Modtalk Optimizations: a Senior Project

Kurt Kilpela
December 2014

Modtalk and its abstract machine

Modtalk is a modular implementation of Smalltalk. (I'm sure you've heard this too many times.) In lieu of the liveness of traditional Smalltalks, we choose to compile Modtalk, separating the development and runtime environments. We can compile a program and give a native executable!

Compilation is to our intermediate form which targets our abstract machine. The abstract machine was inspired by work by Pat Caudill and Allen Wirfs-Brock.

We define a register set, stack, and instruction set.

Register	Use
R	Holds the receiver and return value
A	Holds the first argument
X	Used to index off a pointer
SP	Points to the top slot of our stack
FP	Points to the base of a frame
PC	Program counter
CC	Flags register (specific to that process)

Our instruction set operates on our registers and stack. We offer a variety of options for our intermediate form. Each is useful in one way or another.

Currently our main form of intermediate representation (IR) is called byte code IR (BCIR). We assign an 8-bit value to a type of instruction. This form has a few benefits. Firstly, it's compact. Secondly, it's easily converted to other forms. Lastly, it could be used in a JIT for fast runtime compilation.

Our second form of IR is human readable form which is useful for debugging and testing against.

A more recent form of IR is representation as a control flow graph (CFGIR). This form will be used for control flow optimizations in the future.

Optimizations foundation

The goal of optimizations is to make a program faster, or reduce it's size. That said, an optimization may result in slower runtime or an increase in program size. Detecting when to apply an optimization is a difficult problem. Some optimizations result in a large net positive for the program, trading slight increases in program size for massive gains in performance. Other optimizations might bloat a program while resulting in "little" increase in program speed.

Optimizations can be applied at various times. Compilers can apply optimizations at compile time like I currently do. Runtime statistics can be captured to improve the effectiveness of these optimizations.

Dynamic languages tend to apply optimizations at runtime compiling to a byte code form and optimizing hot spots in programs. Areas of high use receive the highest level of optimization. This can result in near c speeds for programs.

Individual optimizations tend to be easier to apply at a specific level of abstraction. During compilation and optimization, the program is transitioned through various levels of abstraction. Representations closer to source code allow for object optimizations. As you approach machine code, optimizations like peephole optimizations and dead code elimination become easier and more effective.

My original plans

In the beginning, I chose a set of optimizations I thought would provide a good set for the project.

Peephole Optimization:

Removed sections of code which repeat or do not contribute to the output of the program.

Leaf Method Optimization:

Prevent frame building for an activation which does not send a message. (setters/ getters)

Message Send Caching:

There are 3 types of call sites: monomorphic, polymorphic, and megamorphic. Most call sites are monomorphic, meaning they see one type of receiver. Polymorphic sites see a few

receiver types while megamorphic sites see many receiver types. Caching of monomorphic and polymorphic sites can result in very fast message sends.

CFG Generation:

On it's own, a control flow graph is not an optimization. Optimizations can be written that act upon the CFG.

SSA Form/Phi Assignment:

SSA is an intermediate form where each assignment is to a new register. This allows optimizations not available in other forms. Compilers such as clang and gcc use SSA to generate very fast code.

Optimizations I applied

I started out with a pretty broad and naive idea of what I planned to do. I had already applied some optimizations which I did not include in my original proposal but will be described here.

Peephole Optimization:

Peephole optimization was one of the easier optimizations to apply. For these optimizations, I wrote a visitor which walks IR. When the last node in an optimization is encountered, the optimization is checked and applied. Visitation continues at the start of the optimized section to allow all possible peephole optimizations to be applied.

Sample Sequence	Optimized Sequences
push R **pop R**	*(Sequence removed)*
mov [2, FP], R **mov R, A**	**mov [2, FP], A**

Other peephole optimizations exist in Modtalk. Many more could be created.

Leaf Method Optimization:

Optimization of leaf methods was relatively straight forward. During compilation, if a method is detected to be a leaf method, the compiler emits a different code sequence. This optimization allows accessors to skip frame building limiting runtime cost of accessors.

