

Active Harmony User's Guide

4.6.0

Generated by Doxygen 1.8.10

Contents

1	Welcome	3
2	Introduction	4
2.1	Motivating Example	4
2.2	Tuning Variables	4
2.3	Search Spaces	4
2.4	The Feedback Loop	5
2.5	Tuning Session	5
3	Getting Started	8
3.1	Downloading the Source	8
3.2	Installation	8
3.2.1	Building the Documentation	9
3.3	Testing the Installation	9
3.3.1	Standalone Mode	9
3.3.2	Server Mode	10
3.4	Exploring Further	10
4	Harmony Applications	12
4.1	Harmony Information Utility	12
4.2	Harmony Server	13
4.3	Tuna: The Command-line Tuning Shell	15
4.4	Example Applications	16
4.4.1	Synth: A Synthetic Function Test Application	17
5	Configuring Active Harmony	19
5.1	Active Harmony Session Configuration System	19
5.2	Environment Variables	20
6	Plug-ins	21
6.1	Search Strategies	21
6.1.1	Exhaustive (exhaustive.so)	21
6.1.2	Random (random.so)	21
6.1.3	Nelder-Mead (nm.so)	21
6.1.4	Parallel Rank Order (pro.so)	21
6.2	Processing Layers	22
6.2.1	Aggregator (agg.so)	22

6.2.2	Point Caching/Replay layer (cache.so)	22
6.2.3	Code Server (codegen.so)	22
6.2.4	Input Grouping (group.so)	22
6.2.5	Point Logger (log.so)	23
6.2.6	Omega Constraint (constraint.so)	23
6.2.7	TAUdb Interface (TAUdb.so)	23
6.2.8	XML Writer (xmlWriter.so)	24
7	Coding Examples	25
7.1	Establishing Communication with a Tuning Session	25
7.2	Defining and Starting a New Search Task	25
7.3	Advanced Search Task Configuration	26
7.4	Interacting with a Search Task	27
8	Module Index	29
8.1	Modules	29
9	File Index	30
9.1	File List	30
10	Module Documentation	31
10.1	Harmony Descriptor Management Functions	31
10.1.1	Detailed Description	31
10.1.2	Function Documentation	31
10.2	Tuning Search Definition Functions	34
10.2.1	Detailed Description	34
10.2.2	Function Documentation	34
10.3	Tuning Search Task Control Functions	39
10.3.1	Detailed Description	39
10.3.2	Function Documentation	39
10.4	Search Task Interaction Functions	42
10.4.1	Detailed Description	42
10.4.2	Function Documentation	42
10.5	Harmony Error Reporting Functions	47
10.5.1	Detailed Description	47
10.5.2	Function Documentation	47
11	File Documentation	48
11.1	hclient.h File Reference	48

11.1.1 Detailed Description 49

Index **51**

1 Welcome

Welcome to the Active Harmony User's Guide. This manual is designed for users new to auto-tuning, and provides the information necessary for incorporating auto-tuning into a new project. It describes in detail the terms and concepts involved in auto-tuning, how they are implemented in the Active Harmony framework, and how to incorporate them into your client application.

2 Introduction

Auto-tuning refers to the automated search for values to improve the performance of a target application. In this case, performance is an abstract term used to represent a measurable quantity. A common example of performance for auto-tuning is time, where the goal is to minimize execution time. Other possible examples include minimizing power usage or maximizing floating-point operations per second. In general, the Active Harmony framework seeks to minimize performance values and handles maximization via negation.

In order for auto-tuning to be effective, a set of parameters must exist that affect the target application's performance. A simple example is thread count for OpenMP applications, as changing the number of threads involved in executing a parallel program will certainly have an affect on run-time. Target application parameters are represented within Active Harmony as **tuning variables**.

2.1 Motivating Example

As a motivating example, consider the study conducted by Tiwari et al. on optimizing scientific codes. Applications written for scientific computing typically spend the bulk of their execution time in compute-heavy loops. These loops are prime candidates for a compiler optimization known as loop unrolling and tiling. Modifying the number of times a compiler unrolls or tiles a loop results in a distinct binary with different performance properties. However, the optimal number of times to unroll or tile any given loop is virtually impossible to know at compile time since it is dependent on target architecture. Compiling a priori with all possible unrolling and tiling values is prohibitively expensive, but using a sub-optimal binary also wastes valuable compute cycles.

Tiwari solved this problem by allowing an auto-tuner to search for optimal loop unrolling and tiling values. Using this approach, only a small fraction of the possible code variants are built, and an optimal (or near-optimal) version of the code is used for the majority of the execution.

We refer to this example throughout the rest of this manual.

2.2 Tuning Variables

Tuning variables in Active Harmony require a distinct name, and must be declared as one of the following three types:

- **INT** (Integer numbers)
This value range is constrained by a minimum (m), maximum (M), and a stepping value (s) where $m \leq M$ and $s > 0$.
- **REAL** (Real numbers)
This value range is constrained by a minimum (m), maximum (M), and a stepping value (s) where $m \leq M$ and $s > 0$.
- **ENUM** (Enumerated strings)
This value range is constrained by an explicit list of valid values.

2.3 Search Spaces

Each tuning variable may be seen as a 1-dimensional range of values that are valid for a given application parameter. A collection of N tuning variables then creates an N -dimensional Euclidean space. We refer to this in Active Harmony as the **search space**. Points within the space represent a single possible configuration for the target application. For instance, if the following search space is defined:

Variable Name	Bounds
tile	m=1, M=4, s=1
unroll	m=2, M=16, s=2
compiler	list={"GCC", "Intel", "PGI"}

The search will then be conducted within a 3-dimensional space, and (4, 12, "Intel") would be a valid point within that space.

2.4 The Feedback Loop

Active Harmony works in tandem with a target application by manipulating tuning variables and observing the resulting performance. This creates a feedback loop where Active Harmony uses each incoming performance observation to further refine its search for optimal values. In Active Harmony, the tuning element is called the **tuning session** and the target application is called the **client**. They are connected in the following manner:

1. The tuning session **generates** a new point.
2. The client **fetches** the point, and operates for some period of time while measuring performance.
3. The client **reports** the performance value back to the tuning session.
4. The tuning session **analyzes** the report to guide its search for optimal points.
5. Repeat until the search converges.

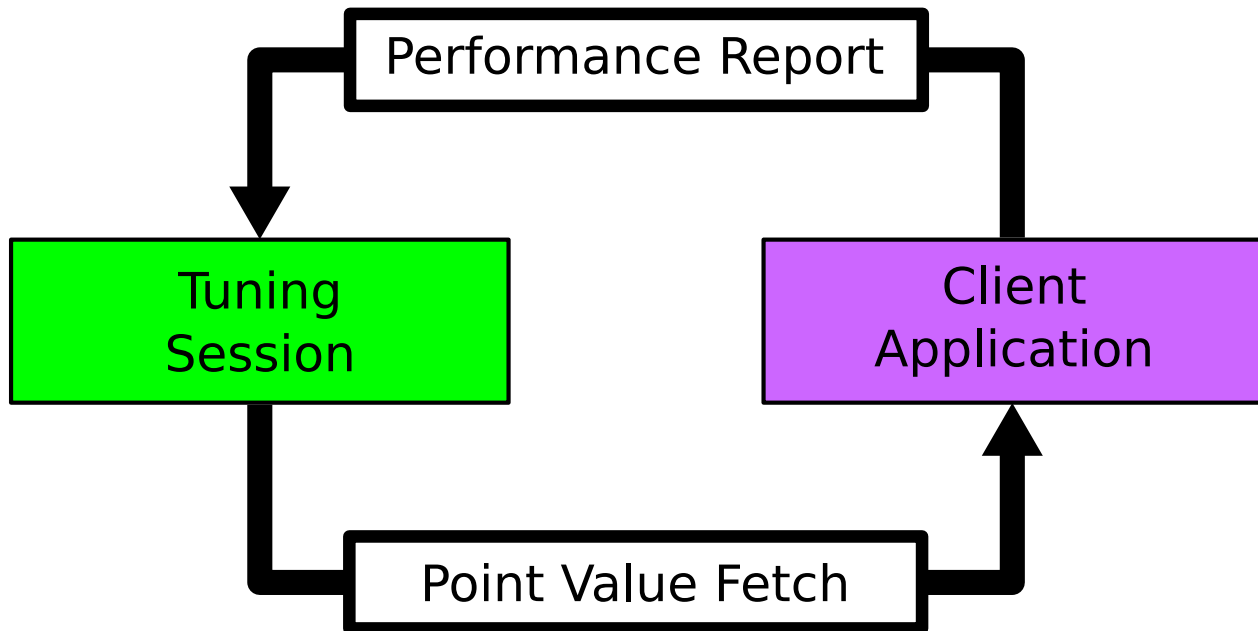


Figure 1: The generalized auto-tuning feedback loop.

2.5 Tuning Session

Conceptually, the tuning session is responsible for generating candidate points. Active Harmony divides this task into two key abstractions, the **search strategy** and the **processing layers**.

The search strategy determines how new candidate points are selected from the search space. For instance, one strategy might be to ignore all performance reports and simply return a random point. Several search strategies come bundled with Active Harmony, each with different properties to support a wide range of client applications. Note that search strategies operate purely at a numeric level by mapping search space points to reported performance values. They have no awareness of how the point will be used by the client.

The processing layers handle any additional tasks that must occur either before or after a point is generated. A prime example is the post processing to convert a numeric candidate point into client usable parameters. Consider the [compiler loop unrolling and tiling example](#) described earlier. The client cannot directly use a numeric point such as (3, 8, "↔ Intel") to execute a code variant. These values must first be sent to a compiler that will produce a binary to be executed by the client.

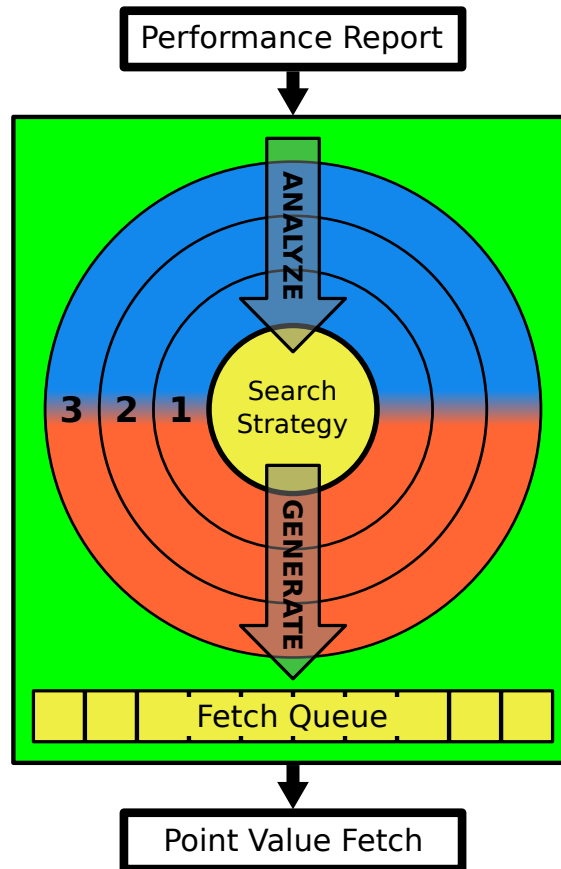


Figure 2: Detailed view of the Active Harmony tuning session.

