Compiler Paper

A minilang Java Compiler

Cole Cummins

Overview

The minilang compiler is a Java, class based compiler with eight major steps. Visitor pattern is upheld in each step of the process, with the three main visitors being the SemanticsVisitor, ASTtoCFGVisitor, and CFGtoLLVMVisitor:

Parsing

The MiniCompiler simply uses the already constructed AST from within the MiniCompiler module, adding no major components to the already functional parser.

Semantics Checking

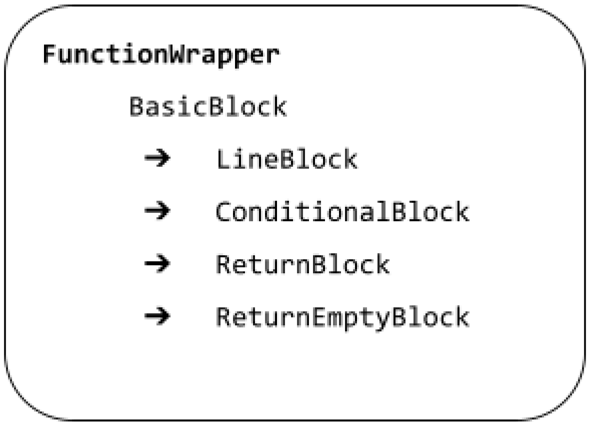
The MiniCompiler traverses the AST to perform a set of static semantics checking and type checking, as well as return equivalency for functions. It uses a symbol table as well as a function table to keep track of return and variable types, traversing the AST to printing out any errors directly to System.out. All type errors are non-breaking, meaning that the SemanticsVisitor will report all type errors in one pass, however assignment errors, struct field errors, and function param errors are all breaking. The SemanticsVisitor performs a return equivalence check using a return table, only counting BlockStatements as return equivalent if there is a ReturnStatement somewhere in them and ConditionalStatements as return equivalent if both branches are return equivalent.

CFG Construction

The MiniCompiler then traverses the AST to construct a Control Flow Graph consisting of BasicBlocks, a parent “abstract” class that holds a set of Statements, and later Instructions, a label, and a list of predecessor blocks. A BasicBlock can be extended to a ConditionalBlock, representing if statements and while loops, a LineBlock, representing a straight line code block, a ReturnBlock, representing a CFG exit, and a ReturnEmptyBlock, representing a CFG exit with no return value.

The BasicBlock body is then wrapped in a FunctionWrapper class, containing a function header and return type. The choice to do a class-based implementation arose from a desire for clarity in terms of debugging and intermediate representation. An easier implementation may have been to just create one single CFGNode class with a list of predecessors and a list of successors, however holding the blocks in lists lacks the distinct information gained from having separate fields for separate classes. The benefit of a class based implementation also allows for clean function overloading, allowing Java to do all the work of deciding which function header to call.

