Quantitative Overhead Analysis for Python

Mohamed Ismail and G. Edward Suh
Cornell University
Ithaca, NY, USA
{mii5, gs272}@cornell.edu

***Abstract*—Dynamic programming languages such as Python are becoming increasingly more popular, yet often show a significant performance slowdown compared to static languages such as C. This paper provides a detailed quantitative analysis of the overhead in Python without and with just-in-time (JIT) compilation. The study identifies a new major source of overhead, C function calls, for the Python interpreter. Additionally, we study the interaction of the run-time with the underlying processor hardware and find that the performance of Python with JIT depends heavily on the cache hierarchy and memory system. We find that proper nursery sizing is necessary for each application to optimize the trade-off between cache performance and garbage collection overhead. Although our studies focuses on Python, we show that our key findings can also apply to other dynamic languages such as Javascript.**

I. Introduction

As software becomes more complex and the costs of developing and maintaining code increase, dynamic programming languages are becoming more desirable alternatives to traditional static languages. Dynamic languages allow programmers to express more functionality with less code. In addition, run-time checks and memory management are built-in, limiting the possibility of low-level program bugs such as buffer overflows. Dynamic languages such as Javascript, Python, PHP, and Ruby consistently rank in the top ten most popular languages across multiple metrics [1]–[3]. These dynamic languages are increasingly utilized in production environments in order to bring new features quickly.

Unfortunately, programs written in a dynamic language often execute significantly slower than an equivalent program written in a static language, sometimes by orders of magnitude. Therefore, the performance overhead represents a major cost of using dynamic languages in high-performance applications. Ideally, companies with enough time and resources may rewrite performance-critical portions of code in faster static languages when they are mature. For example, Twitter initially used Ruby on Rails to build their infrastructure and reported a 3x reduction in search latencies after rewriting it in Java [4]. However, porting code is an expensive proposition and just-in-time (JIT) compilation is often used as a lower-cost alternative to improve the performance of dynamic language programs.

In this paper, we provide a quantitative study on the sources of performance overhead in Python, a popular dynamic language. The overhead of a dynamic language can come from multiple aspects of the language design space. This study explores three different aspects of the overhead to provide a more comprehensive view. First, at the language level, some features of the dynamic language may lead to inherent inefficiency compared to static languages. Second, a language run-time also adds overhead to dynamic languages compared to statically compiled code. We break down Python execution time into language and run-time components as well as core computations to understand overhead sources. Finally, at the hardware level, we study how the dynamic language features impact microarchitecture-level performance through instruction-level parallelism, branch prediction, and memory access characteristics. We compare CPython [5], an interpreter-only design, with PyPy [6], a JIT-based design, to understand the microarchitecture-level differences between the run-time implementations.

The study is broken into two main parts. In the first part, we look at the language and run-time features of Python to understand which aspects of the language and run-time add additional overhead compared to C, the baseline static language. By annotating instructions at the interpreter-level, we can generate breakdowns for a large number of benchmarks. In addition to the sources of overhead already identified by previous work, we find that C function calls represent a major source of overhead that has not been previously identified.

In the second part, we study the interaction of the run-time with the underlying processor microarchitecture. We find that both CPython and PyPy exhibit low instruction-level parallelism. Using PyPy with JIT helps decrease sensitivity to branch predictor accuracy, but increases sensitivity to cache and memory configurations. In particular, we find that the generational garbage collection used in PyPy introduces an inherent trade-off between cache performance and garbage collection overhead. Frequent allocation of objects in dynamic languages increases a pressure on the memory hierarchy. However, increasing the garbage collection frequency to improve cache performance can lead to high garbage collection overhead. Our study shows that the optimal nursery size depends upon application characteristics as well as run-time and cache configurations. If the nursery is sized considering the cache performance and garbage collection overhead trade-off, then there can be significant improvements in program performance.

While we focus primarily on Python for most of our studies, we believe that the main results from our studies are applicable to other dynamic languages as well. For a subset of our findings, we show that the main lessons also apply to V8 [7], a high-performance run-time for Javascript.

The major contributions of this paper include:

1) A comprehensive breakdown study of the CPython interpreter execution time for a large number of benchmarks.

2) Microarchitectural parameter sweeps to better understand which aspects of hardware designs affect performance of both the interpreter-only CPython and PyPy with and without JIT.
3) Analysis of the trade-off of cache performance and garbage collection time for PyPy.

We gain the following new insights regarding the opportunities to improve the performance of Python and other dynamic languages:

1) We find that C function calls represent a major source of overhead not previously identified.
2) Our microarchitectural study shows that dynamic languages exhibit low instruction-level parallelism but presence of JIT lowers sensitivity to branch predictor accuracy and increases sensitivity to memory system performance.
3) We find that nursery sizing has a large impact on dynamic language performance and needs to be done in an application-specific manner, considering the trade-off between cache performance and garbage collection overhead, for the best result.

Our paper is organized as follows. Section II introduces backgrounds on run-time designs. Section III explains our experimental setup. Section IV discusses our study on the sources of overhead for Python. Section V analyzes the interaction between the run-time and the underlying hardware. Section VI discusses related work, and Section VII concludes the paper.

II. Background on Run-Time Design

In this section, we present some background on essential aspects of dynamic languages. Unlike static languages, most dynamic languages translate source code at run-time to an intermediate representation. The language run-time interprets the intermediate representation to execute the program. Since interpretation is slow, just-in-time compilation can be used to further compile the intermediate representation to machine code. Regardless of the execution strategy, automatic memory management ensures that memory is allocated for objects as needed without explicit calls by the programmer. Garbage collection amortizes the cost of freeing memory for dead objects.

A. Basic Interpreter Design

A basic interpreter reads a series of bytecodes and executes necessary actions for each bytecode. Figure 1 shows the stages of interpreter execution for CPython, which is implemented in C as a stack-based virtual machine (VM) with some enhancements to support language-specific constructs. The VM uses `opptr` to index into the bytecode array (`co_code`) and retrieve the appropriate bytecode. The bytecode is then decoded using a *switch-case* construct. Data is read from the stack or other storage variables. The operation specified by the bytecode will be executed using the read data as operands. Some error-checking code will ensure that the execution completed successfully. Finally, data will be written back to the stack or other storage variables. `opptr` will be updated and the process will repeat until the program completes.

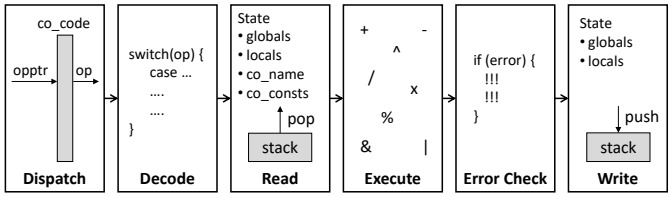


Fig. 1: Overview of CPython virtual machine architecture.

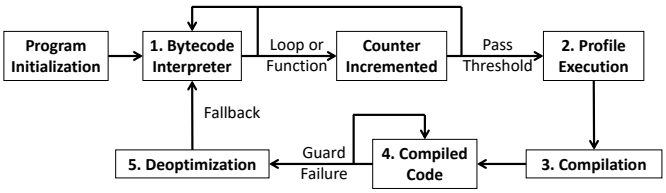


Fig. 2: Steps in just-in-time compilation.

B. Just-in-Time Compilation

Just-in-time compilation can optimize run-time performance by converting interpreted bytecode to machine code. In addition, run-time information about object types and values can be used to perform additional optimizations that cannot be done ahead-of-time. Running the just-in-time compiler during run-time is relatively expensive and the cost of compilation must be amortized by the performance improvement in the compiled code. For this reason, JIT compilers focus on frequently executed code, such as frequently executed loops or functions.

As shown in Figure 2, counters are used to track the number of times that loops or functions execute. Once the counter reaches a threshold, the loop or function is considered a good candidate for compilation. An additional profiling stage collects information for the compiler optimizations. The code is then compiled and the machine code is executed in place of the interpreted bytecode.

