Active Harmony User's Guide

4.5

Generated by Doxygen 1.8.1.1

Mon Sep 30 2013 23:07:08

Contents

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<=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<=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.

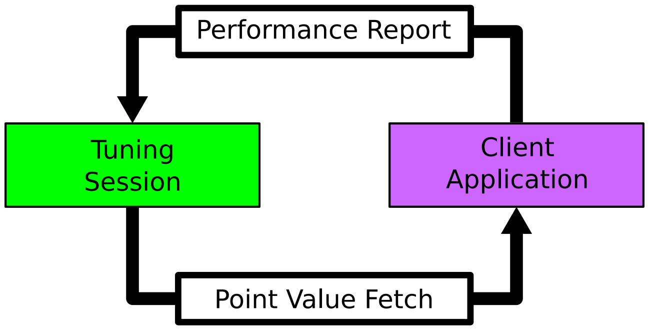


Figure 1: The generalized auto-tuning feedback loop.

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.

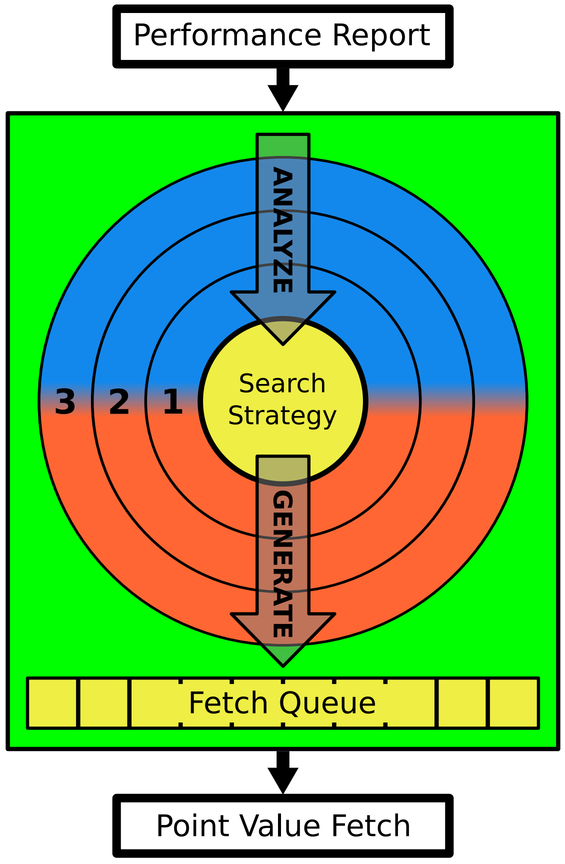


Figure 2: Detailed view of the Active Harmony tuning session.

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-4.5
```

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-4.5
```

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 `example_c` 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:

```
$ ./example_c
```

This should produce the output similar to the following:

```
Starting Harmony...
68, 51, 51, 51, 51, 51 = 5464
48, 68, 51, 51, 51, 51 = 4747
48, 47, 67, 51, 51, 51 = 4040
48, 47, 46, 67, 51, 51 = 3525
48, 47, 46, 45, 66, 51 = 3206

<... snip ...>

9, 35, 49, 53, 79, 75 = 367
9, 36, 47, 53, 77, 74 = 385
9, 34, 48, 53, 76, 74 = 367
8, 32, 45, 56, 77, 74 = 329
9, 32, 48, 53, 77, 75 = 327
```

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 [configuration_file]
```

The `HARMONY_CONFIG` environment variable can also be used to specify which configuration file to use. For example (assuming Bourne shell semantics):

```
$ HARMONY_CONFIG=/some/other/dir/harmony.cfg ./hserver
```

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

```
$ HARMONY_S_HOST=localhost ./example_c
```

If `HARMONY_S_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_S_PORT` environment variable. For example:

```
$ export HARMONY_S_HOST=other.host.net
$ export HARMONY_S_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 SESSION_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:

```
$ ./example_c SESSION_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:

```
$ ./example_c SESSION_LAYER_LIST=log.so LOGFILE=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:

```
$ ./example_c SESSION_LAYER_LIST=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:

```
$ ./example_c SESSION_LAYER_LIST=agg.so:log.so \
              LOGFILE=search.log AGG_TIMES=5 AGG_FUNC=median
```

and

```
$ ./example_c SESSION_LAYER_LIST=log.so:agg.so \
              LOGFILE=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 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.