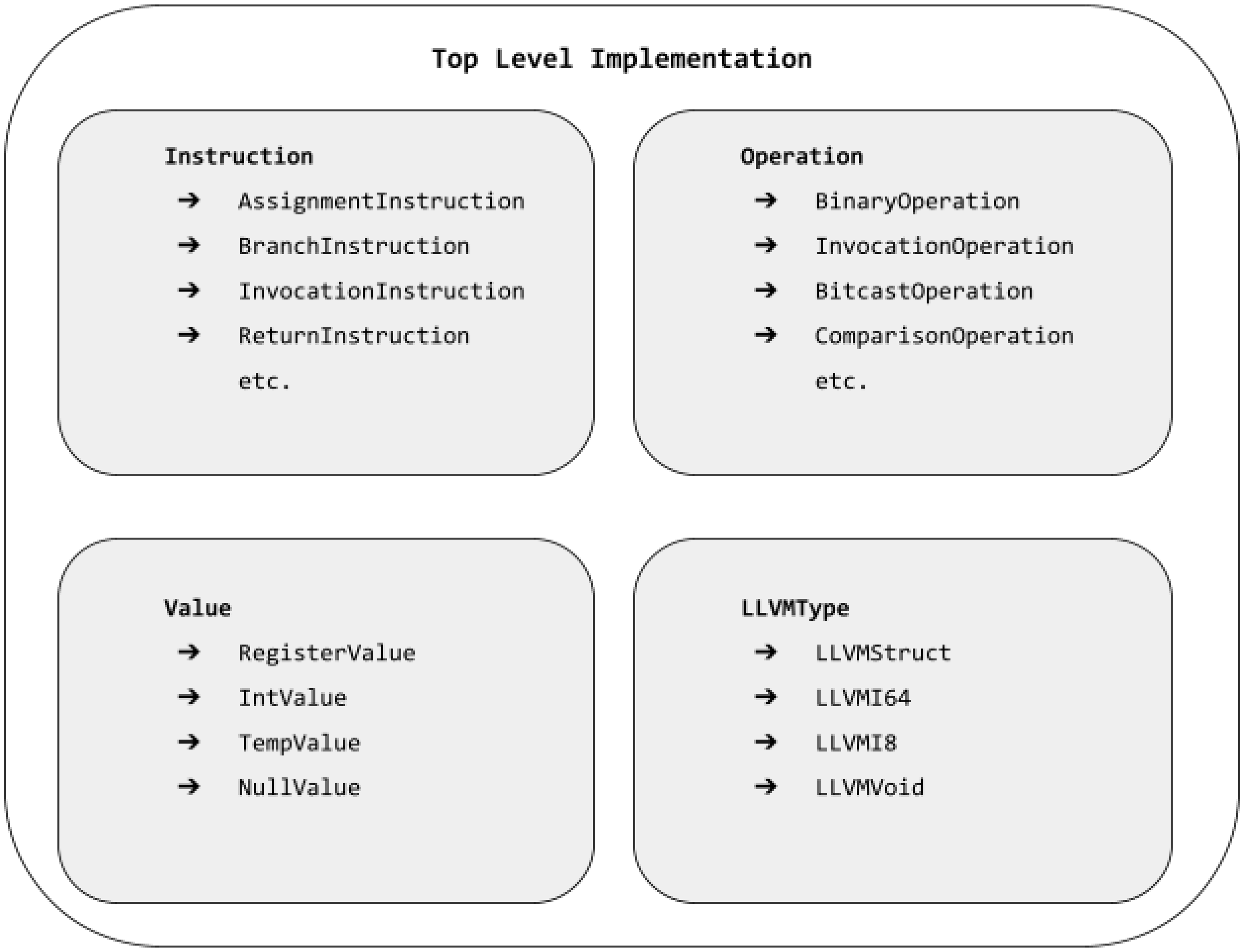
fig. FunctionWrapper with BasicBlocks



LLVM Generation

The MiniCompiler then traverses the CFG to insert LLVM code into the already generated BasicBlocks. The class representation of LLVM code consists of four main super-classes, Instructions, Operations, Values, and LLVMTypes. As LLVM code is inline whereas minilang code is expandable, converting from the AST to LLVM is a simple matter of flattening the code within each BasicBlock. The abstract-intermediate step of LLVM is exceptionally useful in this way, as we are able to conveniently flatten otherwise complex expressions into a simple list of Instructions and Operations. For the class-based LLVM, Instructions are to Statements as Operations are to Expressions, meaning the top-level code flow consists of a list of instructions. Phi Instructions are added to the BasicBlocks upon entry of each block, and as each predecessor enters the block, the local Values from that predecessor are saved in a Phi Instruction.

fig. LLVM Overview



```
ex. AssignmentInstruction
      <Value> = <Operation>
ex. BinaryOperation
      add <LLVMType> <Value>, <Value>
```

CFG Compaction

The first set of optimizations performed by the MiniCompiler is inline CFG compaction, reducing the number of BasicBlocks and therefore branch statements in the code. The CFG compaction is performed on blocks in the graph that have only one successor block, and the successor block has only one predecessor, therefore the connection between them is extraneous. While the first block in the CFG is kept to maintain as a pointer in the FunctionWrapper, the optimizer traverses the remainder of the CFG. The optimizer merges any two blocks that meet the requirements, corrects any phi instructions that may be changed due to graph rearranging, then finally clears any orphaned blocks that are no longer being referenced.

