HITACHI
Inspire the Next

Programming ivy - reference

June 6, 2016

Allart Ian Vogelesang ian.vogelesang@hds.com +1 408 396 6511

Hitachi Data Systems

Two sections in this material

HITACHI
Inspire the Next

1. ivyscript programming language wrapper – for scripting test workflow
   - Similar to a subset of C/C++, with some minor differences.
   - Extensible - parser auto-generated from language grammar. (flex+bison)
   - Longer term - examine pros/cons of making part 2 a CLI that you could use in your favourite scripting language, if so, the current ivyscript programming language wrapper would be dropped.
     - Current idea for this is to transform ivy engine control statements into RESTful API calls in the context of "sessions".
2. ivy engine control statements
   - Operating the underlying ivy engine

HITACHI
Inspire the Next

The ivyscript programming language wrapper

Hitachi Data Systems

ivyscript programming language

HITACHI
Inspire the Next

- Statements in the programming language end with a semi-colon, like C/C++/java.
- C style comments are supported
  - The part from /* to */ is ignored
- C++ style comments are supported
  - From // to the end of the line is ignored.

Nested blocks

HITACHI
Inspire the Next

- Anywhere you can put a statement, you can put a nested block, which starts with "`{`" and ends with "`}`".
- Any variable or function declarations made inside a nested block are not "visible" to code outside the nested block.
- Nested blocks are typically used in `if/then/else` statements, looping constructs, etc.

Types

HITACHI
Inspire the Next

- There are 3 types: `int`, `double`, and `string`.
- Examples of constants, also called literals:

```
int:      0          -5      12345
double:   5.         .5      5.5       5E-2
string:   "house"              ""
```

  - There is also a hex form of `int` literal (constant)
    - `0x0` to `0x7FFFFFFF`
    - The hex form of `int` literal only supports non-negative values.

More on string literals

HITACHI
Inspire the Next

- To include a double quote character in a string constant, escape it with a backslash:
  - `"the word \"house\" is double-quoted"`
- Other escaped characters: `\r, \n, \t`
- An escaped octal character value has 3 digits, e.g. `\001`
- An escaped hex character value has one or two digits, e.g. `\xf` or `\x0f`

Identifiers

HITACHI
Inspire the Next

- "identifiers" are eligible to serve as the name of a variable or function.
- An identifier begins with an alphabetic character (a letter), and continues with letters, digits, and underscore _ characters.

Statement – variable declaration

HITACHI
Inspire the Next

- <type> <list of identifiers with optional = <initializer expression>>;
- Examples:
- ```
  int i, j;
  int k = -1;
  double c;
  double d = 1.5;
  string s;
  string name = "bert";
  ```
  ```

Expressions

HITACHI
Inspire the Next

- A constant (a literal of one of the types) is an expression
  - e.g. `"constant"`
- A variable reference is an expression
  - e.g. `x`
- Expressions may be combined together with operators, which operate the same as in C/C++:
  - +, -, *, /, %, >, <, >=, <=, ==, !=, =, |, &, ^, &&, ||
- Expressions have a type, `int`, `double`, or `string`

Converting an expression to a different type

HITACHI
Inspire the Next

- int(<expression>)
  double(<expression>)
  string(<expression>)
- Some times called a "cast".
- The expression is evaluated and the result is converted to the target type.
- Can result in a run time error
  - E.g. evaluating `int("cow")` would cause a run-time error.

Operators - arithmetic

HITACHI
Inspire the Next

- `+` plus – for numbers, adds, for strings, concatenates
  `-` minus
  `*` multiply
  `/` divide
  `%` remainder from integer division

Logical operators - comparison

HITACHI
Inspire the Next

- `>` greater than
  `<` less than
  `>=` greater than or equal to
  `<=` less than or equal to
  `==` equal to
  `!=` not equal to

- There is no true/false type. Logical operators evaluate to an integer value just like the old C language before there was a `bool` type.
  - An `int` or a `double` value used as a logical expression means "false" if the numeric value is zero, and means "true" for any non-zero value.
  - Use of `int` logical values is transparent – it works the way you expect it to.

Bitwise or, bitwise and, bitwise exclusive or

HITACHI
Inspire the Next

- The bitwise operators operate on the individual bits in an `int` value, exactly like in C/C++.

- | bitwise or
  & bitwise and
  ^ bitwise exclusive or

Logical or, and, not

HITACHI
Inspire the Next

- These operate on logical expressions, which evaluate to an `int` interpreted to mean "false" if the `int` value is zero, "true" otherwise.
  - Like in C/C++ the second expression after the operator is not evaluated if the result is known from evaluating the first expression before the operator.
- || logical or
  - Evaluates the first expression, and if true, returns true. Otherwise, it evaluates the second expression and returns its true/false value.
- && logical and
  - Evaluates the first expression, and if false, returns false. Otherwise it evaluates the second expression and returns its true/false value.
- ! not
  - Evaluates a logical expression and returns the opposite.

Assignment expression

HITACHI
Inspire the Next

- <identifier> = <expression>
- The identifier is looked up in the symbol table at compile time, and if it's valid, at execution the expression is evaluated and the variable is set to that value.
- If the expression is not of the same type as the variable, the value may be coerced / converted, or in some cases a compile time error occurs.

Function call expression

HITACHI
Inspire the Next

- <identifier> (<comma separated list of zero or more expressions>)
  - E.g. `sin(.5)`
- Identifier and parameter list signature are looked up at compile time to and if valid, a function call is built.
- At run time, the expressions are evaluated and the resulting parameter values are passed to the function, the function is executed, and the result is returned.

Operator precedence

HITACHI
Inspire the Next

- Same as C/C++
  - `if ( 3*4+5*6 == fred || ! Person == Nancy = 4)`
  - Means
  - `if ((((3*4)+(5*6))==fred)||(!(Person==(Nancy=4))))`
- If you are not sure, group with parentheses ().

User-defined functions

HITACHI
Inspire the Next

- E.g.