The search strategy sits at the core of the tuning session, surrounded by concentric rings that represent processing layers. As points leave the search strategy, they must pass through the **generation** phase of each processing layer before it is made available to the client. Similarly, as performance reports are returned, they must pass through the **analyze** phase of each processing layer before it is received by the search strategy.

The processing layers are executed sequentially — in forward-order when leaving the search strategy, and backwards-order upon return. An individual processing layer may implement the generate action analyze action independent of one another, or both together to support paired functionality.

Finally, Active Harmony provides flexibility by implementing search strategies and processing layers as plug-ins that are loaded by the tuning session. This structure allows Active Harmony to meet the needs of any auto-tuning application with minimal effort. A specialized auto-tuner can effectively be built by parts.

Active Harmony is distributed with a set of search strategies and processing layers. Detailed information about these plug-ins can be found in the [Plug-Ins](#) section.

3 Getting Started

3.1 Downloading the Source

The latest release of Active Harmony can always be found at:

- <http://www.dyninst.org/harmony>

Release tarballs represent snapshots of our development repository in a tested and stable state. If you'd like to try new features or bug fixes, you can download a version directly from our Git repository:

Directly via git:

- `git clone http://git.dyninst.org/activeharmony.git`

Or, through our repository's web interface at:

- <http://git.dyninst.org/?p=activeharmony.git;a=summary>

3.2 Installation

Once you obtain a copy of the source tree (via tarball or Git), the following utilities are necessary for a successful build:

- GNU Make
- C99-compliant compiler

To build and install Active Harmony, issue the following command:

```
$ make install
```

By default, this will create relevant files within the source tree. If you'd like to install the software elsewhere, use the PREFIX variable:

```
$ make install PREFIX=$HOME/local/harmony-4.6.0
```

This make system is also sensitive to standard build flags supplied by the environment:

```
$ make CFLAGS=-O3 LDFLAGS=-L/usr/local/lib64
```

Furthermore, some processing layers bundled with Active Harmony depend on external packages. Build targets with such a dependency are protected behind specific environment variables. The make system will only attempt to build the processing layer if the associated package variable is defined. Otherwise, that target is skipped.

Dependent Package Variables**

Layer	Variable	Package Download URL
TAUdb	LIBTAUDB	http://www.cs.uoregon.edu/research/tau/downloads.php

XML Writer	LIBXML2	http://www.xmlsoft.org/downloads.html
------------	---------	---

Additional details on any of these build variables can be found in their respective Processing Layer documentation page of the User's Guide.

3.2.1 Building the Documentation

The following software packages will also be necessary to build the documentation:

- Doxygen
- TeX Live (pdflatex)
- Inkscape

To generate documentation in PDF and HTML formats, issue the following command:

```
$ make doc
```

This should leave `.pdf` files and `.html` directories in the `doc/` directory.

3.3 Testing the Installation

There are two separate modes of operation for using Active Harmony, standalone mode and server mode. Both modes are described below and can be used to verify a successful build.

3.3.1 Standalone Mode

This is the default model for Active Harmony clients. When initiating a new tuning session, the target application spawns its own dedicated tuning process automatically.

Correct operation relies on the `HARMONY_HOME` environment variable. It should be set to wherever your version of Active Harmony has been installed. Namely, it should match the `PREFIX` variable used during `make install`. For example, the following command would match the installation described in the [Installation](#) section above (assuming Bourne shell semantics):

```
$ export HARMONY_HOME=$HOME/local/harmony-4.6.0
```

If no `PREFIX` variable was used during `make install`, `HARMONY_HOME` should be set to the base directory of the source tree.

Navigate to the `example/client_api` source directory. The file `minimal` should exist if the build was successful. If `HARMONY_HOME` environment variable is set up correctly, the example should be immediately executable. The example may be run as follows:

```
$ ./minimal
```

This should produce the output similar to the following:

```
Starting Harmony...
( 481, 0.5269, "pineapple") = 874545.454545
( 490, 0.3286, "oranges") = 1119872.185027
( 655, 0.5611, "grapes") = 749438.602745
( 373, 0.5836, "grapes") = 410325.565456
( 516, 0.7858, "peaches") = 478701.959786
```

```
<... snip ...>
( 1, 1.0000, "figs") = 425.000000
( 1, 1.0000, "figs") = 425.000000
( 1, 0.9999, "figs") = 425.042504
( 1, 1.0000, "figs") = 425.000000
( 1, 1.0000, "figs") = 425.000000 (* Best point found. *)
```

This example uses a simple arithmetic function with six variables to produce a synthetic performance value.

3.3.2 Server Mode

This running model requires tuning clients to communicate to a server via a TCP connection. To run the server:

```
$ ./hserver [-p PORT]
```

Once the server is running, we must inform the client to use it. This is accomplished through two environment variables, `HARMONY_HOST` and `HARMONY_PORT`. As a quick example, to run the previous example in server mode, one could use the following command (assuming Bourne shell semantics):

```
$ HARMONY_HOST=localhost ./minimal
```

If `HARMONY_HOST` is defined, the example will attempt to contact a Harmony Server listening on the specified host (in this case, `localhost`) on port 1979. If you wish to connect to a server running on a different port, set the `HARMONY_PORT` environment variable. For example:

```
$ export HARMONY_HOST=other.host.net
$ export HARMONY_PORT=2013
```

An additional perk of running in Server Mode is the web interface. Simply point a javascript-enabled browser to the `host:port` of a running Harmony Server, and it will provide a visual interface for any tuning sessions under its control.

3.4 Exploring Further

Now that you have a sample Harmony client working, you can begin to explore the flexibility of the Active Harmony framework. All the examples below make use of the [Configuration System](#), which is documented in the User's Manual.

By default, the `PRO` search strategy is used. It can easily be changed by setting the `STRATEGY` configuration variable. This way, search strategies are easily comparable. For instance, the following example instructs the session to use a random search strategy instead:

```
$ ./minimal STRATEGY=random.so
```

In addition to changing the core search strategy, additional [processing layers](#) can be easily added to the feedback loop. For instance, if you'd like to write a log of the search session to disk, use the [Point Logger](#) processing layer:

```
$ ./minimal LAYERS=log.so LOG_FILE=search.log
```

Wall-time is a non-deterministic performance metric. Multiple factors such as competing processes, or even CPU temperature can perturb measurement. This phenomenon can be mitigated by performing multiple tests of each point and aggregating the results with the [Aggregator](#) processing layer:

```
$ ./minimal LAYERS=agg.so AGG_TIMES=5 AGG_FUNC=median
```

As described in the [Tuning Session](#), processing layers may be stacked. However, it is important to remember that processing layers are not commutative; their order is important. Consider the following two tuning sessions:

```
$ ./minimal LAYERS=agg.so:log.so \
    LOG_FILE=search.log AGG_TIMES=5 AGG_FUNC=median
```

and

```
$ ./minimal LAYERS=log.so:agg.so \  
    LOG_FILE=search.log AGG_TIMES=5 AGG_FUNC=median
```

Only the order of the processing layers have changed, but the second invocation will produce a much smaller log file. This is because the Aggregator is the outermost layer and receives performance reports before the Point Logger. The Point Logger is then unaware of the repeated experiments and only records the resulting aggregated performance value.

In the first invocation, the Point Logger records each performance report before the Aggregator has a chance to unify them. This results in a log file with many repeated experiments.

A real-world example is provided in `example/code_generation` source directory. That example relies on the code-server, MPI, and CHILL. Additional details can be found in `example/code_generation/README` of the source distribution.

4 Harmony Applications

4.1 Harmony Information Utility

Hinfo is a tool used to print information about an Active Harmony installation and its configuration. The application can also perform basic validation of the installation or included components, and warn the user if any problems are detected.

Usage Syntax

```
hinfo [flags]
```

Flag Information

Flag	Short	Description
<code>--home</code>	<code>-h</code>	Print Active Harmony installation path.
<code>--info=[STRING]</code>	<code>-i</code>	Display detailed information about a specific plug-in. If the argument includes path information (a <code>\</code> character), the string is treated as a file and opened directly. Otherwise, <code>HARMONY_HOME/libexec</code> is searched for a matching plug-in (by title or filename).
<code>--list</code>	<code>-l</code>	List all available Active Harmony plug-ins.
<code>--verbose</code>	<code>-v</code>	Display verbose output during operation.

Usage and Output Examples

Many hinfo operations require a valid Active Harmony installation. The location this directory is inferred or explicitly set using the following rules, in decreasing order of precedence:

1. `HARMONY_HOME` environment variable.
2. Invocation path of hinfo.
3. `PATH` environment variable.

The following example instructs hinfo to print the Active Harmony installation path to be used, and verbosely explain how the path was inferred.

```
$ hinfo --home -v
Inferring home via PATH environment variable.
Harmony home: /usr/local/packages/activeharmony/bin/..

$ ./bin/hinfo --home -v
Inferring home via program invocation path.
Harmony home: ./bin/..
```

The following example instructs hinfo to list all available plug-ins. The plug-ins are listed by title (when available), and file name.

```
$ hinfo --list
Available strategies:
  exhaustive.so
  nemo.so
  nm.so
```

```
pro.so
random.so
```

```
Available processing layers:
agg (agg.so)
cache (cache.so)
codegen (codegen.so)
constraint (constraint.so)
group (group.so)
logger (log.so)
xmlWriter (xmlWriter.so)
```

Hinfo can also provide detailed information about specific plug-ins. Plug-ins may be specified by title or file name, as in the following example:

```
$ hinfo -i logger
Considering 'log.so' as a strategy plug-in:
  Not a strategy: No strategy callbacks defined.

Considering 'log.so' as a processing layer plug-in:
  Detected 'logger' layer. Valid callbacks defined:
    logger_join
    logger_analyze
    logger_init
    logger_fini
```

4.2 Harmony Server

For some advanced uses of auto-tuning, multiple clients need to communicate to a single tuning session. For instance, if an MPI program is being tuned, all the constituent ranks may participate in parallel to expedite the search. In this case, the Active Harmony Server must be used as a central multiplexing unit.