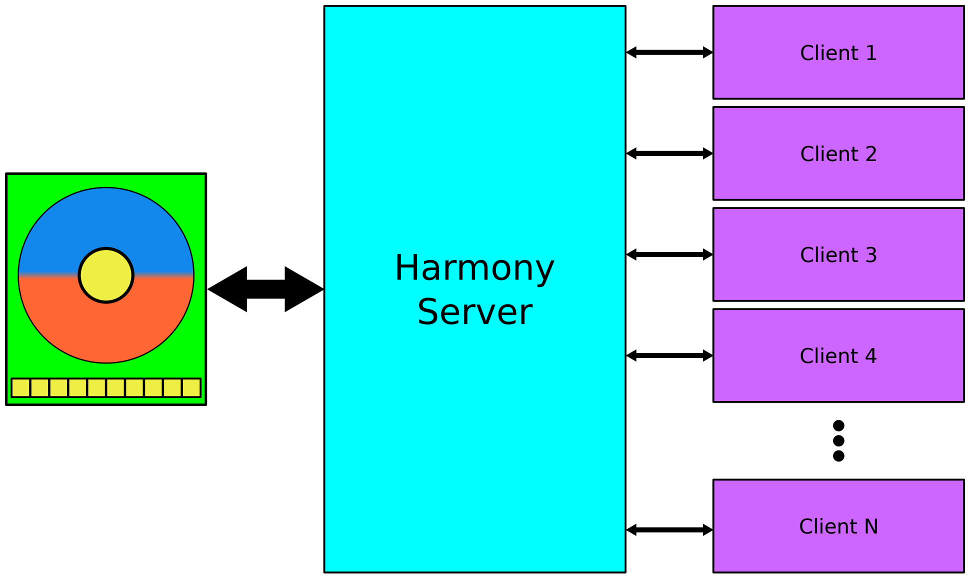


Figure 3: Supporting multiple clients within the feedback loop.

Usage Syntax

```
hserver [config_file_name]
```

The server has no mandatory parameters, and can be started with a plain invocation. A configuration file may optionally be provided as a parameter, otherwise `harmony.cfg` will be used (if it exists). The configuration system provides a unified key/value store that may be queried by all members of the Active Harmony framework. See the Configuration section for more details.

Harmony Server Configuration Variables

Key	Type	Default	Description
SERVER_PORT	Integer	1979	The port that the Harmony Server will bind and listen to. Read only once when the Harmony Server is loaded.

Any configuration directives loaded by the server will be automatically merged with the sessions that it manages. If there is a key conflict between the server and session configuration environment, the session's key takes precedence.

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_S_HOST	Hostname of the machine running the Harmony Server.
HARMONY_S_PORT	TCP/IP port allocated by Harmony Server.

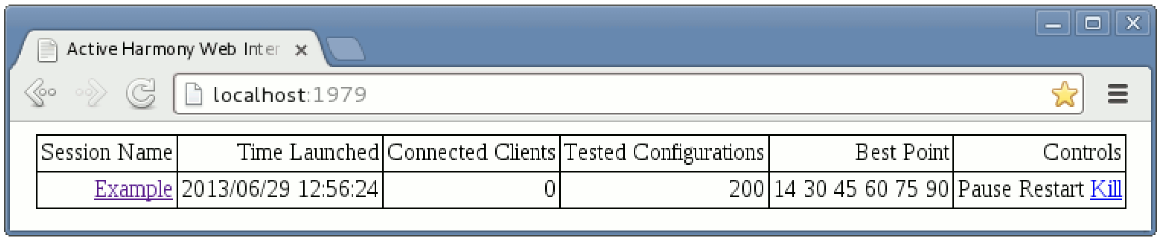
When defined, these environment variables instruct clients to connect to the specified hostname:port pair instead of spawning a local tuning session. Multiple clients may then work together on a single search problem.