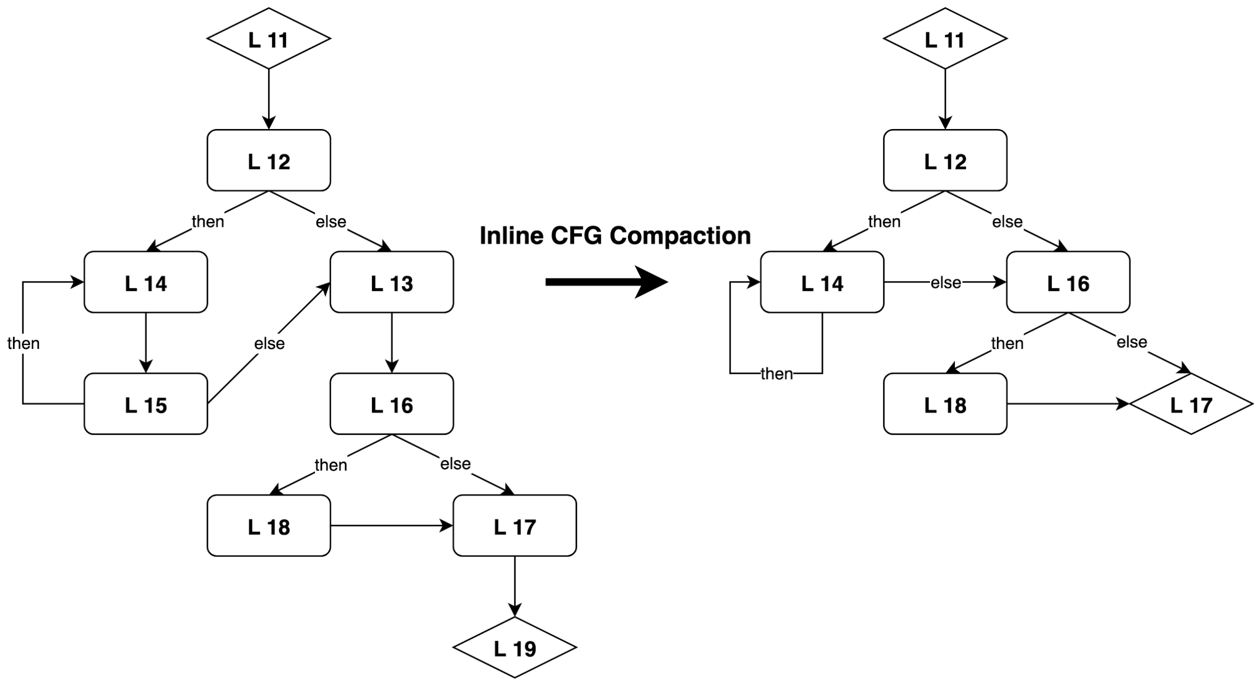
fig. example function, resulting CFG, and compacted CFG for comparison

```
fun calcMean (struct LinkedNums nums) int {
   int sum,num,mean;
   sum = 0;
   num = 0;
   mean = 0;

   while (nums != null) {
     num = num + 1;
     sum = sum + nums.num;
     nums = nums.next;
   }

   if (num != 0){
      mean = sum / num;
   }

   return mean;
}
```



LLVM Flattening

Once the CFG has been compacted, the notion of loops, branches, and nodes is no longer necessary for later steps of optimization and output, and flattening allows for easy traversal. The CFG is then converted into a list of LLVMBlocks, each consisting of a label and a list of instructions. The list of LLVMBlocks is then wrapped in an LLVMFunction, consisting of a list of blocks, a function header, and a return value.