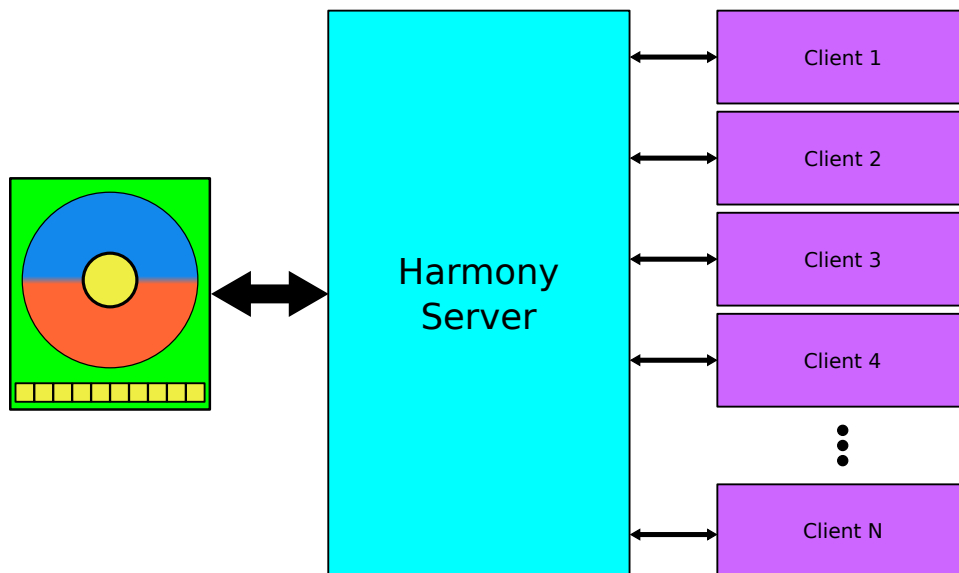


Figure 3: Supporting multiple clients within the feedback loop.

Usage Syntax

```
hserver [-p PORT] [-v]
```

The server has no mandatory parameters, and can be started with a plain invocation. This will launch a Harmony Server and bind it to port 1979. An alternate port may be supplied on the command-line.

Client Modification

Using the Active Harmony server is functionally equivalent to the stand-alone case. Users need only change two environment variables on the client machines.

Environment Variable	Description
HARMONY_HOST	Hostname of the machine running the Harmony Server.
HARMONY_PORT	TCP/IP port allocated by Harmony Server.

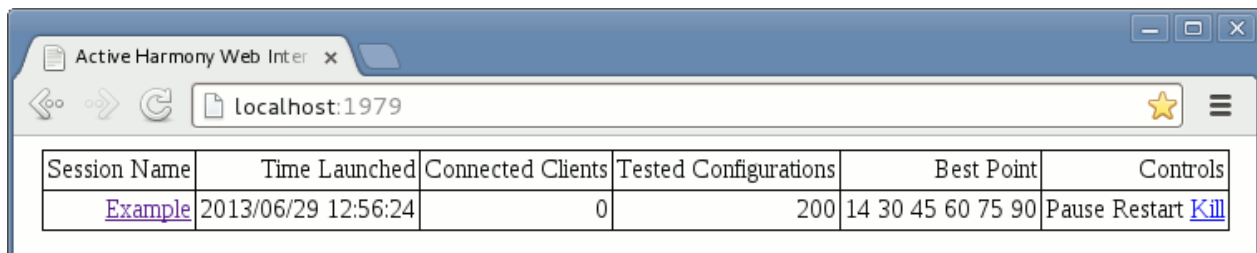
When defined, these environment variables instruct clients to connect to the specified hostname:port pair instead of spawning a local tuning session. Multiple clients may then work together on a single search problem.

Web Server

The Harmony Server also provides a built-in web server as an interface to the sessions it controls. Use a Javascript-enabled web browser to connect to the host and port the server is running on. For example, the URL for connecting to a locally-running server on the default port would be:

```
http://localhost:1979
```

Replace the host or port to match the desired Harmony Server as necessary. A screen similar to the following should appear:



Session Name	Time Launched	Connected Clients	Tested Configurations	Best Point	Controls
Example	2013/06/29 12:56:24	0	200	14 30 45 60 75 90	Pause Restart Kill

Figure 4: Session selection menu of the web interface.

Since the Harmony Server can manage more than one session at a time, there may be multiple lines in the table. You can view detailed information regarding a specific session by clicking its name in the first column. This should produce a screen similar to the following:

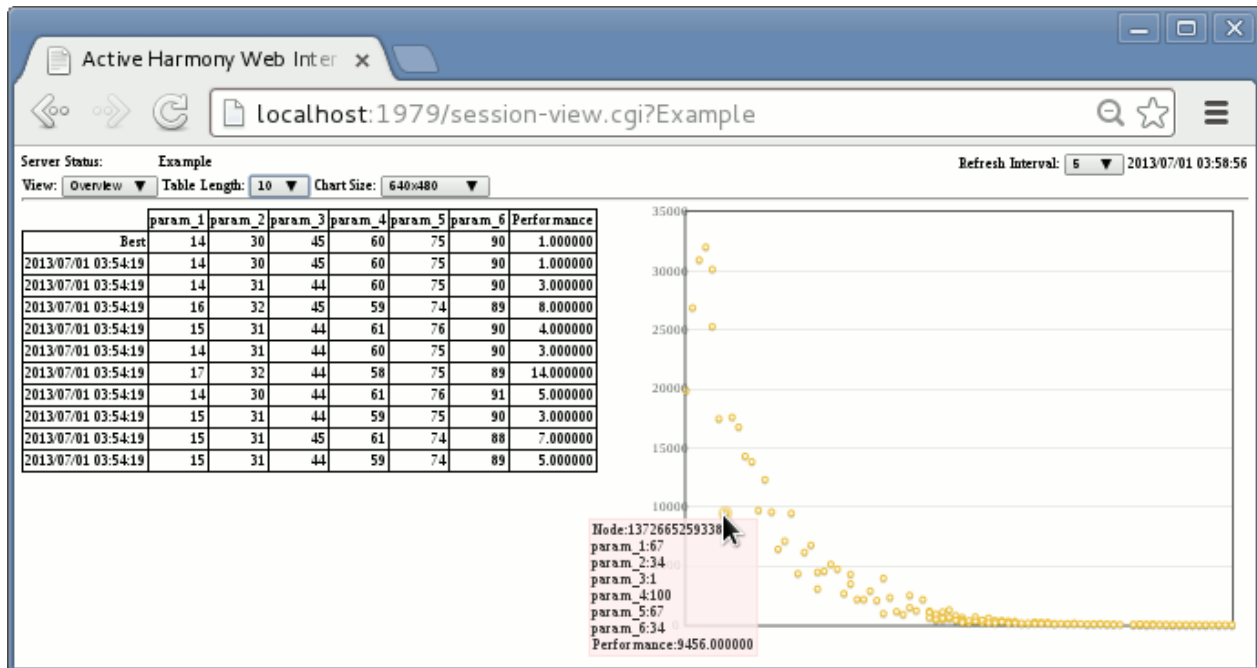


Figure 5: Detailed view of a particular session.

The detailed session view includes several ways to help you visualize the current session. On the left, a table of the ten most recently reported point/performance pairs, as well as the best performing point found thus far. The list can be extended to display as many as 50 entries. On the right, a plot of the entire search is drawn with respect to performance value along the y-axis, and time along the x-axis. Hovering the mouse pointer over any individual marker will produce details of the point/performance pair.

4.3 Tuna: The Command-line Tuning Shell

Tuna is a tool for tuning the parameters of command-line applications. Given a target application and a tuning specification, Tuna is able to perform a search for optimal parameter values automatically.

Usage Syntax

```
tuna <tunable_variables> [options] program [program_args]
```

Tunable Variable Declaration

Flag Usage	Description
<code>-i=<name>, <min>, <max>, <step></code>	Declare an integer number variable called <i>name</i> where valid values fall between <i>min</i> and <i>max</i> with a stride of size <i>step</i> .

<code>-r=<name>, <min>, <max>, <step></code>	Declare a real number variable called <i>name</i> where valid values fall between <i>min</i> and <i>max</i> with a stride of size <i>step</i> .
<code>-e=<name>, <val_1>, .., <val_n></code>	Declare an enumerated variable called <i>name</i> whose values must be <i>val_1</i> or <i>val_2</i> or .. or <i>val_n</i> .

Tuna provides three built-in methods for measuring the performance of a target application. These include wall-time, user-time, or system-time used by the target application. To support a wider range of possible performance metrics, a fourth method monitors output from the target application and parses a floating-point value from its final line as the performance value. This allows virtually any measure of performance, so long as it can be collected by an external wrapper program and printed.

Additional Options

Flag Usage	Description
<code>-m=<metric></code>	Specify how to measure the performance of the child process, where <i>metric</i> is: wall for wall time (<i>default</i>), user for user CPU time, sys for system CPU time, or output to parse a number from the final line of child output.
<code>-q</code>	Suppress client application output.
<code>-v</code>	Print additional informational output.
<code>-n=<num></code>	Run child program at most <i>num</i> times.

If the tunable variables cannot be supplied directly as arguments to the client application, then you must provide additional parameters to describe the format of the argument vector. Each argument (starting with and including the program binary) may include a percent sign (%) followed by the name of a previously defined tunable variable. This identifier may be optionally bracketed by curly-braces. Values from the tuning session will then be used to complete a command-line instance. A backslash (\) may be used to produce a literal %.

Usage Example

```
./tuna -i=tile,1,10,1 -i=unroll,2,12,2 -n=25 ./matrix_mult -t %tile -u %unroll
```

The above usage example defines a parameter space with two integer variables via the `-i` flag. The first variable (*tile*) is permitted to be between 1 and 10, inclusive. The second variable (*unroll*) is permitted to be even numbers between 2 and 12, inclusive. The tuning loop is limited to at most 25 iterations due to the optional `-n` flag. The remaining parameters specify that the target application (`matrix_mult`) should be launched with Harmony-chosen values for *tile* and *unroll* as the second and fourth arguments, respectively. Wall-time of each execution will be measured and reported, as it is the default performance metric.

Tuna makes any command-line application that provides performance related parameters a viable target for auto-tuning. As an example, the GCC compiler suite provides hundreds of command-line arguments to control various details of its compilation process. Finding optimal values for these arguments is a natural task for Tuna. A user need only specify which arguments are relevant for their optimization task and a method to measure the resulting performance.

4.4 Example Applications

Active Harmony comes bundles with a number of example applications to demonstrate different aspects of the framework.

4.4.1 Synth: A Synthetic Function Test Application

Introduction

This example application is designed to test and compare search strategies. It uses purely mathematical multi-variate continuous functions to simulate empirical test results. It requires two parameters: the name of a test function, and the number of tuning variables to use as input for the test function.

The underlying function can be modified several ways.

- **Input Accuracy** This truncates the input values for the underlying function after a certain number of decimal digits. In effect, it alters the resolution of the search strategy by changing the number of valid testing points within the search space. A higher accuracy term results in a more refined mesh. This can be used to test a search strategy's sensitivity to mesh resolution.
- **Output Quantization** This truncates the output of the underlying function after a certain number of decimal digits. In effect, it reduces the continuous nature of the underlying function, where lower quantization values result in a more discrete function. This can be used to test a search strategy's sensitivity to non-continuous functions.
- **Output Perturbation** This adds a random non-negative value to the output of the underlying function. The perturbation occurs after quantization, if both are requested. This can be used to test a search strategy's sensitivity to non-deterministic empirical tests.

