Optimization of List-Based Functional Pipelines

by

Timothy Soehnlin

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 6, 2017

Keywords: optimization, javascript, functional programming

Approved by

Jeffrey Overbey, Chair, Assistant Professor of Computer Science
Kai Chang, Professor of Computer Science
Levent Yilmaz, Professor of Computer Science

Abstract

There has always been a disconnect between how humans organize computer code, and how computers execute it. Functional programming is a paradigm that increases code maintainability and testability, but generally results in poorer resource utilization by the computer. The increased resource cost versus code quality is usually an acceptable cost. However, code that is executed many times over, would benefit from optimization as the cost of the introduced abstractions are amplified. Specifically list-based functional pipelines (map, reduce, filter, etc.) are extremely sensitive to this overhead as the functional overhead is paid for each element in the list. In this paper we provide a JavaScript optimization algorithm that transforms list-based functional pipelines (e.g. `arr.map(x => x * 2).filter(x => x > 2)`) into `for` loops by combining source-level analysis and runtime evaluation to construct and compile optimized code at runtime. This optimization algorithm allows us to dynamically translate common list-based functional pipelines into a form that is optimized for computer execution, without modifying the original source code. This hybrid ultimately provides the best of both worlds, allowing the human to write and manage code in a manner that is familiar and more understandable, and for the final output to be in a form that executes with higher efficiency.

Acknowledgments

For my God, and my family.

Thank you Jenn for supporting me on this journey.

Thank you V for having my back.

Table of Contents

List of Figures

Chapter 1

Introduction

This chapter will introduce the main functional programming concepts of this thesis, discuss the optimization algorithm for converting the functional concepts into procedural code, and then look at the results from the entire process.

1.1 Functional Programming and Pipelines

Functional programming, and by implication functional isolation and purity lend themselves to code that is of higher quality [24], and by implication easier to maintain, and verify. Unfortunately the functional isolation generally results in poorer resource utilization by the computer. The increased resource cost versus code quality is usually an acceptable cost.

1.1.1 Functional Pipelines

It is a common practice, in functional programming, to handle processing of (looping) sequences by function invocation/recursion [15]. Composing processing operators produces a list-based functional pipeline in which a list is transformed. This abstraction comes with the cost of invoking a function per each element of the list and that is multiplied by number of operators in the pipeline. In addition to the cost of function invocation, there is also significant memory usage for any operator that returns a lists that are not the final values. The intermediate lists are temporary and are used for passing inputs between operators (e.g. `map(filter(x, ...)...)`).

```
function findCommonWords(text, threshold) {
  // Split text by non alphanumeric characters
  // For word in words
  //    Skip if the word length is less than three (ignore small words)
  //    Convert the word to lower case
  //    Increment counter for word
  //    If counter passes threshold
  //      Mark word as valid
  // Return all valid words
}
```

Listing 1.1: Pseudo-code for algorithm

Given the algorithm defined in listing 1.1, there are a set of filters, transformations, aggregations that need to be applied in order to produce the desired output. Each functional pipeline operator (`map` , `reduce` , `filter` , etc.) is comprised by two components.

1. The operator's agreed upon meaning.

2. The operator's use of its inputs.

For example, `map` defines an operator that requires its inputs are a list, and a function that can transform every element of the list. Once `map` is invoked with its inputs, it produces a new list with every element replaced with the transformation applied. All the operators have clear inputs and outputs per the JavaScript specification.

1.1.2 Functional Pipeline Operators: JavaScript

An implementation of this concept of list-based functional pipelines can be found in how JavaScript models arrays. The array class implements a super-set of the following functionality:

1. `forEach(operation)` . `operation` represents a generic function whose return value is ignored. This method will invoke `operation` on every element of the array [2].

2. `map(transform)`. `transform` represents a mapping of the array contents. This method will create a new array of identical size to the input, but with every element mapped through the `transform` function [3].

3. `filter(predicate)`. `predicate` represents a mapping of the array elements to a boolean value. This method will return a new array in which every element that has a `predicate` invocation that returns `true` will be in the output [1].

4. `reduce(accumulate, initial)`. `accumulate` represents a function that receives both `accumulator` and an array element. Every time `accumulate` is invoked with `accumulator` and an array element, the output of the function is stored as a new value for `accumulate`. This new value will be used on the next iteration or will be returned if it is the final iteration [4].

Since `map` and `filter` both return arrays, this gives way to method chaining. Method chaining is a design pattern in which an operation on an object returns itself or an instance of the same type as the object. This is the clearest way to create a list-based functional pipeline in JavaScript. An example of this form can be found in listing 1.2.

```
function findCommonWords(text, limit) {
  return text
    .split(/[^A-Za-z]*/)
    .filter(word => word.length >= 4)
    .map(word => word.toLowerCase())
    .reduce((check,  word) => {
      let count = check.all[word] = (check.all[word] || 0)+1;
      if (count > limit) {
        check.common[word] = count;
      }
      return check;
    }, {all:{}, common:{}}).common;
}
```

Listing 1.2: Functional implementation of findCommonWords

1.1.3 Performance Compensation

A common optimization that is encouraged, throughout many programming languages, is to manually convert the functional traversal into a more traditional `for` loop [17] [22] [12]. This involves manually projecting the functional pipeline operators (and the companion functions) into standard procedural code. While increasing performance and decreasing memory usage, the overall cost is tied to code maintainability and quality [24].

The main trade-off of manual operation is pitting the programmer's desire for clean, testable code against the processor's need for tight loops and having as much of the code to be executed in a single function.

1.2 Optimization Algorithm

The JavaScript optimization algorithm works in the following phases.

1. **Source Code Analysis** - The first phase identifies and transforms potential optimization sites at a source code level. This process happens by parsing the JavaScript and

analyzing the resultant abstract syntax tree. We look for patterns that identify potential list-based functional pipelines. Since JavaScript is dynamic we cannot guarantee that the functional pipeline is actually running on a list. Once we identify a potential list-based functional pipeline, we rewrite the potential list-based functional pipeline into a form that allows for runtime analysis. This gives us the ability to efficiently interrogate the data at runtime, to verify that we do have an actual list-based functional pipeline. We also analyze the functional pipeline operator inputs to determine if they are defined within the file, if they are passed in as parameters or if they imported as this distinction has an impact on how the operator is transformed at runtime. We also identify which inline functions read/write closed variables (as these are the only closures we can support since we have access to the enclosing scope).

2. **Runtime Optimization** - The second phase occurs at at runtime, we will verify that the input is indeed a list, and also verify that the set of functional pipeline operators are valid for conversion to a `for` loop. Once we have a valid list-based functional pipeline, we will then generate the resultant `for` loop. We take into consideration closed variables to handle re-assigning modified variables on return. After compilation, we execute the newly generated code

1.3 Results

To validate the findings, we've established four scenarios to test the performance impact of the implementation of our optimization algorithm. The scenarios we test are:

1. **MD5** - An implementation of the MD5 [26] algorithm in JavaScript.

2. **Sort Score Sum** - A simple algorithm that is an implementation of a Project Euler question [13].

3. **Std Dev** - An implementation of the standard deviation calculation.

4. **Text Analysis** - An algorithm that performs word frequency on a large body of text

We test each scenario by running through three different versions of each scenario: functional form, manual form, and optimized form. The functional form and optimized form have identical source code, except for a flag in the optimized form to enable the optimization code. The manual form is a hand coded transformation of the algorithm to provide a theoretical upper bound for performance.

In general the results are extremely promising (figures 3.6, 3.11, 3.18, 3.24). The pattern we seen is that the optimized form actually out performs the manual form when the number of iterations is higher, and that is consistently out performs the functional form in nearly all cases.

This gives way to providing a simple optimization that can be enable for free and provides a fairly dramatic performance improvement without losing any of the benefits of the clarity of the functional form.

1.4 Contributions

This thesis makes the following contributions

1. **Optimization Algorithm** - An algorithm for analyzing JavaScript to transform list-based functional pipelines into standard `for` loops.

2. **Optimization Algorithm Implementation and Framework** - An implementation[30] of the optimization algorithm has been produced in TypeScript and is available for download and usage. It also has a framework for creating new implementations of the algorithm for functional pipeline operators other than JavaScript arrays.

3. **Performance Analysis Framework** - The optimization algorithm implementation[30] also provides a performance optimization algorithm that was used to produce the data used in this paper. In addition to data generation, it also provides functional to generate visual output via gnuplot.

Chapter 2
Optimization Algorithm

In this chapter we provide a JavaScript optimization algorithm that transforms list-based functional pipelines (e.g. `arr.map(x => x * 2).filter(x => x > 2)`) into `for` loops by combining source-level analysis and runtime evaluation to construct and compile optimized code at runtime.

2.1 High Level

The optimization algorithm attempts to mimic the assumptions and behavior of programmers when converting functional pipeline operators (`map`, `reduce`, `filter`, etc.) into `for` loop semantics.

When we deal with the common cases and patterns, the algorithm is simple. The true complexity arises in ensuring all places where this could be used do not violate correctness. There are many cases that are accounted for, but write-dependence is the only major caveat as discussed later.

The algorithm will run in two phases:

1. Source code analysis §2.4
2. Runtime optimization §2.5

The first phase (as discussed in §2.4) will identify potential sites for optimization and will modify the candidate location accordingly. The source code analysis only works on a single file at a time, to minimize the general complexity of the algorithm. In addition the single file perspective helps to keep the algorithm's assumptions as simple as possible, and defer decisions to runtime as often as possible.

The source code analysis will generate a new output file that will then trigger the runtime optimization. At runtime the code will interrogate the candidate optimization, and if everything is in order, will execute a runtime optimization of the code. There are certain scenarios that disqualify optimization but these will not be known until runtime. The end result involves compiling an alternate code path at runtime, and then executing the newly created function.

2.2 JavaScript

JavaScript (officially ECMAScript) is a high-level, dynamic, untyped, programming language. Although there are strong outward similarities between JavaScript and Java, including language name, syntax, and respective standard libraries, the two are distinct languages and differ greatly in their design. JavaScript was influenced by programming languages such as Self and Scheme. JavaScript has been traditionally implemented as an interpreted language, but more recent implementations perform just-in-time compilation. [16]

JavaScript is a very powerful language, supporting functional programming, as well as procedural and object oriented paradigms. Additionally, the ability to compile code at runtime makes it powerful (and dangerous) for expressing complex operations or computations. [9]

2.2.1 ES2015

ES2015 is the 2015 revision of the JavaScript language. This revision introduces new grammar constructs, which in turn have increased the implementation complexity [5] [6] [8]. Here are some of the features introduced in ES2015:

1. **Let, Const** These concepts allow for local variable declaration and local constant declaration. The main change with this feature is that JavaScript now supports block level scoping as opposed to the previous state of only scoping at the functional or global

level [6] [7]. This allows for clearer code (and fewer bugs) but requires a different algorithm for calculating variable scopes. By contrast `var` scopes at the function level, regardless of location within the function (or nested blocks). This is known as variable-hoisting as all variables are assumed to be available at the first point of the function.

2. **Fat Arrow Functions** This is generally a short-hand for anonymous functions with one major caveat. The fat arrow function (`x => x * this.y`) treats the `this` variable differently [5]. Generally most functions have their own context, referenced by a variable `this`. Since functions and classes are nearly identical in JavaScript, `this` is a reference to the instance. Fat arrow functions refuse to create a `this` for reference, and instead defer to the enclosing function scope for the reference of `this`. Again, variable declaration and tracing are more complicated due to the additional scoping methodology.

3. **Object/Array Destructuring** This concept allows a programmer to unpack the contents of an array or an object literal into its constituent pieces, at the time of assignment [8]. This includes variable declarations as well as function parameters. While this is enhances expressive the power of the language, it also increases the complexity of variable tracking.

The reason these features are highlighted is that they all play a role in the final solution. This is also to highlight that as the language grows and changes, the optimization algorithm may need to be modified to stay consistent.

2.3 Assumptions

This dynamic nature of JavaScript lends itself to powerful flexibility, but also makes static analysis difficult [19]. In the case of this optimization, the goal is to mimic a programmer's behavior. If the generated code contains the same errors a programmer would have made in manual conversion, we assume the process to be correct. The code may be broken, but it's broken in a way that is consistent with the programmer's mental model. The goal is not to achieve perfection, but to automate the currently manual process.

2.3.1 Scope and Global

Closed variables play a unique role in JavaScript as the scoping rules are very flexible. Global variables, for instance, are essentially closed variables, originating at the highest scope. Additionally any scope between the current operation and the global scope, can shadow a variable declaration. This means that any static analysis may prove to be incorrect at runtime.

In JavaScript it is a common pattern of adding supplemental functionality to core types by appending to their class structure or even replacing it wholesale. Given that this can easily introduce unforseen bugs, the ECMAScript committee strongly recommends against this. [11]

In addition to shadowing, as with many other languages, property accesses `a.b` may actually be backed by a function invocation [10]. When retrieving `b` from the object `a` a function may be invoked that could modify closed variables or alter global state.

2.3.2 Parameter Types

When dealing with any functional pipeline operator, all the inputs will generally fall into one of three main categories:

1. **Literals** This can be numeric, string, function literals, but the important fact is that they are defined in the scope of the functional pipeline. Additionally, function literals have the added benefit of being able to read/write closed variables.

2. **Static reference** This will generally be constants, static methods, etc that are defined outside of your source file (or specifically the scope of the candidate optimization).

These inputs will be assumed to be static with reality that this assumption maybe wrong. The risk of assuming incorrectly is real, but this is the same risk a programmer would face when manually converting the functional form to the procedural form.