To generate optimized code, the compiler makes assumptions about variable types and values, and it inserts guards to check whether those assumptions are valid during the execution. If there is a failed guard, the compiled state is rolled back to a valid interpreted state and the bytecode interpreter continues execution. This is called *deoptimization* and is a relatively expensive operation that could affect overall performance if it occurs too frequently. Some additional steps can be added to the JIT process to better handle guard failures and optimize a portion of a function or loop if a certain guard continues to fail.

C. Generational Garbage Collection

In a language with automatic memory management, garbage collection is often used to free memory from objects that are no longer in use. The process of determining which objects are live and which are not incurs non-trivial performance overhead. In order to amortize this overhead, garbage collection should be run at infrequent intervals.

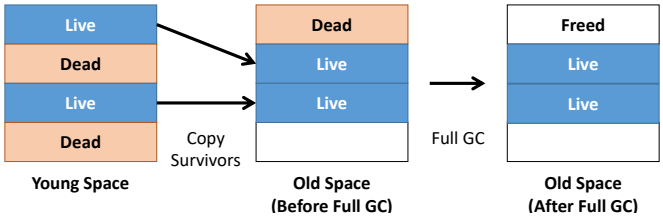


Fig. 3: Graphical depiction of how generational garbage collection works.

TABLE I: ZSim configuration.

Core	4-way OOO, 16B Fetch, 3.40GHz
	2-level 2-bit BP with 2048x18b L1, 16384x2b L2
	224 ROB, 72 Load-Q, 56 Store-Q
L1I	64 kB, 8-way, 4-cycle latency
L1D	64kB, 8-way, 4-cycle latency
L2	256kB, 4-way, 12-cycle latency
L3	2MB, 16-way, 42-cycle latency
Memory	16GB DDR4-2400, 173-cycle latency

To determine which objects are live, the garbage collector will start from a set of root pointers and follow them. This step is often called *tracing*. It will then follow additional pointers that it encounters in the process. Objects with pointers pointing to them are live, while objects without any valid pointers pointing to them are dead and can be collected. In a managed language, the run-time has knowledge of all pointers in the system unlike low-level languages such as C or C++, where any variable can be dynamically cast to represent a pointer [8]. Once the live and dead objects are differentiated, the garbage collector will free memory corresponding to dead objects.

Generational garbage collection [9] is an optimized form of garbage collection that is used in many high performance implementations of modern languages. The memory is separated into subspaces based on object age and different garbage collection algorithms can run on the different subspaces. In the simplest implementation of generational garbage collection, there is one subspace for young objects, sometimes called a *nursery*, and another subspace for old objects. Objects are allocated in the nursery and are moved to the old subspace if they survive long enough. Figure 3 shows how generational garbage collection works.

Efficient generational garbage collection relies on the assumption that most objects in a program die young. Therefore, a copying garbage collector [10] can efficiently move a small number of surviving objects from the nursery to the old subspace. Once the object is in the old subspace, a slower garbage collector, such as a mark-sweep [11] collector, can run less frequently. This can be extended to any number of subspaces based on age. Real implementations of generational garbage collection add variations to this general scheme. For example, the PyPy collector runs the mark-sweep collector incrementally in the old subspace [12].

III. Experimental Setup

We run our experiments on an infrastructure based on Pin [13]. The Pin framework allows us to instrument the various run-times at both the instruction-level and function-level without having to modify the source code and without affecting the instructions executed by the program. We develop Pin tools to capture dynamic instruction counts and other statistics needed for our analysis.

To get cycle count estimates for a variety of memory hierarchies and core configurations, we interface our Pin tools with Zsim [14], a fast x86-64 simulator built on Pin. We run Zsim with a configuration that mimics an Intel Skylake processor. The details of the configuration are shown in Table I. We use an out-of-order core model (OOO) for most of our experiments. For the sources of overhead experiments, we use the simple core model to be able to accurately map individual instructions to their cycle contributions. We assume that each of the four physical cores has one-quarter of the 8MB shared L3 cache available for use. We use DRAMSim2 [15] integrated with ZSim to model DDR4-2400 memory.

For run-times, we use CPython 2.7.10 as our Python interpreter with the standard compiler optimization flags (-O3) and PyPy 5.3.1 as our JIT-based run-time for Python. We use 48 benchmarks gathered from the official Python performance benchmark suite [16] and from the PyPy benchmark suite. The designers of the official Python performance benchmark suite say that it focuses on real-world benchmarks, using whole applications when possible, rather than synthetic benchmarks. We warm up each benchmark by running it 2 times followed by running it 3 times for evaluation.

For some experiments, we additionally use Google V8 4.2.0, a popular high-performance Javascript run-time that uses JIT compilation. For test programs, we use 37 benchmarks from JetStream [17], which "combines a variety of JavaScript benchmarks, covering a variety of advanced workloads and programming techniques" including SunSpider, Octane, and LLVM. We run each benchmark 3 times for evaluation.

IV. Sources of Overhead

In this section, we perform a quantitative study on the overhead of Python compared to static languages such as C. We first categorize various sources of overhead, describe our methodology to experimentally measure a breakdown of a Python execution time, and discuss main findings. The results in this section are reported for CPython [5], the official Python interpreter. We also explore how some of the findings are applicable to both PyPy and V8.

A. Overhead Breakdown

Table II shows the overhead sources that we identify in this study through careful review of language features as well as CPython source code. The overhead categories can be placed into three groups. The language features of Python may incur overhead because they either do not exist in a static language or require additional dynamic operations. The interpreter itself also adds additional performance overhead that compiled code would not have. A majority of the features have been previously identified and evaluated either directly or indirectly (e.g. through an optimization). We list references in the table to the previous work which evaluates the features. In addition, we have identified three new overhead categories

TABLE II: Sources of performance overhead for Python.

Group	Overhead category	Description	Studied by
Additional Language Features	Error check	Check for overflow, out-of-bounds, and other errors	*NEW*
	Garbage collection	Automatically freeing unused memory	[18], [19]
	Rich control flow	Support for more condition cases and control structures	[19], [20]
Dynamic Language Features	Type check	Checking variable type to determine operation	[18], [21]
	Boxing/unboxing	Wrapping or unwrapping integer or float types	[18], [21]
	Name resolution	Looking up variable in a map	[20]
	Function resolution	Dereferencing function pointers to perform an operation	[20]
	Function setup/cleanup	Setting up for a function call and cleaning up when finished	[18]–[20]
Interpreter Operations	Dispatch	Reading and decoding bytecode instruction	[20], [22]
	Stack	Reading, writing, and managing VM stack	[20], [21]
	Const load	Reading constants	[20]
	Object allocation	Inefficient deallocation followed by allocation of objects	[19]
	Reg transfer	Calculating address of VM storage	*NEW*
	C function call	Following the C calling convention in the interpreter	*NEW*

not evaluated in previous work. The different components and overhead categories are described further in the following subsections.

1) Additional Language Features: This category consists of language features that do not exist in static languages such as C. The `errorcheck` overhead comes from run-time checks that Python performs to guarantee safety and robustness. After an operation, Python performs checks such as an overflow check on the `int` types and bound checks on the `list` types. The `garbage collection` overhead comes from operations for run-time garbage collection such as maintaining reference counters and freeing memory. The `rich control flow` overhead results from checking various conditions in the case of richer evaluation of condition or for managing the block stack in the case of support for more control structures.

2) Dynamic Language Features: This category captures language features that exist in C but requires additional run-time operations in Python. A majority of these features are managed statically in C at compile time. Setting up a function call and cleaning up on a return is done dynamically in C through the calling convention, but it requires significantly more computation in Python. The overheads in this group would still be present even if Python programs were compiled ahead-of-time because the compiler lacks necessary run-time information. Python uses dynamic typing, so types of the variables and where they are allocated are not known until run-time. Python cannot resolve types of variables statically because they are not explicitly given in the program. In addition, the variables with unknown types cannot be allocated statically so the locations of variables are only known dynamically.