Building

This example application only relies on a C compiler. It can be built with the following command:

```
$ make all
```

Usage

```
synth [OPTIONS] <fname> <N> [x1..xN] [KEY=VAL ..]
```

OPTION SUMMARY

```
-a, --accuracy=(inf|INT) Allow INT decimal digits of accuracy to the
                           right of the decimal point for function input
                           variables. If 'inf' is provided instead of an
                           integer, the full accuracy of an IEEE double
                           will be used. [Default: %d]
-l, --list                 List available test functions. Full
                           descriptions will be printed when used with -v.
-h, --help                 Display this help message.
-i, --iterations=INT      Perform at most INT search loop iterations.
-o, --options=LIST        Pass a comma (,) separated list of real numbers
                           to the test function as optional parameters.
-p, --perturb=REAL        Perturb the test function's output by a
                           uniformly random value in the range [0, REAL].
-q, --quantize=INT        Quantize the underlying function by rounding
                           output values after INT decimal digits to the
                           right of the decimal point.
-s, --seed=(time|INT)     Seed the random value generator with the
                           provided INT before perturbing output values.
                           If 'time' is provided instead of an integer,
                           the value will be taken from time(NULL).
                           [Default: time]
-t, --tuna=TYPE           Run as a child of Tuna, where only a single
                           point is tested and the output is used to
                           consume a proportional amount of resources as
                           determined by TYPE:
                           w = Wall time.
                           u = User CPU time.
                           s = System CPU time.
                           o = Print value to stdout. (default)
-v, --verbose             Verbose output.
```

Running Mode

There are two ways to use this application, tuning session mode and single evaluation mode.

Tuning Session Mode

By default, this application will launch its own auto-tuning session for the selected underlying function. After a session has been established and joined, it will search the underlying function for optimal points until either the search converges, or the maximum number of iterations has been performed.

Harmony configuration directives can be used to control various aspects of the tuning session, such as search strategy and included processing layers.

Upon completion, the application displays the best point and associated performance value discovered by the search, along with the global optimal performance value, if known.

Single Evaluation Mode

If N real values follow, they are considered to be a single point in the search space to test. The function is evaluated once, and the program exits. This mode is designed pair with Tuna. To further pair with Tuna's functionality, this application can be instructed to consume amount of a system resource that Tuna measuring proportional to the underlying function's output..

Example Invocations

The following invocation displays a list of supported underlying functions verbosely:

```
$ ./synth -l -v
```

To test DeJong's first function in two dimensions:

```
$ ./synth dejong 2
```

To test DeJong's first function in six dimensions:

```
$ ./synth dejong 6
```

To test 1000 iterations of the Random search strategy:

```
$ ./synth -i1000 dejong 6 STRATEGY=random.so
```

To perturb the function by at most 8, and compensate by using the minimum of 5 tests per point:

```
$ ./synth -p8.0 dejong 6 LAYERS=agg.so \  
    AGG_FUNC=min AGG_TIMES=5
```

To run as a client of tuna:

```
$ tuna -r=x1,-64,64,1e-6 -r=x2,-64,64,1e-6 -r=x3,-64,64,1e-6 \  
    -r=x4,-64,64,1e-6 -r=x5,-64,64,1e-6 -r=x6,-64,64,1e-6 \  
    ./synth -v --tuna=w -- dejong 6
```

5 Configuring Active Harmony

The Active Harmony Framework may be configured through two distinct, but similar systems. The session-wide Harmony Configuration System, and general environment variables.

5.1 Active Harmony Session Configuration System

Each tuning session provides a centralized configuration environment. Similar to shell environment variables, tuning session configuration directives take the form of simple key/value string pairs. Key strings are case insensitive, and may only consist of alphanumeric characters or the underscore. Value strings are stored as-is, and interpretation depends on the key.

The backslash (\) character may be used within the value string as a quoting mechanism. A backslash preserves the literal value of the next character that follows.

Configuration directives may also be read from a file containing one key/value pair per line. The first equals sign (=) separates the key from the value string, and the value string extends until a newline (\n) or hash (#) which indicates comments.

Each tuning session has a unique configuration environment. All entities involved with a tuning session (clients, search strategies, processing layers, etc.) may query this system. The configuration environment is initialized at session creation time through `ah_def_cfg()` function calls. After a session has been launched, clients may query the configuration environment through `ah_get_cfg()` and `ah_set_cfg()` function calls.

Note

The Harmony Server is a special case. It contains its own private configuration environment that is merged with any sessions that it manages. If there is a key conflict between the server and session configuration environment, the session's key takes precedence.

This manual documents configuration variables in tables with four columns. The documentation for subsystems such as [Search Strategies](#) and [Processing Layers](#) will contain an individual table that describes the specific configuration variables relevant to themselves. The list will be organized in a table with the following columns:

Column	Description
Key	Key string for configuration system.
Type	How the value string will be interpreted (Integer, Real, String, etc.)
Default	If unspecified, this value will be used instead.
Description	Textual description of the directive's function.

Additionally, there are several session-related configuration variables that are used to control sessions and how they are initialized.

Session-related Configuration Variables**

Key	Type	Default	Description
STRATEGY	String	pro.so	The search strategy to use for a particular session.
LAYERS	String	[none]	The processing layers to use for a particular session.

CLIENT_COUNT	Integer	1	The number of expected clients.
GEN_COUNT	Integer	1	The number of testing points to prepare for each expected client.

5.2 Environment Variables

Certain subsystems of the Active Harmony Framework may require additional information before it can connect to a session and, by extension, the session-wide configuration system. When necessary, this information may be retrieved through the subsystem's environment.

Here are a list of environment variables commonly used:

Environment Variable	Description
HARMONY_HOME	File path of the directory containing an Active Harmony installation. This is effectively the value of PREFIX when Harmony was built from source.
HARMONY_HOST	If defined, tuning clients (using ah_connect()) will attempt to connect to a Harmony Server on this host.
HARMONY_PORT	If defined (along with HARMONY_HOST), tuning clients will use this variable as the port when connecting to a running Harmony Server.

6 Plug-ins

As described in the [Tuning Session](#) section, Active Harmony provides a modular interface for flexible functionality. The session API functions `ah_def_strategy()` and `ah_def_layers()` specify which plug-in's will be loaded by the tuning session. See their individual documentation page for details on their use.

The Active Harmony framework currently allows for two types of plug-ins, [search strategies](#) and [processing layers](#).

6.1 Search Strategies

Search strategies encapsulate the core search logic of an Active Harmony tuning session. Ultimately, it decides the next search space point to be tested. While each search strategy may have radically different methods for selecting the next point, all strategies share the same interface. This allows tuning sessions to easily switch from one strategy to another.

See the [Tuning Session](#) section for an overview of the search strategy's role within the larger tuning session context.

6.1.1 Exhaustive (`exhaustive.so`)

This search strategy starts with the minimum-value point (i.e., using the minimum value for each tuning variable), and incrementing the tuning variables like an odometer until the maximum-value point is reached. This strategy is guaranteed to visit all points within a search space.

It is mainly used as a basis of comparison for more intelligent search strategies.

6.1.2 Random (`random.so`)

This search strategy generates random points within the search space. Using* a pseudo-random method, a value is selected for each tuning variable according to its defined bounds. This search will never reach a converged state.

It is mainly used as a basis of comparison for more intelligent search strategies.

6.1.3 Nelder-Mead (`nm.so`)

This search strategy uses a simplex-based method to estimate the relative slope of a search space without calculating gradients. It functions by evaluating the performance for each point of the simplex, and systematically replacing the worst performing point with a reflection, expansion, or contraction in relation to the simplex centroid. In some cases, the entire simplex may also be shrunken.

Note

Due to the nature of the underlying algorithm, this strategy is best suited for serial tuning tasks. It often waits on a single performance report before a new point may be generated.

For details of the algorithm, see:

Nelder, John A.; R. Mead (1965). "A simplex method for function minimization". *Computer Journal* 7: 308–313. doi:10.1093/comjnl/7.4.308

6.1.4 Parallel Rank Order (`pro.so`)

This search strategy uses a simplex-based method similar to the Nelder-Mead algorithm. It improves upon the Nelder-Mead algorithm by allowing the simultaneous search of all simplex points at each step of the algorithm. As such, it is ideal for a parallel search utilizing multiple nodes, for instance when integrated in OpenMP or MPI programs.

6.2 Processing Layers

Processing layers functionally surround the tuning session. They allow for additional processing before a strategy receives a performance report, or before a strategy receives a performance report, or both. Processing layers are stackable, which allow for an arbitrarily complex auto-tuner to be built by parts.

See the [Tuning Session](#) section for an overview of the processing layer's role within the larger tuning session context.

6.2.1 Aggregator (agg.so)

This processing layer forces each point to be evaluated multiple times before it may proceed through the auto-tuning [feedback loop](#). When the requisite number of evaluations has been reached, an aggregating function is applied to consolidate the set performance values.

6.2.2 Point Caching/Replay layer (cache.so)

This processing layer records point/performance pairs in a local cache as they are reported by clients. If the strategy later generates any points that exist in the cache, this layer will return the associated recorded performance immediately. Note that any outer layers (include the Harmony server and client) will not be notified upon cache hit.

The cache may optionally be initialized by a log file generated from the [Point Logger](#).

6.2.3 Code Server (codegen.so)

This processing layer passes messages from the tuning session to a running code generation server. Details on how to configure and run a code generation tuning session are provided in `code-server/README` of the distribution tarball.

Code Server URLs

The code server uses a proprietary set of URL's to determine the destination for various communications. They take the following form:

```
dir://<path>
ssh://[user@]<host>[:port]/<path>
tcp://<host>[:port]
```

All paths are considered relative. Use an additional slash to specify an absolute path. For example:

```
dir:///tmp
ssh://code.server.net:2222//tmp/codegen
```

6.2.4 Input Grouping (group.so)

This processing layer allows search space input variables (dimensions) to be iteratively search in groups. The value for all input variables not in the current search group remain constant.

Input variables are specified via a zero-based index, determined by the order they are defined in the session search space. For instance, the first variable defined via `harmony_int()`, `harmony_real()`, or `harmony_enum()` can be referenced via the index 0.

Example

Given a tuning session with nine input variables, the following group specification:

$(0, 1, 2), (3, 4, 5, 6), (7, 8)$

would instruct the layer to search the first three variables until the search strategy converges. Input values for the other six variables remain constant during this search.

Upon convergence, the search is restarted and allowed to investigate the next four variables. The first three variables are forced to use the best discovered values from the prior search. This pattern continues until all groups have been searched.