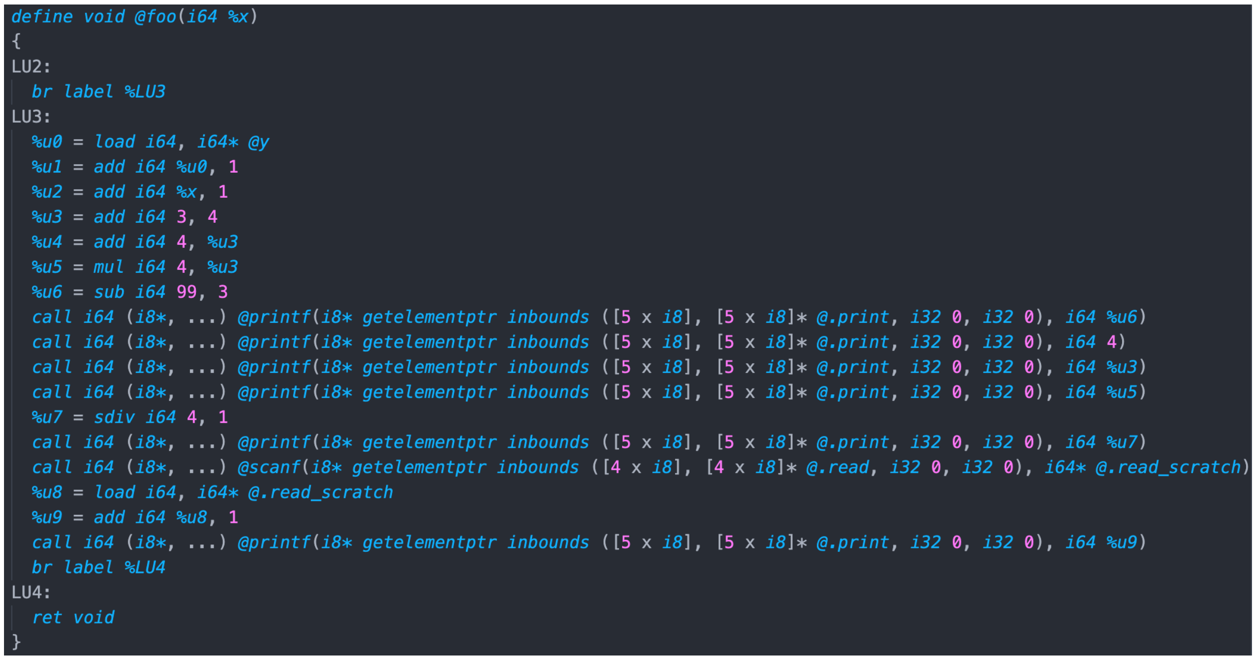
Constant Propagation

The next optimization performed by the MiniCompiler is constant propagation. Constant Propagation is performed by traversing a list of LLVMBlocks, generating a dictionary of Value strings to integer results that can be propagated, then finally traversing the list again to replace the propagated value strings with the integers. This process of fetching then propagating results is performed on the set of blocks until there are no more constants to be propagated. As we had flattened the CFG in the previous step, constant propagation is essentially trivialized, with the only hiccup being propagating through phi instructions. Phis that have similar constant values from all branches can be propagated just the same as other kinds of operations. While constant propagation by itself results in an almost negligible increase in performance, constant propagation in conjunction with useless code elimination, our next optimization, results in a noticeable increase in performance.

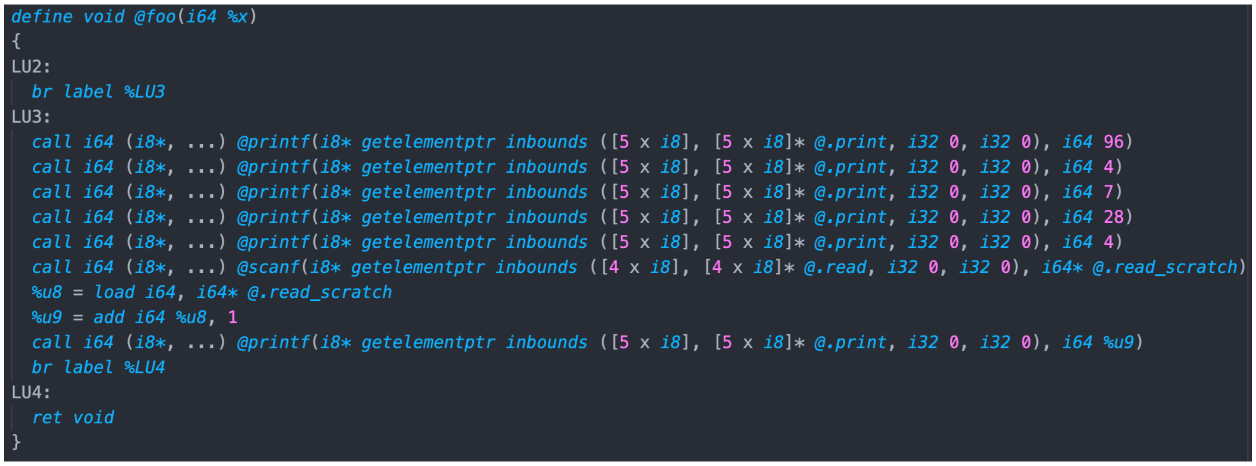
Useless Code Elimination

For our final optimization, the MiniCompiler performs useless code elimination, removing all assignment instructions that are unused. The compiler does this by first traversing through the list of LLVMBlocks, marking all registers as unused at first. The compiler then loops through all other instructions, including BinaryOperations, Comparisons, PhiInstructions etc. marking any registers located in Operations as used. Finally, all instructions with registers still labeled as unused at the end of the process are removed. This process is repeated until all registers are labeled as being used. Again, the process of flattening the CFG helps our optimizer out tremendously, reducing the computational complexity of the implementation. The only small issue with this implementation of useless code elimination is co-dependent PhiInstructions, or PhiInstructions whose only other use on each branch is each other, therefore the resulting registers are marked as used. Fortunately, this is not a common enough issue to create a major slowdown, and useless code elimination otherwise works well.