```
int add_three( int i )
{
    return i+3;
};
```

Semicolon needed, unlike C/C++

- Functions have a type, which is the type of the object they return to the caller.
- Functions can be "declared" without being defined yet:

```
int add_three(int i);
```

Function overloading

HITACHI
Inspire the Next

- It’s OK to have different functions with the same name as long as the sequence of types of the parameters is different so the compiler can tell them apart.

- ```
  int    addtwo(int i)     { return i+2;}
  string addtwo(string s)  { return s + "two";}
  ```
  ```

Ivy-specific builtin functions

HITACHI
Inspire the Next

- `string outputFolderRoot();` defaults to `/scripts/ivy/ivyoutput`
- `string testName();` root part of ivyscript file without `.ivyscript` suffix
- `string masterlogfile();` you can `log(masterlogfile(), "message\n");`
- `string testFolder();` root folder for output from this run
- `string stepNNNN();` from most recent [Go!], e.g. step0002
- `string stepName();` from most recent [Go]
- `string stepFolder();` subfolder for most recent [go] within testFolder()
- `string last_result();` for most recent [Go], returns `"success"` or `"failure"`
- `string show_rollup_structure();` shows type / instance / workload thread hierarchy.

Math builtin functions – same as C/C++

HITACHI
Inspire the Next

- `double sin(double), double cos(double), double tan(double)`
- `double sinh(double), double cosh(double), double tanh(double)`
- `double asin(double), double acos(double),`
  `double atan(double), double atan2(double,double)`
- `double log(double), log10(double),`
  `double exp(double), double pow(double,double)`
- `double sqrt(double)`
- `int abs(int)` - absolute value
- `double pi(), double e()`

String builtin functions

HITACHI
Inspire the Next

- `string substring(string s, int begin_index_from_zero, int number_of_chars);`
- `string left(string s, int n);` like in BASIC, gives you leftmost / rightmost characters
  `string right(string s, int n);`
- `string trim(string s);` removes leading / trailing whitespace
- `string to_lower(string s);`
  `string to_upper(string s);`
- `int stringCaseInsensitiveEquality(string s1, string s2);`
- `string int_to_ldev(int n);` `int_to_ldev(0xFF)` returns `"00:FF"`

regex builtin functions

HITACHI
Inspire the Next

- ivy uses the default flavour of C++ std::regex, which I think uses the ECMAscript dialect

- `int regex_match(std::string s, string regex);`
  E.g. `if ( regex_match("horse","(horse)|(cow)") ) then print("animal\n");`

- `int regex_sub_match_count(string s, string regex);`

- `string regex_sub_match(string s, string regex, int n);`
  `n` must be less than `regex_sub_match_count(s, regex)`

- `int matches_digits(string s);`
  `int matches_float_number(string s);`
  `int matches_float_number_optional_trailing_percent(string s);`
  some ivy parameters can be set to these
  `int matches_identifier(string s);`
  alphabetic, continued with alphanumeric and underscores _
  `int matches_IPv4_dotted_quad(string s);`

Accessing csv files – row and column

HITACHI
Inspire the Next

Header row is row -1

Row 0 is test step 0 or subinterval 0

Use column number from 0, or say "Overall IOPS"

Test step csv files (not shown) have one line per subinterval (both host & subsystem data)

Summary csv files like this one have one line per test step.