At runtime, function references will be interrogated to see if they are able to be inlined or if they fall into the simplistic case. Any function reference that has closed variables cannot be inlined, as we do not have access to the closure to allow reading and writing. This can only be tested at runtime, and so this is by far the most complex parameter type.

Note: If we were developing the implementation inside of the JavaScript engine itself, it may be possible to optimize static references as well, since the closures may be known at that point.

3. **Dynamic reference** This will be function parameters in scope of the candidate optimization. These are assumed to be dynamic by design and will not be trusted with any level of certainty. This does not disqualify a candidate optimization, but it does mean that we will not be able to inline the code without the potential cost of recompiling the code on every invocation.

Given the behavior of programmers to create utilities classes for common, small operations, the static reference case is of critical import with respect to optimization. It would be very easy to treat it as if it were a dynamic reference, but we would lose a fair amount of our performance gains by incurring unnecessary function invocations. Again, this will follow the paradigm a programmer would take if trying to hand optimize any code. If we were using a predicate by the name of `isEven`, we would be inclined to replace that with a check of `x % 2 = 0`.

2.3.3 Write Dependences

The optimization's entire structure is centered around altering the order of execution, specifically to avoid any intermediate storage of results. The issue of write dependence arises from the change in order of operation in visiting each element of the sequence. In the functional pipeline form, each pipeline operator must finish completely before the next operator can start. In the procedural form, each pipeline operator is applied in succession on each sequence element. Again, this is an identical problem a programmer would face if they were to manually convert the candidate optimization into a procedural form by hand. Additionally, since property accesses can actually result in writing values, and methods can be rewritten at runtime, determining when a write actually occurs is fairly complicated.

Ultimately this leads us to the point that we will ignore the concept of write dependences as they cannot be proven except at runtime (if even then). We will behave in a manner that is consistent with a hand generated optimization. If we were to allow optimizations only on literal function parameters (numbers, strings, boolean values) with the rule that no properties can be accessed, we would have a higher degree of certainty that we will not have any write dependences. Ultimately this would make the optimization far less useful, as it would apply in only the simplest cases.

2.4 Source Code Analysis

2.4.1 Identifying Functional Pipelines

As noted before, method chaining is a common paradigm in JavaScript for creating functional pipelines over a list of elements. The final composed entity is identified as a functional pipeline, and that is what we are looking for when optimizing. Essentially, we are looking for a specific call expression, that has an array as the call target.

```
 1 function simple(specialFilter) {
 2   this.limit = 4;
 3   let count = 0;
 4   [1, 2, 3] //pipeline target
 5     .map(x => x * 2)  //pipeline start
 6     .filter(x => x > this.limit)
 7     .map((x, i) => {
 8       count++;
 9       return Math.pow(x, i + 1)
10     })
11     .filter(specialFilter) //pipeline end
12 }
```

Listing 2.1: Simple Functional Example

In listing 2.4, the `map` call expression on line 4 is the the beginning of our pipeline, and the pipeline terminates on line 11 with a `map` call expression. It is easy to recognize this visually, but the source code analysis can only be so certain. In general, it finds a potential call expression, which is defined as a call expression in which the chained operator is a well known identifier (e.g. `map`, `filter`, `reduce`, etc.). Once the base of the functional pipeline is determined, the code will follow the chain until there are no further call expressions or an incompatible call expression is found (e.g. `sort`, `reverse`, etc.)

At this point the candidate optimization is identified. It is still not know if the source of the functional pipeline is truly an array or if it can alternate types at runtime, and this is why the functional pipeline is identified as a candidate for optimization.

2.4.2 Invocation Form

The next step is to process the candidate optimization into the final invocation form. This massages the functional pipeline into a final form that allows for maximal performance at runtime by minimizing any runtime overhead. The overhead for execution of the optimized form, assuming functional pipeline target is an array, will have significant performance impact smaller arrays, and needs to be mitigated as much as possible. For the transformation to be generally useful it needs to perform well on arrays of size 0 as well as arrays of size 100000.

```
function simple(specialFilter) {
  this.limit = 4;
  let count = 0;

  var __thisId = this;

  EXEC(
    [1, 2, 3]/* target*/,
    '__key_RANDOM_UUID', /* callsite id */,
    __operators__UUID,
    [ /* Pipeline inputs */
      [function __uuid1(x) { return x * 2 }],
      [function __uuid2(x) { return x > __thisId.limit }]
      [function __uuid3(x, i) {
        count += 1
        return Math.pow(x, i + 1)
      }],
      [specialFilter]
    ],
    [__thisId, count], /* List of all closed variables */
    (__randomId) => { count = __randomId } /* Closure reassignment*/
  )
}


//Operators and flag to indicate inline vs static vs dynamic
var __operators__UUID = [
  ['map', 0],
  ['filter', 0],
  ['map', 0],
  ['filter', 1]
]
```

Listing 2.2: Simple Functional, Final Form

Given the goal of minimizing the final form's computation, the functional pipeline is restructured into predetermined arrays that can be handed piecemeal to the runtime optimizer without the need for any memory allocations, string concatenations or function invocations.

When invoking an optimized code path, the only data that is needed is the `target`, `context`, `closed` and `closure assignment`. These variables are utilized by the optimized code path in the expressed form.

The final form also needs to accommodate when the `target` is not an array, and needs to be invoked in a manner equivalent to the original form.

```
for (let i = 0; i < chainOperations.length; i++) {
  chainTarget = (chainTarget[chainOperations[i]] as any)(...chainInput[i])
      ;
}
return chainTarget;
```

Listing 2.3: Manual invocation

In the code listing 2.3, each pipeline operator is invoked as a member of the pipeline target (with the appropriate input), and that is stored as the new target for the next pipeline operator. This is functionally equivalent, and should perform comparably to the original form.

2.4.2.1 Closure Analysis

Within the final form (listing 2.2), we analyze closed variables to be able to handle reading and writing of these variables within the optimized code path. Because the optimized code path is not in the same scope as the candidate optimization, we lose access to the closed variables. We need to calculate all the closed variables that we can, and handle them accordingly. Due to the fact that we are only looking at one file at a time, we are only able to analyze closed variables for inline functions, and everything else can only be handled at runtime.

2.4.2.2 Parameters

Another piece of the invocation form is knowledge of which parameters are passed in as function parameters versus which are inline or are globally defined.

The issue here is that we should assume that any parameter passed in will not be static and thus we have to treat as hostile as possible. Given that even analysis of the function itself may take longer than running the code in the original form, we assume the worst case scenario for these functions and limit the optimization to merely invoking the parameter, as opposed to attempting to inline the function at runtime.

The intricacy of the interplay between source code analysis and the runtime optimization is non-trivial and great care is needed to balance the needs of the various phases with overall performance.

2.5 Runtime Optimization

The runtime optimization focuses on compiling the common paradigm of the functional pipeline operators, and how the relationship between the operator and its inputs.

At runtime the code will pass through two separate phases:

1. Transformation and Validation (§2.5.1)

2. Compilation (§2.5.2)

Transformation and validation happens on a per functional pipeline operator basis. This allows for simple transformations as well as a short circuit on any invalid functional pipeline operator.

Once the entire functional pipeline has been transformed, the results are merged together into a final compilation process that results in the optimized code as a new function.

```javascript
function simple(specialFilter) {
  this.limit = 4;
  let count = 0;
  [1, 2, 3] //pipeline target
    .map(x => x * 2)  //pipeline start
    .filter(x => x > this.limit)
    .map((x, i) => {
      count++;
      return Math.pow(x, i + 1)
    })
    .filter(specialFilter) //pipeline end
}
```

Listing 2.4: Simple Functional Implementation

2.5.1 Transformation

The transformation process only deals with a single pipeline operator at a time. Given something as simple as `filter(x => x > 2)`, this could be translated into `if (x <= 2) continue` assuming the code is running a `for` loop.

The goal of the transformation process is do five things:

1. **Verify reference (non-literal) functions are valid** The main verification is determining whether or not the function is not expecting access to the intermediate arrays that we are optimizing away. `map`, `reduce`, `filter`, `forEach`, `some`, `find`, etc. all have an optional last argument that provides read only access to the intermediate array. Since we are skipping creating the intermediate array intentionally, any use of this special form requires that we invalidate the optimization of the entire functional pipeline. Theoretically we could generate sub functional pipelines splitting at the offending operator, but that would require moving even more of the source code analysis

process into the compilation process. Building an intermediate array is not acceptable as we are modifying the order of operations. The intermediate array is meant to represent the entire array at the point in time before the current pipeline operator.

This points to a larger issue, in which any reference is made to the intermediate array (or it's size), generally disqualifies a functional pipeline from optimization.

2. **Determine, for static references, if a function can be inlined** For dynamic references (generally function parameters), we already know these cannot be inlined. For static references, we need to analyze the code determine it's purity. A pure function (without closed variables) can be inlined. If closed variables exist, the static reference must be treated as a dynamic reference.

3. **Determine if a function has need of a positional counter** From this point, we inspect the function parameters and look to see if it has a parameter for the index. If it does not, we skip initializing and incrementing the position for the specific transformation. If we cannot read the function (e.g. it is a native function) or if it does reference the position parameter, then we will initialize and increment counter.

4. **Generate the partial AST** Once all the above is done we generate one of two ASTs. Given a filter predicate of `isEven` we can generate an inline call `if (!(x % 2 == 0)) continue` or we will generate an invocation call `if (!isEven.call(context, x)) continue`. Both are superior to creating and freeing intermediate arrays, but the inline call still offers maximal performance.

5. **Rewrite Local Variables and Parameters** Once the AST is provided, we rewrite all function parameters and references to `this` to allow for multiple operations to reside within the same `for` loop. This means that any variables of the same name, from separate pipeline operators, would collide and produce incorrect code.

2.5.2 Compilation

When constructing the resultant optimized code, we need to initialize the environment the code will be running in. This involves aliasing in closed variables, exposing all the input parameters for the individual transformations (e.g. this context, or the initial value for an accumulator, etc.). The goal here is to provide all the relevant structure needed for runtime.

In addition to the initialization, the compilation process also provides the `for` loop to iterate through all the data elements as well as mapping the closed variables into the final output, for reassignment once the function finishes. A simple example of a transformation is found in listing 2.5.

```javascript
function __compiled(data, context, closed) {
  let __thisId = closed[0], count = closed[1];
  let __pos_1 = 0;
  let __pos_2 = 0;
  let out = [];
  let specialFilter = context[3][0];
  let specialFilterObject = context[3][1];

  fullLoop:
  for (let i = 0; i < data.length; i++) {
    let el = data[i];
    el = el * 2;
    if (!(el > __thisId.limit)) continue fullLoop;
    count += 1;
    el = Math.pow(el, __pos_1);
    __pos_1 ++;
    if (!(specialFilter.call(specialFilterObject, el, __pos_2))) continue
       fullLoop;
    __pos_2++;
    out.push(el);
  }
  return {
    value : out,
    assigned : [count]
  }
}
```

Listing 2.5: Simple Functional Compiled

2.6 Opt-in Optimizations

All of the optimizations that occur are opt-in only. This is managed by the use of a JavaScript pragma directive, e.g.: “use optimize” or “disable optimize”. This allows for optimization to be enabled or disabled at any scope (module, function, etc) and to subsequently disable optimizations in nested scope. The idea being here, that an programmer could add “use optimize” at the beginning of an area of code that could benefit from performance gains. This allows for fine grained control of the usage, and to opt-in only by request.

Chapter 3
Results

In this chapter we describe the test framework, the evaluation criteria, and application of these against four different scenarios.

3.1 Setup

For the majority of our testing, we used NodeJS, and specifically V8 (5.3). Different engines will invariably produce different results on the scenarios we are testing [23]. The goal of the optimization is still to perform what a programmer would do by hand, and so in that vein, the underlying engine is usually unimportant. There may be some additional performance from the runtime optimization process, because we are generating that at runtime, and it is possible to tailor the output to the specific environment the code is being run in.

Additionally, ChromeOS, NodeJs, and Android provide a sufficiently large ecosystem running V8 that any findings are significant regardless of other JavaScript environments [18].

3.2 Criteria

With our testing, there are three main variables that will be analyzed.

1. **Scenario** - These are the different algorithms used to test the optimizations performance. They range from I/O heavy, computationally heavy, and a mix of both. Each scenario must provide three forms to be tested:

    (a) **Functional** - This is the ideal form for writing and testing the function, but is generally not as efficient as manually converting it to a for loop.

   (b) **Manual** - This is the manual transformation to a `for` loop, should be the theoretical upper bound of efficiency, but may not always hold true given the nature of JavaScript

   (c) **Optimized** - This is the functional form transformed via the optimization process.

   The goal here is to provide three views of each scenario, to see what an average user can expect when using this optimization.

2. **Input Size** - This is the input size the final compiled function will process. This represents whether or not we will be dealing with large streams of data (> 10), or very small sets of data (< 10) or no data at all ($= 0$). Each of these scenarios can happen for any given program, but small data sets are more common in UI work (mouse input, keyboard input, retrieving data from a remote endpoint). Large data sets are more likely to occur in a server environment when processing information. This is not hard and fast, but more a factor of how client/server architectures work.

3. **Iteration Count** - This is the number of times the compiled function is called. This represents the frequency of invocation, which can be seen as infrequent (< 10) or very frequent (> 10). The underlying VM will optimize the code as it detects hot spots, and so the overall cost of execution will modify as processing occurs.

The end result will be measured as **Processing Time**. This is the number of nanoseconds (test hardware supports nanonsecond resolution for timer) it takes to process one unit of input. Given a scenario, input size and iteration count, we will see how the different algorithms perform. The times will be determined using a high resolution timer provided by the Node.js engine [14]. This is fairly, accurate but is affected by garbage-collection, and so there are outliers in the test runs. To handle some of these outliers, we focus on the median, as well as the weighted average. The weighted average is defined as the central 80% of the data, excluding the top and bottom 10%.