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
```

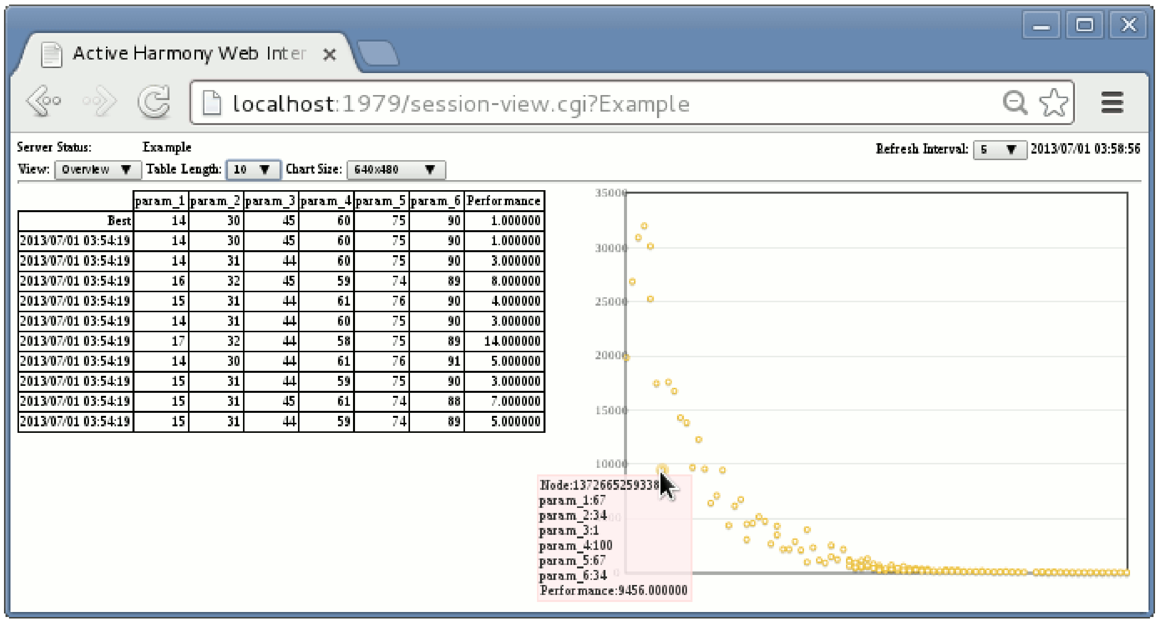
Replace the host or port to match the desired Harmony Server as necessary. A screen similar to the following should appear:



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:



Server Status: Example
View: Overview Table Length: 10 Chart Size: 640x480
Refresh Interval: 5 2013/07/01 03:58:56

	param_1	param_2	param_3	param_4	param_5	param_6	Performance
Best	14	30	45	60	75	90	1.000000
2013/07/01 03:54:19	14	30	45	60	75	90	1.000000
2013/07/01 03:54:19	14	31	44	60	75	90	3.000000
2013/07/01 03:54:19	16	32	45	59	74	89	8.000000
2013/07/01 03:54:19	15	31	44	61	76	90	4.000000
2013/07/01 03:54:19	14	31	44	60	75	90	3.000000
2013/07/01 03:54:19	17	32	44	58	75	89	14.000000
2013/07/01 03:54:19	14	30	44	61	76	91	5.000000
2013/07/01 03:54:19	15	31	44	59	75	90	3.000000
2013/07/01 03:54:19	15	31	45	61	74	88	7.000000
2013/07/01 03:54:19	15	31	44	59	74	89	5.000000

```
Node:1372665259338
param_1:67
param_2:34
param_3:1
param_4:100
param_5:67
param_6:34
Performance:9456.000000
```

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.

4.2 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
`-i=<name>,<min>,<max>,<step>`	Declare an integer number variable called *name* where valid values fall between *min* and *max* with a stride of size *step*.
`-r=<name>,<min>,<max>,<step>`	Declare a real number variable called *name* where valid values fall between *min* and *max* with a stride of size *step*.
`-e=<name>,<val_1>,..,<val_n>`	Declare an enumerated variable called *name* whose values must be *val_1* or *val_2* or .. or *val_n*.

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
`-m=<metric>`	Specify how to measure the performance of the child process, where *metric* is: **wall** for wall time (*default*), **user** for user CPU time, **sys** for system CPU time, or **output** to parse a number from the final line of child output.

`-q`	Suppress client application output.
`-v`	Print additional informational output.
`-n=<num>`	Run child program at most *num* 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.

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 harmony_setcfg() function calls. After a session has been launched, clients may query the configuration environment through harmony_getcfg() and harmony_setcfg() 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
SESSION_STRATEGY	String	pro.so	The search strategy to use for a particular session.
SESSION_LAYER_LIST	String	[none]	The processing layers to use for a particular session.
CLIENT_COUNT	Integer	1	The number of expected clients.
PER_CLIENT_STORAGE	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_S_HOST	If defined, tuning clients (using harmony_join()) will attempt to connect to a Harmony Server on this host.
HARMONY_S_PORT	If defined (along with HARMONY_S_HOST), tuning clients will used 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 harmony_strategy() and harmony_layer_list() 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.

Configuration Variables

Key	Type	Default	Description
PASSES	Integer	1	Number of passes through the search space before the search is considered converged.

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.

Configuration Variables

Key	Type	Default	Description
RANDOM_SEED	Integer	time()	Value to seed the pseudo-random number generator. Default is to seed the random generator by time.

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

Configuration Variables

Key	Type	Default	Description
SIMPLEX_SIZE	Integer	N+1	Number of vertices in the simplex. Defaults to the number of tuning variables + 1.
RANDOM_SEED	Integer	time()	Value to seed the pseudo-random number generator. Default is to seed the random generator by time.
INIT_METHOD	String	point	Initial simplex generation method. Valid values are "point", "point_fast", and "random" (without quotes).
INIT_PERCENT	Real	0.35	Initial simplex size as a percentage of the total search space. Only for "point" and "point_fast" initial simplex methods.
REJECT_METHOD	String	penalty	How to choose a replacement when dealing with rejected points: **Penalty** - Use this method if the chance of point rejection is relatively low. It applies an infinite penalty factor for invalid points, allowing the Nelder-Mead algorithm to select a sensible next point. However, if the entire simplex is comprised of invalid points, an infinite loop of invalid points may occur. **Random** - Use this method if the chance of point rejection is high. It reduces the risk of infinitely selecting invalid points at the cost of increasing the risk of deforming the simplex.
REFLECT	Real	1.0	Multiplicative coefficient for simplex reflection step.
EXPAND	Real	2.0	Multiplicative coefficient for simplex expansion step.
SHRINK	Real	0.5	Multiplicative coefficient for simplex shrink step.
FVAL_TOL	Real	0.0001	Convergence test succeeds if difference between all vertex performance values fall below this value.
SIZE_TOL	Real	0.05∗r	Convergence test succeeds if simplex size falls below this value. Default is 5% of the initial simplex radius.

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.

Configuration Variables

Key	Type	Default	Description
SIMPLEX_SIZE	Integer	N+1	Number of vertices in the simplex. Defaults to the number of tuning variables + 1.
RANDOM_SEED	Integer	time()	Value to seed the pseudo-random number generator. Default is to seed the random generator by time.
INIT_METHOD	String	point	Initial simplex generation method. Valid values are "point", "point_fast", and "random" (without quotes).
INIT_PERCENT	Real	0.35	Initial simplex size as a percentage of the total search space. Only for "point" and "point_fast" initial simplex methods.
REJECT_METHOD	String	penalty	How to choose a replacement when dealing with rejected points: **Penalty** - Use this method if the chance of point rejection is relatively low. It applies an infinite penalty factor for invalid points, allowing the PRO algorithm to select a sensible next point. However, if the entire simplex is comprised of invalid points, an infinite loop of invalid points may occur. **Random** - Use this method if the chance of point rejection is high. It reduces the risk of infinitely selecting invalid points at the cost of increasing the risk of deforming the simplex.
REFLECT	Real	1.0	Multiplicative coefficient for simplex reflection step.
EXPAND	Real	2.0	Multiplicative coefficient for simplex expansion step.
SHRINK	Real	0.5	Multiplicative coefficient for simplex shrink step.
FVAL_TOL	Real	0.0001	Convergence test succeeds if difference between all vertex performance values fall below this value.
SIZE_TOL	Real	0.05∗r	Convergence test succeeds if simplex size falls below this value. Default is 5% of the initial simplex radius.

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.

Configuration Variables

Key	Type	Default	Description
AGG_FUNC	String	[none]	Aggregation function to use. Valid values are "min", "max", "mean", and "median" (without quotes).
AGG_TIMES	Integer	[none]	Number of performance values to collect before performing the aggregation function.

6.2.2 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
```

