

Finch Version 12-01-03

This release includes a simple, sequential priority function optimizer. It also includes a parallel optimizer that relies on the PBS job queuing system and NFS; PBS distributes the computation, and the file system is used for communication.

Changes since Version 6-9-03

1. The Finch library now compiles with g++ version 3.2.
2. Added profiling tools to help analyze expressions.
3. The genetic programming library now uses the ramped-half-and-half method for initialization.
4. Added a parallel harness (pbs_harness) that uses the PBS job queuing system.
5. Fixed a bug in the sequential harness that prevented it from working with anything but the example “proxycompiler”.

Introduction

This document describes *finch*, a tool that automatically fine-tunes compiler heuristics. Rather than trying to fine-tune an entire heuristic, the system focuses on *priority functions*. Priority functions—sometimes called cost functions—simply assign weights to the options available to a compiler heuristic. For instance, list schedulers use a priority function to determine the order in which instructions in the ready worklist should be scheduled. Such priority functions often consider latencies and dependence heights of instructions in the worklist.

Priority functions have several nice properties that make them amenable to machine learning algorithms. First of all, it would be infeasible to learn an entire *valid* heuristic. Any heuristic that the learning algorithm discovers must enforce program correctness for *all* input programs. Priority functions allow the learning algorithm to concentrate on a very small portion of a compiler algorithm—among other things, the baseline algorithm enforces correctness. Nevertheless, even small changes to a priority function can drastically affect the quality of a compiler heuristic.

In addition, priority functions have a clear specification: we know that a priority function expects measurable program characteristics as input arguments, and we know that the function will return a cost¹.

Finch uses genetic programming to automatically search for priority functions. The algorithm starts by creating an initial *population* of automatically created random priority functions. In this release, the ramped-half-and-half method is used to create the initial population [3]. Then, until a user-defined number of iterations has been reached, the algorithm constantly updates the population of priority functions in a process akin to Darwinian evolution. By default the top individual is guaranteed survival in the following iteration. After every iteration, a new working population is created by crossing over and mutating priority functions in the current population. Please see [1] for a complete description of the algorithm employed by finch.

Building finch and Supporting Libraries

You can obtain finch's source code at <http://www.cag.lcs.mit.edu/metaopt>. The latest version of finch will be listed in the *Software Downloads* section of the webpage. Related papers and presentations are also available at the URL.

Before describing the build process, here is what needs to be built:

- [finch](#): this is the program that steers the search process.
- [libfinch.a](#): this library contains a genetic programming expression class, and a random number generator. This library will have to be linked in with your compiler.

¹ Constants can also be used as input arguments. For instance, when searching for an instruction scheduling priority function, the issue width of a processor might be useful.

Figure 1 shows the directory structure of the package. All of the source code is in the `src` directory. The `bench` directory contains a sample benchmark as well as the benchmark specification file that finch needs to compile and run it. The (GNU) makefiles in the source directory will build the binaries and libraries and copy them to `bin` and `lib` directories respectively. In addition, include files will be copied to the `include` directory.

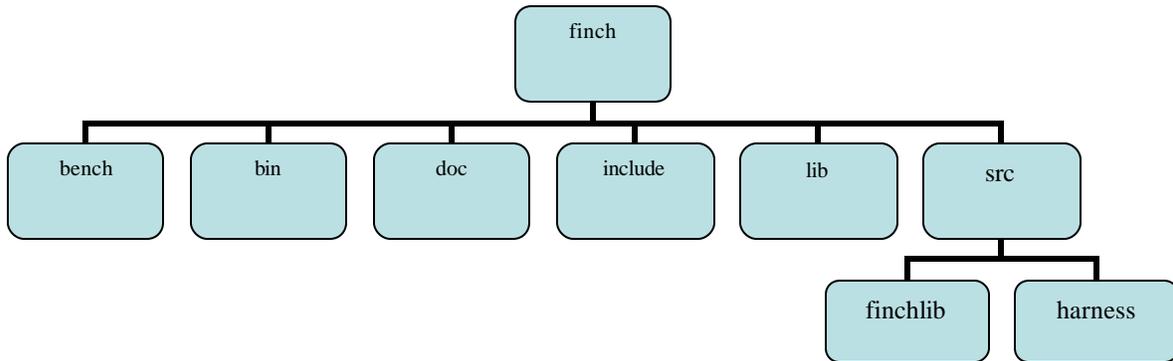


Figure 1. Directory structure of the distribution.