	A	B	C	Q	R	S	AW	AX
1	Test Name	Step Number	Step Name	iogenerator type	blocksize	maxTags	Overall IOPS	Overall Decimal MB/s
2	demo9	step0000	iops_max	random_independent	4 KiB	32	2597.16	10.638
3	demo9	step0001	baseline_service_time	random_independent	4 KiB	32	25.81	0.105718
4	demo9	step0002	1.125_x_baseline	random_independent	4 KiB	32	555.849	2.27676
5	demo9	step0003	1.25_x_baseline	random_independent	4 KiB	32	1097.86	4.49682
6	demo9	step0004	1.5_x_baseline	random_independent	4 KiB	32	1486.01	6.08671
7	demo9	step0005	1.75_x_baseline	random_independent	4 KiB	32	1722.35	7.05476
8	demo9	step0006	2_x_baseline	random_independent	4 KiB	32	1898.07	7.77448
9	demo9	step0007	3_x_baseline	random_independent	4 KiB	32	2353.68	9.64067
10	demo9	step0008	4_x_baseline	random_independent	4 KiB	32	2602.77	10.6609
11	demo9	step0009	5_x_baseline	random_independent	4 KiB	32	2601.7	10.6566

AY	AZ	BA
Overall Average Blocksize (KiB)	Overall Little's Law Avg Q	Overall Average Service Time (ms)
4	64.0024	24.6432
4	0.164588	6.37691
4	4.08014	7.34037
4	8.74646	7.96686
4	14.2187	9.56836
4	19.2298	11.1649
4	24.2199	12.7603
4	45.0346	19.1337
4	63.9994	24.589
4	63.9962	24.5978

Csv file builtin functions 1/3

HITACHI
Inspire the Next

- `set_csvfile(string filename);`
  - Loads csv file into a kind of spreadsheet object, if it's not already loaded into memory.
  - You can load multiple csv files and switch back and forth.
  - All subsequent csvfile calls refer to the currently set csvfile.
- `drop_csvfile(string filename);`
  - If you are done with it and you would like to release the space.
- `int csvfile_rows();`
  - Number of rows following the header row.
  - Returns -1 if invalid file or file empty. Returns 0 if there was only a header row.
- `int csvfile_columns_in_row(int row);`
  `int csvfile_header_columns();` same as `csvfile_columns_in_row(-1)`

Csv file builtin functions 2/3 – individual cells

HITACHI
Inspire the Next

- `string csvfile_cell_value(int row, int column);`
  `string csvfile_cell_value(int row, string column_header_text);`
  - You can refer to a column using an int, the column index from zero.
  - You can refer to a column using a string, the column header text.

- `string csvfile_raw_cell_value(int row, int column);`
  `string csvfile_raw_cell_value(int row, string column_header_text);`
  - ivy "wraps" text fields as a formula with a string constant, e.g. `="horse"`
    - This stops Excel from interpreting 1-1 as January 1st, and 00:00 from interpreting as a time.
  - The csv file functions normally "unwrap" csv column values, removing this kind of wrapper or removing simple double quotes surrounding a value, to treat `="horse"`, `"horse"` and `horse` the same
  - Retrieving the raw value give you exactly what was between the commas in the csv file.

Csv file builtin functions 3/3 – headers & slices

HITACHI
Inspire the Next

- `string csvfile_column_header(int col);`
  - Give you the text of the column header

- `string csvfile_column(int col);`
  `string csvfile_column(string column_header);`
  - Gives you a "column slice" of the spreadsheet showing "raw" values.
  - E.g. `"IOPS,55,66,55,44"`
  - Demo number 8 shows iterating through the column slices to write out the transpose of a csv file.

- `string csvfile_row(int row);`
  - Gives you a "row slice" of the spreadsheet showing the "raw" values.
  - E.g. `="random_independent",="4 KiB",32,2601.7`

utility functions

HITACHI
Inspire the Next

- `string print(string), double print(double), int print(int)`
  - Prints the specified value to stdout and then returns that value.
- `int fileappend(string filename, string s)`
  - One way to write output. Does not append a newline to `s`.
- `int log(string filename, string s)`
  - Writes a timestamp prefix before the string, and adds terminating newline if the last line in `s` doesn't already have one.
  - E.g. `log(masterlogfile(),"message");`
- `trace_evaluate(int)`
  - Turns execution tracing on/off. Zero means off, otherwise on.