Configuration Variables

Key	Type	Default	Description
SERVER_URL	URL	[none]	Destination of messages from session to code server.
TARGET_URL	URL	[none]	Destination of binary files from code server to target application.
REPLY_URL	URL	[tmpdir]	Destination of reply messages from code server to session. A reasonable directory in /tmp will be generated by default if left blank.
TMPDIR	String	/tmp	Session local path to store temporary files.
SLAVE_LIST	String	[none]	Comma separated list of [`host n`] pairs, where `n` slaves will run on `host`. For example `bunker.cs.umd.edu 4, nexcor.cs.umd.edu 4` will instruct the code server to spawn 8 total slaves between two machines.
SLAVE_PATH	String	[none]	Path on slave host to store generated binaries before being sent to the target application.

6.2.3 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.

Configuration Variables

Key	Type	Default	Description
LOGFILE	String	(none)	Name of point/performance log file.
LOGFILE_MODE	String	a	Mode to use with `fopen()`. Valid values are "a" and "w" (without quotes).

6.2.4 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/`.

Configuration Variables

Key	Type	Default	Description
OC_BIN	Filename	oc	Location of the Omega Calculator binary. The `PATH` environment variable will be searched if not found initially.
CONSTRAINTS	String	[none]	Constraint statements to be used during this session. This variable has precedence over `CONSTRAINT_FILE`.
CONSTRAINT_FILE	Filename	[none]	If the `CONSTRAINTS` variable is not specified, they will be loaded from this file.
QUIET	Boolean	0	Bounds suggestion and rejection messages can be suppressed by setting this variable to 1.

Note

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.

6.2.5 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.

Configuration Variables

Key	Type	Default	Description
TAUDB_NAME	String	[none]	Name of the PostgreSQL database.
TAUDB_STORE_METHOD	String	one_time	Determines when statistics are computed: **one_time** - With each database write. **real_time** - At session cleanup time.
TAUDB_STORE_NUM	Integer	0	Number of reports to cache before writing to database.
CLIENT_COUNT	Integer	1	Number of participating tuning clients.

6.2.6 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.

Configuration Variables

Key	Type	Default	Description
XML_FILENAME	String	[none]	XML output file.

7 Coding Examples

Active Harmony provides two separate interfaces for the general user. One for establishing tuning sessions and the other for interacting with existing tuning sessions.