To build the system, you need to run GNU make in the `src` directory. The makefiles currently use the GNU g++ compiler, gcc compiler, flex++, and yacc. If you'd like to use the PBS, parallel system, you have to modify the Makefile in `src/harness` by including the following definition at the top of the Makefile: `HAS_PBS_SERVER = 1`.

Validated Systems

Finch has been validated on the following systems:

Architecture	Operating System
x86	Redhat Linux v7.2, v8.0
Itanium® I and II	Redhat Linux v7.2

The system was compiled with g++ version 2.95.3. It has been tested less rigorously with g++ version 3.2. Furthermore, lex and yacc are required to build the libraries. If you manage to get the system working on another system, please let us know (mstephen@mit.edu).

Plugging in the System

This section describes how to wrap finch around your compiler infrastructure. To start with, you must identify a candidate priority function to optimize. Presumably you already have a priority function in mind if you downloaded this document.

Finding a Suitable Priority Function

If you are searching for priority functions to optimize, here are a few simple, and perhaps obvious tips. First, grepping for “cost”, “priority”, “prioritize”, or “sort” in your compiler’s source directory will likely reveal several candidates.

Optimizing a priority function is a time-intensive process. For the benchmarks that we surveyed in [1], our native sequential system took anywhere from 1 to 3 days per benchmark in the training set (running on Itanium). Thus, a quick check to see if you have found a priority function that is amenable to our system could save a lot of wasted cycles. Fortunately, simply replacing your compiler’s priority function with a constant value (or with a call to a random number generator) can tell you a lot about the importance of the function. If the constant priority function noticeably affects the performance of binaries created by your compiler, running our system is a noble use of computer cycles.

Is finch Suitable for your System?

Finch requires feedback from the system to assign fitnesses to expressions. If this feedback is noisy—as is the case on non-simulated systems—then finch may not be up to task. As a prerequisite, make sure that back-to-back-to-back runs of the same application return *similar* measurements (regardless of the metric that you are measuring). For instance, on an Itanium® machine running Linux, most benchmarks have a fairly consistent running time, making finch an appropriate optimization tool. On the other hand, the dynamic nature of the Pentium® 4 processor (with SMT enabled) makes back-to-back executions very inconsistent. As a result, finch is not appropriate for optimizing compilers that target Pentium 4’s.

As a rule of thumb, differences of less than a couple of percent are tolerable; anything more, may provide problems—unless of course the returns of using our system far outweigh noise margins.

Hooking in finch

There are three general library calls that must be made: a call for initialization, a call for evaluation of the priority function, and finally a call for finalization. Finch uses the file system to communicate between the harness (the finch executable) and the compiler that you wish to optimize. Finch expects you to set the \$FINCH_HOME environment variable. \$FINCH_HOME should point to a directory that will be used for communication between finch and the compiler.

The following numbered list describes Finch’s communication protocol. **Please note that it is not necessary to understand the details of the protocol in order to use finch.**