Additionally, to provide consistent results from run to run, we pre-compile the code. When running the initial tests, the compilation cost was fairly small, but since each test run is run in the same V8 instance, the caching applies across all test runs and would unfairly burden the initial test.

3.3 Scenarios

3.3.1 MD5

The MD5 algorithm [26] is a hashing algorithm that generates a 128-bit signature for any input of bytes. This algorithm is computationally heavy. A unique feature of this test, is that it is the only one in which the manual form was pulled from a preexisting library that had already optimized heavily for cpu performance.

Looking at the results, the general trend is the optimized code outperforming the functional form, and the manual transformation outperforming the optimized code. The graphs reveal some more detailed analysis with respect to specific performance characteristics.

Looking at 3.1, this test highlights variability in code that deals with large amount of input, but is run infrequently. As compared to the other graphs, this graph shows the least consistency in time per array element. The underlying engine will be affected greatly by its internal caches, and hotspot optimizations. Overall the ordering of the implementation efficiency holds true, but there is a point where the functional form outperformed the optimized form even at 10k iterations.

With 3.2, this test highlights how the scenario handles the overhead of the optimization process, as the frequency of iteration inside the generated code is much less than the frequency of execution of the generated code. The overall performance at 1 iteration does not have enough iterations to amortize the startup costs. At iteration count 5000 and beyond you can see the code generally runs at the same cost per array element.

Comparing the input size focused figures (3.3, 3.6) versus the iteration focused figures (3.4, 3.5) it is apparent that the cost per iteration is much lower when the input size increases as opposed to the number of iterations. This points to the underlying engine being able to optimize tight loops better than repeated function invocations. Comparing 3.6 with 3.5, the times per array element start to converge. This would imply that there is some minimal threshold that reaps the full benefit ot the underlying engine's hotspot optimizations.

What is also clear, is that from the functional form to the optimized functional form, there is a dramatic increase in performance. What we see is about a 10x speedup between the optimized and non-optimized algorithm, and about a 3x speedup between the optimized and manual implementations.