Sample getter	Optimized getter

push FP
move SP, FP
pushMethod
push nil
push R
move [3, FP], X
move [2, X], R
move [1, FP], X
move FP, SP
pop FP
ret
	move [2, R], R
	ret

I need to explain how message sends work in Modtalk before we continue. Methods are hashed into a method dictionary. The slot immediately following an object's header stores it's method dictionary. (This is why we use the method dictionary for caching as described in a moment.) A method dictionary is chained to the method dictionary for it's superclass. When an object is sent a message, lookup will traverse the linked list of method dictionary checking each for a method with that selector.

Message Send Caching:

As I've said before, most call sites are monomorphic. Implementing a monomorphic inline cache can provide a significant win. For polymorphic and monomorphic call sites, I chose to optimize them without caching. Our monomorphic sites now store the MethodDictionary of an object and the method which should be activated if the same type of receiver is being sent that message. If the method dictionary does not match, a lookup occurs and the cache is updated for the new receiver.

Normal Send	Cached Send

`mt_lookup,`
`(OP)&Symbol_standardBenchmark,`
`callX,`
	`mt_lookupAndCache,`
	`(OP)NULL, //method dictionary slot`
	`(OP)NULL, //method slot`
	`(OP)&Symbol_standardBenchmark,`
	`callX,`

The sequences above are for our threaded runtime and the cache is stored in code stream. In our x64 native runtime, the cache is not in the code stream.

Inverted Lookup:

A site may only activate one method but have many receivers. Imagine a class implements the method #new. All of it's subclasses also respond to #new but in our cache implementation, the method dictionary is used as the key. Unfortunately, linear search would be too slow when there are more than (approximately) 3 different types. We can however make lookup faster. While maintaining the original lookup method (for use by perform:), we are able to redirect lookup at a call site to look in a different location.

For each selector sent in a Modtalk program, I generate an inverse lookup dictionary. Instead of using the selector as a key, I use the method dictionary of the object. We look in a selectors dictionary to obtain a nearly constant time lookup. For a message like #new, this can be a tremendous win. In the case where the method is in the receivers local method dictionary, the lookup time ***should*** be about the same.

Open Coding:

I owe you an explanation of open coding. In Smalltalk, everything is a message send. When I say everything, I mean everything! The language does not define control structures as part of it's implementation. Control structures such as conditionals and loops are implemented via message sends and blocks (closures). Blocks aren't required for control structures but make them a lot nicer to implement and use. This causes performance to suffer. For loops, the stack would grow an activation for each run through the loop and lookup time for the conditional messages is noticeable.

When Smalltalk was originally released, books were released to explain aspects of the system and provide advice for implementors. There is a list of selectors which can be "open coded" for performance reasons. With this, you lose polymorphism around these specific selectors. So, this #whileTrue: would be open coded as shown below. I'll discuss the performance implications later.

```
[ aNumber <  5]
        whileTrue:
                [aNumber := aNumber + 1]
start:
        aNumber < 5
        booleanCheck
        branchFalse end
        aNumber := aNumber + 1
        branch start
```

end:

Control Flow Graph:

As discussed before, generating the control flow graph (CFG) is not an optimization on it's own. Optimizations can be applied to a method in this form. Compiler intention (like storing into a variable) is maintained at this level. This is one of the trickier portions of this project. The relationship between the compiler and it's encoder needed to change. Each encoder (the thing that converts intention into intermediate code) is now responsible for interpreting more intentional messages from the compiler.

I'd like to note, the level of abstraction between the compiler and encoder does not seem to have reached a fixed point.

The Effect of Optimizations:

At this point, it's important to talk about benchmarks. For the test below, I'm running DeltaBlue, a constraints solver. 29,330,470 message sends take place in the course of a DeltaBlue run.