Grouping should be beneficial for search spaces whose input variables are relatively independent with respect to the reported performance.

6.2.5 Point Logger (log.so)

This processing layer writes a log of point/performance pairs to disk as they flow through the auto-tuning [feedback loop](#).

6.2.6 Omega Constraint (constraint.so)

Active Harmony allows for basic bounds on tuning variables during session specification, where each tuning variable is bounded individually. However, some target applications require tuning variables that are dependent upon one another, reducing the number of valid parameters from the strict Cartesian product.

This processing layer allows for the specification of such variable dependencies through algebraic statements. For example, consider a target application with tuning variables x and y . If x may never be greater than y , one could use the following statement:

$$x < y$$

Also, if the sum of x and y must remain under a certain constant, one could use the following statement:

$$x + y = 10$$

If multiple constraint statements are specified, the logical conjunction of the set is applied to the [Search Space](#).

Note

This processing layer requires the Omega Calculator, which is available at:

<https://github.com/davewathaverford/the-omega-project/>.

Some search strategies provide a `REJECT_METHOD` configuration variable that can be used to specify how to deal with rejected points. This can have great affect on the productivity of a tuning session.

6.2.7 TAUdb Interface (TAUdb.so)

Warning

This processing layer is still considered pre-beta.

This processing layer uses the TAU Performance System's C API to keep a log of point/performance pairs to disk as they flow through the auto-tuning [feedback loop](#).

The `LIBTAUDB` [build variable](#) must be defined at build time for this plug-in to be available, since it is dependent on a library distributed with TAU. The distribution of TAU is available here:

- <http://www.cs.uoregon.edu/research/tau/downloads.php>

And `LIBTAUDB` should point to the `tools/src/taudb_c_api` directory within that distribution.

6.2.8 XML Writer (xmlWriter.so)

Warning

This processing layer is still considered pre-beta.

This processing layer writes an XML formatted log of point/performance pairs to disk as they flow through the auto-tuning [feedback loop](#).

The `LIBXML2` [build variable](#) must be defined at build time for this plug-in to be available. The libxml2 library is available in multiple forms here:

- <http://www.xmlsoft.org/downloads.html>

And `LIBXML2` should point wherever libxml2 has been installed. For Linux distributions that include libxml2 as a package, using `/usr` may be sufficient.

7 Coding Examples

Active Harmony Client API enables the user to complete two distinct tasks: establishing tuning sessions and interacting with existing tuning sessions. In the following sections, we provide examples for both of these tasks.

7.1 Establishing Communication with a Tuning Session

This example demonstrates how to use the session API to establish a connection with a Harmony search session. It defines a function called `init_tuner()` which, similar to the MPI's `mpi_init()` routine, is designed to accept the `argc` and `argv` parameters from `main()`.

Code similar to `init_tuner()` should exist in any Harmony client application. The same set of API calls may be used to establish a connection with a local tuning process, or an existing tuning session through the [Harmony Server](#).

Note

To increase clarity, return values of API calls are not checked in this example. This is not recommended for production code.

```
#include <stdio.h>
#include "hclient.h"

hdesc_t* init_tuner(int* argc, char** argv)
{
    hdesc_t* hdesc;

    /* Initialize a Harmony tuning session handle. */
    hdesc = ah_alloc();

    /* Read any Harmony configuration directives passed on the
     * command line. This step is optional (but often helpful!).
     */
    ah_args(hdesc, argc, argv);

    /* Connect to a Harmony session. Since we're passing NULL as
     * our hostname, the HARMONY_HOST environment variable will be
     * used if defined. Otherwise, a local session will be spawned.
     */
    ah_connect(hdesc, NULL, 0);

    return hdesc;
}
```

7.2 Defining and Starting a New Search Task

This example demonstrates how to use the session API to establish a session with four variables. It defines a function called `new_search()` which takes a Harmony session descriptor, and returns a Harmony search task handle. The task handle will be used in the [task interaction](#) example.

Note

To increase clarity, return values of API calls are not checked in this example. This is not recommended for production code.

```
#include <stdio.h>
#include "hclient.h"

htask_t* new_search(hdesc_t* hdesc)
{
    hdef_t* hdef;
    htask_t* htask;

    /* Allocate a new Harmony search definition handle. */
    hdef = ah_def_alloc();
```

```

/* Give the tuning session a unique name. */
ah_def_name(hdef, "Example");

/* Add an integer variable called "intVar1" to the session.
 * Its value may range between 1 and 100 (inclusive).
 */
ah_def_int(hdef, "intVar1", 1, 100, 1, NULL);

/* Add another integer variable called "intVar2" to the session.
 * Its value may range between 2 and 200 (inclusive) by
 * strides of 2.
 */
ah_def_int(hdef, "intVar2", 2, 200, 2, NULL);

/* Add a real-valued variable called "realVar" to the session.
 * Its value may range between 0.0 and 0.5 (inclusive), using the
 * full precision available by an IEEE double.
 */
ah_def_real(hdef, "realVar", 0.0, 0.5, 0.0, NULL);

/* Add a string-valued variable called "strVar" to the session. */
ah_def_enum(hdef, "strVar", NULL);

/* The variable "strVar" may be "apples", "oranges", "peaches",
 * or "pears".
 */
ah_def_enum_value(hdef, "strVar", "apples");
ah_def_enum_value(hdef, "strVar", "oranges");
ah_def_enum_value(hdef, "strVar", "peaches");
ah_def_enum_value(hdef, "strVar", "pears");

/* Here we pass the definition handle to a subroutine to further
 * define the search task. See the "Advanced Search Task
 * Configuration" example for more details.
 */
advanced_config(hdef);

/* Start a new tuning task in the Harmony session. It will be
 * based on the search we've described above.
 */
htask = ah_start(hdesc, hdef);

/* Don't forget to free the definition handle! */
ah_def_free(hdef);

return htask;
}

```

7.3 Advanced Search Task Configuration

This example demonstrates the ability to specify [session plug-ins](#), and how to [configure](#) them. Detailed descriptions of the valid configuration keys for each of the three plug-ins used in this example can be found in their respective manual pages:

- [Parallel Rank Order Search Strategy](#)
- [Point Logger Processing Layer](#)
- [Aggregator Processing Layer](#)

The following code snippet extends the search definition handle created in the prior example. See the [search definition](#) example to see how definition handles are created.

Note

To increase clarity, return values of API calls are not checked in this example. This is not recommended for production code.

```
#include <stdio.h>
```

```

#include "hclient.h"

void advanced_config(hdef_t* hdef)
{
    /* Use the Parallel Rank Order simplex-based as the strategy for this
     * session, instead of the default Nelder-Mead.
     */
    ah_def_strategy(hdef, "pro.so");

    /* Instruct the strategy to use an initial simplex roughly half
     * the size of the search space.
     */
    ah_def_cfg(hdef, "INIT_RADIUS", "0.50");

    /* This tuning session should surround the search strategy with
     * a logger layer first, and an aggregator layer second.
     */
    ah_def_layers(hdef, "log.so:agg.so");

    /* Instruct the logger to use /tmp/tuning.run as the logfile. */
    ah_def_cfg(hdef, "LOG_FILE", "/tmp/tuning.run");

    /* Instruct the aggregator to collect 10 performance values for
     * each point, and allow the median performance to continue through
     * the feedback loop.
     */
    ah_def_cfg(hdef, "AGG_TIMES", "10");
    ah_def_cfg(hdef, "AGG_FUNC", "median");
}

```

7.4 Interacting with a Search Task

The example demonstrates how to use the client API to fetch from and report to the search task started in the [task definition and start](#) example.

Note

To increase clarity, return values of API calls are not checked in this example. This is not recommended for production code.

```

#include <stdio.h>
#include "hclient.h"

int main(int argc, char* argv[])
{
    /* Harmony data types. */
    hdesc_t* hdesc;
    htask_t* htask;

    /* Variables for this application's run-time tunable parameters. */
    long    var1;
    long    var2;
    double  var3;
    const char* var4;

    /* These functions are defined in the examples above. */
    hdesc = init_tuner(&argc, argv);
    htask = new_search(hdesc);

    /* Bind local memory to values fetched from Harmony.
     * This allows ah_fetch() to modify local variables automatically.
     */
    ah_bind_int(htask, "intVar1", &var1);
    ah_bind_int(htask, "intVar2", &var2);
    ah_bind_real(htask, "realVar", &var3);
    ah_bind_enum(htask, "strVar", &var4);

    /* Loop until the session has reached a converged state. */
    while (!ah_converged(htask))
    {
        /* Define a variable to hold the resulting performance value. */
        double perf;

        /* Retrieve new values from the Harmony Session. */
        ah_fetch(htask);
    }
}

```

```
/* The local variables var1, var2, var3, and var4 have now
 * been updated and are ready for use.
 *
 * This is where a normal application would do some work
 * using these variables and measure the performance.
 * Since this is a simple example, we'll pretend the
 * function "work()" will take the variables, and produce a
 * performance value.
 */
perf = work(var1, var2, var3, var4);

/* Report the performance back to the session. */
ah_report(htask, &perf);
}

/* Leave the search task. */
ah_leave(htask);

/* Free the session descriptor.
 * All associated task handles are also destroyed by this call. */
ah_free(hdesc);

return 0;
}
```

8 Module Index

8.1 Modules

Here is a list of all modules:

Harmony Descriptor Management Functions	31
Tuning Search Definition Functions	34
Tuning Search Task Control Functions	39
Search Task Interaction Functions	42
Harmony Error Reporting Functions	47

9 File Index

9.1 File List

Here is a list of all documented files with brief descriptions:

[hclient.h](#)

Harmony client application function header

48

10 Module Documentation

10.1 Harmony Descriptor Management Functions

Functions

- `hdesc_t * ah_alloc (void)`
Allocate and initialize a new Harmony descriptor.
- `int ah_args (hdesc_t *hdesc, int *argc, char **argv)`
Find configuration directives in the command-line arguments.
- `int ah_id (hdesc_t *hdesc, const char *id)`
Assign an identifying string to this client.
- `int ah_connect (hdesc_t *hdesc, const char *host, int port)`
Establish a connection with a Harmony tuning session.
- `int ah_close (hdesc_t *hdesc)`
Close the connection to a Harmony tuning session.
- `void ah_free (hdesc_t *hdesc)`
Release resources associated with a Harmony client descriptor.

10.1.1 Detailed Description

A Harmony descriptor is an opaque structure that stores state associated with a client's connection to a tuning session. A tuning session hosts one or more tuning search tasks, and is the only intermediary by which clients may communicate with search tasks. The functions within this section allow the user to create and manage the resources controlled by the descriptor.

10.1.2 Function Documentation

10.1.2.1 `hdesc_t* ah_alloc (void)`

Allocate and initialize a new Harmony descriptor.

All client API functions require a valid Harmony descriptor to function correctly. The descriptor is designed to be used as an opaque type and no guarantees are made about the contents of its structure.

