# Just-in-time Parameter Specialization

15-745: Optimizing Compilers

Aram Ebtekar, Alejandro Carbonara

April 30, 2014

## 1 Introduction

Dynamic languages incur the runtime overhead of an interpreter or JIT (just in time) compiler. While this real-time setting allows less room for complex optimization passes than the preprocessed nature of compiled code, it also presents new opportunities for dynamic optimizations that depend on data collected during program execution.

Costa et al (2013)[1] proposed specializing a JavaScript function based on the values that are being passed in, compiling the function with its parameters replaced by constants. This transformation enables additional instances of standard code optimizations such as constant folding and dead-code elimination. However, it's too costly to compile and store a new version of the function each time it's called with new arguments. Hence, they specialize each function only once and, if a later call uses different arguments, the function is compiled in its general form and will never again be specialized.

Despite its overhead next to the simplicity of most JavaScript functions, this approach usually manages to boost performance because, as Costa's data shows, about half of all functions in a typical JavaScript program are called in only one way. Nonetheless, their method wastes numerous opportunities. Consider a frequently called function that gets the same argument list 90% of the time. Costa would permanently discard the specialized function after observing a single instance of changing arguments. A better way to handle this would be to keep the specialized version, as well as a general version to fall back on for the remaining 10%. Or consider the case where only a subset of the arguments ever change, so that a partial specialization would still be very useful. The most frequent argument values may even change over long periods of time. It's costly to generate, maintain, and select among multiple versions of each function, so we must strike some balance.

Note that for very simple or infrequently called functions, the overhead of a more elaboration specialization scheme is sure to exceed its benefits. This project is motivated by functions which are more computationally intensive, yet frequently called with common arguments. This is often the case in more demanding applications such as physics, graphics and video games where, for instance, we may need to render or simulate objects with recurring values for gravity, timestep, or ambient light. In these settings, we'll afford some overhead as the potential benefits are greater. For simpler functions, we may revert to Costa's scheme, or use no specialization at all.

As explained later in this report, we failed to obtain experimental results. Therefore, we note with big apologies that this document will read like a theoretical report with some discussion and speculation of the practical consequences. We present a parameter specialization algorithm, discuss its advantages and pitfalls, and prove a bound on the performance overhead.

# 2 Core Framework

From now on, we use the term **specialization** to mean a compiled and optimized function with none, some or all of its parameters replaced by constant values. When designing a specialization scheme that uses runtime data, we must answer some questions: what data should we collect, and how should we organize it? Which specializations should we generate or delete, and how should we organize those? How do we efficiently select an existing specialization to use for a new function call? For the most part, this report is concerned with time but not memory, so let's ignore the deletion question.

If we allow specializations which fix arbitrary subsets of the parameter list, then exponentially many of them might match a given function call, making it difficult to sort through them. Furthermore, we cannot afford the overhead of scanning through the argument list multiple times during a single function call, to see which of our specializations' parameter lists match it.

For these reasons, we restricted our attention to specializing over suffixes of the parameter list. Why suffixes instead of prefixes? We could have chosen any ordering, but it's common programming practice to place parameters with default values last, so it's more often fruitful to replace the latter parameters by constants. Regardless of the ordering, this restriction ensures there is a unique optimal match among a given set of specializations for any given function call: that which fixes the largest number of parameters, all of which match the new call.

Even so, it's too costly to find the longest match by sequentially scanning over all specializations. Fortunately, suffixes of an argument list can be efficiently stored and

matched in a tree structure which we'll call the **Parameter Specialization Tree (PST)** of a function: let the root (at depth 0) correspond to the trivial specialization, i.e. the function's general form. Edges from the root are labeled by possible values for the final parameter, and lead to nodes corresponding to the specialization of that parameter. In general, edges from depth $d-1$ to depth $d$ are labelled by possible values for the $d$'th-to-last parameter, and lead to nodes corresponding to the specialization in which the last $d$ parameters are set to the values on the node's path from the root. The PST begins as a lone root and grows as new function calls are made, accomodating the paths corresponding to each argument list. During each call to an $n$-ary function, we traverse the $n+1$ nodes corresponding to its argument list, and each node counts the number of times it was visited.