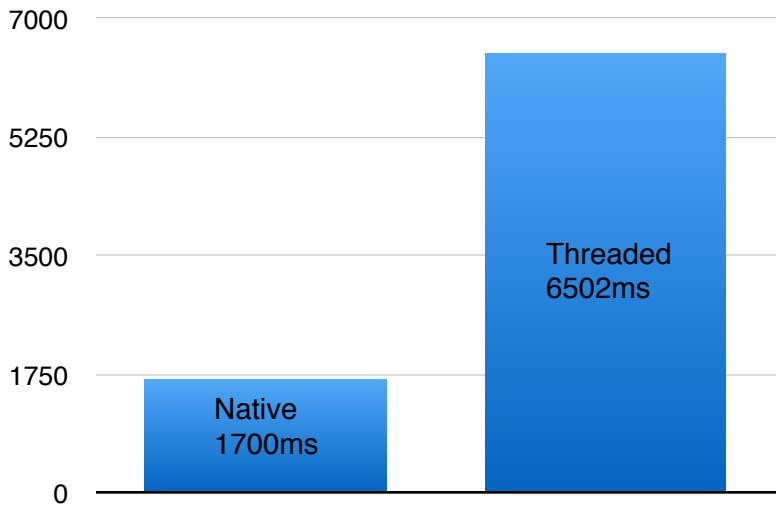
In general, benchmarks tend to rely heavily on integer arithmetic. In a niche mathematical language, this is alright. When looking at a general purpose programming language, this is a terrible metric. DeltaBlue is targeted as a garbage collection test. It allocates many object during the course of its run. It's a better analog than many benchmarks for object oriented languages.

I'm going to consider each optimization individually. Modtalk currently has three runtimes: bytecode interpreted in Smalltalk, threaded interpreter, and native. Support for bytecode interpreted has lagged behind the other two.

Our native runtime compiles the code for our methods to assembly. Instructions in our human readable IR are in many cases 1-to-1 with native instructions. Maintaining the abstract machine previously described forces some instructions (mostly stack instructions) to take a few assembly instructions. This runtime targets x86_64 native intel processors on OS X and linux.

Our threaded interpreter compiles our IR to an array of function pointers. Arguments to these instructions are placed in the code stream as well. Our threaded runtime is a close analog to our abstract machine. This runtime is meant for quick and easy porting to new platforms.

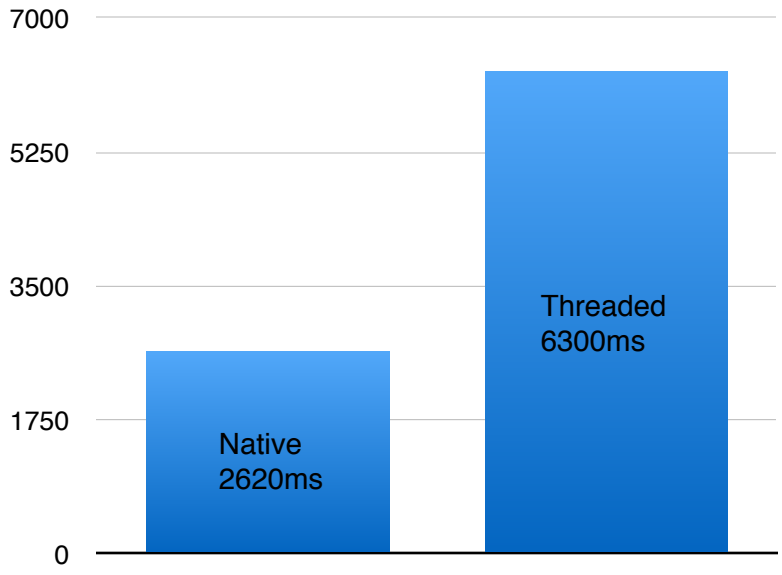
The effect of a specific optimization is different for each runtime. An unoptimized run is reported first for each target.



The effect of producing native code is apparent.