7.1 Launching a New Session

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

Code similar to `launch_tuner()` should be added to any target application that must establish a new tuning session for itself (as opposed to connecting to 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"

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

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

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

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

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

    /* 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.
     */
    harmony_real(hdesc, "realVar", 0.0, 0.5, 0.0);

    /* Add a string-valued variable called "strVar" to the session.
       Its value may be "apples", "oranges", "peaches", or "pears".
     */
    harmony_enum(hdesc, "strVar", "apples");
    harmony_enum(hdesc, "strVar", "oranges");
    harmony_enum(hdesc, "strVar", "peaches");
    harmony_enum(hdesc, "strVar", "pears");

    /* Connect to a Harmony Server at the default host ("localhost"),
       on the default port (1979), and initiate the tuning session
       we've described using the above API calls.
     */
    if (harmony_launch(hdesc, NULL, 0) != 0) {
        fprintf(stderr, "Could not launch tuning session: %s\n",
                harmony_error_string(hdesc));
        return -1;
    }

    return 0;
}
```

7.2 Advanced Session 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:

- Nelder-Mead Search Strategy
- Point Logger Processing Layer
- Aggregator Processing Layer

The following code snippet extends the code example in the session launch example. It may be inserted at any point before the call to `harmony_launch()`.

Note

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

```
    /* Use the Nelder-Mead simplex-based as the strategy for this
       session, instead of the default Parallel Rank Order.
     */
    harmony_strategy(hdesc, "nm.so");

    /* Instruct the strategy to use an initial simplex roughly half
       the size of the search space.
     */
    harmony_setcfg(hdesc, "INIT_PERCENT", "0.50");

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

    /* Instruct the logger to use /tmp/tuning.run as the logfile. */
    harmony_setcfg(hdesc, "LOGFILE", "/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.
     */
    harmony_setcfg(hdesc, "AGG_TIMES", "10");
    harmony_setcfg(hdesc, "AGG_FUNC", "median");
```

7.3 Using the Client API

The example demonstrates how to use the client API to join, fetch, and report to the tuning session established in the session launch example.

If this client was required to establish its own tuning session, a call to launch_tuner() from the example above could be added prior to the call to harmony_init() on line 15.

Note

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

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

    /* Variables to hold the application's runtime tunable parameters.
       Once bound to a Harmony tuning session, these variables will be
       modified upon harmony_fetch() to a new testing configuration.
```

```
     */
    long        var1;
    long        var2;
    double      var3;
    const char *var4;

    /* Initialize the Harmony client descriptor. */
    hdesc = harmony_init();

    /* Bind the session variables to local variables. */
    harmony_bind_int(hdesc,  "intVar1", &var1);
    harmony_bind_int(hdesc,  "intVar2", &var2);
    harmony_bind_real(hdesc, "realVar", &var3);
    harmony_bind_enum(hdesc, "strVar",  &var4);

    /* Join the tuning session we established above. */
    harmony_join(hdesc, NULL, 0, "Example");

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

        /* Retrieve new values from the Harmony Session. */
        harmony_fetch(hdesc);

        /* 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. */
        harmony_report(hdesc, perf);
    }

    /* Leave the session. */
    harmony_leave(hdesc);

    return 0;
}
```

8 Module Index

8.1 Modules

Here is a list of all modules:

9 File Index

9.1 File List

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

10 Module Documentation

10.1 Harmony Descriptor Management Functions

Functions

- hdesc_t ∗ harmony_init (int ∗argc, char ∗∗∗argv)

  Allocate and initialize a new Harmony descriptor.
- void harmony_fini (hdesc_t ∗hdesc)

  Release all 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 particular tuning session. 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 void harmony_fini (hdesc_t ∗ *hdesc*)

Release all resources associated with a Harmony client descriptor.

Parameters

hdesc	Harmony descriptor returned from harmony_init().

10.1.2.2 hdesc_t∗ harmony_init (int ∗ *argc,* char ∗∗∗ *argv*)

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.

Similar to MPI's initialization function mpi_init(), harmony_init() optionally accepts the address of the two parameters to main (`argc` and `argv`). If provided, any string that resembles a Harmony configuration directive will removed from argv and saved elsewhere. Scanning ends early if a -- parameter is found. These directives are then applied immediately prior to session launch or client join, whichever comes first.

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

Parameters

argc	Address of argc parameter from main().
argv	Address of argv parameter from main().

Returns

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

10.2 Session Setup Functions

Functions

- int harmony_session_name (hdesc_t ∗hdesc, const char ∗name)

  Specify a unique name for the Harmony session.
- int harmony_int (hdesc_t ∗hdesc, const char ∗name, long min, long max, long step)

  Add an integer-domain variable to the Harmony session.
- int harmony_real (hdesc_t ∗hdesc, const char ∗name, double min, double max, double step)

  Add a real-domain variable to the Harmony session.
- int harmony_enum (hdesc_t ∗hdesc, const char ∗name, const char ∗value)

  Add an enumeration variable and append to the list of valid values in this set.
- int harmony_strategy (hdesc_t ∗hdesc, const char ∗strategy)

  Specify the search strategy to use in the new Harmony session.
- int harmony_layer_list (hdesc_t ∗hdesc, const char ∗list)

  Specify the list of plug-ins to use in the new Harmony session.
- int harmony_launch (hdesc_t ∗hdesc, const char ∗host, int port)

  Instantiate a new Harmony tuning session.