While this PST structure still presents considerably higher overhead than Costa's method, it is fairly reasonable, at least for the more expensive functions: a traversal demands only about as many operations as scanning through the argument list. Although each node should be thought of as corresponding to a particular specialization, only a small fraction of these specializations will ever actually be compiled; the remaining nodes will instead carry a null pointer. Choosing which nodes to compile is an interesting problem: a variety of statistical criteria are compatible with this framework, but finding one that's likely to yield a performance advantage is a challenge.

Given a set of compiled nodes, selecting the one to execute for a particular function call is easy: always use the deepest non-null pointer found during the call's PST traversal. It points to the matching specialization with the greatest number of parameter-by-constant replacements, making it the most optimized.

# 3 Statistical Method

Our milestone report presented a very simple statistical criterion: we would compile a node when its visit count exceeded that of its most visited child by at least a specified constant, say 10. However, we soon found flaws with this criterion, a major one being that it tends to compile redundantly. For instance, suppose each of the calls $foo(1,4)$, $foo(2,4)$ and $foo(3,5)$ is made 12 times, totalling 36 function calls. Then of course, we compile all three full specializations. However, we would also compile the redundant partial specializations $foo(\_,4)$ and $foo(\_,\_)$ because $24 > \max\{12,12\}+10$ and $36 > \max\{24,12\}+10$, respectively.

Therefore, we will propose a new criterion with better theoretical properties. Before doing so, let's briefly remark that no matter what criterion is employed, we should take care always to have compiled a common ancestor of the function calls

to date (where the node corresponding to each call is the deepest in its traversal path, the same as its full specialization). Otherwise, there may be no valid node to execute. The simplest way to achieve this is to compile the root upon the first call to the function. A greedier strategy is to always ensure the **least** common ancestor is compiled; this method provides deeper specializations, but may compile up to $n$ additional nodes for an $n$-ary function in the worst case. As a middle ground, we may apply a close analogue of Costa's method: begin by compiling the full specialization corresponding to the first call, and then, the first time a different argument list is seen, compile the unspecialized root.

In order to explain the statistical criterion by which the remaining node compilations are made, define non-negative weights for each parameter of each function in the program. These weights should represent the approximate benefit of replacing that parameter by a constant value for a single execution of the specialized function. It can be estimated by a static analysis of the function code, or the weights might simply all be set to 1.

For a given PST containing some compiled specializations, define its utility with respect to a function call to be the total weight of all parameters that would be replaced by constants in the specialization it selects. Let the PST's **total utility** be the sum of its utilities with respect to every function call in its history. For a candidate specialization which isn't yet compiled, let its **marginal utility** be the amount by which the PST's total utility would increase if the compilation were performed. The marginal utility can be computed as the product of two quantities: (1) the sum of weights of the parameters which this specialization replaces by constants, but none of its compiled ancestors in the PST do; and (2) the number of recorded calls which visited this specialization's node without visiting any of its compiled descendants. During a function call, if any of the nodes in its path have a marginal utility exceeding some specified threshold, we compile the deepest of those.

Pseudocode is listed in Algorithms 1 and 2. node.count is the number of traversals that visited a particular node, and node.descCount is the number of traversals that visited at least one of its compiled proper descendants.

**Algorithm 1** InitPST()

```
root ← new Node(parent=NULL)
root.spec ← compileParameterSpecialization(root)
```

**Algorithm 2** CallUpdateAndSelect(args)

```
Node cur ← root
Node opt ← root
int sumWeights ← 0
for all arg : args in reverse order do
  NodeRef next ← root.child[arg]
  if next = NULL then
    next ← new Node(parent=cur)
  end if
  cur ← next
  cur.count ← cur.count + 1
  sumWeights ← sumWeights + weight(arg)
  marginalUtility ← (cur.count - cur.descCount) * sumWeights
  if cur.spec ≠ NULL then
    opt ← cur
    sumWeights ← 0
  else if marginalUtility ≥ COMPILE_THRESHOLD then
    opt ← cur
  end if
end for
int ΔCount ← 1
if opt.spec = NULL then
  opt.spec = compileParameterSpecialization(opt)
  ΔCount ← opt.count - opt.descCount
end if
cur ← opt
repeat
  cur ← cur.parent
  cur.descCount ← cur.descCount + ΔCount
until cur.spec ≠ NULL
return opt.spec;
```