Heap memory is allocated for the descriptor, so be sure to call `ah_free()` when it is no longer needed.

Returns

Returns a new Harmony descriptor upon success, and `NULL` otherwise.

10.1.2.2 `int ah_args (hdesc_t * hdesc, int * argc, char ** argv)`

Find configuration directives in the command-line arguments.

This function iterates through the command-line arguments (as passed to the `main()` function) to search for Harmony configuration directives in the form `NAME=VAL`. Any arguments matching this pattern are added to the configuration and removed from `argv`.

Iteration through the command-line arguments stops prematurely if a double-dash ("`--`") token is found. All non-directive arguments are shifted to the front of `argv`. Finally, `argc` is updated to reflect the remaining number of program arguments.

Parameters

<i>hdesc</i>	Harmony descriptor returned from ah_alloc() .
<i>argc</i>	Address of the <code>argc</code> variable.
<i>argv</i>	Address of the command-line argument array (i.e., the value of <code>argv</code>).

Returns

Returns the number of directives successfully taken from `argv`, or -1 on error.

10.1.2.3 `int ah_close (hdesc_t * hdesc)`

Close the connection to a Harmony tuning session.

Terminates communication with a Harmony session. Task descriptors associated with this session may continue to be used in a limited capacity. For example, best point retrieval is available ([ah_best\(\)](#)), but new testing values are not ([ah_fetch\(\)](#)).

Parameters

<i>hdesc</i>	Harmony descriptor returned from ah_alloc() .
--------------	---

Returns

Returns 0 on success, and -1 otherwise.

10.1.2.4 `int ah_connect (hdesc_t * hdesc, const char * host, int port)`

Establish a connection with a Harmony tuning session.

This function enables communication with a tuning session, Active Harmony's search management process. It must succeed prior to calling any search interface functions (e.g., [ah_join\(\)](#), [ah_launch\(\)](#), etc.).

If *host* is `NULL`, its value will be taken from the environment variable `HARMONY_HOST`. If `HARMONY_HOST` is not defined, the environment variable `HARMONY_HOME` will be used to initiate a private tuning session, which will be available only to the local process.

If *port* is 0, its value will be taken from the environment variable `HARMONY_PORT`, if defined. Otherwise, its value will be taken from the `src/defaults.h` file.

Parameters

<i>hdesc</i>	Harmony descriptor returned from ah_alloc() .
<i>host</i>	Host of the Harmony server (or <code>NULL</code>).
<i>port</i>	Port of the Harmony server.

Returns

Returns 0 on success. Otherwise, -1 is returned and a string indicating the nature of the failure may be retrieved from [ah_error\(\)](#).

10.1.2.5 `void ah_free (hdesc_t * hdesc)`

Release resources associated with a Harmony client descriptor.

If the descriptor is connected to a search session, this function will call [ah_leave\(\)](#) on any active search tasks this client is currently participating in.

Note

This function will not free memory for search definition structures built using this descriptor. They must be released separately using `ah_def_fini()`.

Warning

This function will free memory for search tasks associated with this session descriptor. This will invalidate pointers that were returned from functions like `ah_start()` and `ah_join()`. Be sure these are no longer in use before calling this function.

Parameters

<i>hdesc</i>	Harmony descriptor returned from <code>ah_alloc()</code> .
--------------	--

10.1.2.6 int ah_id (hdesc_t * hdesc, const char * id)

Assign an identifying string to this client.

Set the client identification string. All messages sent to the tuning session will be tagged with this string, allowing the framework to distinguish clients from one another. As such, care should be taken to ensure this string is unique among all clients participating in a tuning session.

If this function is not called, the client will be assigned an ID based on its hostname and process ID.

Parameters

<i>hdesc</i>	Harmony descriptor returned from <code>ah_alloc()</code> .
<i>id</i>	Unique identification string.

Returns

Returns 0 on success, and -1 otherwise.

10.2 Tuning Search Definition Functions

Functions

- `hdef_t * ah_def_alloc (void)`
Generate an empty search definition.
- `int ah_def_name (hdef_t *hdef, const char *name)`
Generate an empty search definition.
- `hdef_t * ah_def_load (const char *filename)`
Generate a tuning session description from a file.
- `void ah_def_free (hdef_t *hdef)`
Release the resources used by this definition descriptor.
- `int ah_def_int (hdef_t *hdef, const char *name, long min, long max, long step, long *ptr)`
Add an integer-domain variable to the search definition.
- `int ah_def_real (hdef_t *hdef, const char *name, double min, double max, double step, double *ptr)`
Add a real-domain variable to the search definition.
- `int ah_def_enum (hdef_t *hdef, const char *name, const char **ptr)`
Add an enumerated-domain variable to the search definition.
- `int ah_def_enum_value (hdef_t *hdef, const char *name, const char *value)`
Append a value to an enumerated-domain variable.
- `int ah_def_strategy (hdef_t *hdef, const char *strategy)`
Specify the strategy to use for this search.
- `int ah_def_layers (hdef_t *hdef, const char *list)`
Specify the list of plug-ins to use for this search.
- `int ah_def_cfg (hdef_t *hdef, const char *key, const char *val)`
Modify the initial configuration of a new Harmony search.

10.2.1 Detailed Description

These functions are used to define a Harmony tuning search. They allow users to articulate the details of each tuning variable, such as value domain type (e.g., integers vs. real numbers) and valid value ranges. To be valid, a search definition must contain at least one tuning variable.

10.2.2 Function Documentation

10.2.2.1 `hdef_t* ah_def_alloc (void)`

Generate an empty search definition.

This structure is used with functions like `ah_def_int()` and `ah_def_enum()` to programmatically define the parameters and configuration environment for a new search.

Returns

Returns a new Harmony search definition descriptor on success, and `NULL` otherwise.

10.2.2.2 `int ah_def_cfg (hdef_t* hdef, const char * key, const char * val)`

Modify the initial configuration of a new Harmony search.

This function allows the user to specify key/value pairs that will exist in the search prior to calling `ah_start()`.

Parameters

<i>hdef</i>	Definition descriptor returned from ah_def_alloc() or ah_def_load() .
<i>key</i>	List of plug-ins to load with this session.
<i>val</i>	List of plug-ins to load with this session.

Returns

Returns 0 on success, and -1 otherwise.

10.2.2.3 `int ah_def_enum (hdef_t * hdef, const char * name, const char ** ptr)`

Add an enumerated-domain variable to the search definition.

Parameters

<i>hdef</i>	Definition descriptor returned from ah_def_alloc() or ah_def_load() .
<i>name</i>	Name to associate with this variable.
<i>ptr</i>	Address inside the client where <code>sizeof(char*)</code> bytes may be written when new values are retrieved via ah_fetch() or ah_best() .

Returns

Returns 0 on success, and -1 otherwise.

10.2.2.4 `int ah_def_enum_value (hdef_t * hdef, const char * name, const char * value)`

Append a value to an enumerated-domain variable.

If the variable does not exist in the search definition, it will be created.

Parameters

<i>hdef</i>	Definition descriptor returned from ah_def_alloc() or ah_def_load() .
<i>name</i>	Name to associate with this variable.
<i>value</i>	String that belongs in this enumeration.

Returns

Returns 0 on success, and -1 otherwise.

10.2.2.5 `void ah_def_free (hdef_t * hdef)`

Release the resources used by this definition descriptor.

Parameters

<i>hdef</i>	Definition descriptor returned from ah_def_alloc() or ah_def_load() .
-------------	---

10.2.2.6 `int ah_def_int (hdef_t * hdef, const char * name, long min, long max, long step, long * ptr)`

Add an integer-domain variable to the search definition.

Parameters

<i>hdef</i>	Definition descriptor returned from ah_def_alloc() or ah_def_load() .
<i>name</i>	Name to associate with this variable.
<i>min</i>	Minimum range value (inclusive).
<i>max</i>	Maximum range value (inclusive).
<i>step</i>	Minimum search increment.
<i>ptr</i>	Address inside the client where <code>sizeof(long)</code> bytes may be written when new values are retrieved via ah_fetch() or ah_best() .

Returns

Returns 0 on success, and -1 otherwise.

10.2.2.7 `int ah_def_layers (hdef_t* hdef, const char * list)`

Specify the list of plug-ins to use for this search.

Plug-in layers are specified via a single string of filenames separated by colon (:), semicolon (;), comma (,), or whitespace characters. The layers are loaded in list order, with each successive layer placed further from the search strategy in the center.

Parameters

<i>hdef</i>	Definition descriptor returned from ah_def_alloc() or ah_def_load() .
<i>list</i>	List of plug-ins to load with this session.

Returns

Returns 0 on success, and -1 otherwise.

10.2.2.8 `hdef_t* ah_def_load (const char * filename)`

Generate a tuning session description from a file.

Opens and parses a file for configuration and tuning variable declarations. The session name is copied from the filename. If `filename` is the string "-", `stdin` will be used instead. To load a file named "-", include the path (i.e., "./-").

The file is parsed line by line. Each line may contain either a tuning variable or configuration variable definitions. The line is considered a tuning variable declaration if it begins with a valid tuning variable type.

Here are some examples of valid integer-domain variables:

```
int first_ivar = min:-10 max:10 step:2
int second_ivar = min:1 max:100 # Integers have a default step of 1.
```

Here are some examples of valid real-domain variables:

```
real first_rvar = min:0 max:1 step:1e-4
real second_rvar = min:-0.5 max:0.5 step:0.001
```

Here are some examples of valid enumerated-domain variables:

```
enum sort_algo = bubble, insertion, quick, merge
enum search_algo = forward backward binary # Commas are optional.
enum quotes = "Cogito ergo sum" \
              "To be, or not to be." \
              "What's up, doc?"
```

Otherwise, the line is considered to be a configuration variable definition, like the following examples:

```
STRATEGY=pro.so
INIT_POINT = 1.41421, \
            2.71828, \
            3.14159
LAYERS=agg.so:log.so

AGG_TIMES=5
AGG_FUNC=median

LOG_FILE=/tmp/search.log
LOG_MODE=w
```

Variable names must conform to C identifier rules. Comments are preceded by the hash (#) character, and long lines may be joined with a backslash () character.

Parsing stops immediately upon error and [ah_error\(\)](#) may be used to determine the nature of the error.

Parameters

<i>filename</i>	Name to associate with this variable.
-----------------	---------------------------------------

Returns

Returns a new `hdef_t` pointer on success, and NULL otherwise.

10.2.2.9 `int ah_def_name (hdef_t * hdef, const char * name)`

Generate an empty search definition.

This structure is used with functions like [ah_def_int\(\)](#) and [ah_def_enum\(\)](#) to programmatically define the parameters and configuration environment for a new search.

Parameters

<i>hdef</i>	Definition descriptor returned from ah_def_alloc() or ah_def_load() .
<i>name</i>	Name to associate with this search. If NULL, then the library will attempt to generate a unique identifier during ah_start() .