10.2.1 Detailed Description

These functions are used to describe and establish new tuning sessions. Valid sessions must define at least one tuning variable before they are launched.

10.2.2 Function Documentation

10.2.2.1 int harmony_enum (hdesc_t ∗ *hdesc*, const char ∗ *name*, const char ∗ *value*)

Add an enumeration variable and append to the list of valid values in this set.

Parameters

hdesc	Harmony descriptor returned from harmony_init().
name	Name to associate with this variable.
value	String that belongs in this enumeration.

Returns

Returns 0 on success, and -1 otherwise.

10.2.2.2 int harmony_int (hdesc_t ∗ *hdesc*, const char ∗ *name*, long *min*, long *max*, long *step*)

Add an integer-domain variable to the Harmony session.

Parameters

hdesc	Harmony descriptor returned from harmony_init().
name	Name to associate with this variable.
min	Minimum range value (inclusive).
max	Maximum range value (inclusive).
step	Minimum search increment.

Returns

Returns 0 on success, and -1 otherwise.

10.2.2.3 int harmony_launch (hdesc_t ∗ *hdesc,* const char ∗ *host,* int *port*)

Instantiate a new Harmony tuning session.

After using the session establishment routines (harmony_int, harmony_name, etc.) to specify a tuning session, this function launches a new tuning session. It may either be started locally or on the Harmony Server located at *host*:*port*.

If *host* is `NULL`, its value will be taken from the environment variable `HARMONY_S_HOST`. If `HARMONY_S_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_S_PORT`, if defined. Otherwise, its value will be taken from the src/defaults.h file, if needed.

Parameters

hdesc	Harmony descriptor returned from harmony_init().
host	Host of the Harmony server (or `NULL`).
port	Port of the Harmony server.

Returns

Returns 0 on success, and -1 otherwise.

10.2.2.4 int harmony_layer_list (hdesc_t ∗ *hdesc,* const char ∗ *list*)

Specify the list of plug-ins to use in the new Harmony session.

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

Parameters

hdesc	Harmony descriptor returned from harmony_init().
list	List of plug-ins to load with this session.

Returns

Returns 0 on success, and -1 otherwise.

10.2.2.5 int harmony_real (hdesc_t ∗ *hdesc,* const char ∗ *name,* double *min,* double *max,* double *step*)

Add a real-domain variable to the Harmony session.

Parameters

hdesc	Harmony descriptor returned from harmony_init().
name	Name to associate with this variable.
min	Minimum range value (inclusive).
max	Maximum range value (inclusive).
step	Minimum search increment.

Returns

Returns 0 on success, and -1 otherwise.

10.2.2.6 int harmony_session_name (hdesc_t * *hdesc*, const char * *name*)

Specify a unique name for the Harmony session.

Parameters

hdesc	Harmony descriptor returned from harmony_init().
name	Name to associate with this session.

Returns

Returns 0 on success, and -1 otherwise.

10.2.2.7 int harmony_strategy (hdesc_t * *hdesc*, const char * *strategy*)

Specify the search strategy to use in the new Harmony session.

Parameters

hdesc	Harmony descriptor returned from harmony_init().
strategy	Filename of the strategy plug-in to use in this session.

Returns

Returns 0 on success, and -1 otherwise.

10.3 Tuning Client Setup Functions

Functions

- int harmony_id (hdesc_t ∗hdesc, const char ∗id)

  Assign an identifying string to this client.
- int harmony_bind_int (hdesc_t ∗hdesc, const char ∗name, long ∗ptr)

  Bind a local variable of type `long` to an integer-domain session variable.
- int harmony_bind_real (hdesc_t ∗hdesc, const char ∗name, double ∗ptr)

  Bind a local variable of type `double` to a real-domain session variable.
- int harmony_bind_enum (hdesc_t ∗hdesc, const char ∗name, const char ∗∗ptr)

  Bind a local variable of type `char ∗` to an enumerated string-based session variable.
- int harmony_join (hdesc_t ∗hdesc, const char ∗host, int port, const char ∗sess)

  Join an established Harmony tuning session.
- int harmony_leave (hdesc_t ∗hdesc)

  Leave a Harmony tuning session.

10.3.1 Detailed Description

These functions prepare the client to join an established tuning session. Specifically, variables within the client application must be bound to session variables. This allows the client to change appropriately upon fetching new points from the session.

10.3.2 Function Documentation

10.3.2.1 int harmony_bind_enum (hdesc_t ∗ *hdesc,* const char ∗ *name,* const char ∗∗ *ptr*)

Bind a local variable of type `char ∗` to an enumerated string-based session variable.

This function associates a local variable with a session variable declared using harmony_enum(). Upon harmony_fetch(), the value chosen by the session will be stored at the address `ptr`.

This function must be called for each string-based variable defined in the joining session. Otherwise harmony_join() will fail when called.