Figure 3.1: MD5: Input Size vs Time (ns) with 2 Iterations

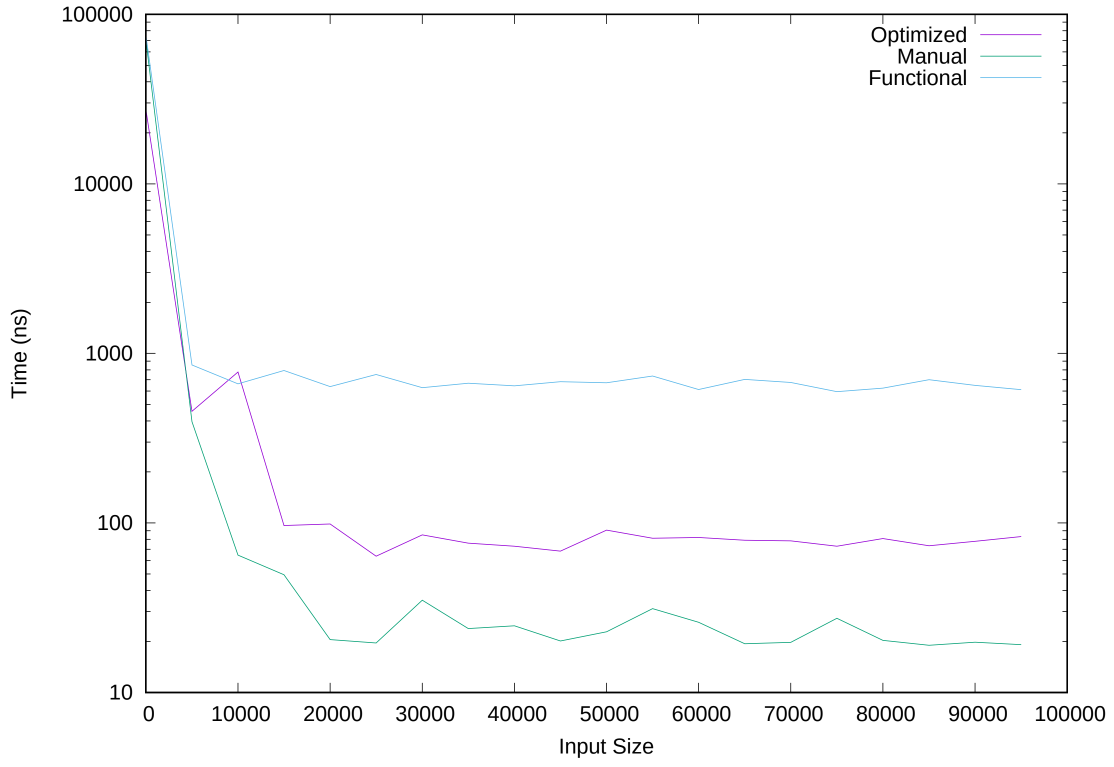


Figure 3.2: MD5: Iterations vs Time (ns) with an Input Size of 2

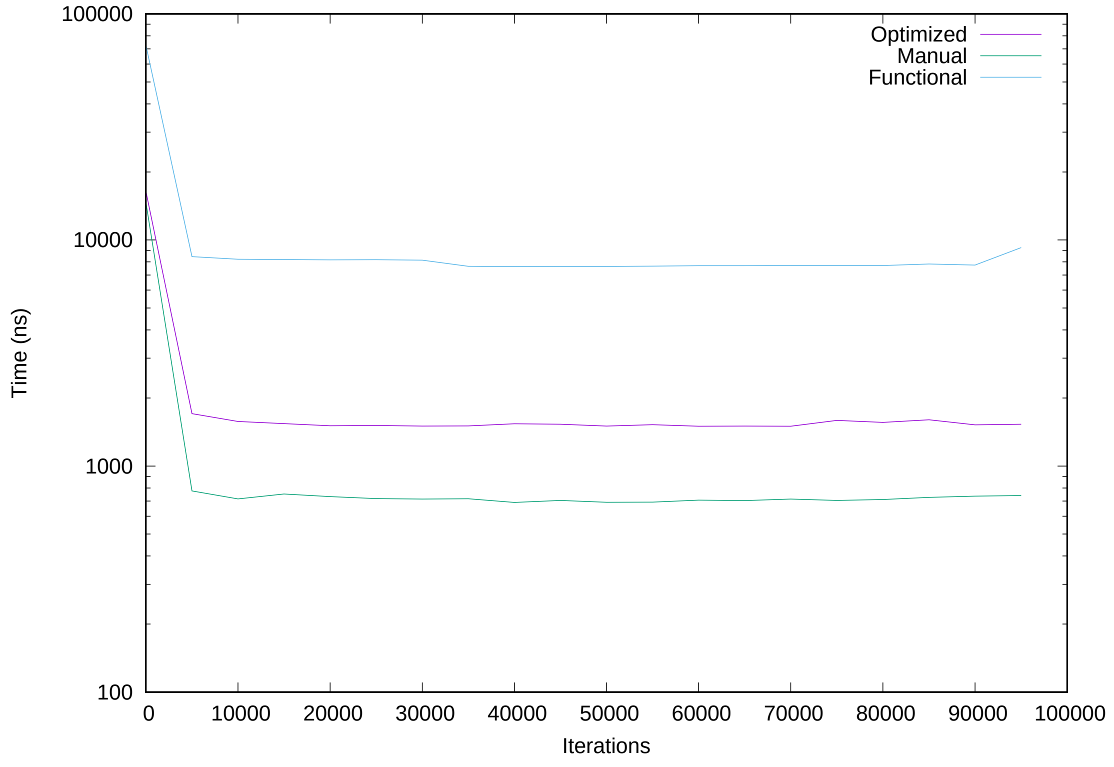


Figure 3.3: MD5: Input Size vs Time (ns) with 10 Iterations

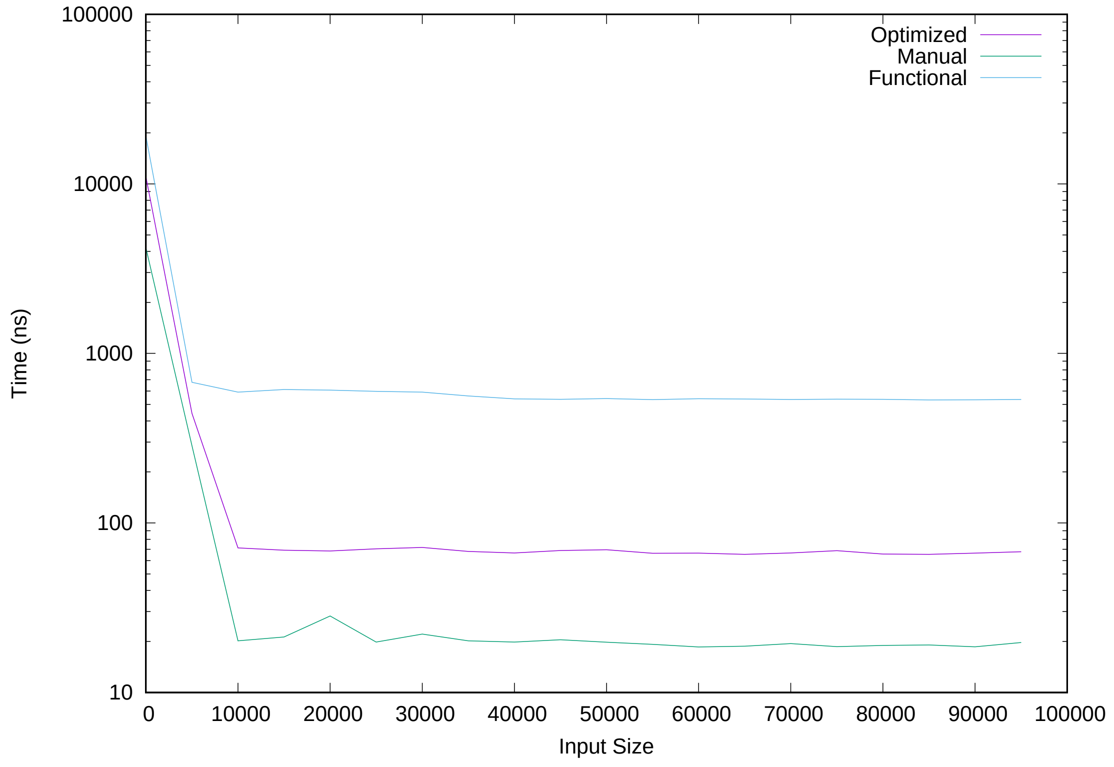


Figure 3.4: MD5: Iterations vs Time (ns) with an Input Size of 10

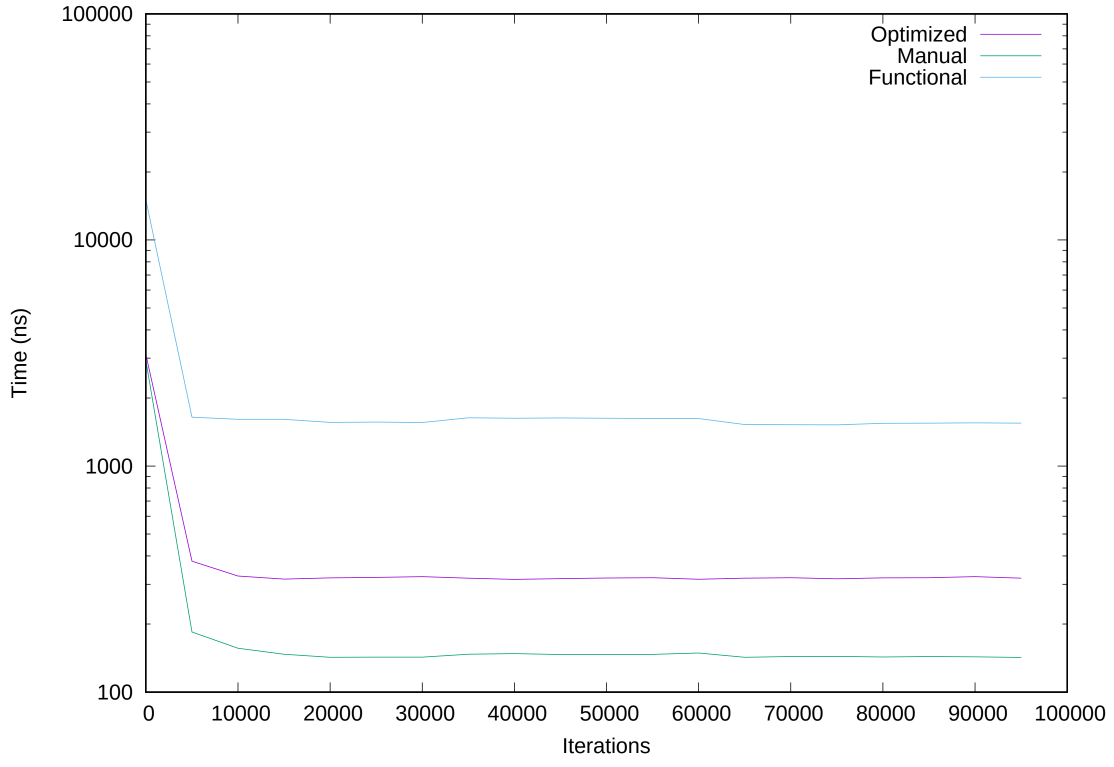


Figure 3.5: MD5: Iterations vs Time (ns) with an Input Size of 100

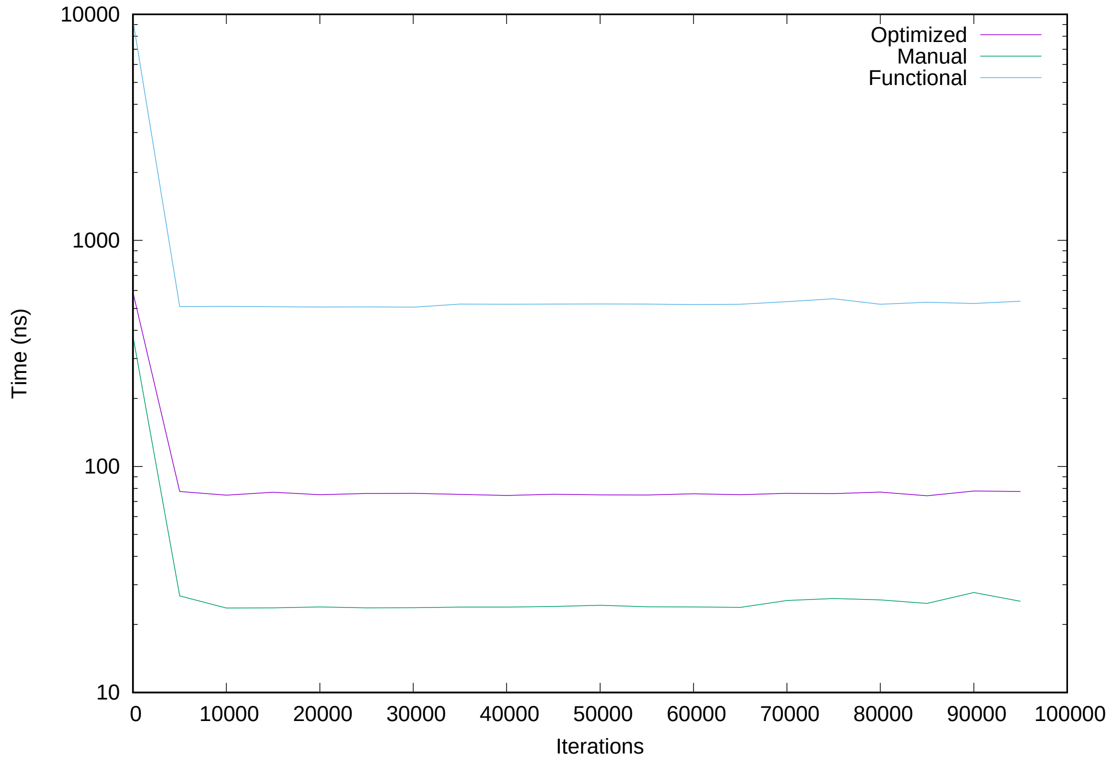
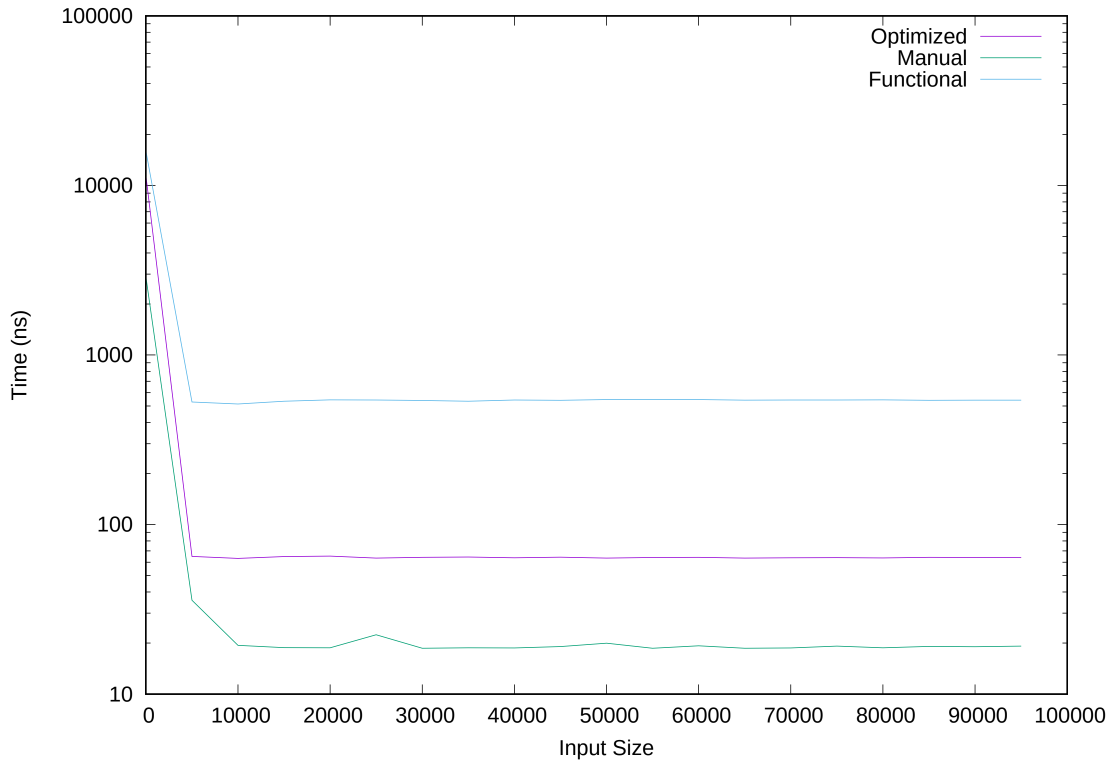


Figure 3.6: MD5: Input Size vs Time (ns) with 100 Iterations



3.3.2 Sort Score Sum

This is a functional implementation of a simple problem from the algorithm puzzle website Project Euler. The nature of the problem is a combination of I/O operations and simple computation resulting in a single number to represent the data set. [13]

In every test (figure 3.7, 3.8, 3.9, 3.10, 3.11, 3.12), the optimized form out performs the manual and the nominal functional form (on average). This is a little surprising given the fact that the generated code is nearly identical (in function and form) to the manual transformation. Part of what is happening, is that the nature of compiling the code at runtime affords some interesting performance boosts for a language like JavaScript. Some of the reasons for increased performance are:

1. **No enclosing scopes** All variables are passed in directly to the function because the compiled code no longer has access to any closed variables.

2. **Parameter variables** Within V8, function parameters versus local parameters seem to have a faster access time. Google's own closure compiler also utilizes this knowledge and optimizes as many variables as possible to be directly accessed via the function parameters.

3. **"use strict"**. When compiling the code at runtime, we are able to make stronger assumptions about the code we are generating, and enforce strict mode.

Looking at figures 3.7, 3.10, 3.12 you can also see that the input size is 1 or the number of iterations is 1, that the performance cannot be guaranteed to outperform the manual transformation. This points back to the runtime startup cost of running the compiled code is higher than just running the manual form directly.

These rules may not apply to other languages if this was to be ported from JavaScript to target a different language.