Returns

Returns 0 on success, and -1 otherwise.

10.2.2.10 `int ah_def_real (hdef_t * hdef, const char * name, double min, double max, double step, double * ptr)`

Add a real-domain variable to the search definition.

Parameters

<i>hdef</i>	Definition descriptor returned from ah_def_alloc() or ah_def_load() .
<i>name</i>	Name to associate with this variable.
<i>min</i>	Minimum range value (inclusive).
<i>max</i>	Maximum range value (inclusive).

<i>step</i>	Minimum search increment.
<i>ptr</i>	Address inside the client where <code>sizeof(double)</code> bytes may be written when new values are retrieved via ah_fetch() or ah_best() .

Returns

Returns 0 on success, and -1 otherwise.

10.2.2.11 int ah_def_strategy (hdef_t * hdef, const char * strategy)

Specify the strategy to use for this search.

Parameters

<i>hdef</i>	Definition descriptor returned from ah_def_alloc() or ah_def_load() .
<i>strategy</i>	Filename of the strategy plug-in to use in this session.

Returns

Returns 0 on success, and -1 otherwise.

10.3 Tuning Search Task Control Functions

Functions

- `htask_t * ah_start (hdesc_t *hdesc, hdef_t *hdef)`
Start a new Harmony tuning search task.
- `htask_t * ah_join (hdesc_t *hdesc, const char *name)`
Join an established Harmony tuning search task.
- `int ah_pause (htask_t *htask)`
Pause a tuning search task.
- `int ah_resume (htask_t *htask)`
Resume a tuning search task.
- `int ah_restart (htask_t *htask)`
Resume a tuning search task.
- `int ah_leave (htask_t *htask)`
Leave a tuning search task.
- `int ah_kill (htask_t *htask)`
Kill a tuning search task.

10.3.1 Detailed Description

These functions are used to control Harmony tuning search tasks. They allow client programs to participate in a search by either starting a new search or joining an existing task. Once participating, the client may control the search by pausing or resetting it.

10.3.2 Function Documentation

10.3.2.1 `htask_t* ah_join (hdesc_t * hdesc, const char * name)`

Join an established Harmony tuning search task.

After connecting to a session using `ah_connect()`, this function initiates a new tuning search task as defined by the given search definition.

Parameters

<i>hdesc</i>	Harmony descriptor returned from <code>ah_alloc()</code> .
<i>name</i>	Name of an existing tuning search in the session.

Returns

Returns a new `htask_t` pointer on success. Otherwise, NULL is returned and `ah_error()` may be used to determine the nature of the error.

10.3.2.2 `int ah_kill (htask_t * htask)`

Kill a tuning search task.

If the task descriptor is currently attached, this function will send a kill request to the remote session. This instructs the session to release the resources it holds for the task. Any future messages received by the session destined for this task will result in error.

Note

Errors encountered while sending the kill request will be ignored, in case the search has already been killed by a competing client.

If the task descriptor is currently detached (via [ah_leave\(\)](#)), the kill request is skipped.

Either way, this function will release client-local resources reserved for this task descriptor. Using the now defunct task descriptor is an error and will result in undefined behavior.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
--------------	---

Returns

Returns 0 on success, and -1 otherwise.

10.3.2.3 int ah_leave (htask_t * htask)

Leave a tuning search task.

End participation with the search task. The task descriptor will remain available for limited set of functions (e.g., [ah_get_int\(\)](#), etc.), but functions that participation in a task (e.g., [ah_fetch\(\)](#)) will result in error.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
--------------	---

Returns

Returns 0 on success, and -1 otherwise.

10.3.2.4 int ah_pause (htask_t * htask)

Pause a tuning search task.

This instructs the session to halt the production of new trial points for this search task, and return only the best point until the search is resumed.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
--------------	---

Returns

Returns 0 on success, and -1 otherwise.

10.3.2.5 int ah_restart (htask_t * htask)

Resume a tuning search task.

This instructs the session to continue the production of new trial points for this search task.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
--------------	---

Returns

Returns 0 on success, and -1 otherwise.

10.3.2.6 int ah_resume (htask_t * htask)

Resume a tuning search task.

This instructs the session to continue the production of new trial points for this search task.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
--------------	---

Returns

Returns 0 on success, and -1 otherwise.

10.3.2.7 htask_t* ah_start (hdesc_t * hdesc, hdef_t * hdef)

Start a new Harmony tuning search task.

After connecting to a session using [ah_connect\(\)](#), this function initiates a new tuning search task as defined by the given search definition.

Parameters

<i>hdesc</i>	Harmony descriptor returned from ah_alloc() .
<i>hdef</i>	Definition descriptor returned from ah_def_alloc() or ah_def_load() .

Returns

Returns a new Harmony search task descriptor on success, and `NULL` otherwise.

10.4 Search Task Interaction Functions

Functions

- int [ah_bind_int](#) (htask_t *htask, const char *name, long *ptr)
Bind a client address to an integer-domain search variable (dimension).
- int [ah_bind_real](#) (htask_t *htask, const char *name, double *ptr)
Bind a client address to a real-domain search variable (dimension).
- int [ah_bind_enum](#) (htask_t *htask, const char *name, const char **ptr)
Bind a client address to an enumerated string-based search variable (dimension).
- long [ah_get_int](#) (htask_t *htask, const char *name)
Return the current value of an integer-domain search variable.
- double [ah_get_real](#) (htask_t *htask, const char *name)
Return the current value of a real-domain search variable.
- const char * [ah_get_enum](#) (htask_t *htask, const char *name)
Return the current value of an enumerated-domain search variable.
- const char * [ah_get_cfg](#) (htask_t *htask, const char *key)
Get a key value from the search's configuration.
- const char * [ah_set_cfg](#) (htask_t *htask, const char *key, const char *val)
Set a new key/value pair in the search's configuration.
- int [ah_fetch](#) (htask_t *htask)
Fetch a new configuration from the Harmony search.
- int [ah_report](#) (htask_t *htask, double *perf)
Report the performance of a configuration to the Harmony search.
- int [ah_report_one](#) (htask_t *htask, int index, double value)
Report a single performance value for the current configuration.
- int [ah_best](#) (htask_t *htask)
Set values under Harmony's control to the best known configuration.
- int [ah_converged](#) (htask_t *htask)
Retrieve the convergence state of the current search.

10.4.1 Detailed Description

These functions are used by the client after it is participating in a running search. This is how clients perform activities such as retrieving a point to test and reporting its associated performance.

10.4.2 Function Documentation

10.4.2.1 int [ah_best](#) (htask_t * htask)

Set values under Harmony's control to the best known configuration.

Bound memory (from using functions like [ah_bind_int\(\)](#)) or values retrieved (from functions like [ah_get_real\(\)](#)) will be the best known input values from the search thus far.

If no best configuration exists (i.e., before any configurations have been evaluated), this function will return an error.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
--------------	---

Returns

Returns 1 on success, and -1 otherwise.

10.4.2.2 `int ah_bind_enum (htask_t * htask, const char * name, const char ** ptr)`

Bind a client address to an enumerated string-based search variable (dimension).

This function associates an address inside the client with a search variable defined using [ah_def_enum\(\)](#). Upon [ah_fetch\(\)](#), the value retrieved for this tuning variable will be stored in `sizeof(char*)` bytes starting at address `ptr`.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
<i>name</i>	Search variable defined using ah_def_int() .
<i>ptr</i>	Address inside the client where <code>sizeof(char*)</code> bytes may be written to store the current testing value.

Returns

Returns 0 on success, and -1 otherwise.

10.4.2.3 `int ah_bind_int (htask_t * htask, const char * name, long * ptr)`

Bind a client address to an integer-domain search variable (dimension).

This function associates an address inside the client with a search variable defined using [ah_def_int\(\)](#). Upon [ah_fetch\(\)](#), the value retrieved for this tuning variable will be stored in `sizeof(long)` bytes starting at address `ptr`.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
<i>name</i>	Search variable defined using ah_def_int() .
<i>ptr</i>	Address inside the client where <code>sizeof(long)</code> bytes may be written to store the current testing value.

Returns

Returns 0 on success, and -1 otherwise.

10.4.2.4 `int ah_bind_real (htask_t * htask, const char * name, double * ptr)`

Bind a client address to a real-domain search variable (dimension).

This function associates an address inside the client with a search variable defined using [ah_def_real\(\)](#). Upon [ah_fetch\(\)](#), the value retrieved for this tuning variable will be stored in `sizeof(double)` bytes starting at address `ptr`.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
<i>name</i>	Search variable defined using ah_def_real() .
<i>ptr</i>	Address inside the client where <code>sizeof(double)</code> bytes may be written to store the current testing value.

Returns

Returns 0 on success, and -1 otherwise.

10.4.2.5 int ah_converged (htask_t * htask)

Retrieve the convergence state of the current search.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
--------------	---

Returns

Returns 1 if the search has converged, 0 if it has not, and -1 on error.

10.4.2.6 int ah_fetch (htask_t * htask)

Fetch a new configuration from the Harmony search.

If a new configuration is available, this function will update the values of all registered variables. Otherwise, it will configure the system to run with the best known configuration thus far.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
--------------	---

Returns

Returns 0 if no registered variables were modified, 1 if any registered variables were modified, and -1 otherwise.

10.4.2.7 const char* ah_get_cfg (htask_t * htask, const char * key)

Get a key value from the search's configuration.

Retrieve the string value associated with the given key in the search task's configuration system.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
<i>key</i>	Config key to search for on the session.

Returns

Returns a c-style string on success, and NULL otherwise.

10.4.2.8 const char* ah_get_enum (htask_t * htask, const char * name)

Return the current value of an enumerated-domain search variable.

Finds an enumerated-domain tuning variable given its name and returns its current value.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
<i>name</i>	Name of a tuning variable declared with ah_def_enum() .

Returns

Returns the current value of a tuning variable, if the given name matches an enumerated-domain search variable. Otherwise, NULL is returned and [ah_error\(\)](#) may be used to determine the nature of the error.

10.4.2.9 `long ah_get_int (htask_t * htask, const char * name)`

Return the current value of an integer-domain search variable.

Finds an integer-domain tuning variable given its name and returns its current value.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
<i>name</i>	Name of a tuning variable declared with ah_def_int() .

Returns

Returns the current value of a tuning variable, if the given name matches an integer-domain search variable. Otherwise, LONG_MIN is returned and [ah_error\(\)](#) may be used to determine the nature of the error.

10.4.2.10 `double ah_get_real (htask_t * htask, const char * name)`

Return the current value of a real-domain search variable.

Finds a real-domain tuning variable given its name and returns its current value.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
<i>name</i>	Name of a tuning variable declared with ah_def_real() .

Returns

Returns the current value of a tuning variable, if the given name matches a real-domain search variable. Otherwise, NAN is returned and [ah_error\(\)](#) may be used to determine the nature of the error.

10.4.2.11 `int ah_report (htask_t * htask, double * perf)`

Report the performance of a configuration to the Harmony search.