Builtin functions – `shell_command`

HITACHI
Inspire the Next

- `string shell_command(string)`
  - Executes the shell command and returns its output.
    - Runs as **`root`**. You have been warned.
    - Ivy runs as `root` in our lab because ivy uses ssh to fire up ivyslave and ivy_cmddev on test hosts, and "`root`" has been set up to not require a password to ssh. Ivy may also need to run as `root` to do I/O to raw LUNs – not sure.
    - The only ivy component that definitely requires to run as `root` is the SCSI Inquiry tool, which has the executable that issues "SCSI Inquiry" marked `setuid` as `root`, and thus works for any user.
  - Use `shell_command()` to do almost anything
    - `grep` in an ivy output folder to find a csv file name
    - Get a time or date stamp

Builtin functions – `exit()`

HITACHI
Inspire the Next

- As in
  - ```
    if ( last_result() != "success" )
    {
        print "timed out without making a valid measurement.\n";
        exit();
    }
    ```
    ```

Statements: expression statement

HITACHI
Inspire the Next

- <expression> ;
- Executes the expression and discards the result.

Statements – if / then / else

HITACHI
Inspire the Next

- `if ( ` <logical expression> `) then` <statement>
- `if ( ` <logical expression> `) then` <statement> `else` <statement>
- Note that the keyword "`then`" is used in ivyscript, unlike C/C++.

Statements – traditional C style for loop

HITACHI
Inspire the Next

- `for ( <initializer expression> ; <logical expression>; <epilogue expression> )`
  `<loop body statement>`

- The initializer expression is run.

- Then the logical expression is evaluated, if false, execution of the statement is complete.

- Otherwise, the loop body statement is run, then the epilogue expression is run, then we loop back to where we will evaluate the logical expression.

Example of traditional for loop

HITACHI
Inspire the Next

```
int i;
for ( i=0; i<10; i=i+1 )
{
      print( "i = " + string(i) + "\n");
}
```

- Note that it's not `for (int i=0; i<10; i++)`
  1. The initializer is an expression, not a statement, so can't declare `i` to be an `int`.
  2. There is no C++ increment operator ++.

Statement – list-style for loop

HITACHI
Inspire the Next

- For <identifier> = { <list of expressions> } statement
- E.g.