The `typecheck` overhead relates to all checks the interpreter must perform to determine the type of the variable. In Python there is usually a check for variable type before an operation is performed on the object. The `boxing and unboxing` overhead relates to reading integer and float primitives values from the object and writing back these primitive values to the object. These primitives would normally be stored in machine registers for a C program, but are represented as objects with type information in Python. For example, in an add operation, the values of the two variables to be added will be read from the corresponding object. The sum will be computed and will be written back to another object representing the sum.

The `name resolution` overhead relates to looking up the variable pointer in a map by using the variable name as the key. Types of global variables are not known and they can be created and destroyed dynamically, so Python uses maps to store pointers to the variables. The `function resolution` overhead relates to dereferencing of function pointers. Functions in Python are first class objects that can be created and destroyed, so Python stores function pointers for common operations related to an object.

The `function setup/cleanup` overhead relates to setting up a call to a function and cleaning up on a return. In order to setup up a call, Python needs to determine the function type (both Python and C functions are supported). If it is a C function, then the inputs passed in through the C extension interface and the output needs to be returned. If it is a Python function, an execution frame for the function needs to be allocated. Functions that require variable arguments require special attention. Once the function returns, Python needs to deallocate the frame and pass the return value to the caller.

3) Interpreter Operations: In addition to categories relating to language features, there are categories related to the overhead of running the interpreter. These relate to the cost of emulating a virtual machine on a physical machine. The `dispatch` overhead relates to reading the bytecode and decoding it to perform the correct operation. This includes the execution of the dispatch loop and a switch statement for decoding.

CPython is a stack-based virtual machine. The `stack` overhead relates to operations for managing the stack. Operations read from the stack and write to the stack. The stack is local storage for the VM similar to the register file for the CPU. The stack is not meant to store program state, but act as local storage for intermediate values. There are some bytecodes for explicitly managing the stack, such as `DUP_TOP` which duplicates the top entry. In addition to the stack, there are data structures which store constant values. The `const load` overhead is the overhead of loading constants to the stack. Constants are stored in the `co_const` array. The values first need to be loaded to the stack before they are used by other bytecodes.

In the interpreter implementation, there are certain objects that could be reused but are instead deallocated and reallocated. The `object allocation` overhead captures the case an object is deallocated then reallocated. For example, most method frames are allocated during execution of the method and deallocated when it finishes. In addition, arithmetic operations take operands from the stack and generate a new value. When the operation completes, the original operands are deallocated and a new object is allocated for the value.

Since CPython is written in C, there may be additional inefficiencies introduced by the compiler. The `C function call` overhead captures the additional cost of setting and cleaning up C functions in the interpreter. This includes the cost of creating and destroying stack frames and performing the call. The use of a C function to write good refactored code results in many function calls per bytecode instruction. These calls cannot be inlined in most cases because function pointers are used.

When reading a VM data structure, such as the stack, the CPU will first load the address of the data structure first to the machine registers. Then it will compute the effective address of the Python variable (e.g. top of stack). Finally, it will load the Python variable into the machine register. This additional step of finding the data structure of the VM and loading it to machine registers is categorized as `reg transfer`.

B. Analysis Approach

Once we define our overhead categories, our goal is to develop an analysis tool that can return the contribution of each category to the overall execution time. Our approach takes advantage of the fact that the Python program is running on a statically-compiled interpreter. We annotate each instruction of the interpreter with a category label. When running the program with our annotated interpreter, we can generate a breakdown of the time spent in each category for any Python program with no additional effort.

Our annotations of the execution must relate to the execution of the whole Python program and not just the sources of overhead. Some instructions can be directly annotated with the overhead categories summarized in Table II. Other instructions are needed to execute the program and cannot be annotated with an overhead category. For example, a Python `BINARY_ADD` bytecode has overheads associated with type checking, unboxing, error checking, etc., but also performs an ALU add operation between the two variables. We annotate the instructions needed to execute the program with an `execute` label. Our analysis breakdown includes the contribution of the `execute` category in addition to the overhead categories.

Annotating each static instruction alone cannot provide an accurate breakdown. There are cases where a function's annotation depends on the calling function. For example, CPython uses the same dictionary lookup function for both looking up a variable in a global map and for performing a lookup operation on a map data structure used in the Python program. In the case of looking up in the global map, the function should be annotated with the `name resolution` overhead category. In the other case, the function should be annotated with the `execute` category. We consider the call sites of these functions to support different labels.

An alternative analysis approach to quantifying the different overheads would be to start with C code of a program and transform it to a Python program while iteratively introducing the necessary language features and implementation details. At each step, we would quantify the slowdown of introducing the additional feature. Based on this, we could better understand the performance gap between Python and C for a given program. This is very tedious and cannot be generalized across many programs. Similarly, we can start with a Python program and introduce more static features into the language to eliminate the dynamic run-time overhead. This would also be tedious and hard to generalize across many programs.

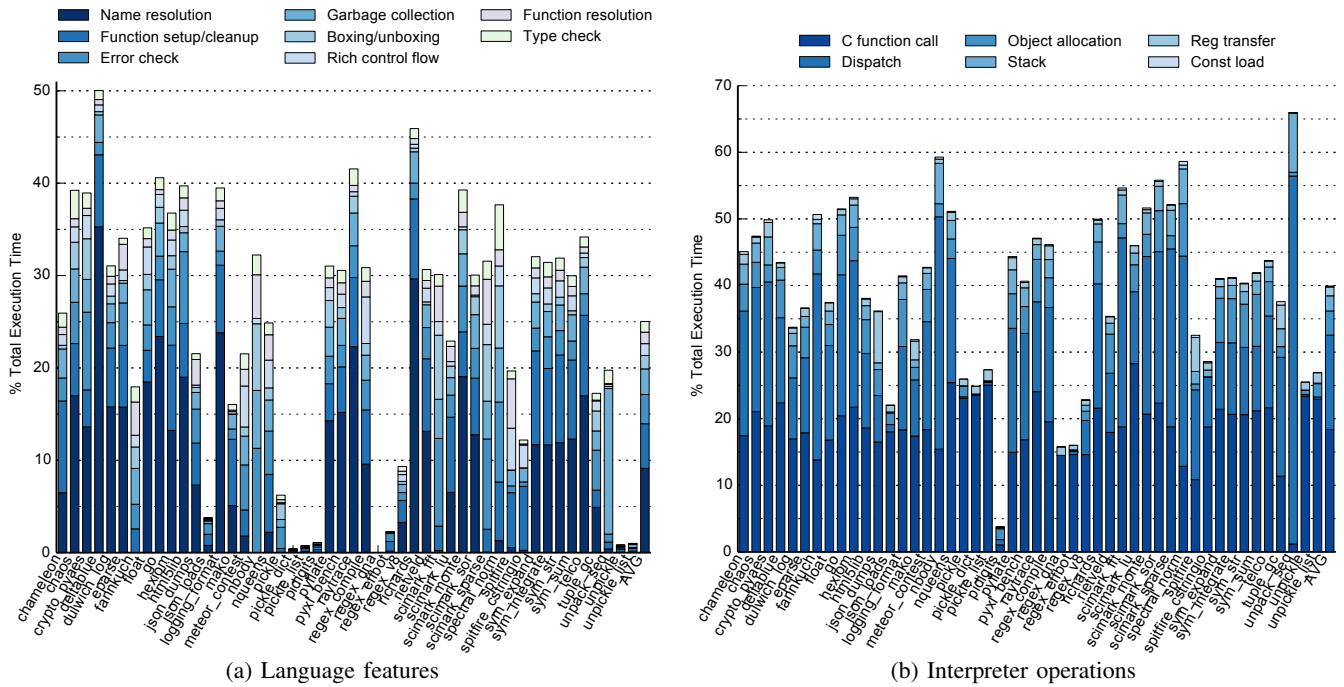
1) Gathering Statistics with Pin: In order to implement the analysis method, we use Pin [13] to instrument the CPython interpreter. To make our analysis more flexible, we write a Pin tool to export essential run-time statistics and perform a post-processing step to generate our breakdown. In our Pin tool, we export statistics for some of the functions in the interpreter at the instruction granularity. For these functions, we export the total execution time of the static instruction at the given PC value.