fig. Unoptimized SSA code optimized with constant propagation and useless code elimination



```
define void @foo(i64 %x)
{
LU2:
  br label %LU3
LU3:
  %u0 = load i64, i64* @y
  %u1 = add i64 %u0, 1
  %u2 = add i64 %x, 1
  %u3 = add i64 3, 4
  %u4 = add i64 4, %u3
  %u5 = mul i64 4, %u3
  %u6 = sub i64 99, 3
  call i64 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.print, i32 0, i32 0), i64 %u6)
  call i64 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.print, i32 0, i32 0), i64 4)
  call i64 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.print, i32 0, i32 0), i64 %u3)
  call i64 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.print, i32 0, i32 0), i64 %u5)
  %u7 = sdiv i64 4, 1
  call i64 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.print, i32 0, i32 0), i64 %u7)
  call i64 (i8*, ...) @scanf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.read, i32 0, i32 0), i64* @.read_scratch)
  %u8 = load i64, i64* @.read_scratch
  %u9 = add i64 %u8, 1
  call i64 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.print, i32 0, i32 0), i64 %u9)
  br label %LU4
LU4:
  ret void
}
```



```
define void @foo(i64 %x)
{
LU2:
  br label %LU3
LU3:
  call i64 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.print, i32 0, i32 0), i64 96)
  call i64 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.print, i32 0, i32 0), i64 4)
  call i64 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.print, i32 0, i32 0), i64 7)
  call i64 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.print, i32 0, i32 0), i64 28)
  call i64 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.print, i32 0, i32 0), i64 4)
  call i64 (i8*, ...) @scanf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.read, i32 0, i32 0), i64* @.read_scratch)
  %u8 = load i64, i64* @.read_scratch
  %u9 = add i64 %u8, 1
  call i64 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.print, i32 0, i32 0), i64 %u9)
  br label %LU4
LU4:
  ret void
}
```