Parameters

hdesc	Harmony descriptor returned from harmony_init().
name	Session variable defined using harmony_enum().
ptr	Pointer to a local `char ∗` variable that will hold the current testing value.

Returns

Returns a harmony descriptor on success, and -1 otherwise.

10.3.2.2 int harmony_bind_int (hdesc_t ∗ *hdesc,* const char ∗ *name,* long ∗ *ptr*)

Bind a local variable of type `long` to an integer-domain session variable.

This function associates a local variable with a session variable declared using harmony_int(). Upon harmony_fetch(), the value chosen by the session will be stored at the address `ptr`.

This function must be called for each integer-domain variable defined in the joining session. Otherwise harmony_join() will fail when called.

Parameters

hdesc	Harmony descriptor returned from harmony_init().
name	Session variable defined using harmony_int().
ptr	Pointer to a local `long` variable that will hold the current testing value.

Returns

Returns a harmony descriptor on success, and -1 otherwise.

10.3.2.3 int harmony_bind_real (hdesc_t ∗ *hdesc,* const char ∗ *name,* double ∗ *ptr*)

Bind a local variable of type `double` to a real-domain session variable.

This function associates a local variable with a session variable declared using harmony_real(). Upon harmony_fetch(), the value chosen by the session will be stored at the address `ptr`.

This function must be called for each real-domain variable defined in the joining session. Otherwise harmony_join() will fail when called.

Parameters

hdesc	Harmony descriptor returned from harmony_init().
name	Session variable defined using harmony_real().
ptr	Pointer to a local `double` variable that will hold the current testing value.

Returns

Returns a harmony descriptor on success, and -1 otherwise.

10.3.2.4 int harmony_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.

By default, a string is generated from the hostname, process id, and socket descriptor.

Parameters

hdesc	Harmony descriptor returned from harmony_init().
id	Unique identification string.

Returns

Returns 0 on success, and -1 otherwise.

10.3.2.5 int harmony_join (hdesc_t ∗ *hdesc,* const char ∗ *host,* int *port,* const char ∗ *sess*)

Join an established Harmony tuning session.

Establishes a connection with the Harmony Server on a specific host and port, and joins the named session. All variables must be bound to local memory via harmony_bind() for this call to succeed.

If `host` is `NULL` or `port` is 0, values from the environment variable `HARMONY_S_HOST` or `HARMONY_S_PORT` will be used, respectively. If either environment variable is not defined, values from defaults.h will be used as a last resort.

Parameters

hdesc	Harmony descriptor returned from harmony_init().
host	Host of the Harmony server.
port	Port of the Harmony server.
sess	Name of an existing tuning session on the server.

Returns

Returns 0 on success, and -1 otherwise.

10.3.2.6 int harmony_leave (hdesc_t ∗ *hdesc*)

Leave a Harmony tuning session.

End participation in a Harmony tuning session by closing the connection to the Harmony server.

Parameters

hdesc	Harmony descriptor returned from harmony_init().

Returns

Returns 0 on success, and -1 otherwise.

10.4 Client/Session Interaction Functions

Functions

- char ∗ harmony_getcfg (hdesc_t ∗hdesc, const char ∗key)

    Get a key value from the session's configuration.
- char ∗ harmony_setcfg (hdesc_t ∗hdesc, const char ∗key, const char ∗val)

    Set a new key/value pair in the session's configuration.
- int harmony_fetch (hdesc_t ∗hdesc)

    Fetch a new configuration from the Harmony server.
- int harmony_report (hdesc_t ∗hdesc, double value)

    Report the performance of a configuration to the Harmony server.
- int harmony_best (hdesc_t ∗hdesc)

    Sets variables under Harmony's control to the best known configuration.
- int harmony_converged (hdesc_t ∗hdesc)

    Retrieve the convergence state of the current search.
- const char ∗ harmony_error_string (hdesc_t ∗hdesc)

    Access the current Harmony error string.
- void harmony_error_clear (hdesc_t ∗hdesc)

    Clears the error status of the given Harmony descriptor.

10.4.1 Detailed Description

These functions are used by the client after it has joined an established session.

10.4.2 Function Documentation

10.4.2.1 int harmony_best (hdesc_t ∗ *hdesc*)

Sets variables under Harmony's control to the best known configuration.

Parameters

hdesc	Harmony descriptor returned from harmony_init().

Returns

Returns 0 on success, and -1 otherwise.

10.4.2.2 int harmony_converged (hdesc_t ∗ *hdesc*)

Retrieve the convergence state of the current search.

Parameters

hdesc	Harmony descriptor returned from harmony_init().

Returns

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

10.4.2.3 void harmony_error_clear (hdesc_t ∗ *hdesc*)

Clears the error status of the given Harmony descriptor.

Parameters

hdesc	Harmony descriptor returned from harmony_init().

10.4.2.4 const char∗ harmony_error_string (hdesc_t ∗ *hdesc*)

Access the current Harmony error string.

Parameters

hdesc	Harmony descriptor returned from harmony_init().