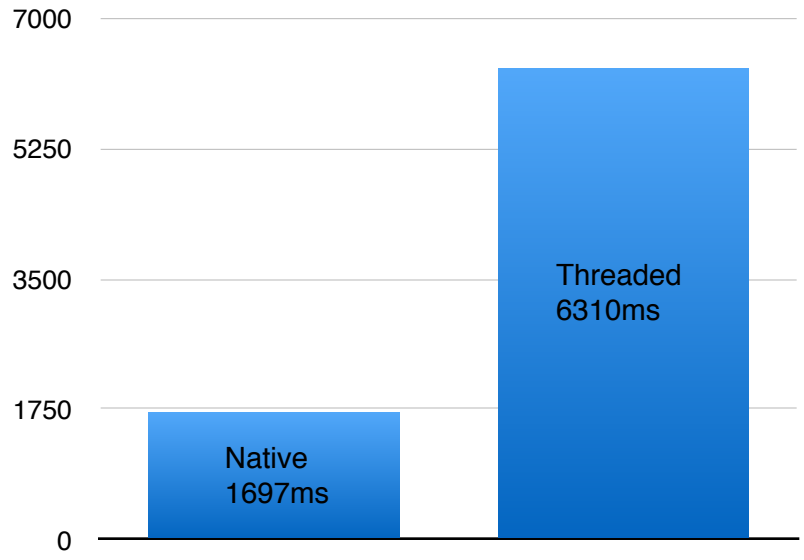
Peephole Optimization:

Peephole optimization gives results that are counter-intuitive at first glance. You'll notice we lost over 900ms. This leads me to believe pipelining might be coming into play here. In a CPU, there are instructions which are in the process of being decoded and executed. As each instruction is decoded and executed, more operations may be in the process of being interpreted. With a peephole optimizer, it is possible we are reaching branches sooner resulting in more places where the CPU's branch prediction is wrong.



Leaf Method Optimization:

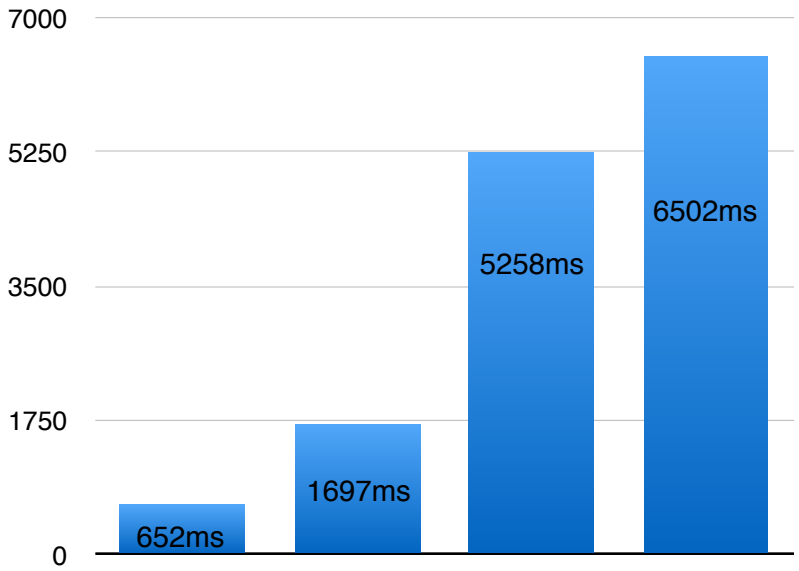
Like peephole optimization, leaf method optimization doesn't offer much performance for native. Threaded results are similar to that of peephole optimization.



Message Send Caching:

Monomorphic inline caching offers the largest performance boost for a single optimization.

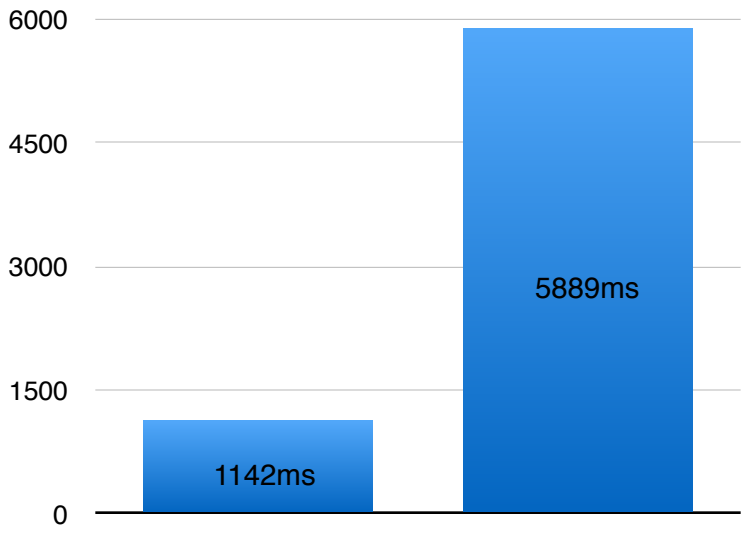
I've provided the unoptimized average in addition to the cached average so you can visually see the effect. Native run time was cut 1045 ms, or 62%. That is a huge win.



