-frame Transformer model (188M parameters) is much larger than a small embedding layer and LSTM (4M parameters). So, we create a comparable signature-only 32 frame Transformer model. Given the same token and frame budgets (*i.e.*, 32 frames, 256 tokens per frame), we subtokenize and embed the signatures of each frame, rather than code. Interestingly, this Transformer-based signature-only model outperforms the code model by achieving 57% MPCA on the holdout dataset, 3 percentage points higher than the code version of the model.

A major problem is the number of code tokens. While a window of code around the call site is straightforward to collect, it may not be the right set of tokens. Tokens that are lexically far away from the callsite (*e.g.*, control flow such as *for* or *while*) may have a large impact on the prediction, but will not be captured by the window. This has been observed before [3, 4, 29, 48] and motivates the next approach.

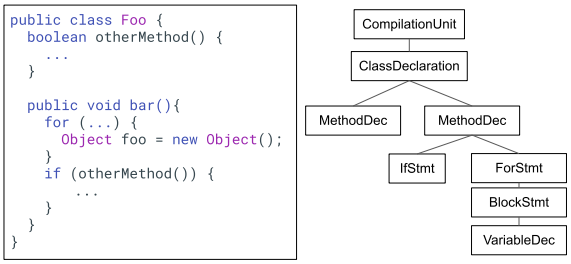

Figure 5. A code snippet and associated AST

5.5 Representing Code with Abstract Syntax Trees

Another potential direction is to represent code using Abstract Syntax Trees (ASTs). ASTs are a tree representation of source code structure. As opposed to concrete syntax trees, or parse trees, ASTs do not capture every detail of the source code, but do capture important structural details. For example, an AST might represent an `if` statement with a handful of nodes: a node for the `if`-statement itself, and three child nodes, representing the condition, `then` branch, and `else` branch. We parse our source code using javalang[3], an open source Python library providing a lexer and parser for Java.

An example of this representation is shown in Figure 5. Note that some details are omitted from the figure for clarity. To represent an allocation site, we find the path to the allocation site on the AST. For stack frames that represent a call site, we instead find the appropriate method invocation node. To generate a token sequence, we simply use the names of the nodes on the AST. So, the *foo* allocation site would be represented by the tokens: CompilationUnit, ClassDeclaration, MethodDec, ForStmt, BlockStmt, VariableDec.

While this approach might lose some finer-grained information about the code, namely the variable names, the AST nodes may be useful for capturing high-level structural information, such as loop keywords. We train and test the same 32 frame model on the same DaCapo dataset as before, but using AST tokens. It achieves 55% MPCA, which is in between the code and signature performance.

5.6 Multi-modal Features

LLAMA showed that function names are sufficient in some cases. Code precisely defines program behavior and captures programmer intent (with variable names), but is verbose. ASTs are concise but lose fine-grained information.

Instead of using a single representation, we can try to combine them in an attempt to capture the best properties of each representation. This model is shown in Figure 4. We instantiate 3 Transformers, with sequence lengths 176, 16, and 64 for code, AST, and signature tokens, respectively. At each frame, the three Transformers produce one embedding each, representing their specific token-type embedding for

[3]https://github.com/c2nes/javalang

the frame. The 3 embeddings are then max-pooled to produce a single embedding of the frame. We repeat this for each of the 32 frames, and use an LSTM to produce a single embedding, followed by a Dense layer and a softmax to produce a prediction. While this multi-modal model processes the same number of tokens per frame (256) as the single type model, it has about 190M trainable parameters, compared to the 70M parameters of the previous single-type models.

We train and test this combined-embeddings model on our DaCapo dataset, and find that it achieves 59% MPCA on the holdout dataset. While it is not by much, it was the best performing model by a couple of percentage points.

5.7 Challenges

Despite the extra signal we provide the model, and despite out-performing signature-only methods, this prediction accuracy is still not high enough to be usable. The challenges in Section 4 are also relevant here, as a dataset can strongly affect a model's performance. However, there are also a number of challenges specific to this family of models.