Figure 3.7: sort-score-sum: Input Size vs Time (ns) with 2 Iterations

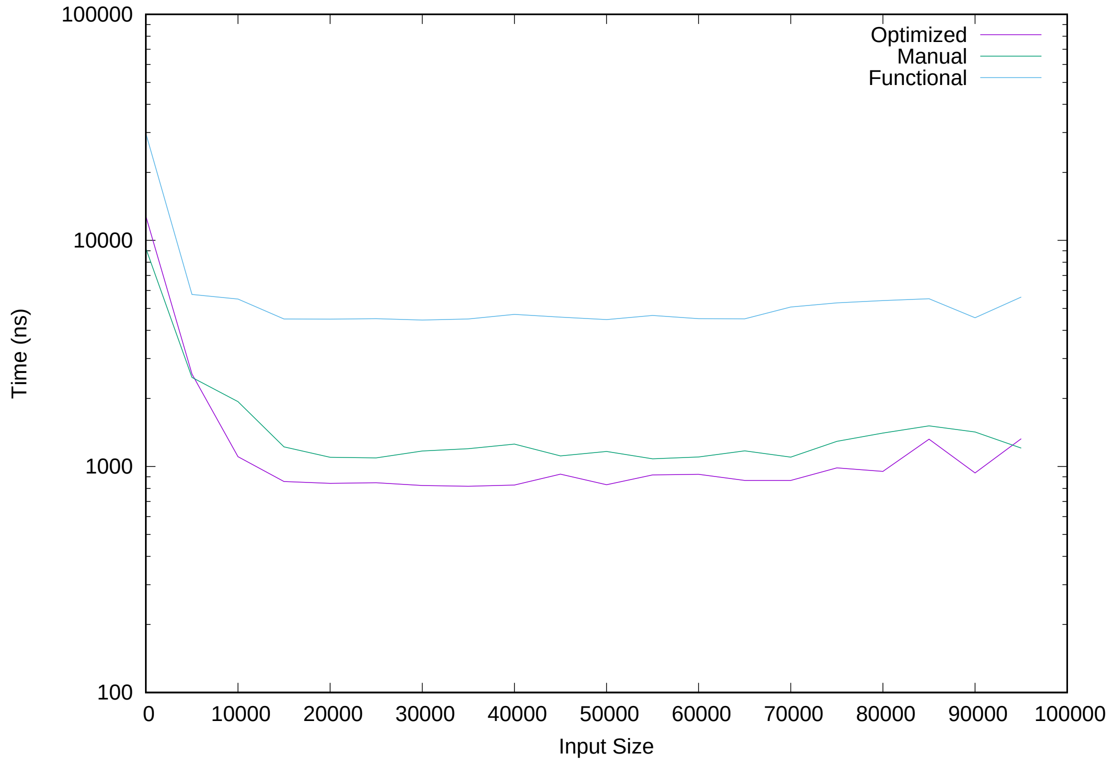


Figure 3.8: sort-score-sum: Iterations vs Time (ns) with an Input Size of 2

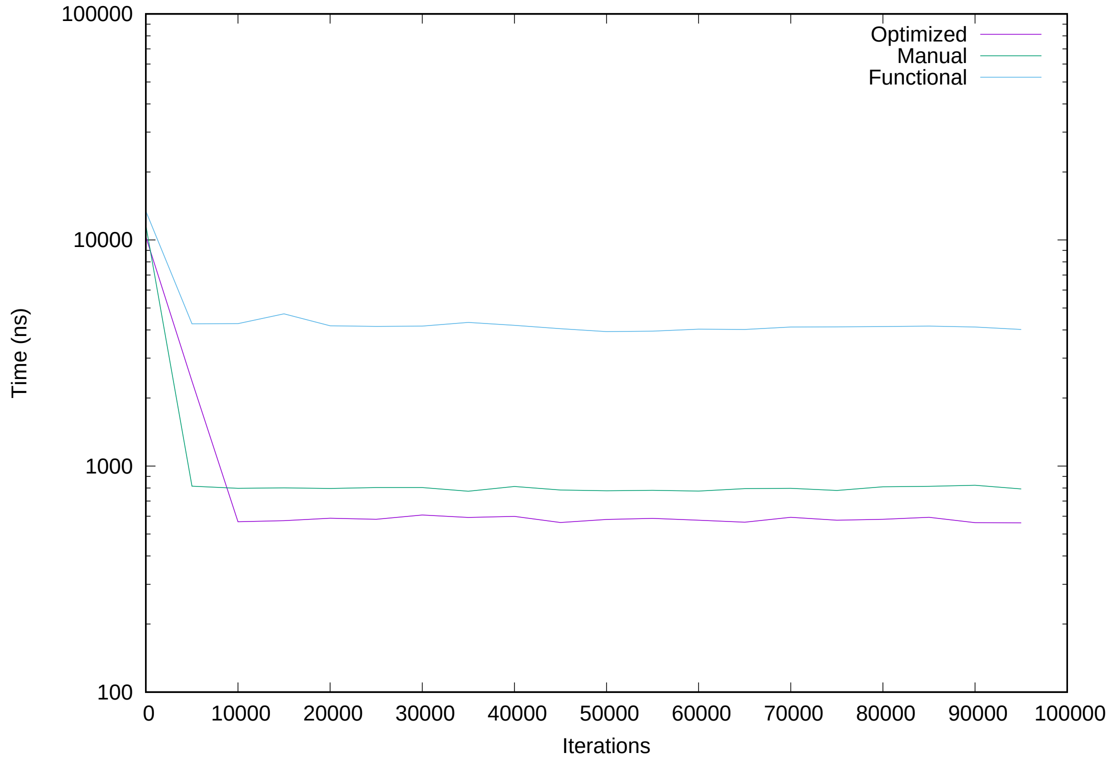


Figure 3.9: sort-score-sum: Input Size vs Time (ns) with 10 Iterations

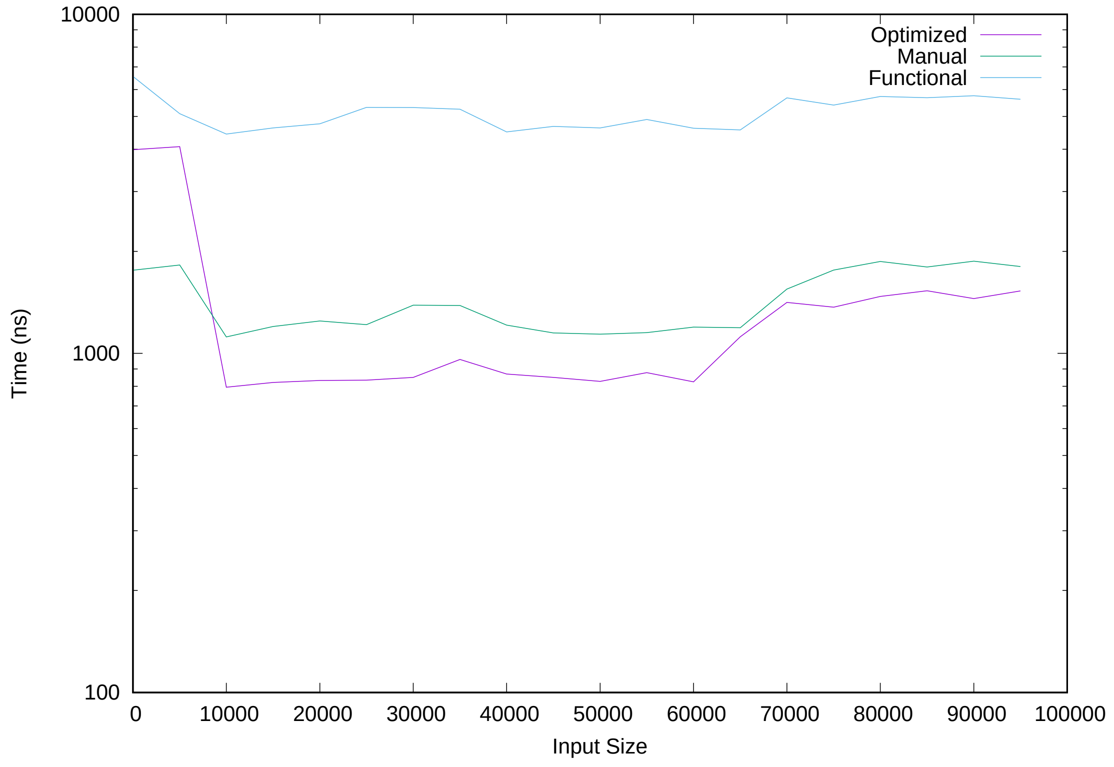


Figure 3.10: sort-score-sum: Iterations vs Time (ns) with an Input Size of 10

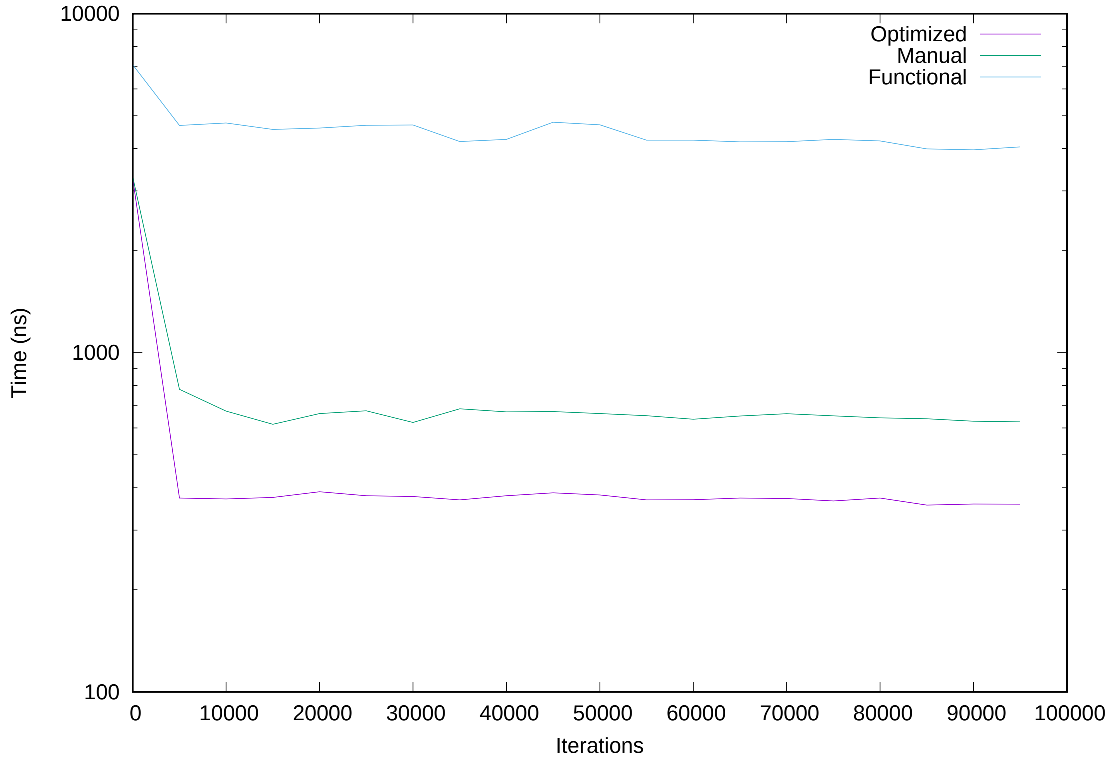


Figure 3.11: sort-score-sum: Iterations vs Time (ns) with an Input Size of 100

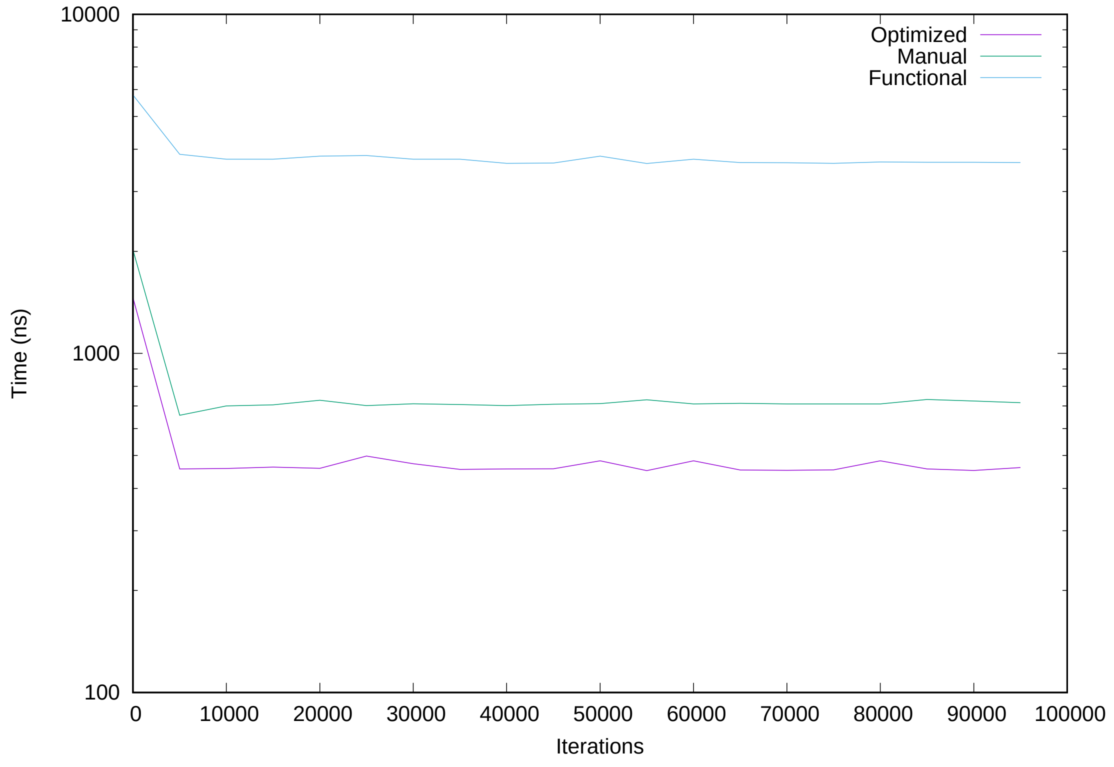
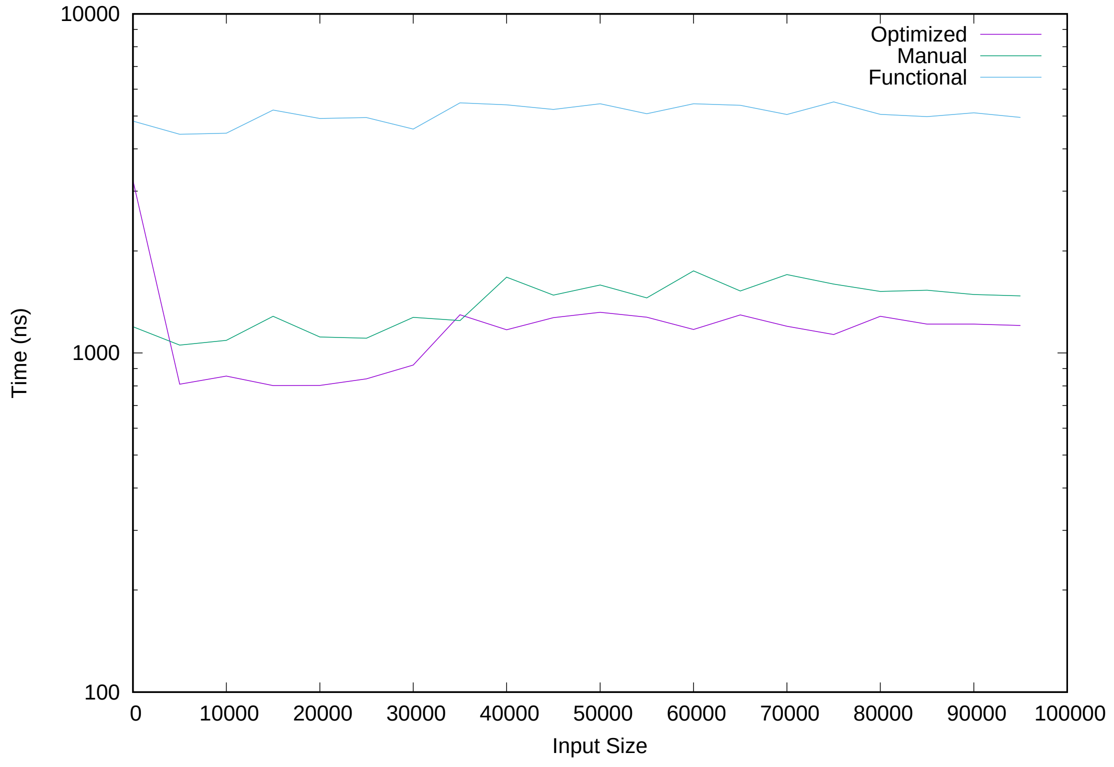


Figure 3.12: sort-score-sum: Input Size vs Time (ns) with 100 Iterations



3.3.3 Standard Deviation

A simple algorithm to compute the standard deviation of an array of numbers. It is computationally heavy.

For the most part, the optimized form outperforms the standard functional form (figures 3.14, 3.13, 3.15, 3.16, 3.17, 3.18). It also outperforms the manual form in all tests except figure 3.14.

This is a surprising result in that the manual form should generally be better than optimized form. This has a similar pattern to that of §3.3.2, and shows how the underlying engine is sensitive to certain optimizations (e.g. removing closures and function parameters).

What you can also see, specifically in the test with a small input size of two (Iterations vs Time (ns) with an Input Size of two), is that there is a cost to the overhead of the optimized code. It is small, but the underlying JavaScript engine is able to execute the operations on a two element array faster than the generated `for` loop.

As with all optimizations of this nature, there will be edge cases that the optimization does not perform as fast. This comes down to the trade-off analysis of determining if the increased performance is worth the cost of code maintenance, in addition to looking at the other cases where the optimized form does outperform.