Analysis

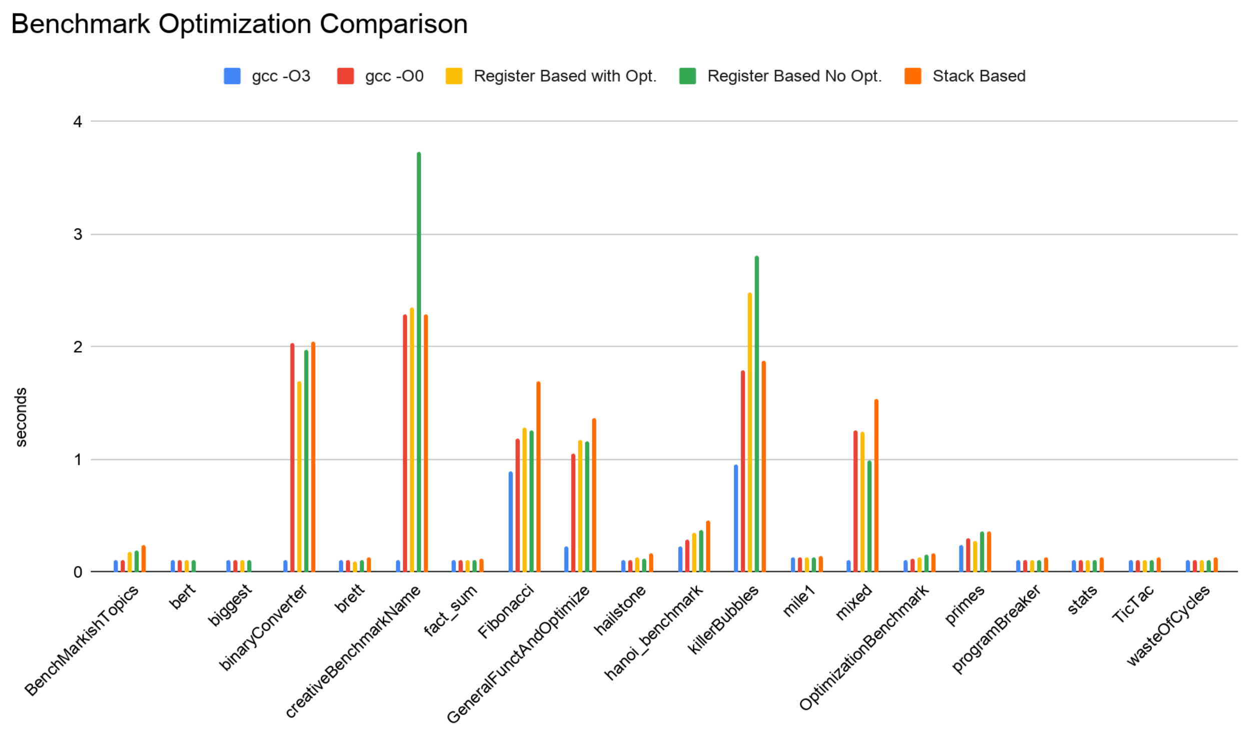
We see that, as expected, gcc with all optimizations completely blows the MiniCompiler out of the water, resulting in an on average ~350% increase over the stack based implementation of the MiniCompiler. The register based implementation is surprisingly only 103% more optimized than the stack based implementation, and given the reduction of number of instructions and removal of loads and stores of variables almost entirely, one would expect the register base approach to perform at least slightly better. Each timing shown in the chart represents only a single run of the given compiler and benchmark, and while I would have liked to perform an aggregate of multiple runs in my data, for whatever reason the python subprocess module seems to fork bomb itself after repeated calls to the same command in a loop. The python testing script to perform the analysis took a number of optimizations flags, ran a make clean, then make, wrote all benchmarks .ll files with the given optimizations, checked for clang syntax errors, then finally ran all passing a.out files, diffing expected outputs vs actual outputs.

fig. Table of all benchmarks timings

	gcc -O3	**gcc -O0**	**Register Based with Opt.**	**Register Based No Opt.**	**Stack Based**
BenchMarkishTopics	0.10804	0.11122	0.18265	0.19173	0.242304
bert	0.10898	0.10497	0.10932	0.11071	
biggest	0.1042	0.10516	0.1079	0.10602	
binaryConverter	0.1063	2.03004	1.7006	1.9763	2.051256
brett	0.11054	0.10353	0.10236	0.11279	0.135216
creativeBenchmarkName	0.10481	2.28831	2.35556	3.72897	2.293356
fact_sum	0.10473	0.11431	0.10521	0.11076	0.125172
Fibonacci	0.89169	1.18884	1.28129	1.25535	1.700928
GeneralFunctAndOptimize	0.23017	1.05766	1.1716	1.16309	1.366164
hailstone	0.11254	0.11073	0.13416	0.11771	0.167556

hanoi_benchmark	0.22935	0.2868	0.35559	0.37356	0.465672
killerBubbles	0.96199	1.7902	2.48088	2.80612	1.87374
mile1	0.13112	0.12898	0.12833	0.13438	0.147168
mixed	0.10691	1.2543	1.25342	0.999	1.538868
OptimizationBenchmark	0.10751	0.12219	0.12866	0.15581	0.170004
primes	0.23727	0.29886	0.2811	0.36757	0.36204
programBreaker	0.11289	0.10721	0.11064	0.10865	0.131748
stats	0.10591	0.10827	0.10358	0.10909	0.131544
TicTac	0.10885	0.10969	0.10782	0.1087	0.132288
wasteOfCycles	0.11228	0.10687	0.10421	0.1078	0.131832
Average Time (sec)	0.209804	0.576407	0.615244	0.7072055	0.731492
Average % Increase	348.60%	126.90%	118.80%	103.30%	~

fig. Chart of all benchmarks timings


Benchmark Optimization Comparison
gcc -O3
gcc -O0
Register Based with Opt.
Register Based No Opt.
Stack Based
seconds
BenchMarkishTopics
bert
biggest
binaryConverter
brett
creativeBenchmarkName
fact_sum
Fibonacci
GeneralFunctAndOptimize
hailstone
hanoi_benchmark
killerBubbles
mile1
mixed
OptimizationBenchmark
primes
programBreaker
stats
TicTac
wasteOfCycles