However, we still spend the majority of our program in method lookup. The time spent in lookup for threaded dropped about 35% but run time did not reflect this.

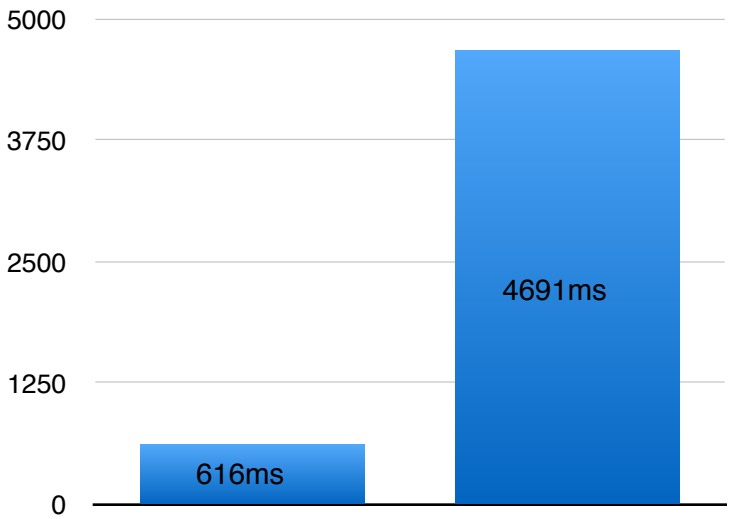
Inverted Lookup:

Inverted lookup offered similar performance improvements to caching. This suggests a significant portion of the messages sent were implemented in a class close to the type of the object.



We still drop 33% of the total run time for native. I expected a larger win for inverted lookup.

Caching + Inverted Lookup + Leaf Methods + Peephole Optimization:

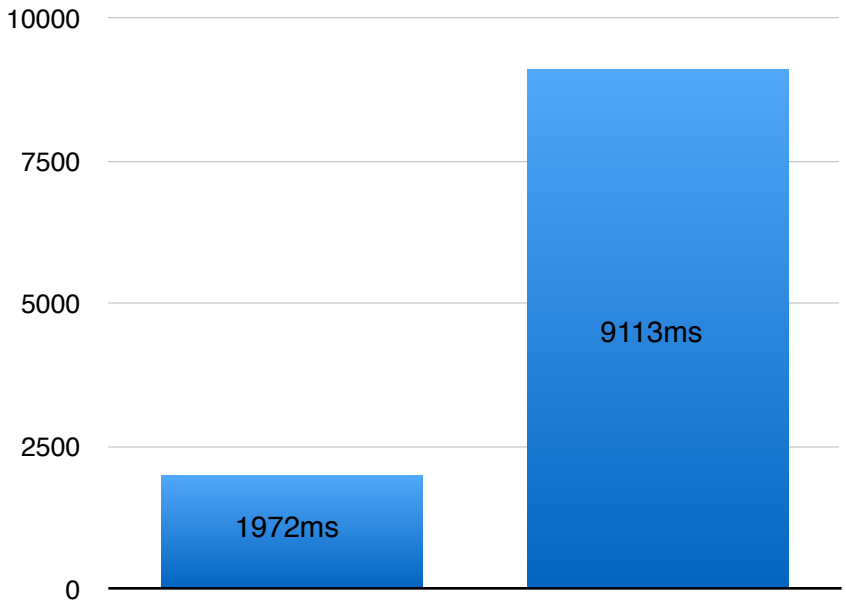


In both runtimes, enabling all optimizations resulted in the fastest running program (of my test set.) I've included all of the data I've collected in the appendix. For native, we saved 64% of the programs total run time! For threaded, we were able to save 28%.