Figure 3.13: std-dev: Input Size vs Time (ns) with 2 Iterations

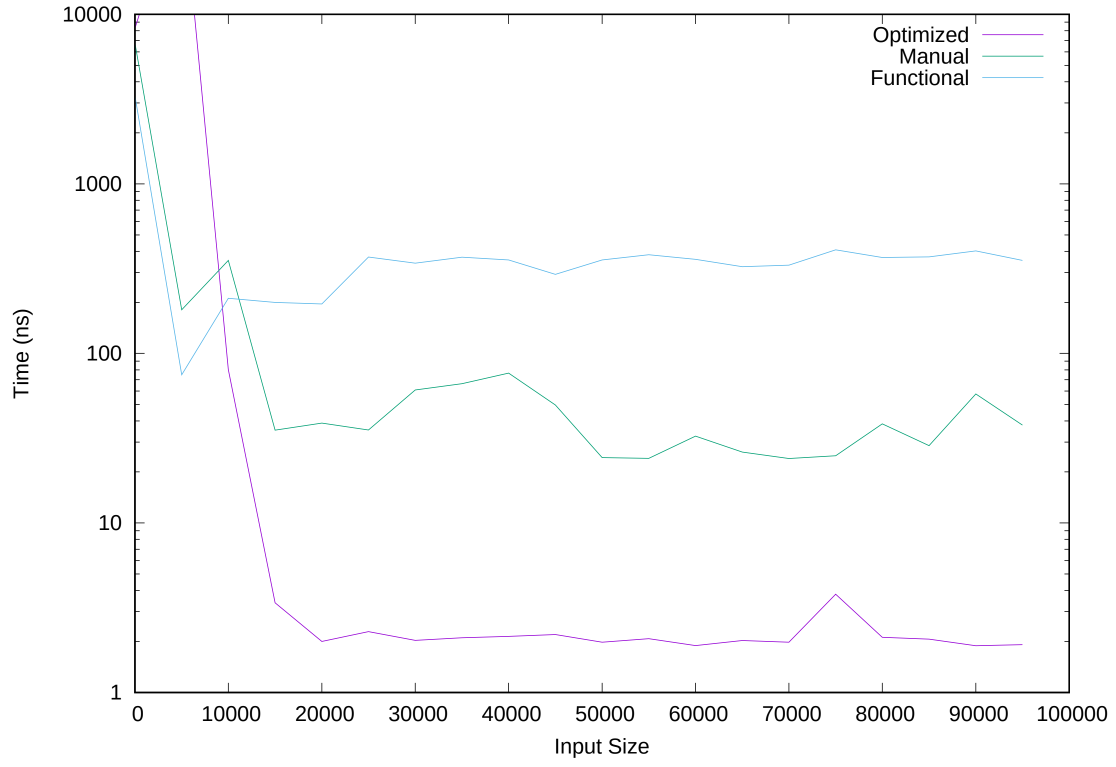


Figure 3.14: std-dev: Iterations vs Time (ns) with an Input Size of 2

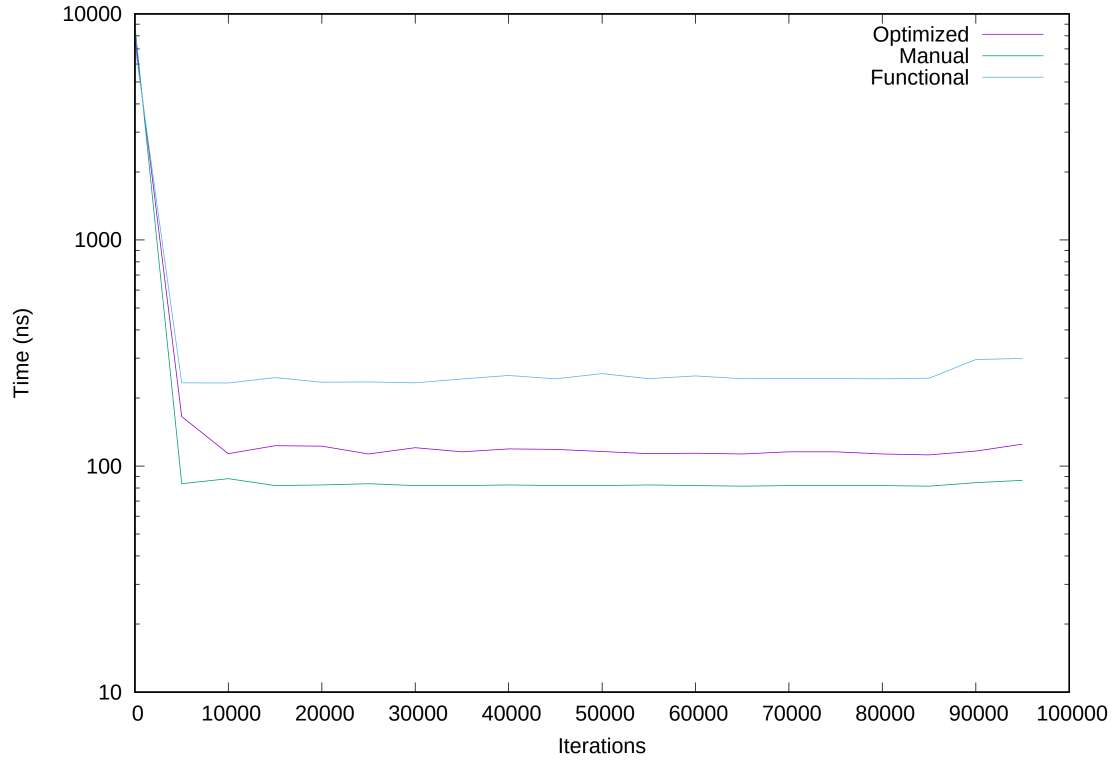


Figure 3.15: std-dev: Input Size vs Time (ns) with 10 Iterations

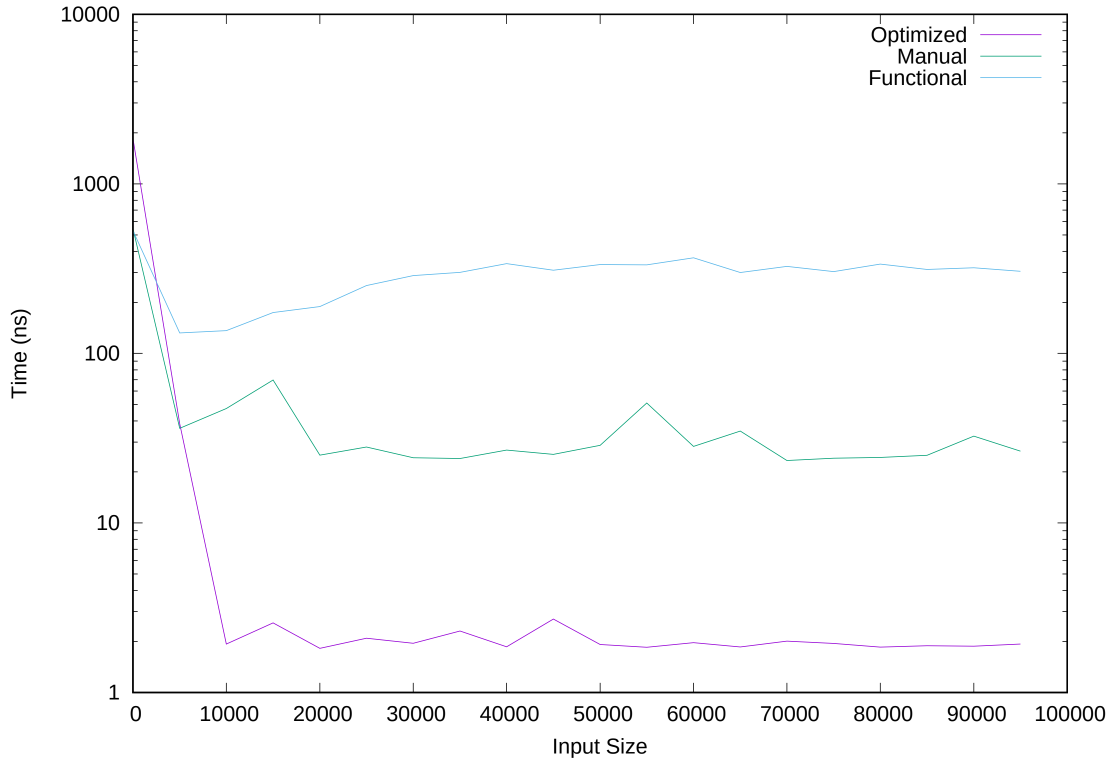


Figure 3.16: std-dev: Iterations vs Time (ns) with an Input Size of 10

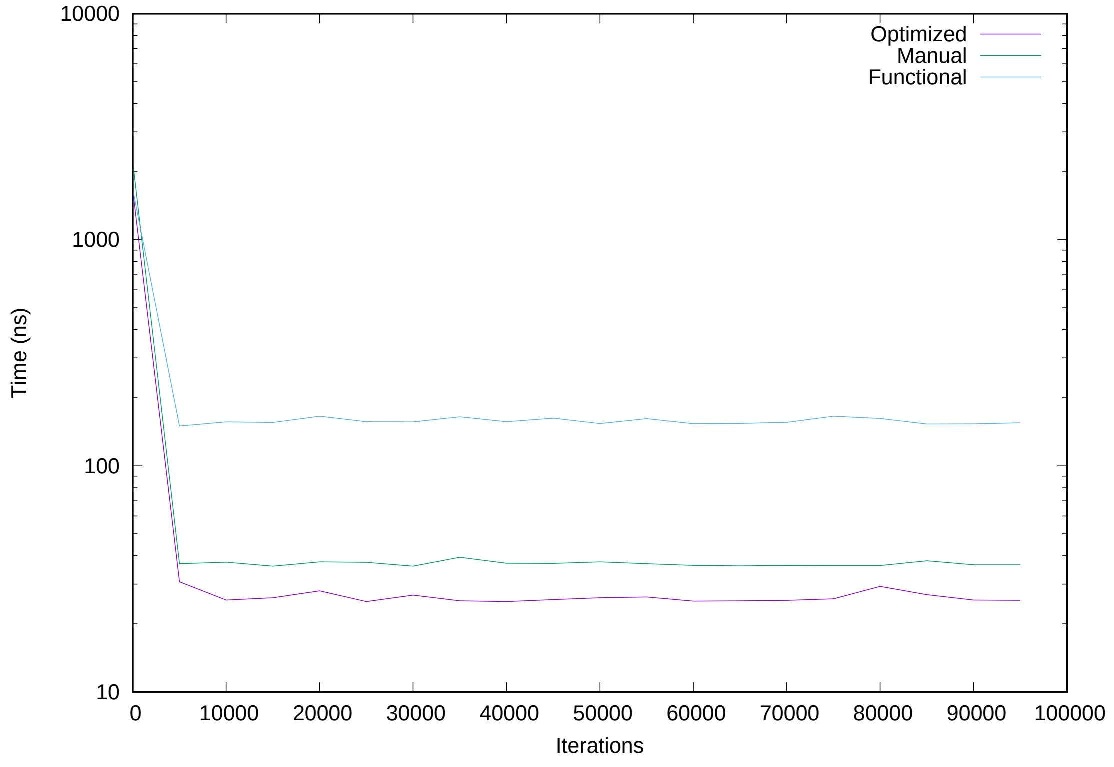


Figure 3.17: std-dev: Iterations vs Time (ns) with an Input Size of 100

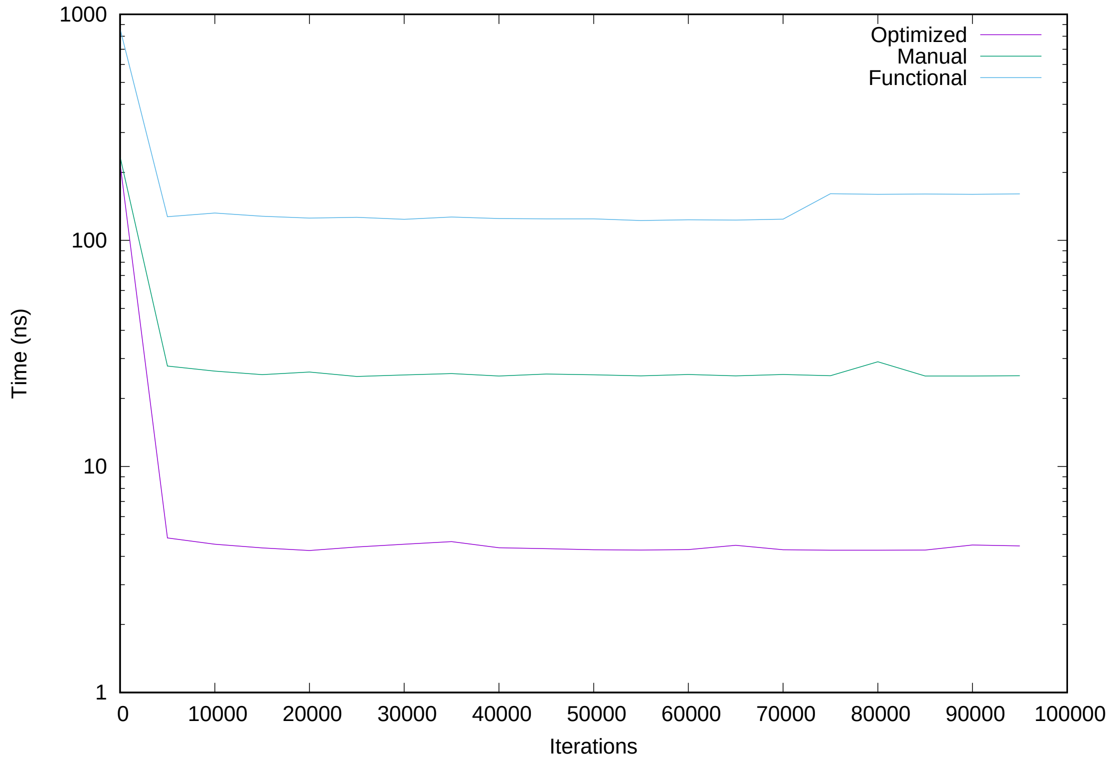
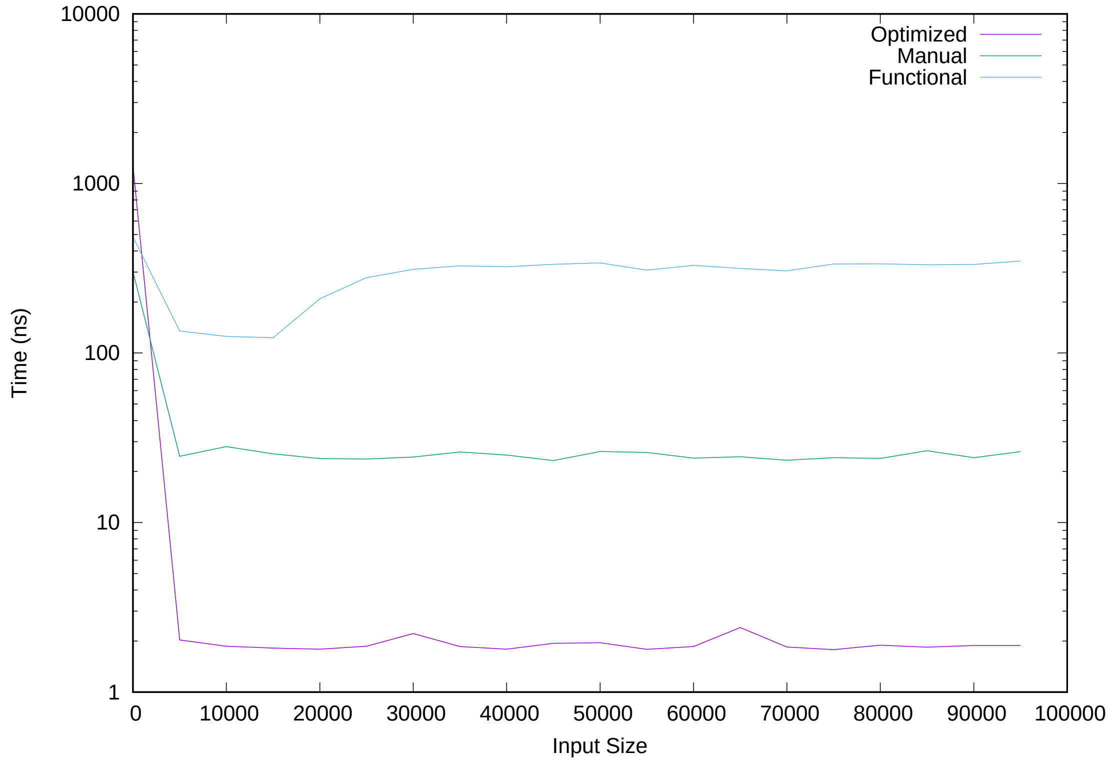


Figure 3.18: std-dev: Input Size vs Time (ns) with 100 Iterations



3.3.4 Text Analysis

This algorithm is doing frequency analysis on a large body of text. The cost of loading the text and splitting into separate words is not counted. The primary operations of this algorithm are basic comparison and object assignment.