# 4 Analysis

What can we say about the algorithm's performance? We already discussed that the tree traversal constitutes a significant enough overhead that the simplest functions should disregard the PST method altogether. However, for those which do use the PST, the compilation of multiple parameter specializations may be even more costly. Ideally, we'd like to prove that the cost of our PS compilations is always less than the performance benefit they provide. However, such a guarantee is too tall an order because parameter specialization is by its nature speculative: a single execution of the specialized code can't be expected to cover for its own compilation cost, so we can only hope to cover it by later executions which we cannot predict. Accepting this, it's encouraging that we can guarantee the next best thing: the compiled specializations are worthwhile in the context of the call history; in other words, provided the parameter weights were accurate, had all of our current parameter specializations been available from the start, they would have yielded a net grain!

**Theorem 1.** *Algorithms 1 and 2 compile at most* $1 + \frac{totalUtility}{COMPILE\_THRESHOLD}$ *parameter specializations. Therefore, if the cost of compilation is COMPILE_COST, a non-negative counterfactual net gain (relative to compiling only the root) is assured by setting COMPILE_THRESHOLD > COMPILE_COST.*

**Proof:** We proceed inductively. Initially, we have zero utility and just the trivial root specialization, so the relation holds. Notice that while the set of specializations remains static, the total utility may increase but never decrease. Thus, the only case left to consider is when a new specialization is added. In this case, the total utility increases by the new specialization's marginal utility, which is at least COMPILE_THRESHOLD. Hence, by the induction hypothesis, we get at most
$1 + (1 + \frac{totalUtility\_old}{COMPILE\_THRESHOLD})$
$\leq 2 + \frac{totalUtility - COMPILE\_THRESHOLD}{COMPILE\_THRESHOLD}$
$= 1 + \frac{totalUtility}{COMPILE\_THRESHOLD}$ parameter specializations. □

**Remarks:** For conciseness, the above algorithm and proof never consider deleting a specialization. Since the introduction of certain specializations decreases the marginal utility of others, we may want to free memory by removing specializations which have become redundant. We can extend the algorithm to delete any specializations it encounters with marginalUtility ≤ DELETE_THRESHOLD. With this modification, we can still bound the **total** number of compilations, both stored and deleted, by $1 + \frac{totalUtility}{COMPILE\_THRESHOLD - DELETE\_THRESHOLD}$.

So what does it all mean? What seems like a nice result must be interpreted with care in the context of typical computer programs. Once again, it should be

emphasized that the PST is of no use if the overhead of tree traversal dominates the potential benefits of parameter specialization. In most simple cases, Costa's method suffices. Now, if the cost of tree traversal is ignored, then we've seen that by appropriately tuning the COMPILE_THRESHOLD parameter, we can avoid further losses. In the extreme case, setting it to a very large value prevents any nontrivial specialization from taking place at all! At more reasonable levels, the counterfactual benefit will not translate to real gains if the function is called in an adversarial manner (e.g. never calling with arguments for which we've already specialized). However, if the call pattern is reasonably consistent, then it seems we might do well by setting the threshold to some multiple of COMPILE_COST. Ultimately, the only way to find out is by running high-quality experiments, something which we sadly don't have.

# 5 Experimental Setup

We worked with the codebase from Costa's just-in-time value specialization paper. They build their code directly off of ionmonkey, the JavaScript compiler used by Firefox. They provided a number of test suites and scripts in order to automatically test the speedups for various optimizations they performed on the functions. As our algorithm is an extension of theirs, we figured our algorithm would naturally implement on top of theirs.