Open Coding:

I need to preface this section. A few selectors are open coded in Modtalk. The selectors used to implement if, if/else, logical and and or, identity (==), and looping selectors. For this test, I disabled all open coding except open coding of the looping selectors. Looping without open coding results in a frame activation for each run through the loop. I did not want to worry about this specific issue.

For my open coding test results, I am only considering the previous case where all optimizations are turned on. In ANSITester, our Modtalk kernel tester, the runtime jumps from about a second to almost 13 without open coding. This is why I chose to look at our most optimized programs only.



In our otherwise most optimized situation, we are slower than the unoptimized state for each runtime. The effect is most apparent in native where the run time more than triples. We hope to remove the need for open coding of these selectors via more advanced optimizations. This would allow for polymorphic use of the restricted selectors and for the optimizations to apply to the code of the developer.

How we compare:

The big question that comes to mind at this point is how we relate to other languages. Two similar object-oriented languages (with implementations of DeltaBlue) are Java and Python.

A DeltaBlue run in Python averages 2370ms. Our threaded runtime is significantly slower across the board. However, our native runtime outperforms python in all cases (except when peephole optimization is the single optimization). Now, you might be thinking, python is slow…

In Java, DeltaBlue runs in 232ms on average. I'd like to note, in the first run, it takes around 460ms. Across runs, Java must store optimization information. In our most optimized runs, we are 2.65x slower than java. Considering Java's implementors have been working for two decades on optimizing Java, it's impressive for a semester of work.

My Thoughts:

As a more research and results oriented senior project, I chose to omit details of implementation in this paper. During my presentation, if desired (Randy), I can show code samples and explain in more detail how things are implemented.

Not knowing exactly what I was getting into in the beginning, I naively chose optimizations and forms for optimization I heard about. I wasn't able to finish all of the optimizations I set out to do but I've been able to develop ideas for continued work.

Now for my thoughts on grading. As it's more research based, as time went along, I've discovered new ideas I chose to explore in-place of those in my original presentation. My original grade scale follows.

Optimization	Points
Peephole optimizer	10
Leaf method optimization	10
Message Send Caching	30
CFG	30
SSA Form and Phi assignment	30
Control Flow Optimizations	30
Dead code elimination	10

To be more along the lines of what I would suggest if I were to resubmit my original proposal, I've created the following.

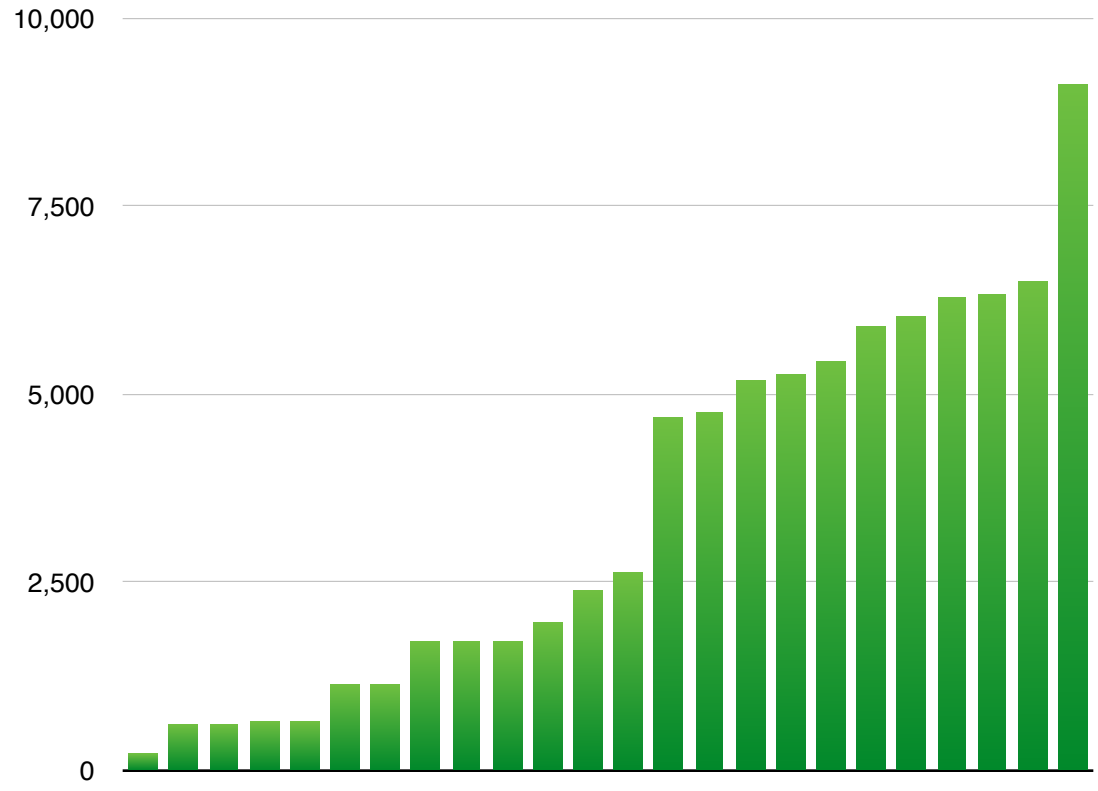
Optimization	Points	Claiming
Peephole optimizer	10	10
Leaf method optimization	10	10
Message Send Caching	30	30
Inverted Lookup	30	30
CFG	30	30
Open Coding	20	20