1. Finch initially removes the `$FINCH_HOME/config` file. This file is used to specify the number and types of arguments that the learning algorithm should consider when trying to learn a priority function.
2. Finch then invokes the compiler. If the library calls are made correctly—which will be described later—the system will create a new `$FINCH_HOME/config` file. The numbers in the config file tell finch vital information about input arguments to the priority function—namely how many double-type arguments and how many Boolean-type arguments finch should consider.
3. Finch will then use the information in this file to create an initial population of random priority functions.
4. The harness will then write the priority functions in the population to disk. For the sequential version the priority functions are written one at a time to the file called `$FINCH_HOME/evaluate/pfun.l`. Note that the evaluate directory is created by the harness. For the parallel version, the priority functions are written to `$FINCH_HOME/evaluate/<bench_num>/<expr_num>/pfun.l`.
5. As described later, an input file is used to specify how to run the benchmarks in your *training* set. Finch uses this file to compile, link, and run the benchmarks. When an appropriately modified compiler is run, it will use the priority function that finch wrote to disk.
6. The sequential harness expects a “fitness” file to be placed in `$FINCH_HOME/evaluate/fitness`. The parallel version looks for the fitness file in `$FINCH_HOME/evaluate/<bench_num>/<expr_num>/evaluate/fitness`. The user is responsible for communicating this information back to finch. For the Itanium results presented in [1], we simply used the time command when we ran the benchmarks: `“/usr/bin/time –output=fitness –format=%U”`. By default, the sequential version of finch compiles and runs benchmarks in the `$FINCH_HOME/evaluate` directory, so the time command shown above would create a file named `$FINCH_HOME/evaluate/fitness` with the total user time required to run the benchmark in the file. **Execution time is not the only metric you can use, but please note that finch considers lower fitnesses to be better.**

The following sections describe the available interfaces to the finch library. They describe files that must be included, as well as the necessary library calls.

C++ Interface

All of the functions in this section can be found in `finch_interface.h`, and their corresponding definitions are in `finch_interface.cc`.

The first function that has to be called is shown below:

```
void FINCH_initialize_lib()
```

It initializes the finch library. It should only be called once for every invocation of the compiler pass that you are trying to optimize. The function opens and parses priority functions created by the harness. This call should generally be made with your compiler pass’s other initialization code.

The initialization call dynamically creates a new priority function. This priority function will replace the static priority function that you are trying to replace. Instead of using the “stock” priority function that came with your compiler, use a call to one of the following functions:

```
double FINCH_evaluate_real_expr(uint32 num_dbls, uint32 num_bools, ...)
bool   FINCH_evaluate_binary_expr(uint32 num_dbls, uint32 num_bools, ...)
```

Both of these functions use varargs. The first and second arguments specify the number of double-valued and Boolean-valued parameters respectively that the priority function should consider. All of the double-valued arguments come next, followed by the Boolean valued arguments. Thus, an example usage of the library call is:

```
p = FINCH_evaluate_real_expr(2, 1,
                             num_ops, num_branches,
                             is_predictable);
```

Here `p` is the double-typed priority value returned by the function call, `num_ops` and `num_branches` are the two double-typed arguments, and `is_predictable` is the one Boolean-typed argument. Other similar calls that do not use varargs are listed in [finch_interface.h](#).

Finally, be sure to clean up the memory that the finch library created. A call to the following function will do this:

```
void FINCH_finalize_lib()
```

This should be placed with your pass’s other cleanup code.

Linking in the Libraries

Now that your compiler contains calls to the finch library, you must link your compiler with `$FROOT/lib/$MACHTYPE/libfinch.a`, where `$FROOT` is the location of the base of the distribution shown in Figure 2.

Creating a Training Set

This section describes how to create a training set. As mentioned above, finch reads an input file that specifies which applications are in the training set, as well as how to compile and run each of them. The best way to describe the input file is by example:

```
newbench {
    $time      = "/usr/bin/time --output=fitness --format=%U ";
    $benchmark = "aps";
    $rootdir   = "/home/bench/$benchmark";
    $srcs      = "$rootdir/APS.f";
    $clean     = "rm -f *";
    setup      = "$clean", "ln -s $rootdir/API9 .";
```

```

profile      = "";
compile     = "gcc -o $bench.o -c -O3 $srcs";
link        = "gcc -o $bench $bench.o";
run         = "$time $bench";
check       = "grep -w VALID APV";
clean       = "$clean";
}

```

This example assumes that you have already modified `gcc` by replacing one of its priority functions as described in the last section. The `newbench` keyword specifies that you are adding a new benchmark to the training set. Thus, if you were to create a training set that consisted of four benchmarks, your training set specification file would have ten such entries:

```

newbench {
    ...
}
newbench {
    ...
}
newbench {
    ...
}
newbench {
    ...
}

```

Finch uses the `C system` command to execute the specifications. More specifically, `finch` executes the commands in the following order:

1. `setup`: commands listed in this phase can be used to copy any files into the `evaluate` directory where the benchmark will be compiled. Notice how this phase specifies two comma separated commands. In any phase, any number of comma separated commands can be executed.
2. `profile`: this phase can be used to profile your code. This phase is empty, and thus, we did not even have to specify it.
3. `compile`: commands listed in this phase actually compile the benchmark.
4. `link`: If not done in the previous phase, this phase can be used to link the object files.
5. `run`: this phase runs the benchmark. Notice how `time` is used to record the fitness of the benchmark.
6. `check`: this phase is used to check the validity of the benchmark. In theory, any priority function should be legal, but sometimes a particular priority function will break the compiler algorithm. Thus, `finch` may also be useful for correctness checking.
7. `clean`: this phase cleans up any files that any of the phases might have created. It is not strictly necessary however, since `finch` does this anyway.

Note that if any of the calls to `system` return non-zero, `finch` will halt with an error message that reports the offending command.

It is sometimes useful to create variables that represent strings such as commonly referenced directories. For instance the above specification created the variables named `$rootdir` and `$bench` that were used in the phase specifications. These variables are expanded within the aforementioned phase descriptions. **Note: You can specify shell environment variables such as `$PWD`. Variables defined in a `newbench` clause are not exported to the shell, and such definitions supercede environment variables.**

Running the System

With the library calls instantiated properly, and a benchmark specification file created, it is time to actually run the system. This section describes how to do this.

Setting Environment Variables

Set the `$FINCH_HOME` environment variable to point to any directory. **Note: Finch deletes some files such as `$FINCH_HOME/config`, `$FINCH_HOME/evaluate`, `$FINCH_HOME/srate`, and `$FINCH_HOME/notes`.** The `finch` binary—as well as the `finch` library—relies on this environment variable. You may also wish to include the location of the `finch` binary in your path:

`$FROOT/bin/$MACHTYPE/`,

where `$FROOT` is the root of the Finch source tree.

Finch Usage

Finch has the following usage: `finch [options] <benchmark specification file>`

The options that `finch` accepts are described in the following table:

Option	Argument	Default	Description
<code>-classifier</code>		Off	Creates classification (binary) priority functions. For some problems you may wish to create Boolean- rather than double-typed priorities.
<code>-dsssize</code>	Uint	6	Specifies size of the subset of the training set that <code>finch</code> should use. This dramatically increases execution time. Note that this value changes dynamically. If you specify a subset size of 6, <code>finch</code> will probabilistically chose a subset of size 5, 6, or 7. The minimum of the number of benchmarks and the <code>dss</code> set size will be used.
<code>-elitist</code>	uint	1	Specifies the number of top expressions that are guaranteed survival in the next generation.
<code>-initial</code>	char*	""	Specifies the filename of an initial population. At the beginning of each generation, the population is written to a file named <code>_state_x</code> , where <code>x</code> is the

			generation number. This file can be read directly into finch.
-gens	uint	100	Specifies the number of generations to run for.
-gensize	uint	180	Places a cap on the size of expressions that are generated.
-help			Displays the usage.
-height	uint	3	Specifies initial expression heights.
-mutate	uint	8	Percentage of the population that should be mutated after each generation
-mortality	uint	22	Percentage of the population that is replaced using crossover after each generation. By default this number is low because of dynamic subset selection; we may not want to replace an expression before it has been evaluated on several benchmarks.
-popsize	Uint	400	Specifies the population size.
-quantize	Uint		Keeps a set of all fitnesses that have been computed thus far. Useful to gauge the fitness landscape.
-savepop	char*	""	Saves the population to a file at the end of each generation. The string specified with this option is used as the base of the file's name (e.g., _pop12).
-seed	uint	42	Seeds the random number generator.
-sunset	uint	1	If an expression is the best expression in any generation, it is marked as the best expression. The sunset parameter specifies how long it should be retained before being a candidate for replacement. This option is useful for dynamic subset selection (DSS) where some expressions may perform well on certain subsets.
-tournament	uint	7	Specifies the tournament size for selection. Tournament selection is a standard genetic programming technique for selecting the fittest individual. The fittest of N randomly chosen expressions is selected as the fittest expression, where N is the tournament size.
-usepbs		Off	Uses the PBS job queuing system to parallelize computation. This option only works if PBS is installed on your system.