If a function can be labeled by a single category, then we export statistics at the function granularity to limit the size of the statistics files and to make annotations more feasible. In addition to the function name and total execution time, we also export the origin PC. The origin PC is the most recent PC in the call trace that belongs to a function that we are annotating at an instruction granularity.

Some categories relating to patterns in the assembly code can be automatically annotated by the Pin tool. For example, we can directly use the Pin tool to identify and categorize the assembly instructions relating to the `C function call` and `reg transfer` overhead. We export the total execution time for these categories.

2) Cycle Count Estimates Using Zsim: While instruction count may be a good first-order estimate, it does not capture micro-architectural aspects such as memory latency and branches. We interface our Pin tool with the Zsim simulator (which is also a Pin tool) to get modeled cycle counts in addition to the instruction counts. Zsim has an out-of-order core model that can model an out-of-order pipeline as well as branch mispredicts and cache misses. However, attributing cycle counts to a single instruction becomes challenging for an out-of-order core because the latency of an instruction in the pipeline can be affected by other instructions also in the pipeline.

Instead, we use the simple core model and use the number of cycles each instruction takes to execute. In the simple core model, instruction latency is only affected by misses in the instruction and data caches. Otherwise, an instruction takes a single cycle. This gives us a better first-order estimate of the sources of overhead than just dynamic instruction counts.



(a) Language features

(b) Interpreter operations

Fig. 4: Overhead breakdown for CPython.

3) Post-Processing: During post-processing, we use source line numbers in the interpreter for annotation and map PC values in the exported statistics file to source lines numbers. For functions which we are annotating at the instruction granularity, we annotate each line of the CPython source code with a category. For those functions that we are annotating at the function granularity, we can either annotate only by the name of the function or by both the name of the function and the origin PC if the function category is caller-specific. We need to compile CPython with the (`-g`) flag to be able to match source lines with PC values. After running the post-processing, we get a breakdown of categories and the execution time (in CPU cycles) for those categories.

We only need to annotate the CPython interpreter once and not for each Python program. Since all Python programs run on the interpreter, the PC values and source line number mappings for the interpreter remain the same and the annotations can be reused. As a result, we can analyze and compare a large number of Python programs with the same set of annotations.

C. Experimental Results

1) Execution Time Breakdown: Figure 4(a) shows the contributions of language features (both additional and dynamic) come from many categories, all adding up to a significant portion of the total execution time. Among these categories, `name resolution` and `function setup/cleanup` overheads dominate with 9.1% and 4.8% average overhead respectively. To reduce the impact of `function setup/cleanup`, we could inline Python functions. For `name resolution`, we can cache variable look-ups [20].

Figure 4(b) shows the contributions of interpreter operations to the execution time. We find that `C function calls` and `dispatch` are major contributors overall with 18.4% and 14.2% average overhead. In previous work, `dispatch` has been repeatedly identified as a high source of overhead [22], [23]. However, `C function calls` have not been identified as a major source of overhead in the context of interpreters.

Some work has focused on indirect branches and calls to improve BTB performance in interpreters [24], [25]. Our analysis shows that indirect calls (but not indirect branches) account for an average of 11.9% and up to 19.0% of the C function call overhead, representing an average of 1.9% and up to 4.1% of the overall execution time. Therefore, other aspects of the C function call overhead that account for a larger portion of the execution time, such as setting up and destroying the stack frame, should also be studied and optimized.

On average, the identified overheads account for 64.9% of the overall execution time. The remaining 35.1% is used for the execution of the program. Therefore, there is at least 2.8x increase in execution time on average moving from a C-like program to a Python program running on CPython due to language and interpreter overheads. In reality, the program written in C can run one or two orders of magnitude faster than the program written in Python [19] because the C compiler can further optimize the program using static information about types and memory layout of objects. During execution, the programs spend an average of 7.0% of their overall execution time in C library code. However, benchmarks such as `pickle_dict`, `pickle_list`, `regex_dna`, `regex_effbot`, `regex_v8`, `unpickle`, and `unpickle_list` spend more than 64% of their time in C library code. As a result, there is very little contribution

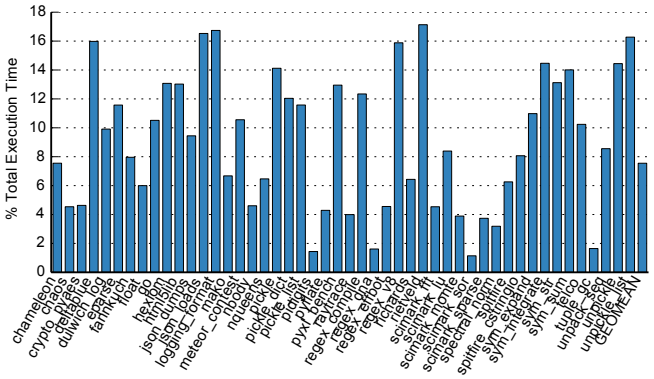


Fig. 5: C function call overhead for PyPy.

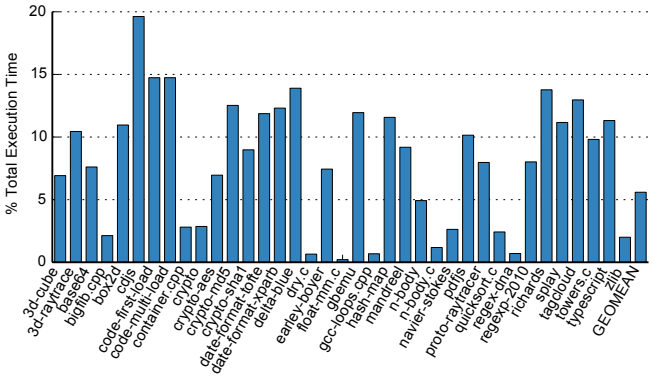


Fig. 6: C function call overhead for V8.

from the overhead categories. `C function call` overhead exists and is still significant even in the C library code.

2) Applicability to Other Run-times and Languages: Since C function call overhead can be automatically annotated by the Pin tool by detecting the instructions related to the calling convention, we extend our tool to analyze the C function call overhead of PyPy and V8. Figure 5 and Figure 6 show that this overhead is significant in these run-times as well with 7.5% and 5.6% average overhead for PyPy and V8 respectively. The JIT compilation helps reduce some of the overhead through inlining methods and generating traces, so the contribution is less than in the interpreter. Based on our results, we believe that optimizing C function call overhead is important for achieving good performance in dynamic languages in general.

V. Hardware Interaction

So far, we have not considered how underlying hardware affects program behavior. In this section we explore how the run-times interact with the underlying hardware. We first study the sensitivity of the run-time performance to various microarchitectural parameters. We find that PyPy performance is sensitive to cache hierarchy and memory system parameters. Since memory management is a key contributor to cache performance, we study the interaction of memory management with the underlying hardware.

A. Microarchitecture Parameter Sweeps

In this section, we explore the sensitivity of run-time performance to various microarchitectural parameters. We run the benchmarks on CPython and PyPy with and without JIT to see if there are differences in the sensitivity between an interpreter-based run-time and a run-time that additionally uses JIT compilation. Figure 7 shows how the CPI (cycle-per-instruction) changes as we sweep various microarchitecture parameters. Here, the average CPI numbers across all benchmarks are shown. The PyPy with JIT execution is additionally broken down into different phases of execution by annotating PyPy at the function granularity using Pin. For the issue width sweep we set the fetch width to be large to prevent it from becoming a bottleneck. The fetch width sweep results are not shown but show a similar trend as issue width.

The results show that the performance of both CPython and PyPy are relatively insensitive to processor fetch width and issue widths, suggesting that there is low instruction-level parallelism. The branch results indicate that merely increasing the branch table size does not improve branch prediction accuracy enough to impact performance. However, when the table is too small and prediction accuracy suffers, the interpreter-based run-times suffer more than a run-time with JIT. This indicates that JIT helps lower sensitivity to branch prediction accuracy.