Unfortunately, we ran into major troubles in trying to understand ionmonkey, their changes on top of it, and any attempts to implement meaningful changes of our own, however small. Just-in-time compilation has a number of peculiarities that makes it hard to work with. The code is not analyzed in passes. The code is broken up into a flow graph, and each block is ran as it is being compiled.

In the base compiler code, replacement is done in two different places. The compiler first does certain compilations when a function is first run. If the piece of code is seen again, they substitute the compiled code directly in. Costa's algorithm can handle this in a simple way. At most they need to compile a function twice: once with their optimizations and once without. Our algorithm would require us to be able to do the parameter checks, replacements, and decisions to compile in both sections of code. We were not able to figure out how to implement our algorithm in these sites, not to mention how to construct our data structure so that it would be properly handled by the garbage collector. Even the smallest changes we made would make the compiler run unpredictably.

In fact, even Costa's theoretically simple method seemed to require a lot of changes to the code occurring in a lot of different places. ionmonkey has very high coupling, making it impossible to focus on just the relevant features as a blackbox.

For a long time we thought we might have just overlooked something simple, as we're not particularly confident of our industrial programming abilities; even now we can't be totally sure, but after spending far too many frustrating hours trying we think this might just be too difficult an engineering task for us to complete in any reasonable time.

# 6 Experimental Evaluation

Costa's test suite includes a large set of realistic JavaScript on which we could compare our performance with their code as well as the baseline, had our implementation succeeded. It would also have been helpful to look at the properties of functions on which we perform well or poorly, to see if it may be possible to detect these cases to automatically decide when to apply our method.

Despite being unable to run our tests, our examination suggests that our method would not attain notable speedups in most cases, and would cause slowdowns for a large portion. From our tests on the provided JIT code, Costas algorithm is very judicious in deciding whether a function would be optimized, and his changes still resulted in a noticeable slowdown in many test cases. In the cases where optimization did happen, they worked well. Not many of the tests involved much of the interesting sort of call patterns that we try to exploit. Nonetheless, it bears mentioning that our method might be better suited to different non-web applications, in JITs for other languages such as Java and for intensive uses such as real-time simulation and rendering.

# 7 Surprises and Lessons Learned

Working with JIT compilation was a lot harder than expected. Implementing any sort of optimization required strong knowledge of how the system worked. On the plus side, we learned from analyzing Costa's paper and going through a careful thought experiment based on the problem they tackled and the apparent opportunities for improvement which their method ignored. We manage to exploit more of those opportunities, but it's unclear whether it pays off. Although we tried to make our data structures highly efficient, what we try to do necessarily requires substantially more computational resources than the basic alternatives.

# 8 Conclusions and Future Work

We proposed a novel framework, the Parameter Specialization Tree, for storing and selecting parameter specializations to execute at runtime. Furthermore, we present statistical criteria for the creation and deletion of specializations, and discuss some of their properties. However, without proper testing, it's hard to say anything conclusive.

There are many possible directions in which to further develop the method, though these should only be attempted after learning more about its practical performance via experiments. We can be more careful in choosing the weights and orderings of parameters, perhaps by incorporating a just-in-time static analysis phase, or with real-time data. If certain parameters don't seem important enough for optimization purposes, we can ignore them for the purposes of the PST, thus reducing the overhead substantially. We might also consider decaying the node visit counts exponentially over time, so as to limit memory use in cases where the call distribution of functions changes gradually over long periods of time.

Finally, it might be exciting to see whether some elements of our techniques can apply to static languages. A mixture of static analysis and sample runs of a program can feed data into a compiler, which might then it to generate multiple specialized versions of certain functions. While the data may be less dynamic in this case, we benefit from a relative lack of performance overhead since the data structure is only needed during offline compilation.

# Split

Whatever makes us pass, pretty please? We worked a lot on this and, though we have little to show for it, most of that time was spent on our attempts at understanding ionmonkey in order to construct the experiments.

# References

[1] Igor Costa, Pericles Alves, Henrique Nazare Santos, Fernando Magno Quintao Pereira. "Just-in-Time Value Specialization" in Proceedings of the 2013 International Symposium on Code Generation and Optimization (CGO), February 2013.