Sends a complete performance report regarding the current configuration to the Harmony session. The `perf` parameter should point to a floating-point double.

For multi-objective search problems, the `perf` parameter should point to an array of floating-point doubles containing all performance values. If all performance values have already been reported via [ah_report_one\(\)](#), then NULL may be passed as the performance pointer. Unreported performance values will result in error.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
<i>perf</i>	Performance vector for the current configuration.

Returns

Returns 0 on success, and -1 otherwise.

10.4.2.12 `int ah_report_one (htask_t * htask, int index, double value)`

Report a single performance value for the current configuration.

Allows performance values to be reported one at a time for multi-objective search problems.

Note that this function only caches values to send to the Harmony search. Once all values have been reported, [ah_report\(\)](#) must be called (passing `NULL` as the performance argument).

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
<i>index</i>	Objective index of the value to report.
<i>value</i>	Performance measured for the current configuration.

Returns

Returns 0 on success, and -1 otherwise.

10.4.2.13 `const char* ah_set_cfg (htask_t * htask, const char * key, const char * val)`

Set a new key/value pair in the search's configuration.

Writes the new key/value pair into the search's run-time configuration database. If the key exists in the database, its value is overwritten. If `val` is `NULL`, the key will be erased from the database.

Parameters

<i>htask</i>	Task descriptor returned from ah_start() or ah_join() .
<i>key</i>	Config key to modify in the search.
<i>val</i>	Config value to associate with the key.

Returns

Returns the original key value on success. If the key did not exist prior to this call, an empty string ("") is returned. Otherwise, `NULL` is returned on error.

Note

The buffer which stores the previous configuration value may be used by other Harmony Client API functions. If you need the value to persist beyond the next API call, you must make a copy.

10.5 Harmony Error Reporting Functions

Functions

- `const char * ah_error (void)`
Retrieve the most recent Harmony API error string.
- `void ah_error_clear (void)`
Clear the current task-level error string.

10.5.1 Detailed Description

These functions may be used to retrieve information about the most recent failure within Harmony API functions.

10.5.2 Function Documentation

10.5.2.1 `const char* ah_error (void)`

Retrieve the most recent Harmony API error string.

Returns

Returns a pointer to a string that describes the latest Harmony error, or `NULL` if no error has occurred since the last call to `ah_error_clear()`.

Note

The buffer which stores the most recent error string may be overwritten by other Harmony Client API functions. If you need the value to persist beyond the next API call, you must make a copy.

11 File Documentation

11.1 hclient.h File Reference

Harmony client application function header.

Functions

- `hdesc_t * ah_alloc` (void)
Allocate and initialize a new Harmony descriptor.
- `int ah_args` (hdesc_t *hdesc, int *argc, char **argv)
Find configuration directives in the command-line arguments.
- `int ah_id` (hdesc_t *hdesc, const char *id)
Assign an identifying string to this client.
- `int ah_connect` (hdesc_t *hdesc, const char *host, int port)
Establish a connection with a Harmony tuning session.
- `int ah_close` (hdesc_t *hdesc)
Close the connection to a Harmony tuning session.
- `void ah_free` (hdesc_t *hdesc)
Release resources associated with a Harmony client descriptor.
- `hdef_t * ah_def_alloc` (void)
Generate an empty search definition.
- `hdef_t * ah_def_load` (const char *filename)
Generate a tuning session description from a file.
- `int ah_def_name` (hdef_t *hdef, const char *name)
Generate an empty search definition.
- `int ah_def_int` (hdef_t *hdef, const char *name, long min, long max, long step, long *ptr)
Add an integer-domain variable to the search definition.
- `int ah_def_real` (hdef_t *hdef, const char *name, double min, double max, double step, double *ptr)
Add a real-domain variable to the search definition.
- `int ah_def_enum` (hdef_t *hdef, const char *name, const char **ptr)
Add an enumerated-domain variable to the search definition.
- `int ah_def_enum_value` (hdef_t *hdef, const char *name, const char *value)
Append a value to an enumerated-domain variable.
- `int ah_def_strategy` (hdef_t *hdef, const char *strategy)
Specify the strategy to use for this search.
- `int ah_def_layers` (hdef_t *hdef, const char *list)
Specify the list of plug-ins to use for this search.
- `int ah_def_cfg` (hdef_t *hdef, const char *key, const char *val)
Modify the initial configuration of a new Harmony search.
- `void ah_def_free` (hdef_t *hdef)
Release the resources used by this definition descriptor.
- `htask_t * ah_start` (hdesc_t *hdesc, hdef_t *hdef)
Start a new Harmony tuning search task.
- `htask_t * ah_join` (hdesc_t *hdesc, const char *name)
Join an established Harmony tuning search task.
- `int ah_pause` (htask_t *htask)

- Pause a tuning search task.*

 - int `ah_resume` (htask_t *htask)
- Resume a tuning search task.*

 - int `ah_restart` (htask_t *htask)
- Resume a tuning search task.*

 - int `ah_leave` (htask_t *htask)
- Leave a tuning search task.*

 - int `ah_kill` (htask_t *htask)
- Kill a tuning search task.*

 - int `ah_bind_int` (htask_t *htask, const char *name, long *ptr)
- Bind a client address to an integer-domain search variable (dimension).*

 - int `ah_bind_real` (htask_t *htask, const char *name, double *ptr)
- Bind a client address to a real-domain search variable (dimension).*

 - int `ah_bind_enum` (htask_t *htask, const char *name, const char **ptr)
- Bind a client address to an enumerated string-based search variable (dimension).*

 - long `ah_get_int` (htask_t *htask, const char *name)
- Return the current value of an integer-domain search variable.*

 - double `ah_get_real` (htask_t *htask, const char *name)
- Return the current value of a real-domain search variable.*

 - const char * `ah_get_enum` (htask_t *htask, const char *name)
- Return the current value of an enumerated-domain search variable.*

 - const char * `ah_get_cfg` (htask_t *htask, const char *key)
- Get a key value from the search's configuration.*

 - const char * `ah_set_cfg` (htask_t *htask, const char *key, const char *val)
- Set a new key/value pair in the search's configuration.*

 - int `ah_fetch` (htask_t *htask)
- Fetch a new configuration from the Harmony search.*

 - int `ah_report` (htask_t *htask, double *perf)
- Report the performance of a configuration to the Harmony search.*

 - int `ah_report_one` (htask_t *htask, int index, double value)
- Report a single performance value for the current configuration.*

 - int `ah_best` (htask_t *htask)
- Set values under Harmony's control to the best known configuration.*

 - int `ah_converged` (htask_t *htask)
- Retrieve the convergence state of the current search.*

 - const char * `ah_error` (void)
- Retrieve the most recent Harmony API error string.*

 - void `ah_error_clear` (void)
- Clear the current task-level error string.*

11.1.1 Detailed Description

Harmony client application function header.

All clients must include this file to participate in a Harmony tuning session.

Index

- ah_alloc
 - Harmony Descriptor Management Functions, 31
 - ah_args
 - Harmony Descriptor Management Functions, 31
 - ah_best
 - Search Task Interaction Functions, 42
 - ah_bind_enum
 - Search Task Interaction Functions, 43
 - ah_bind_int
 - Search Task Interaction Functions, 43
 - ah_bind_real
 - Search Task Interaction Functions, 43
 - ah_close
 - Harmony Descriptor Management Functions, 32
 - ah_connect
 - Harmony Descriptor Management Functions, 32
 - ah_converged
 - Search Task Interaction Functions, 44
 - ah_def_alloc
 - Tuning Search Definition Functions, 34
 - ah_def_cfg
 - Tuning Search Definition Functions, 34
 - ah_def_enum
 - Tuning Search Definition Functions, 35
 - ah_def_enum_value
 - Tuning Search Definition Functions, 35
 - ah_def_free
 - Tuning Search Definition Functions, 35
 - ah_def_int
 - Tuning Search Definition Functions, 35
 - ah_def_layers
 - Tuning Search Definition Functions, 36
 - ah_def_load
 - Tuning Search Definition Functions, 36
 - ah_def_name
 - Tuning Search Definition Functions, 37
 - ah_def_real
 - Tuning Search Definition Functions, 37
 - ah_def_strategy
 - Tuning Search Definition Functions, 38
 - ah_error
 - Harmony Error Reporting Functions, 47
 - ah_fetch
 - Search Task Interaction Functions, 44
 - ah_free
 - Harmony Descriptor Management Functions, 32
 - ah_get_cfg
 - Search Task Interaction Functions, 44
 - ah_get_enum
 - Search Task Interaction Functions, 44
 - ah_get_int
 - Search Task Interaction Functions, 45
 - ah_get_real
 - Search Task Interaction Functions, 45
 - ah_id
 - Harmony Descriptor Management Functions, 33
 - ah_join
 - Tuning Search Task Control Functions, 39
 - ah_kill
 - Tuning Search Task Control Functions, 39
 - ah_leave
 - Tuning Search Task Control Functions, 40
 - ah_pause
 - Tuning Search Task Control Functions, 40
 - ah_report
 - Search Task Interaction Functions, 45
 - ah_report_one
 - Search Task Interaction Functions, 46
 - ah_restart
 - Tuning Search Task Control Functions, 40
 - ah_resume
 - Tuning Search Task Control Functions, 41
 - ah_set_cfg
 - Search Task Interaction Functions, 46
 - ah_start
 - Tuning Search Task Control Functions, 41
- Harmony Descriptor Management Functions, 31
- ah_alloc, 31
 - ah_args, 31
 - ah_close, 32
 - ah_connect, 32
 - ah_free, 32
 - ah_id, 33
- Harmony Error Reporting Functions, 47
- ah_error, 47
- hclient.h, 48
- Search Task Interaction Functions, 42
- ah_best, 42
 - ah_bind_enum, 43
 - ah_bind_int, 43
 - ah_bind_real, 43
 - ah_converged, 44
 - ah_fetch, 44
 - ah_get_cfg, 44
 - ah_get_enum, 44
 - ah_get_int, 45
 - ah_get_real, 45
 - ah_report, 45
 - ah_report_one, 46
 - ah_set_cfg, 46

Tuning Search Definition Functions, [34](#)

- [ah_def_alloc, 34](#)
- [ah_def_cfg, 34](#)
- [ah_def_enum, 35](#)
- [ah_def_enum_value, 35](#)
- [ah_def_free, 35](#)
- [ah_def_int, 35](#)
- [ah_def_layers, 36](#)
- [ah_def_load, 36](#)
- [ah_def_name, 37](#)
- [ah_def_real, 37](#)
- [ah_def_strategy, 38](#)

Tuning Search Task Control Functions, [39](#)

- [ah_join, 39](#)
- [ah_kill, 39](#)
- [ah_leave, 40](#)
- [ah_pause, 40](#)
- [ah_restart, 40](#)
- [ah_resume, 41](#)
- [ah_start, 41](#)