A major open question is how to solve the context selection process. Due to memory constraints in large Transformer models, selecting the right tokens is paramount. Even with TPUv3s, we found it difficult to train on more than 32 frames, even with tiny batch sizes, and had to ignore the excess frames. However, almost 93% of the stack traces we collect have more than 32 frames, meaning that almost every stack trace has frames missing in its representation. Additionally, some individual frames are very large and must be trimmed down. Of the stack frames that have source code (78% of frames), 18% have too many tokens, and must lose some of their tokens before being presented to the model.

We choose to use the top of the stack as these stack frames are "closer" to the allocation site, and may be more relevant. However, it could be the case that other frames (or even auxiliary features like its height) might be predictive as well. For example, objects allocated with a certain library call on the bottom of the stack are possibly longer-lived.

Code selection within individual frames is important as well. While ASTs might help alleviate the problem of lexically-far tokens, it is not perfect. Carefully selecting the "important" code tokens could greatly improve model performance, as the "non-important" tokens can almost be considered noise that hurts the model. We are considering techniques that prioritize tokens the model is likely to consider "important" (in Transformer parlance, have high *attention* scores), based on an Expectation-Maximization formulation akin to CodeTrek [47].

6 Part 3: Application

Our proposed approach could be used for a number of optimization tasks. While we show how it can be used to predict object lifetimes, it could also be used for predicting properties such as object hotness. In general, the predictions are not the end goal: the predictions themselves are used by some downstream task. In the case of our lifetime predictions, this could be deciding to pre-tenure [8, 9] or stack allocate an object [17]. LLAMA used object lifetime predictions to create a memory manager that organized its heap into lifetime classes, rather than size classes. Analogously, object hotness predictions could be used for learned remote-memory prefetching. While there are works that improve prefetching in this setting [42], there may be good opportunities for ML-based approaches because accurate predictions could reduce very expensive remote memory fetches.

An important consideration is the required accuracy of the downstream task. Before modifying an existing system, it is useful to first quantify potential speedups and required model accuracy. As a thought experiment, we can take the example of learned remote-memory prefetching. Before modifying the prefetching system, we can simulate the effect of a model. First, for some application, we could collect a representative sequence of memory accesses, using a tool such as Intel's Pin [40]. Given the sequence of memory accesses, we can compare the page fault rate of the existing system and a model. We might hypothetically observe that the existing prefetching system addresses 65% of page faults, *i.e.*, 35% of requests must fetch remote memory, while the rest avoids a page fault because of the prefetching. If a model could accurately predict 50% of pages to prefetch, the ML-based prefetching would perform worse than the existing system, as it would have a higher page fault rate. If a model could accurately predict 65% of pages to prefetch, it still may perform worse than the existing system because of the cost of running the model. We might then find that, given the cost of a remote-memory page fault, and the cost of running the model online, that the ML-based system breaks even when the accuracy is 75%. A user might then decide that the ML-based system is only worth implementing if they can train a model that achieves 85% accuracy.

Assuming a model achieves sufficient accuracy, we can integrate its predictions into a system in a number of ways, each making a tradeoff between richness of input features and the effect of model overhead.

6.1 Online Prediction

In this approach, we run the model at the time a decision is made. For example, at the time an object is allocated, a runtime system could run a model to make the online decision to pre-tenure an object. While this approach was suitable to LLAMA, it may not be possible with our larger code-embedding models. Object allocations are latency-sensitive and must finish in nanoseconds, which is not enough time to run a large model. LLAMA proposes amortizing the cost of running the model by caching model predictions and only running the prediction if the result is not already cached.

However, Transformer-based models, such as the ones we use, can easily take milliseconds to run. Even with caching, this may be too expensive to run online in some applications.

One important benefit is that it may be possible to include other live state (*e.g.* the value of certain variables, cache-line state) as a feature to the model. These features would not be available to non-online predictions.

6.2 Prediction in JIT Compilers

Another possible use case could be during JIT compilation. For example, OpenJDK runs interpreted bytecode, but when a method is executed enough times, will profile it at run time and compile it with increasing amounts of optimizations. Since a JITed method is hot, any performance optimizations will have a large effect. Because the JIT compilation typically occurs in the background, running a model for some milliseconds could have less of an overhead than running it on the critical path, as an object is allocated. Similar to online prediction, live program state could be used as a feature in the prediction. However, managed runtimes have been tuned for decades and might already do a "good enough" job with their online-profiling and optimizations, even if they are simpler than stack trace predictions we could produce.