On the other hand, cache and memory parameters have significant impacts on performance of PyPy with JIT. This indicates that the JIT significantly increases pressure on the memory hierarchy. In particular, the performance depends heavily on cache sizes. Interestingly, the same programs running on the CPython interpreter and PyPy without JIT do not require a large cache. This indicates that the working set of an application itself is not large. Therefore, there is no fundamental reason why the bytecode interpreter and compiled code phases of PyPy with JIT require a large cache to run efficiently. In addition, the CPI for PyPy with JIT is greater than the CPI for CPython and PyPy without JIT. This indicates that while the JIT lowers the number of instructions executed, each instruction takes more cycles to execute due to longer average memory access latency. This is further shown by the sensitivity of the PyPy with JIT to memory latency and bandwidth.

The cache line size sweep shows that PyPy with JIT benefits more using larger cache line sizes, while the interpreter run-times do not. After a closer study, the need for a large cache and cache line sizes appears to come from the interaction of the memory management system with the caches. This observation introduces an interesting opportunity for performance optimization and is discussed in more detail in the subsequent section.

Figure 8 shows the results of microarchitecture parameter sweep when the overall CPI is shown for a few of the benchmarks. The general trend is the same as the previous figure. Yet, this figure shows that the performance impacts of microarchitecture parameter changes depend on individual application characteristics. Note that the benchmarks perform 1.4% better on average with a memory latency of 100 vs. 50 cycles. This is due to a timing anomaly in the out-of-order instruction scheduling. The sweeps for V8, another JIT-based

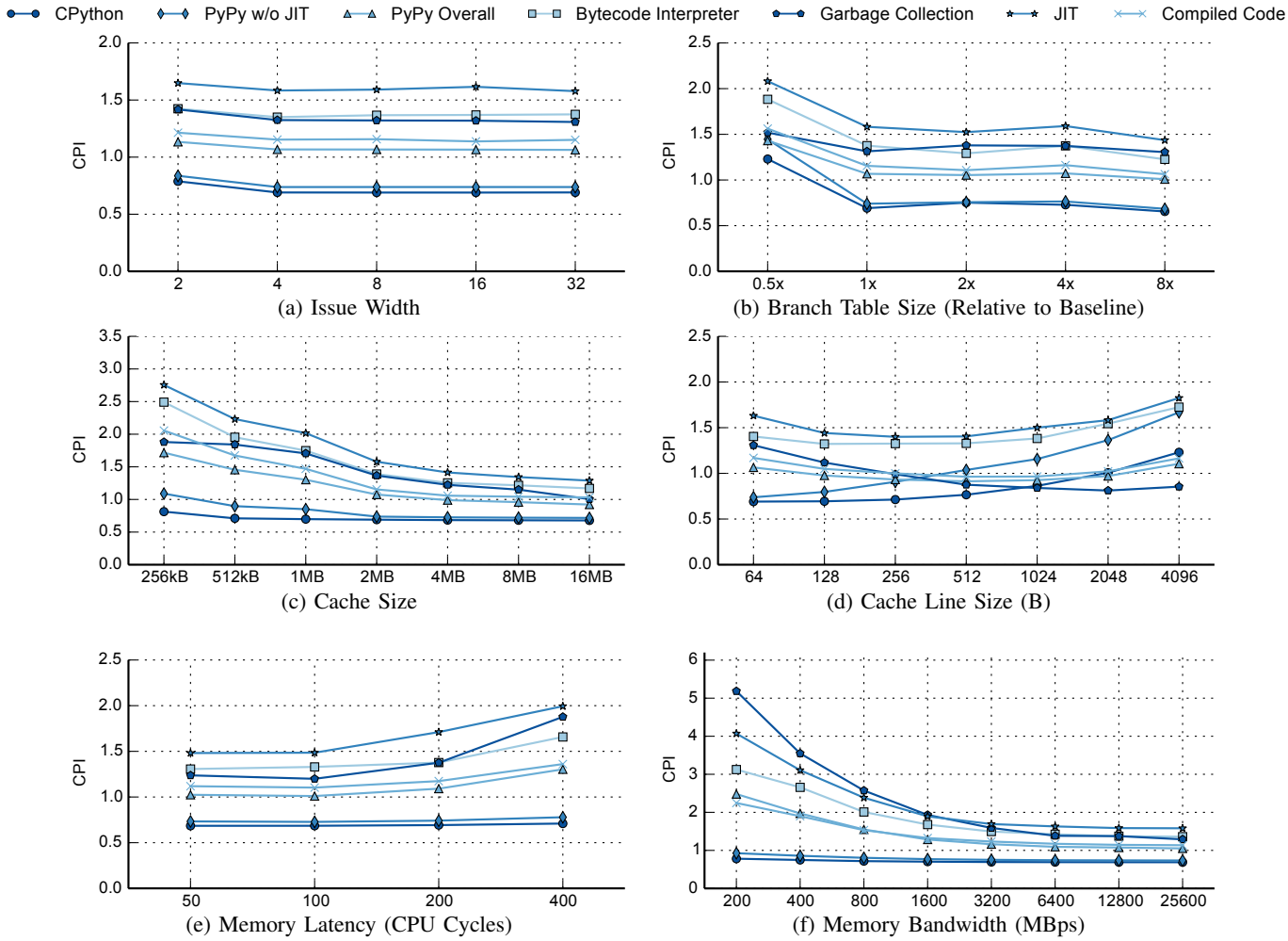


Fig. 7: CPI with microarchitecture parameter sweeps. A line is shown for each run-time as well as phases in PyPy execution.

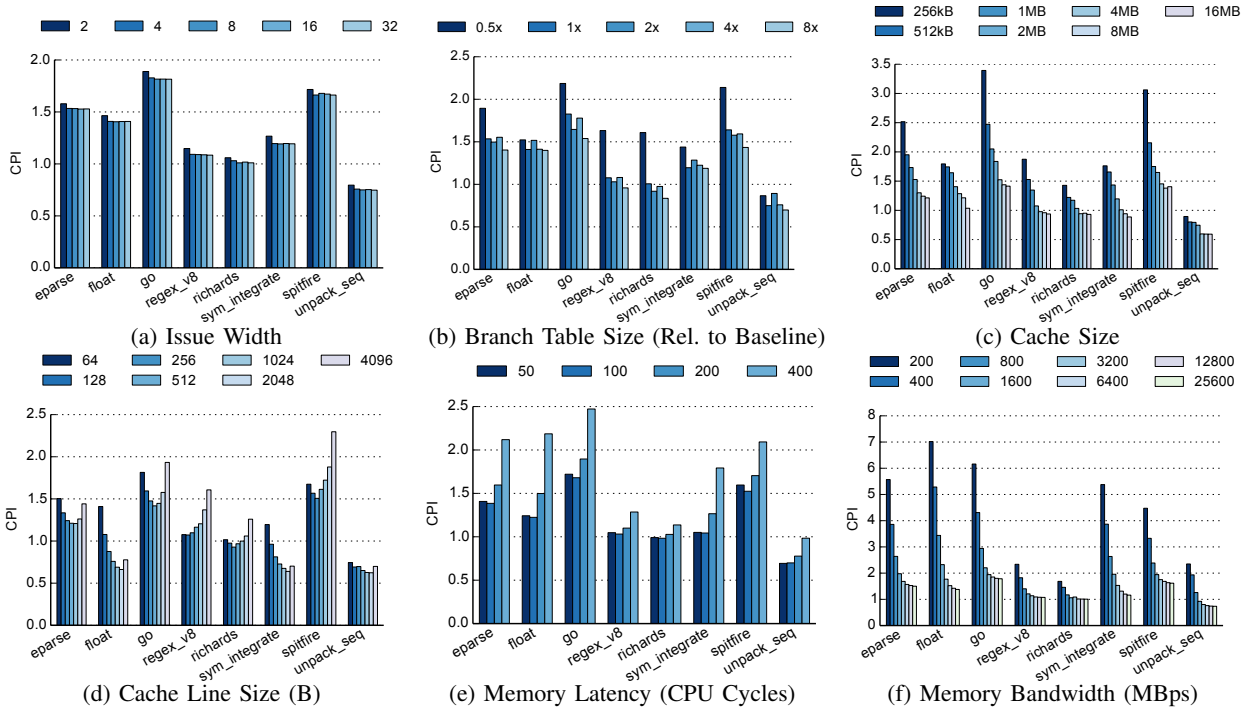
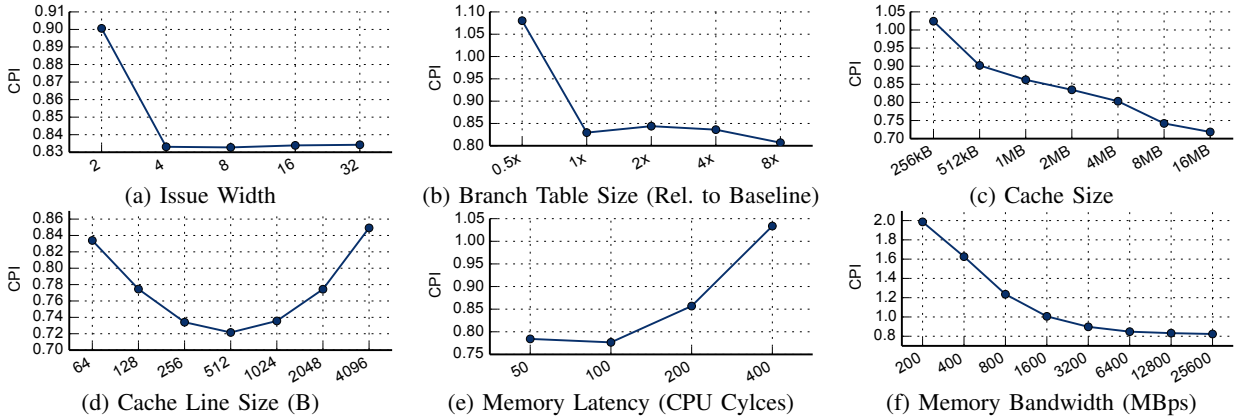