Restarting the System

The system relies on the file system. If at any point the file system becomes unstable, finch may break. Because of this, finch constantly writes its state to disk. Once the file system has been restored, finch can be restarted with the last known state. Finch can be started with the last known state by using the `-initial` option. In other words, if the last known state is `_state5`, use `"-initial_state5"`. In addition to this, after every generation, finch writes the DSS state to disk to a file called `_dssstate`. This file is a modified version of the benchmark specification file that includes up-to-date state used by dynamic subset selection.

Interpreting Results

After every generation the best expression from that generation is written to `best.l`. In addition, the `_state*` files contain the entire population of expressions for a given generation.

Let's say your call to evaluate the current priority function looks like this:

```
p = FINCH_evaluate_real_expr(2, 1,
                             num_ops, num_branches,
                             is_predictable);
```

The solutions that finch comes up with will look something like the following:

```
(cmul (barg $b0) (dconst 1.3) (darg $d1))
```

Here “(barg \$b0)” simply represents the 0th binary argument, in this case `is_predictable`. Likewise “(darg \$d1)” represents the 1st double argument, in this case `num_branches`. Finch can create a double argument that corresponds to any of the double arguments in the evaluation library call. Likewise, any Boolean argument in the call might be created by finch. “(dconst 1.3)” is the constant 1.3, and `cmul` is the conditional multiply opcode. Thus, the above expression represents the following computation:

```
if is_predictable then
  return 1.3*num_branches
else
  return num_branches
```

All of the opcodes used in this distribution are described in [1].

Using fanalyzer

`fanalyzer` is a tool that we developed to help analyze priority functions created by Finch. It relies on [dot](#), a graphing utility that draws directed graphs [4]. `fanalyzer` essentially profiles a given priority function on a set of input benchmarks. The input benchmark specification is the same as that required by Finch.

The usage is as follows:

```
fanalyzer [options] -expr <expression_file> -benches <benchmarks>
```

options:

```
-dotfname    the dot output file name.  
-help        print help information.  
-srate       the sample rate [1..100].
```

`fanalyzer` creates a DOT graph description of the priority function. Each node in the graph is annotated with its variance during compilation of the benchmarks in the benchmark specification file. Furthermore, the grayscale color of each node reflects the covariance of each subexpression with the root node (the whole expression). The darker the node is, the higher the covariance between it and the root node. Thus, the root node will be black, and constant nodes will be white; the colors highlight the most important subexpressions. Figure 2 shows an example dot-generated graph.

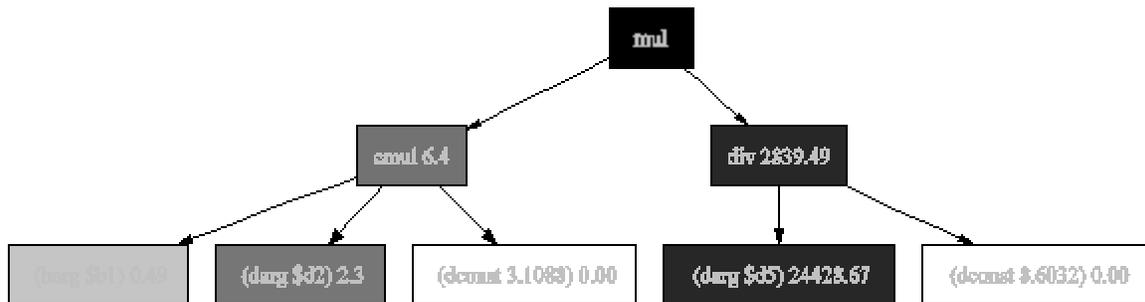


Figure 2. An example “dot” graph produced by `fanalyzer`.