6.3 Offline Prediction

On the other end of the spectrum is moving the prediction completely offline. The benefit in this case is that there is no runtime overhead to run an expensive model. One way to use this type of prediction is to generate annotations that can be used by the runtime system [8, 44] or optimizations like ThinLTO [33]. However, these predictions can only use source code features, as dynamic state is not available.

7 Discussion

While the multi-modal representation of stack traces seems like a promising direction, the accuracy that these models achieve is not yet usable. We see a number of opportunities for improvement.

7.1 Token selection

One major open problem is selecting code tokens. There are far too many tokens in a stack trace to include them all. We could try to solve this problem at a function granularity, or by selecting only certain stack frames, or combining the two.

Within a single function, the most valuable tokens might be lexically far from the allocation site, and are missed by our simple window of code selection. One potential direction is to augment the stack traces using static analysis. For example, the code that defines and uses the object, may be more important than lexically-nearby code. These def-use chains may point to the most useful source code statements: the ones that actually affect the object. However, def-use (especially inter-procedural) chains may be difficult to gather on a large scale. This is a trade-off: using an analysis has a time cost, but may produce better predictions.

We select a fixed number of stack frames from the top of the stack, but other stack frames may be more important. Per-frame token budgets also do not need to be fixed. If could rank every token's importance, and we see that a certain function contains many useful tokens, we could increase the frame's token budget (at the expense of another frame).

Finally, we currently ignore comments because of token budget constraints. However, they may provide useful natural language hints to the model.

7.2 Modalities

Token-based representations are not the only option. Given recent success in graph neural networks (GNNs) and GNN-based code embeddings, including a graph embedding of the code might prove very useful. Additionally, there might be better ways of handling the different modalities. For example, we used separate Transformers for each embedding type, then a maxpool operation to combine the embeddings per-frame. However, an attention model might be better than a maxpool, or it might be better to keep all of the embeddings rather than aggregating them per-frame. There is a large design space still to be explored.

7.3 Labelling Examples

Stack trace-based representation can be ambiguous. Many objects can be allocated at the same site, and will be represented by the same stack trace even if they behave differently. While some tasks might be able to tolerate this label ambiguity by choosing one label (*e.g.* most common or max), others might not. One idea is to augment the data with some dynamic features, for example, GC-related data, current CPU load, or current memory load. While not available to offline-only predictions, this would be a useful way to disambiguate object behavior, even if they come from the same stack trace.

8 Conclusion

In this paper, we present a framework for reasoning about the design space of predicting heap allocation properties with machine learning. We believe that our paper provides evidence that this is a promising approach, but a number of challenges need to be solved to make it practical. We hope that this intellectual abstract opens up a new research direction for the ISMM community and that our discussions of challenges and trade-offs in the design space of this problem will lead to more work that takes advantage of advancements in code embedding models within memory managers.

Acknowledgments

We thank the anonymous reviewers for their insightful and thorough comments. We would also like to thank Steve Blackburn and Chandu Thekkath for their feedback.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJOFETxR-

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. arXiv:1808.01400 [cs.LG]

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. code2vec: Learning Distributed Representations of Code. https://doi.org/10.48550/ARXIV.1803.09473

[6] David A. Barrett and Benjamin G. Zorn. 1993. Using Lifetime Predictors to Improve Memory Allocation Performance. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 187–196. https://doi.org/10.1145/155090.155108

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[8] Stephen M. Blackburn, Matthew Hertz, Kathryn S. Mckinley, J. Eliot B. Moss, and Ting Yang. 2007. Profile-Based Pretenuring. *ACM Trans. Program. Lang. Syst.* 29, 1 (jan 2007), 2–es. https://doi.org/10.1145/1180475.1180477

[9] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinely, and J. Eliot B. Moss. 2001. Pretenuring for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Tampa Bay, FL, USA) *(OOPSLA '01)*. Association for Computing Machinery, New York, NY, USA, 342–352. https://doi.org/10.1145/504282.504307

[10] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. 2016. Enriching Word Vectors with Subword Information. *CoRR* abs/1607.04606 (2016). arXiv:1607.04606 http://arxiv.org/abs/1607.04606

[11] Rodrigo Bruno, Duarte Patricio, José Simão, Luis Veiga, and Paulo Ferreira. 2019. Runtime Object Lifetime Profiler for Latency Sensitive Big Data Applications. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 28, 16 pages. https://doi.org/10.1145/3302424.3303988

[12] Brad Calder, Peter Feller, and Alan Eustace. 1999. Value Profiling and Optimization. *Journal of Instruction Level Parallelism* 1 (1999).