I'm claiming 130 on the new grade scale. In my new grading range, it follows I should receive an A. In my old scale, I should have received a C.

Lets average them together and call it a B.

Appendix:

Name	1	2	3	4	5	6	7	8	9	10		
java	460	208	207	205	206	204	208	204	206	208		232
nativeInvertedCachedPeepholeLeaf	616	616	617	614	619	616	609	620	619	617		616
nativeInvertedCached	619	620	621	612	617	621	620	609	618	617		617
nativeCachedPeepholeLeaf	620	618	667	617	621	611	663	670	617	614		632
nativeCached	664	615	663	669	667	665	614	669	668	622		652
nativeInvertedPeepholeLeaf	1130	1136	1132	1131	1125	1128	1139	1135	1127	1132		1132
nativeInverted	1177	1134	1175	1135	1135	1136	1136	1137	1130	1121		1142
nativePeepholeLeaf	1698	1696	1698	1697	1700	1694	1690	1690	1697	1689		1695
nativeLeaf	1693	1705	1708	1693	1695	1692	1689	1702	1697	1697		1697
native	1702	1696	1697	1701	1696	1699	1705	1705	1699	1704		1700
nativeInvertedCachedPeepholeLeafNoOpenCo	1954	2010	1956	1957	1961	2015	2004	1954	1954	1951		1972
python	2423	2373	2369	2370	2366	2360	2361	2353	2359	2361		2370
nativePeephole	2676	2628	2620	2635	2626	2611	2624	2621	2661	2620		2632
threadedInvertedCachedPeepholeLeaf	4712	4362	4846	4796	4372	4769	4788	4700	4787	4779		4691
threadedCachedPeepholeLeaf	4834	4782	4637	4825	4415	4721	4883	4845	4887	4873		4770
threadedInvertedCached	5143	5287	5124	5249	5186	5136	5237	5285	5186	5144		5198
threadedCached	5196	5389	5394	5075	5089	5503	5199	5086	5357	5289		5258
threadedInvertedPeepholeLeaf	5497	5571	5190	5500	5501	5549	5487	5193	5189	5508		5419
threadedInverted	5976	5965	5951	6009	5917	5594	6015	5907	5905	5648		5889
threadedPeepholeLeaf	6070	5913	6073	5922	6125	6065	6103	6122	6128	5757		6028
threadedPeephole	6271	6478	6281	6422	6282	6233	6443	6335	5967	6283		6300
threadedLeaf	5970	6324	6321	6383	6330	6408	6378	6368	6410	6209		6310
threaded	6167	6638	6378	6519	6525	6575	6533	6635	6539	6509		6502
threadedInvertedCachedPeepholeLeafNoOpen	9114	9214	9158	8613	9218	9189	9157	9253	9075	9141		9113



(The order in the graph corresponds to the order in the chart)