Fig. 8: CPI with microarchitecture parameter sweeps. Each bar shows the overall CPI for one benchmark running on PyPy.



(a) Issue Width (b) Branch Table Size (Rel. to Baseline) (c) Cache Size

(d) Cache Line Size (B) (e) Memory Latency (CPU Cylces) (f) Memory Bandwidth (MBps)

Fig. 9: CPI with microarchitecture parameter sweeps. The line shows the average CPI for V8.

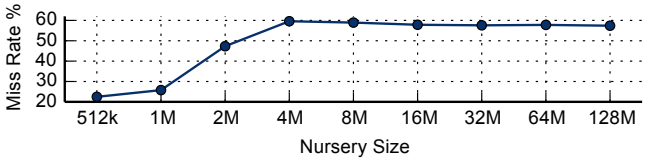


Fig. 10: LLC miss rate as a function of nursery size.

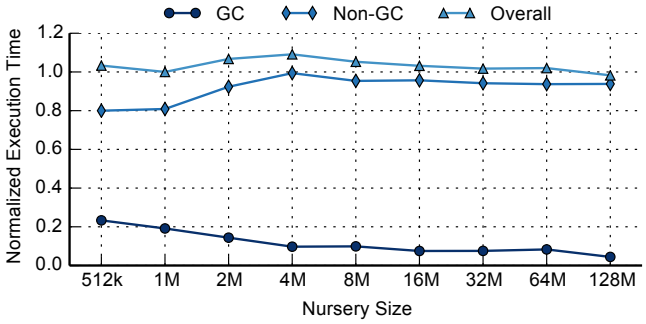


Fig. 11: PyPy execution breakdown for different nursery sizes.

run-time, are shown in Figure 9. They show trends similar to PyPy with JIT indicating that the memory management interaction is important for other JIT-based run-times as well.

B. Memory Management Interaction

After further study of the cache interactions in PyPy, we found that the memory management system contributes to the sensitivity in cache performance. Proper sizing of the nursery is essential to achieve good cache performance. However, sizing the nursery to improve cache performance may not lead to better overall program performance due to the increased overhead from garbage collection. In this section, we consider the interaction of the memory management system with the cache hierarchy in more detail.

Figure 10 shows the last level cache (LLC) miss rates as a function of nursery size. When the nursery is smaller than the cache size (i.e. 2MB), new objects can be allocated directly in the cache and miss rates are low. Once the nursery is too large to fit in the cache, cache trashing occurs and most object initializations miss in the cache. The miss rate increases significantly by almost a factor of 2.4.

Figure 11 shows the breakdown of the execution time normalized to the overall execution time of running with a

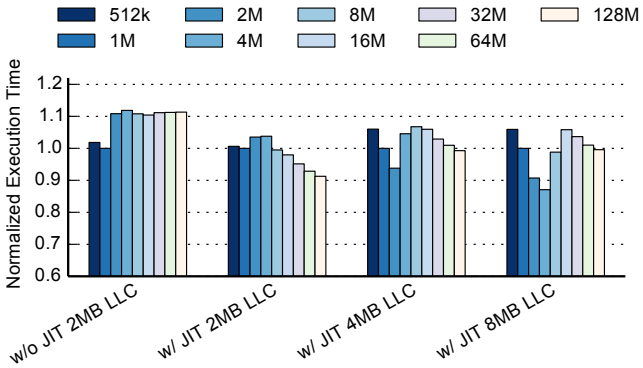


Fig. 12: Nursery sweep for PyPy with different run-time configurations and last level cache sizes.

nursery that is half the cache size (i.e. 1MB nursery for 2MB cache) averaged across all of the benchmarks. It shows that on average the increase in cache miss rate hurts overall performance for nursery sizes slightly larger than the cache size. However, as nursery size increases, the garbage collector runs less frequently and the overhead due to garbage collection decreases. With a much larger nursery, the lower garbage collection overhead offsets the increase in execution time for the rest of the application (i.e. Non-GC) due to the poor cache performance. These results suggest that nursery sizing purely for good cache performance may not always result in good overall performance.

The choice of run-time configuration and the amount of cache space also affect the performance trade-off. Figure 12 shows the average execution time of four configurations for the different nursery sizes normalized to the 1MB nursery case. The first two configurations use a LLC size of 2MB without and with JIT. The next two configurations use PyPy with JIT with different LLC sizes (8MB is the on-chip shared L3 size for Skylake processors).

For PyPy without JIT, the average trend suggests that sizing the nursery size for cache performance is beneficial for overall performance. As shown in Figure 13, this is due to the fact that the contribution of garbage collection to the overall execution time is small. By optimizing the program execution with JIT, the contribution of garbage collection increases by 4.6x from

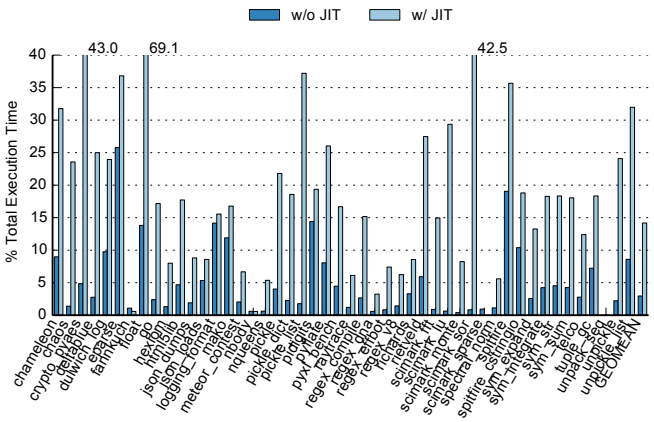


Fig. 13: Garbage collection time as a percent of program execution time.

3% to 14% on average. As a result, sizing the nursery for cache performance can hurt the overall performance due to the larger relative garbage collection overhead. Note that although garbage collection contribution changes significantly when using JIT, the absolute garbage collection time only increases by 5.4% on average when using JIT.