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 harmony_error_clear().

10.4.2.5 int harmony_fetch (hdesc_t ∗ *hdesc*)

Fetch a new configuration from the Harmony server.

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

hdesc	Harmony descriptor returned from harmony_init().

Returns

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

10.4.2.6 char∗ harmony_getcfg (hdesc_t ∗ *hdesc,* const char ∗ *key*)

Get a key value from the session's configuration.

Searches the server's configuration system for key, and returns the string value associated with it if found. Heap memory is allocated for the result string.

Warning

This function allocates memory for the return value. It is the user's responsibility to free this memory.

Parameters

hdesc	Harmony descriptor returned from harmony_init().
key	Config key to search for on the server.

Returns

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

10.4.2.7 int harmony_report (hdesc_t ∗ *hdesc*, double *value*)

Report the performance of a configuration to the Harmony server.

Parameters

hdesc	Harmony descriptor returned from harmony_init().
value	Performance measured for the current configuration.

Returns

Returns 0 on success, and -1 otherwise.

10.4.2.8 char∗ harmony_setcfg (hdesc_t ∗ *hdesc*, const char ∗ *key*, const char ∗ *val*)

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

Writes the new key/value pair into the server'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. These key/value pairs exist only in memory, and will not be written back to the server's configuration file.

Warning

This function allocates memory for the return value. It is the user's responsibility to free this memory.

Parameters

hdesc	Harmony descriptor returned from harmony_init().
key	Config key to modify on the server.
val	Config value to associate with the key.

Returns

Returns the original key value string on success and `NULL` otherwise, setting errno if appropriate. Since this function may legitimately return `NULL`, errno must be cleared pre-call, and checked post-call.

11 File Documentation

11.1 hclient.h File Reference

Harmony client application function header.

Functions

- hdesc_t ∗ harmony_init (int ∗argc, char ∗∗∗argv)

  Allocate and initialize a new Harmony descriptor.
- void harmony_fini (hdesc_t ∗hdesc)

  Release all resources associated with a Harmony client descriptor.
- int harmony_session_name (hdesc_t ∗hdesc, const char ∗name)

  Specify a unique name for the Harmony session.
- int harmony_int (hdesc_t ∗hdesc, const char ∗name, long min, long max, long step)

  Add an integer-domain variable to the Harmony session.
- int harmony_real (hdesc_t ∗hdesc, const char ∗name, double min, double max, double step)

  Add a real-domain variable to the Harmony session.
- int harmony_enum (hdesc_t ∗hdesc, const char ∗name, const char ∗value)

  Add an enumeration variable and append to the list of valid values in this set.
- int harmony_strategy (hdesc_t ∗hdesc, const char ∗strategy)

  Specify the search strategy to use in the new Harmony session.
- int harmony_layer_list (hdesc_t ∗hdesc, const char ∗list)

  Specify the list of plug-ins to use in the new Harmony session.
- int harmony_launch (hdesc_t ∗hdesc, const char ∗host, int port)

  Instantiate a new Harmony tuning session.
- int harmony_id (hdesc_t ∗hdesc, const char ∗id)

  Assign an identifying string to this client.
- int harmony_bind_int (hdesc_t ∗hdesc, const char ∗name, long ∗ptr)

  Bind a local variable of type `long` to an integer-domain session variable.
- int harmony_bind_real (hdesc_t ∗hdesc, const char ∗name, double ∗ptr)

  Bind a local variable of type `double` to a real-domain session variable.
- int harmony_bind_enum (hdesc_t ∗hdesc, const char ∗name, const char ∗∗ptr)

  Bind a local variable of type `char *` to an enumerated string-based session variable.
- int harmony_join (hdesc_t ∗hdesc, const char ∗host, int port, const char ∗sess)

  Join an established Harmony tuning session.
- int harmony_leave (hdesc_t ∗hdesc)

  Leave a Harmony tuning session.
- char ∗ harmony_getcfg (hdesc_t ∗hdesc, const char ∗key)

  Get a key value from the session's configuration.
- char ∗ harmony_setcfg (hdesc_t ∗hdesc, const char ∗key, const char ∗val)

  Set a new key/value pair in the session's configuration.
- int harmony_fetch (hdesc_t ∗hdesc)

  Fetch a new configuration from the Harmony server.
- int harmony_report (hdesc_t ∗hdesc, double value)

  Report the performance of a configuration to the Harmony server.
- int harmony_best (hdesc_t ∗hdesc)

Sets variables under Harmony's control to the best known configuration.

- int harmony_converged (hdesc_t ∗hdesc)

  Retrieve the convergence state of the current search.
- const char ∗ harmony_error_string (hdesc_t ∗hdesc)

  Access the current Harmony error string.
- void harmony_error_clear (hdesc_t ∗hdesc)

  Clears the error status of the given Harmony descriptor.

11.1.1 Detailed Description

Harmony client application function header. All clients must include this file to participate in a Harmony tuning session.

Index