[13] Binghong Chen, Daniel Tarlow, Kevin Swersky, Martin Maas, Pablo Heiber, Ashish Naik, Milad Hashemi, and Parthasarathy Ranganathan. 2022. Learning to Improve Code Efficiency. arXiv:2208.05297 [cs.SE]

[14] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications. In *CGO 2016 Proceedings of the 2016 International Symposium on Code Generation and Optimization*. New York, NY, USA, 12–23.

[15] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2021. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. arXiv:arXiv:2104.08308

[16] Trishul M. Chilimbi and Martin Hirzel. 2002. Dynamic Hot Data Stream Prefetching for General-Purpose Programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 199–209. https://doi.org/10.1145/512529.512554

[17] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. 2003. Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis. *ACM Trans. Program. Lang. Syst.* 25, 6 (nov 2003), 876–910. https://doi.org/10.1145/945885.945892

[18] François Chollet et al. 2015. Keras. https://keras.io.

[19] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. 2015. Memento Mori: Dynamic Allocation-Site-Based Optimizations. *SIGPLAN Not.* 50, 11 (jun 2015), 105–117. https://doi.org/10.1145/2887746.2754181

[20] David Cohn and Satinder Singh. 1996. Predicting Lifetimes in Dynamically Allocated Memory. In *Advances in Neural Information Processing Systems*, M.C. Mozer, M. Jordan, and T. Petsche (Eds.), Vol. 9. MIT Press. https://proceedings.neurips.cc/paper/1996/file/a9078e8653368c9c291ae2f8b74012e7-Paper.pdf

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2018).

[22] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 990–998. https://proceedings.mlr.press/v70/devlin17a.html

[23] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2021. Structured Neural Summarization. arXiv:1811.01824 [cs.LG]

[24] Andreas Gal, Christian W. Probst, and Michael Franz. 2006. HotpathVM: An Effective JIT Compiler for Resource-Constrained Devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments* (Ottawa, Ontario, Canada) *(VEE '06)*. Association for Computing Machinery, New York, NY, USA, 144–153. https://doi.org/10.1145/1134760.1134780

[25] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. 1995. Profile-Guided Receiver Class Prediction. *SIGPLAN Not.* 30, 10 (Oct. 1995), 108–123. https://doi.org/10.1145/217839.217848

[26] Ameer Haj-Ali, Qijing Jenny Huang, John Xiang, William Moses, Krste Asanovic, John Wawrzynek, and Ion Stoica. 2020. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. *Proceedings of Machine Learning and Systems* 2 (2020), 70–81.

[27] Timothy L. Harris. 2000. Dynamic Adaptive Pre-Tenuring. In *Proceedings of the 2nd International Symposium on Memory Management* (Minneapolis, Minnesota, USA) *(ISMM '00)*. Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/362422.362476

[28] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena

Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 152–162. https://doi.org/10.1145/3236024.3236051

[29] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global Relational Models of Source Code. In *International Conference on Learning Representations*. https://openreview.net/forum?id=B1lnbRNtwr

[30] Matthew Hertz, Stephen M Blackburn, J Eliot B Moss, Kathryn S. McKinley, and Darko Stefanović. 2002. Error-Free Garbage Collection Traces: How to Cheat and Not Get Caught. *SIGMETRICS Perform. Eval. Rev.* 30, 1 (jun 2002), 140–151. https://doi.org/10.1145/511399.511352

[31] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[32] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. 2021. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 257–273. https://www.usenix.org/conference/osdi21/presentation/hunter

[33] Teresa Johnson, Mehdi Amini, and Xinliang David Li (Eds.). 2017. *ThinLTO: Scalable and incremental LTO.*

[34] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. https://doi.org/10.48550/ARXIV.2001.00059

[35] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 12-18 July 2020 (Proceedings of Machine Learning Research)*. PMLR.