The following call to `fanalyzer` will sample twenty percent of the evaluations of the priority function (`best.l`) over the benchmarks specified (those in `bench.spec`).

```
fanalyzer -srate 20 -expr best.l -benches bench.spec
```

Putting it all Together: A Small Example

We have put together a small example to illustrate Finch’s usage. To begin with, we have created a fake compiler called the `proxycompiler`. The source code for `proxycomp` is in `$FROOT/src/proxycomp`, where `$FROOT` is the root of the Finch distribution. The `proxycomp` illustrates the three library calls that must be made:

```
FINCH_initialize_lib() at the beginning of the compiler pass,  
FINCH_evaluate_real_expr() in place of the compiler’s priority function, and  
FINCH_finalize_lib() before the pass exits.
```

Before continuing, it is important to note that the proxycomp differs from a normal usage of Finch in several ways. First, the proxycomp assigns a fitness to the priority function that it uses. Usually, the compiler would simply compile the code using the specified priority function, and the fitness – or value – of the priority function would not be determined until the resulting executable was run. Secondly, the proxycomp does not compile code. It assigns a fitness to the given priority function according to how closely it matches the arithmetic expression

$$f = \begin{cases} i \cdot j - j^2 + 2.0 & \text{if } b = 0 \\ i^2 & \text{if } b = 1 \end{cases}$$

In other words, the proxycompiler simply uses the sum of squared errors to regression match.

Go to `$FROOT/testdir`. Here you'll see a file called `bench.spec`, the contents of which are repeated here:

```
newbench {
  $bindir      = "$PWD/../../bin/$MACHTYPE";
  $bench       = "$bindir/proxycomp";
  $clean       = "rm -f *";
  setup        = "";
  compile      = "$bench";
  link         = "";
  run          = "";
  check        = "";
  clean        = "$clean";
}
```

This file is the training set specification. In this case, there is only one “benchmark” in the training set. A normal benchmark specification file might contain several `newbench` clauses, each of which would specify how to compile, link, run, and check correctness of a particular benchmark. The Makefiles that you used to build `finch` and the `proxycomp` put the executables in the `$FROOT/bin/$MACHTYPE` directory. Thus, the `$bindir` variable points to that directory.

Now, set the `$FINCH_HOME` environment variable: `setenv FINCH_HOME `pwd``

Now, issue the command:

```
finch -gens 50 -popsize 750 -height 4 -gensize 60 -mortality 99
bench.spec
```

This command starts the finch harness. This will run Finch for fifty generations which may take a while, so be patient. You should see the fitness of the best expression remain the same or decrease after every generation. Finch will create 750 expressions, ranging from height one to height 4. Whenever Finch creates a new expression that exceeds 60 nodes, it discards the expression. The `-mortality` flag specifies that 99% of the

population will be replaced after each generation (though regardless of this setting, the best expression is always kept).

You should also see that Finch creates a file called `_state_X`, where X is the generation number that corresponds to the expressions in the file. Each of the expressions in one of these state files is annotated with its fitness on the problem.

Finch scores the fitnesses of each of the expressions in the population relative to the 0th expression in the *last* population. Thus an entry from a state file will look something like the following:

```
// 3 : [0.741433,1]
[ 0:1975087.856500 ]
(tern (barg $b0) (darg $d3) (darg $d3))
```

The first line tells us that this expression is the third expression in the population. Furthermore, it tells us that this expression is 1/0.74 times better than the 0th expression from the last population. This line also tells us that this expression is one generation old.

The second line is the raw fitness of the expression. The [`<bench num>` :`<fitness>`] means that the fitness for benchmark number `<bench num>` is `<fitness>`.

Finally, the priority function itself is on the third line.

I think you have all the information you need to start using finch. Please refer to the bench directory in the distribution for an example benchmark specification file.

Bibliography

- [1] M. Stephenson, M. Martin, U. O'Reilly, and S. Amarasinghe. **Meta Optimization: Improving Compiler Heuristics with Machine Learning**. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.
- [2] C. Gathercole. **An Investigation of Supervised Learning in Genetic Programming**. *PhD thesis, University of Edinburgh*, 1998.
- [3] J. Koza. **Genetic Programming: On the Programming of Computers by means of Natural Selection**. *MIT Press*, 1992.
- [4] AT&T Research. <http://www.research.att.com/sw/tools/graphviz/refs.html>