Out of all scenarios, this one shows the most mixed results, but is also more representative of normal business logic, specifically the act of reading and writing to objects, and text processing.

Looking at figures 3.19 and 3.20 that deal with small input size (2) or a small number of iterations (2), there is no clear winner (as opposed to the other scenarios). There are points (especially with lower input size or iteration count) in which the manual form is outperformed by functional form, and the optimized form is randomly outperformed by both the manual and functional form throughout. This test highlights the cost of memory reading/writing and the variability it brings.

Once we get past the smaller input size/iteration count, a pattern establishes itself. Figures 3.21, 3.23, and 3.24 show the optimized form generally outperforming the manual and functional forms. This lines up with other scenarios but not in all cases.

Figure 3.23 shows the manual form consistently outperforming the optimized form. This uniquely highlights that there will be usage patterns that will benefit from the optimized form, but additional performance can be gotten from manual optimization.

Figure 3.19: text-analysis: Input Size vs Time (ns) with 2 Iterations

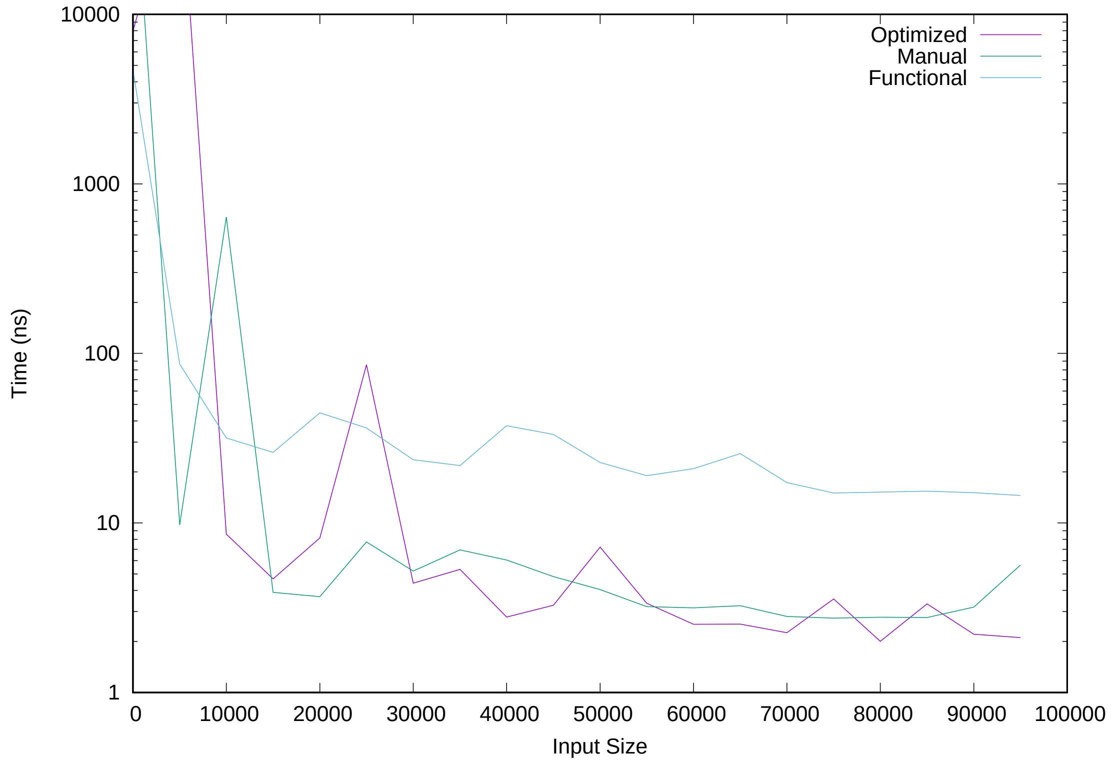


Figure 3.20: text-analysis: Iterations vs Time (ns) with an Input Size of 2

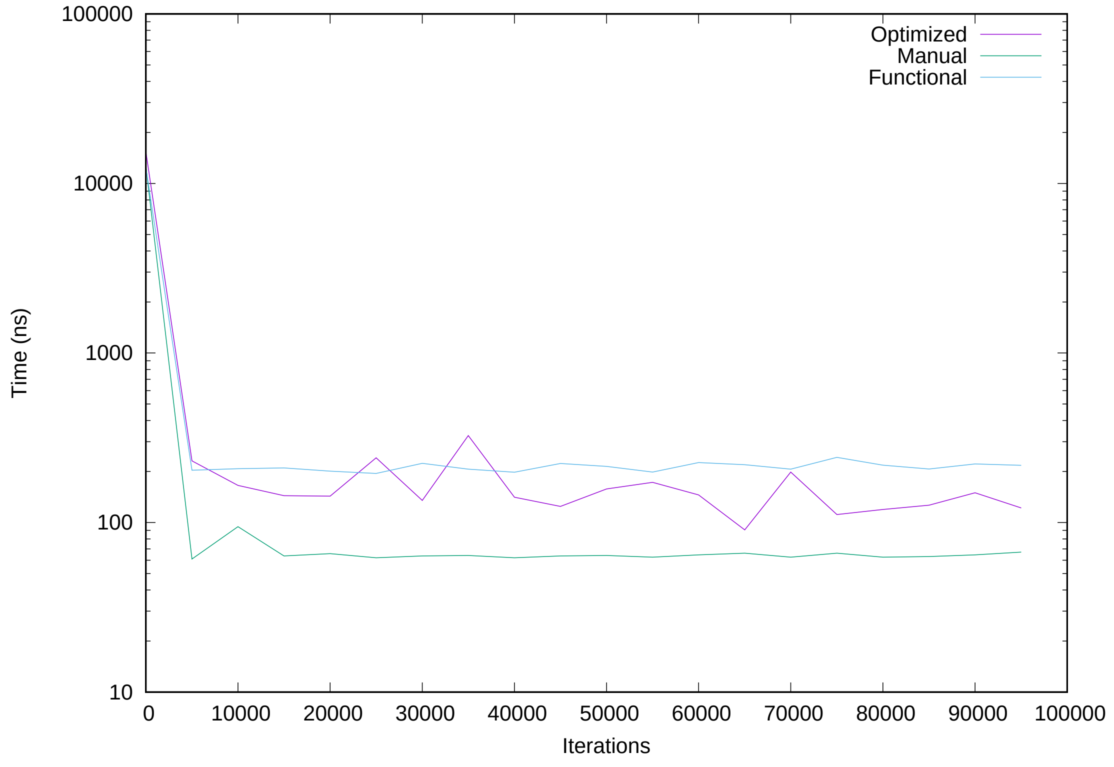


Figure 3.21: text-analysis: Input Size vs Time (ns) with 10 Iterations

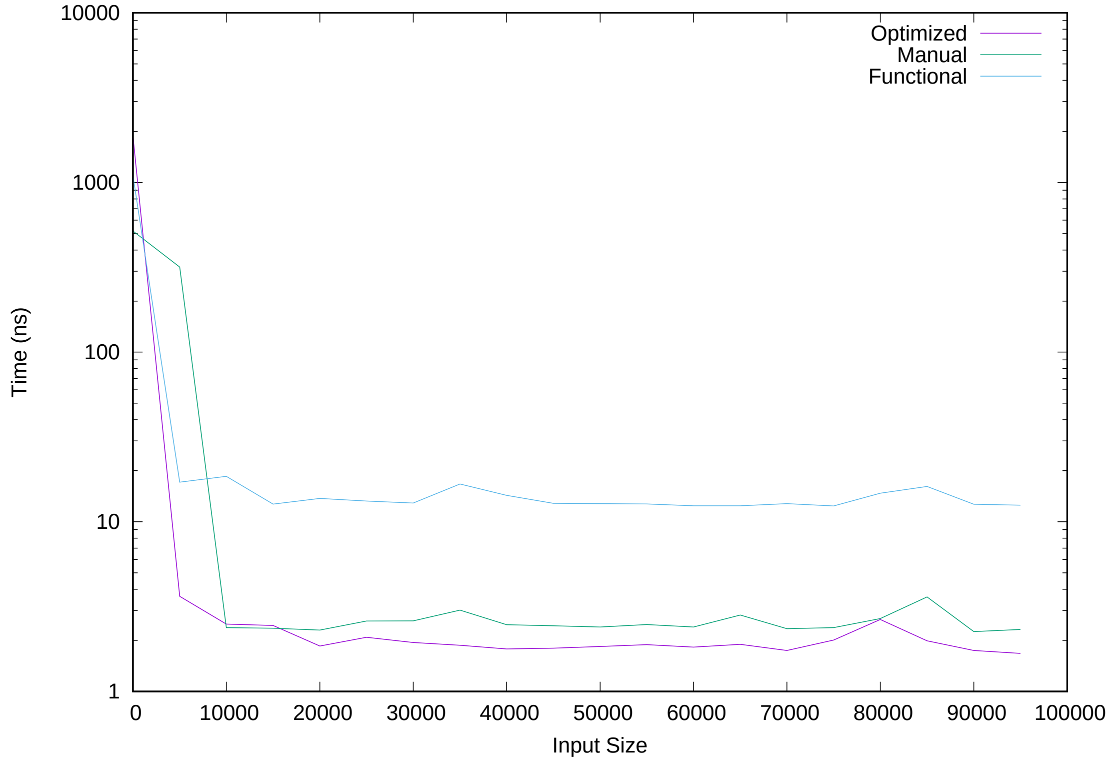


Figure 3.22: text-analysis: Iterations vs Time (ns) with an Input Size of 10

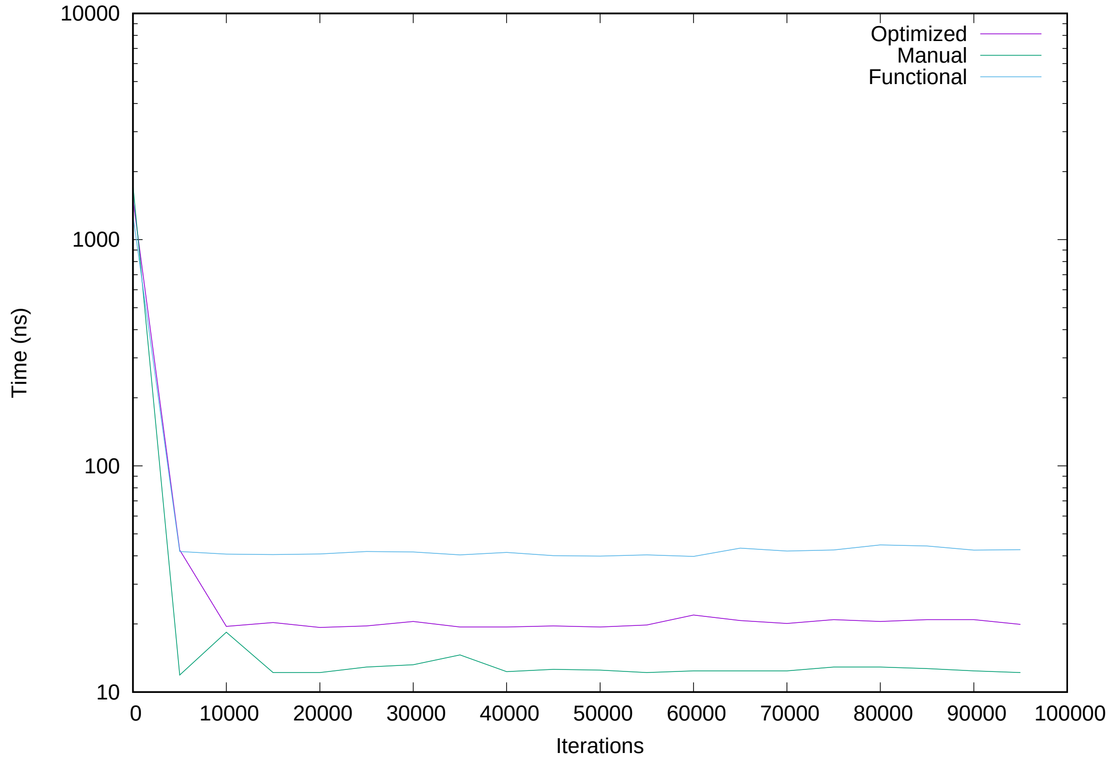


Figure 3.23: text-analysis: Iterations vs Time (ns) with an Input Size of 100

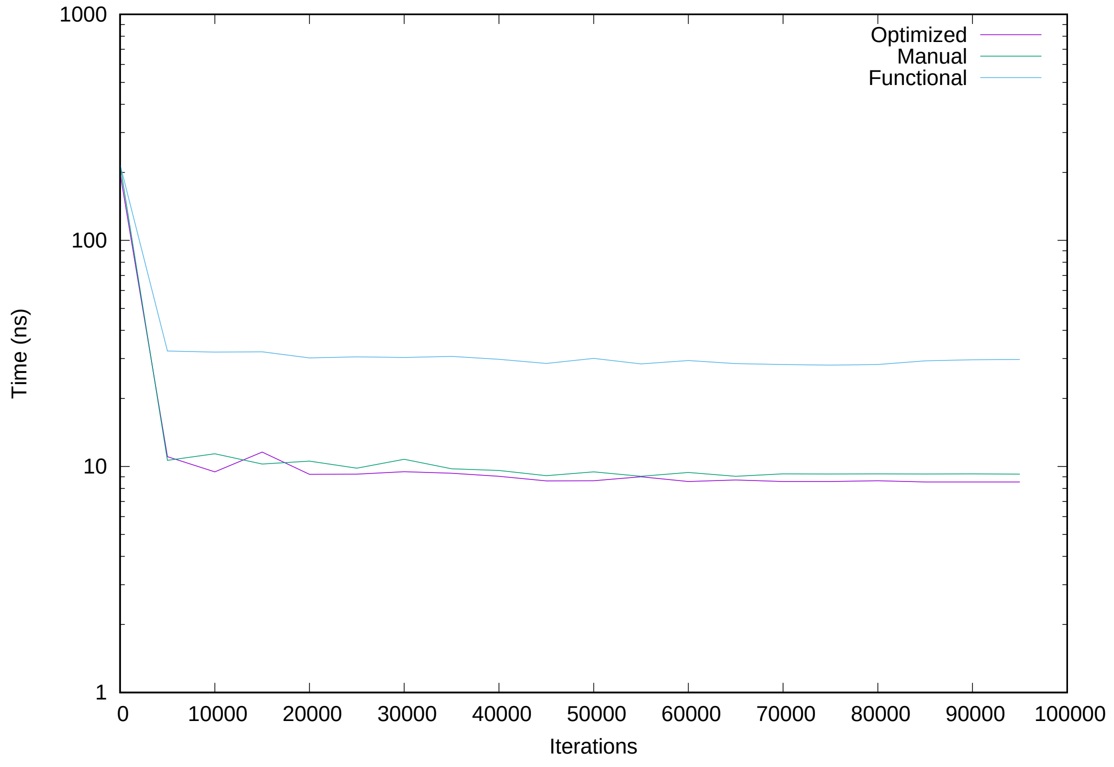
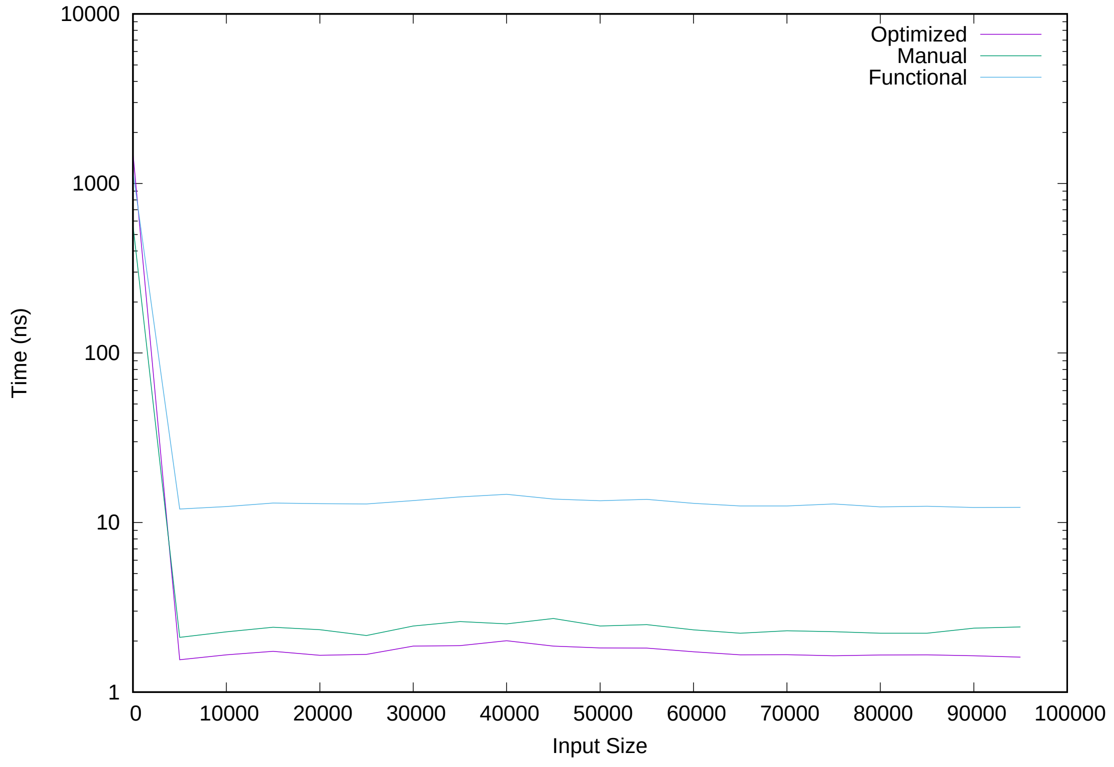


Figure 3.24: text-analysis: Input Size vs Time (ns) with 100 Iterations



Chapter 4

Conclusion

4.1 Related Work

JavaScript, as a research topic, has matured greatly in recent years. More and more research is being done on JavaScript the language, and the underlying functionality versus applications of JavaScript to specific problem domains.

One of the closest works is a paper that focuses on converting the functional pipelines in JavaScript [28], to functional composition. In many ways, it embodies the same concept as our work, but achieves the results through very different means. It requires the use of a special set of functions, requiring the programmer to write code that is fixed to a non-standard paradigm. It also still requires invoking the functions directly instead of converting existing functional pipelines into generated code.

There has also been work focusing on other looking at server workload characteristics [21] which will help to identify the general benefits of these optimizations, as the ratio of computation to idling gives an upper bound to the possible performance increases.

Other relevant work, which our work subsumes, is focusing on detecting function purity in JavaScript [20]. The paper's main focus is on determining whether or not a function has side effects and does so with formal language analysis. The code that we used is not as formally rigid, but is sufficient in looking for writes before reads, and accessing variables in different scopes.

JavaScript, as a research topic, has increased in popularity. Unfortunately, apart from work focused on functional pipeline optimization via composition [28], most of the work is tangential at best.

4.2 Future Work

The concept of composing functional operations, at runtime or compile time, has implications beyond JavaScript and list-based functional pipelines. Working with a language that is statically typed would remove a lot of the guess work from the process in general, and would allow for more consistent and rigorous optimizations. Taking the optimization out of the JavaScript code and moving it into the engine would open up many more avenues for optimizations and would also minimize the duplicated effort in parsing and compiling the JavaScript code at runtime.

Another area of optimization would be moving the optimization algorithm towards stream-based programming. Streams (generators) are powerful constructs, and stream transformation/filtering/reduction is useful. RxJS [27] is a stream based framework for JavaScript that is gaining in popularity. It is an implementation of the general design pattern called Reactive Extensions [25]. The core of the framework are streams(Observable) and this same concept is poised to gain first class status within the JavaScript language [29]. The ability to create new streams, and to compile the composition to a single function versus a nested of set of function invocations could provide a significant boost to the general performance of the framework and all that rely upon it.

Beyond all of this, there are more general optimizations that could be made, by the nature of recompiling the code at runtime, and specifically merging multiple bodies of code together into a single function. The V8 JavaScript engine is already great at optimizing where it can, but even just reducing the amount of code it has to evaluate (constant folding, algebraic simplifications, dead code elimination, etc. would be useful enhancements).

4.3 Wrapping Up

The concept of this algorithm has broad reaching implications. At a high level, it is clear that removing extraneous array creations and function invocations will always produce faster code. More specifically, it is fairly straightforward to convert a simple functional pipeline paradigm into procedural code, and maintain the correctness of the code. While being a seemingly simple operation, but the details are deceptively complex.