Figure 14 and Figure 15 show the sweeps for individual benchmarks for PyPy with and without JIT respectively. The results suggest that one sizing policy is not good for all the benchmarks and the optimal nursery size also depends on the run-time configuration being used (i.e. with or without JIT). Some applications like `eparse` which have a large garbage collection contribution for both PyPy with and without JIT will benefit from a large nursery. Other applications like `fannkuch` which have low garbage collection contribution for both PyPy with and without JIT will benefit from a nursery sized for good cache performance. There are also some applications like `pyxl_bench` which may benefit from a large nursery size for PyPy with JIT and a small nursery size for PyPy without JIT due to the large change in the garbage collection contribution as a result of running JIT.

Figure 12 also shows that the overall cache size affects the trade-off. With larger cache sizes, a larger nursery can fit in the cache and the better cache performance contributes to better overall performance. Figure 16 shows that this trend also exists for V8 suggesting that this trade-off will be important to explore for implementations beyond PyPy.

Figure 17 shows that by choosing the best nursery size for each application, the normalized execution time can drop by an average 21.4% over the baseline static allocation of half of the cache (i.e. 1MB nursery for 2MB cache). In comparison, simply increasing the nursery to the maximum size for all applications would only result in 9.8% average execution time reduction. These results further suggest that nursery sizing should be done considering cache performance, run-time configuration, and application characteristics.

VI. Related Work

There are a few previous studies that break down and quantitatively analyze the language execution of Python [18], [20],

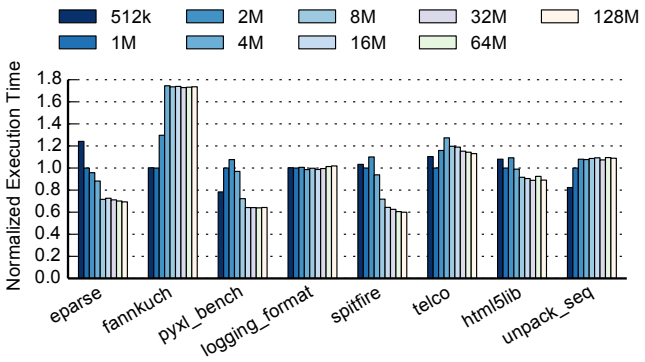


Fig. 14: Nursery sweep for individual benchmarks running on PyPy with JIT.

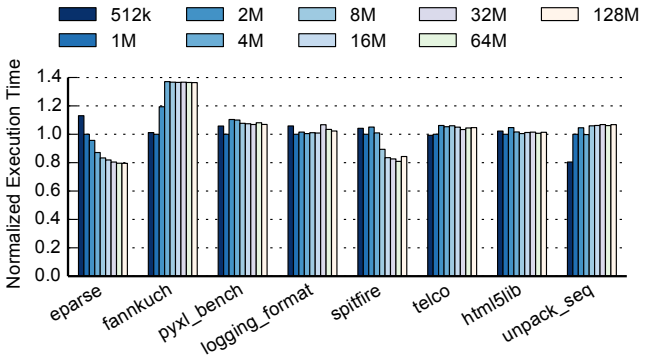


Fig. 15: Nursery sweep for individual benchmarks running on PyPy without JIT.

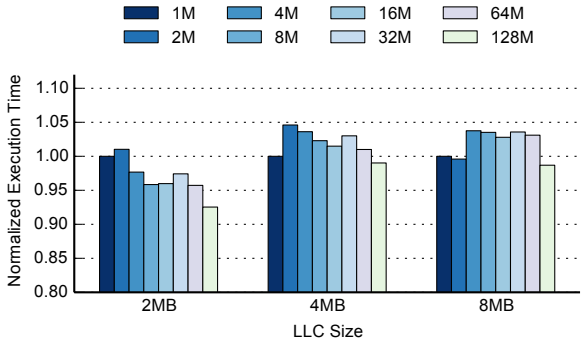


Fig. 16: Nursery sweep for V8 with different last level cache sizes.

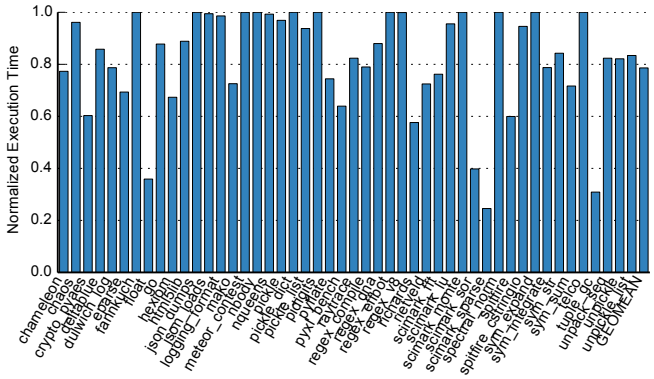


Fig. 17: Normalized execution time for best nursery size per benchmark.

[22]. One study by Barany [18] deconstructs the interpreter performance by identifying common sources of overhead in dynamic languages. Their `pylibjit` tool uses a just-in-time compiler that optimizes Python code to quantify the effects of various sources of overhead. The limitations of the previous work is that they require custom annotations in Python programs and as a result can only perform analysis on a small number of benchmarks. By annotating the interpreter, we can run any Python program and get a breakdown of the execution time. In addition, we are the first to point out C function calls as a significant overhead for CPython.

There exists a number of studies [26]–[28] that use benchmarking to understand which kinds of code work well for different Python implementations. For example, Heynssens [26] has a masters thesis on benchmarking methodology and analysis for Python programs. He draws his conclusions based on the results from various benchmarks running on different Python implementations. These studies compare execution times but don't breakdown sources of overhead.

Some studies have directly proposed modifications to the CPython interpreter to improve its performance. Cao et al. [22] find that Python dispatch overheads can be 25% of the execution time and use pretranslation to get up to 18% improvement. Power and Rubinsteyn [29] converts the stack-based bytecodes to a register-based format that exposes more possibilities for optimization. The optimizations focus on improving a specific aspect of the interpreter instead of breakdown of various overheads.

Ilbeyi et al. [19] analyze the performance of Python in the context of a meta-JIT framework. They present overall execution time comparisons between CPython and PyPy with and without JIT in addition to a detailed breakdown of the overhead in the context of the JIT framework. Our work is complementary as we focus on execution time breakdowns of the interpreter and sweep microarchitectural and run-time parameters. Our findings on sensitivity to cache and memory parameters and the interactions of garbage collection with the cache are new insights for PyPy.

There is more extensive work in Javascript to quantitatively understand the sources of overheads [30]–[36]. Some work provides microarchitectural characterization of Javascript workloads, but they do so to study the differences in benchmark suites and how well the benchmarks match real workloads. For example, Tiwari and Solihin [30] analyze the difference between the Sunspider and V8 benchmarks. Dot et al. [32] identify checks as a major source of overhead in V8. Our findings of C function calls and sensitivity to cache and memory designs, which are generalized to V8, are not discussed in the previous work.

Our work points out C function calls as a major contributor of overhead for dynamic languages. There is much work in improving the speed of indirect branches and calls in the context of C++. Some work rely on intelligent interprocedural analysis and inlining to optimize indirect calls [37]–[40]. Unlike a static language, most of the interprocedural analysis techniques would not work in the context of dynamic languages and new approaches need to be proposed. Other work improves BTB performance for indirect branches [24], [25], [41]–[46]. Casey et al. [24] and Ertl and Gregg [25] identify indirect branches as a significant overhead source in interpreters and discuss how to improve the BTB performance. While their proposed optimizations improve the indirect calls, they would not eliminate the majority of the C function call overhead.

Some previous work in nursery sizing suggests that the nursery should fit in the cache [47], [48], while other work suggests that a larger nursery is better [49]. Reddy et al. [47] identify the trade-off of cache performance and garbage collection overhead. They pin the nursery to the cache to improve the cache performance and overall execution time of the program as well as garbage collection pause times. On the other hand, an in-depth study by Blackburn et al. [49] on the micro-architectural behaviors of various garbage collection algorithms suggests that a larger nursery size will result in better performance. They find that sizing the nursery larger than the last level cache results in lower garbage collection overhead without significant change to the application performance. In our work, we argue that nursery sizing should be done in a manner that considers application-specific characteristics, run-time configuration, and cache performance. In some cases, a smaller nursery will be better while in other cases, a larger nursery could be better for overall performance.