[36] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 158–169. https://doi.org/10.1145/2749469.2750392

[37] Hugh Leather and Chris Cummins. 2020. Machine Learning in Compilers: Past, Present and Future. In *2020 Forum for Specification and Design Languages (FDL)*. 1–8. https://doi.org/10.1109/FDL50818.2020.9232934

[38] Bolun Li, Pengfei Su, Milind Chabbi, Shuyin Jiao, and Xu Liu. 2023. DJXPerf: Identifying Memory Inefficiencies via Object-Centric Profiling for Java. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (Montréal, QC, Canada) *(CGO 2023)*. Association for Computing Machinery, New York, NY, USA, 81–94. https://doi.org/10.1145/3579990.3580010

[39] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. 2016. Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 383–400. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/lion

[40] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[41] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-based Memory Allocation for C++ Server Workloads. In *25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[42] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 843–857.

[43] Todd Mytkowicz, Devin Coughlin, and Amer Diwan. 2009. Inferred Call Path Profiling. *SIGPLAN Not.* 44, 10 (oct 2009), 175–190. https://doi.org/10.1145/1639949.1640102

[44] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 349–365.

[45] Deok-Jae Oh, Yaebin Moon, Eojin Lee, Tae Jun Ham, Yongjun Park, Jae W. Lee, and Jung Ho Ahn. 2021. MaPHeA: A Lightweight Memory Hierarchy-Aware Profile-Guided Heap Allocation Framework. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Virtual, Canada) *(LCTES 2021)*. Association for Computing Machinery, New York, NY, USA, 24–36. https://doi.org/10.1145/3461648.3463844

[46] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-Symbolic Program Synthesis. https://doi.org/10.48550/ARXIV.1611.01855

[47] Pardis Pashakhanloo, Aaditya Naik, Hanjun Dai, Petros Maniatis, and Mayur Naik. 2022. Learning to Walk over Relational Graphs of Source Code. In *Deep Learning for Code (DL4C) Workshop at the International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=SubGAoOWJWc

[48] Pardis Pashakhanloo, Aaditya Naik, Yuepeng Wang, Hanjun Dai, Petros Maniatis, and Mayur Naik. 2022. CodeTrek: Flexible Modeling of Code using an Extensible Relational Representation. In *International Conference on Learning Representations*. https://openreview.net/forum?id=WQc075jmBmf

[49] Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-based Bug Detection. arXiv:arXiv:1805.11683

[50] Veselin Raychev, Martin Vechev, and Andreas Krause. 2019. Predicting Program Properties from 'Big Code'. *Commun. ACM* 62, 3 (Feb. 2019), 99–107. https://doi.org/10.1145/3306204

[51] Nadav Rotem and Chris Cummins. 2021. Profile Guided Optimization without Profiles: A Machine Learning Approach. https://doi.org/10.48550/ARXIV.2112.14679

[52] Joe Savage and Timothy M. Jones. 2020. HALO: Post-Link Heap-Layout Optimisation. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 94–106. https://doi.org/10.1145/3368826.3377914

[53] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. 2021. MLGO: a Machine Learning Guided Compiler Optimizations Framework. https://doi.org/10.48550/ARXIV.2101.04808

[54] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural Program Repair by Jointly Learning to Localize and Repair. arXiv:arXiv:1904.01720

[55] Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. 2018. Tensor2Tensor for Neural Machine Translation. *CoRR* abs/1803.07416 (2018). http://arxiv.org/abs/1803.07416

[56] Yijian Wang and David Kaeli. 2003. Profile-Guided I/O Partitioning. In *Proceedings of the 17th Annual International Conference on Supercomputing* (San Francisco, CA, USA) *(ICS '03)*. Association for Computing Machinery, New York, NY, USA, 252–260. https://doi.org/10.1145/782814.782850

[57] Guoqing Xu. 2013. Resurrector: A Tunable Object Lifetime Profiling Technique for Optimizing Real-World Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 111–130. https://doi.org/10.1145/2509136.2509512

[58] Giulio Zhou and Martin Maas. 2021. Learning on Distributed Traces for Data Center Storage Systems. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 350–364. https://proceedings.mlsys.org/paper/2021/file/82161242827b703e6acf9c726942a1e4-Paper.pdf

Received 2023-03-03; accepted 2023-04-24