There may be gaps in the algorithm, but its utility and benefits are obvious. When used as a supplement to optimize functional code versus doing a manual transformation, it is able to gain immediate performance benefits without losing code clarity or maintainability. And if manual optimization is still required to get the absolute performance out of a system, the optimization makes no requirement on the source code itself.

From the scenario data, the algorithm performs well even while dealing with the very flexible nature of JavaScript. It is not perfect or maximal in all scenarios, but it is consistently faster than the un-optimized functional form by a sizable factor.

Bibliography

[1] Mozilla Corporation. *Array.prototype.filter()*. 2016. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter` (visited on 09/25/2016).

[2] Mozilla Corporation. *Array.prototype.forEach()*. 2016. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach` (visited on 09/25/2016).

[3] Mozilla Corporation. *Array.prototype.map()*. 2016. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map` (visited on 09/25/2016).

[4] Mozilla Corporation. *Array.prototype.reduce()*. 2016. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce` (visited on 09/25/2016).

[5] Mozilla Corporation. *Arrow functions*. 2016. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions` (visited on 09/25/2016).

[6] Mozilla Corporation. *const*. 2016. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let` (visited on 09/25/2016).

[7] Mozilla Corporation. *const*. 2016. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const` (visited on 09/25/2016).

[8] Mozilla Corporation. *Destructuring assignment*. 2016. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment` (visited on 09/25/2016).

[9] Mozilla Corporation. *eval()*. 2016. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval` (visited on 09/25/2016).

[10] Mozilla Corporation. *getter*. 2016. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get` (visited on 09/25/2016).

[11] Mozilla Corporation. *Inheritance and the prototype chain*. 2016. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain` (visited on 09/25/2016).

[12] Kunal Dabir. *Groovy performance : iterating with closure vs loop*. 2011. URL: `http://kunaldabir.blogspot.it/2011/07/groovy-performance-iterating-with.html` (visited on 09/25/2016).

[13] Project Euler. *Names scores*. 2005. URL: `https://projecteuler.net/index.php?section=problems&id=022` (visited on 09/25/2016).

[14] Node.js Foundation. *process.hrtime([time])*. 2016. URL: `https://nodejs.org/api/process.html#process_process_hrtime_time` (visited on 09/25/2016).

[15] Wikimedia Foundation. *Functional Programming Recursion*. 2016. URL: `https://en.wikipedia.org/wiki/Functional_programming#Recursion` (visited on 09/25/2016).

[16] Wikimedia Foundation. *JavaScript*. 2016. URL: `https://en.wikipedia.org/wiki/JavaScript` (visited on 09/25/2016).

[17] Ben Hollis. *Investigating JavaScript Array Iteration Performance*. 2009. URL: `http://benhollis.net/blog/2009/12/13/investigating-javascript-array-iteration-performance/` (visited on 09/25/2016).

[18] StatCounter Inc. *StatCounter Global Stats Top 9 Browsers on Aug 2016*. 2016. URL: `http://gs.statcounter.com/#all-browser-ww-monthly-201608-201608-bar` (visited on 09/25/2016).

[19] Benjamin Livshits Magnus Madsen and Michael Fanning. *Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries*. Tech. rep. Aarhus University, Microsoft Research, and Microsoft Corporation, 2012.

[20] J. Nicolay et al. "Detecting function purity in JavaScript". In: *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*. Sept. 2015, pp. 101–110. DOI: `10.1109/SCAM.2015.7335406`.

[21] T. Ogasawara. "Workload characterization of server-side JavaScript". In: *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. Oct. 2014, pp. 13–21. DOI: `10.1109/IISWC.2014.6983035`.

[22] Paul Phillips. *optimizing simple fors*. 2010. URL: `http://scala-language.1934581.n4.nabble.com/optimizing-simple-fors-td2545502.html` (visited on 09/25/2016).

[23] J. Radhakrishnan. "Hardware dependency and performance of JavaScript engines used in popular browsers". In: *2015 International Conference on Control*

Communication Computing India (ICCC). Nov. 2015, pp. 681–684. DOI: 10.1109/ICCC.2015.7432981.

[24] Baishakhi Ray et al. “A Large Scale Study of Programming Languages and Code Quality in Github”. In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 155–165. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635922. URL: http://doi.acm.org/10.1145/2635868.2635922.

[25] *ReactiveX*. 2016. URL: http://reactivex.io/ (visited on 10/30/2016).

[26] R. Rivest. *The MD5 Message-Digest Algorithm*. 1992. URL: https://www.ietf.org/rfc/rfc1321.txt (visited on 10/30/2016).

[27] *RxJS 5 (release candidate)*. 2016. URL: https://github.com/ReactiveX/rxjs (visited on 10/30/2016).

[28] P. Singh, R. Mathew, and V. Singh. “Rethinking javascript loops as combinators”. In: *2015 International Conference on Computing and Network Communications (CoCoNet)*. Dec. 2015, pp. 288–294. DOI: 10.1109/CoCoNet.2015.7411200.

[29] Kevin Smith. *ECMAScript Observable*. 2016. URL: https://github.com/tc39/proposal-observable (visited on 10/30/2016).

[30] Timothy Soehnlin. *Functional Pipeline Optimization*. 2016. URL: https://github.com/arciisine/functional-pipeline-optimization (visited on 10/30/2016).