VII. Conclusion

In this paper, we perform an extensive characterization of Python at various levels to provide insight into new opportunities for optimizations in dynamic languages. When looking at the sources of overhead, we find that C function call overhead is repeatedly a major source of overhead in addition to other sources of overhead identified by previous work. In studying the interaction of the run-time with the underlying hardware, we find that PyPy with JIT is sensitive to cache and memory parameters. Through further investigation, we find that nursery size needs to be tuned at the application-specific level, considering the run-time and underlying hardware. While we focus primarily on Python for most of our studies, we believe that the main results from our studies can be applicable to other dynamic languages.

VIII. Acknowledgments

This work was partially supported by the Office of Naval Research grant N00014-15-1-2175.

References

[1] Coding Dojo, "The 9 most in-demand programming languages of 2016," 2016, http://www.codingdojo.com/blog/9-most-in-demand-programming-languages-of-2016/.
[2] Spectrum IEEE, "The 2017 top ten programming languages," Jul 2017, https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages.
[3] S. O'Grady, "The RedMonk programming language rankings: January 2016," Feb 2016, http://redmonk.com/sogrady/2016/02/19/language-rankings-1-16/.
[4] "Twitter's shift from Ruby to Java helps it survive US election," Nov 2012, https://www.infoq.com/news/2012/11/twitter-ruby-to-java.
[5] "Python," http://www.python.org/.

[6] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, "Tracing the meta-level: Pypy's tracing JIT compiler," in *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)*. ACM, 2009.

[7] Google, "Chrome V8," 2018, https://developers.google.com/v8/.

[8] H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Software Practice & Experience*, vol. 18, Sep. 1988.

[9] H. Lieberman and C. Hewitt, "A real-time garbage collector based on the lifetimes of objects," *Communications ACM*, vol. 26, Jun. 1983.

[10] R. R. Fenichel and J. C. Yochelson, "A LISP garbage-collector for virtual-memory computer systems," *Commun. ACM*, vol. 12, Nov. 1969.

[11] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part I," *Commun. ACM*, vol. 3, Apr. 1960.

[12] The PyPy Project, "Garbage collection in PyPy," 2014, https://pypy.readthedocs.io/en/release-2.4.x/garbage_collection.html.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2005.

[14] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2013.

[15] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, Jan 2011.

[16] PyPerformance, "Python performance benchmark suite," 2017, http://pyperformance.readthedocs.io/.

[17] Browserbench, "JetStream 1.1," 2017, http://browserbench.org/JetStream/.

[18] G. Barany, "Python interpreter performance deconstructed," in *Proceedings of the Workshop on Dynamic Languages and Applications (Dyla)*. ACM, 2014.

[19] B. Ilbeyi, C. F. Bolz-Tereick, and C. Batten, "Cross-layer workload characterization of meta-tracing JIT VMs," in *Proceedings of the 2017 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2017.

[20] M. Chandra, N. Krintz, C. Cascaval, D. Edelsohn, P. Nagpurkar, and P. Wu, "Understanding the potential of interpreter-based optimizations for Python," UC Santa Barbara Computer Science, Tech. Rep. 2010-14, August 2010, https://www.cs.ucsb.edu/research/tech-reports/2010-14.

[21] S. Brunthaler, "Speculative staging for interpreter optimization," *CoRR*, vol. abs/1310.2300, 2013.

[22] H. Cao, N. Gu, K. Ren, and Y. Li, "Performance research and optimization on CPython's interpreter," in *Proceedings of the 2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 2015.

[23] C. Kim, S. Kim, H. G. Cho, D. Kim, J. Kim, Y. H. Oh, H. Jang, and J. W. Lee, "Short-circuit dispatch: Accelerating virtual machine interpreters on embedded processors," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. IEEE, 2016.

[24] K. Casey, M. A. Ertl, and D. Gregg, "Optimizing indirect branch prediction accuracy in virtual machine interpreters," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 29, 2007.

[25] M. A. Ertl and D. Gregg, "The structure and performance of efficient interpreters," *Journal of Instruction-Level Parallelism*, vol. 5, 2003.

[26] R. Heynssens, "Performance analysis and benchmarking of Python, a modern scripting language," Master's thesis, Universiteit Gent, Belgium, 2014.

[27] X. Cai, H. P. Langtangen, and H. Moe, "On the performance of the Python programming language for serial and parallel scientific computations," *Scientific Programming*, vol. 13, 2005.

[28] H. P. Langtangen and X. Cai, "On the efficiency of Python for high-performance computing: A case study involving stencil updates for partial differential equations," in *Modeling, Simulation and Optimization of Complex Processes*. Springer, 2008, pp. 337–357.

[29] R. Power and A. Rubinsteyn, "How fast can we make interpreted Python?" *CoRR*, vol. abs/1306.6047, 2013.

[30] D. Tiwari and Y. Solihin, "Architectural characterization and similarity analysis of Sunspider and Google's V8 JavaScript benchmarks," in *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 2012.

[31] G. Southern and J. Renau, "Overhead of deoptimization checks in the V8 JavaScript engine," in *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016.

[32] G. Dot, A. Martinez, and A. Gonzalez, "Analysis and optimization of engines for dynamically typed languages," in *Proccedings of the 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2015.

[33] Y. Zhu, D. Richins, M. Halpern, and V. J. Reddi, "Microarchitectural implications of event-driven server-side web applications," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*. ACM, 2015.

[34] T. Ogasawara, "Workload characterization of server-side javascript," in *Proceedings of the 2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014.

[35] M. Musleh and V. Pai, "Architectural characterization of client-side JavaScript workloads & analysis of software optimizations," Purdue University Department of Electrical and Computer Engineering, Tech. Rep. 467, 2015, https://docs.lib.purdue.edu/ecetr/467/.

[36] A. Srikanth, "Characterization and optimization of JavaScript programs for mobile systems," Ph.D. dissertation, University of Texas at Austin, May 2013.

[37] K. Hazelwood and D. Grove, "Adaptive online context-sensitive inlining," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO)*. IEEE, 2003.

[38] D. P. Grove, "Effective interprocedural optimization of object-oriented languages," Ph.D. dissertation, University of Washington, 1998.

[39] G. Aigner and U. Hölzle, "Eliminating virtual function calls in C++ programs," in *Proceedings of the 10th European Conference on Object-Oriented Programming (ECCOP)*. Springer-Verlag, 1996.

[40] D. F. Bacon, "Fast and effective optimization of statically typed object-oriented languages," Ph.D. dissertation, University of California, Berkeley, 1997.

[41] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. Cohn, "VPC prediction: Reducing the cost of indirect branches via hardware-based dynamic devirtualization," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2007.

[42] M. U. Farooq, L. Chen, and L. Kurian, "Value based BTB indexing for indirect jump prediction," in *Proceedings of the IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2010.

[43] J. A. Joao, O. Mutlu, H. Kim, R. Agarwal, and Y. N. Patt, "Improving the performance of object-oriented languages with dynamic predication of indirect jumps," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2008.

[44] K. Driesen and U. Hölzle, "Multi-stage cascaded prediction," in *European Conference on Parallel Processing*. Springer, 1999.

[45] P.-Y. Chang, E. Hao, and Y. N. Patt, "Target prediction for indirect jumps," in *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 1997.

[46] D. S. McFarlin and C. Zilles, "Bungee jumps: Accelerating indirect branches through HW/SW co-design," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*. ACM, 2015.

[47] V. K. Reddy, R. K. Sawyer, and E. F. Gehringer, "A cache-pinning strategy for improving generational garbage collection," in *International Conference on High-Performance Computing*. Springer, 2006.

[48] P. R. Wilson, M. S. Lam, and T. G. Moher, "Caching considerations for generational garbage collection," in *ACM SIGPLAN Lisp Pointers*, no. 1. ACM, 1992.

[49] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Myths and realities: The performance impact of garbage collection," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance)*. ACM, 2004.