```
for i = { 0, 1, 2 }
        print("i = " + string(i) + "\n");
```

Statement – while loop

HITACHI
Inspire the Next

- `while` (<logical expression>) <loop body statement>
- The logical expression is evaluated, and if false, execution of the statement is complete.
- Otherwise, the loop body statement is executed and then we loop back to evaluating the logical expression again.

Statement – do - while loop

HITACHI
Inspire the Next

- `do` <loop body statement> `while` (<logical expression>) ;
- The loop body statement is executed, and then the logical expression is evaluated, and if the result was "false", execution of the statement is complete.
- Otherwise, and then we loop back to running the loop body statement again.

HITACHI
Inspire the Next

Operating the ivy engine

Hitachi Data Systems

The "test name"

HITACHI
Inspire the Next

- When ivy is invoked on the command line like
  - `ivy some/path/henri.ivyscript`
- The **`henri`** in `some/path/henri.ivyscript`, the part of the ivyscript filename discarding the path and the .ivyscript suffix, is called the "**test name**".
- It's used as the subfolder name off of the `[OutputFolderRoot]` folder.

"test name" – used in output filename prefixes

HITACHI
Inspire the Next

- The test name is also used as part of the prefix of ivy output filenames.
  - So that you can combine together in one folder any files from multiple ivy runs and there wouldn't be name collisions as long as the test names were different.
  - So that if you threw all the output files in one folder they would sort on nicely on filename, grouping like they were grouped in the original folders.
  - So that if all you get is the file, you still knew which folder it came from.

Statements – `[OutputFolderRoot]`

HITACHI
Inspire the Next

- `[OutputFolderRoot] <string literal>;`
  - Specifies a root folder which must already exist.
  - The default is `"."` (the current folder).
  - Specifies the root folder in which ivy will make a subfolder to record the output from running an `.ivyscript` program.
- A string literal (string constant) is required, because the output root folder name is captured at compile time.
  - This way, the output folder structure and log files can be all in place before the ivyscript program starts running.
  - At most one `[OutputFolderRoot]` statement, anywhere in your program.

Statement – [Hosts]

HITACHI
Inspire the Next

- [Hosts] <list of hosts> [Select] <select spec> ;
- Forms of specifying test hosts:
  - <string expression for ivyscript_hostname>
    - E.g. "sun159" [must look like an identifier]
  - <dotted quad - *not in quotes*>
    - E.g. 192.168.1.1
  - <starting hostname> to <ending hostname or number>
    - Shorthand for a series of hostnames with numeric suffixes.
    - E.g. "cb16" to "cb31" or just "cb16" to "31" or even "cb16" to 31

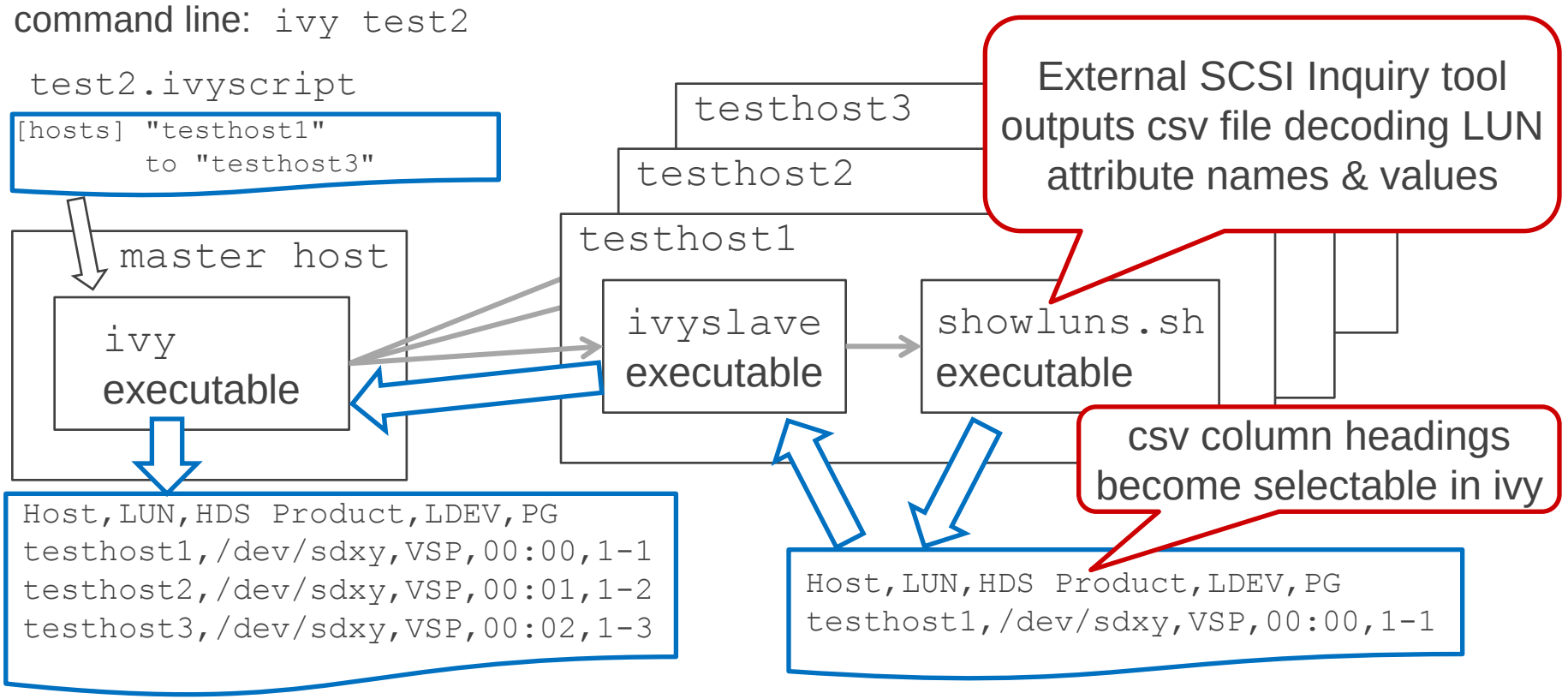
[Hosts] statement starts up ivy on test hosts

HITACHI
Inspire the Next

- On each specified host, the "ivyslave" executable is started using an ssh command.
  - Master host must be set up to ssh to test hosts without a password (that is, using certificate-based authentication.)
  - Have only tried running ivy as root.
    Don't know if a regular user is permitted to do raw LUN I/O.
- Each test host discovers all its storage LUNs, using a SCSI Inquiry-based LUN lister utility program.
  - ivy uses Ian's "showluns.sh" that decodes Hitachi proprietary attributes like subsystem type, serial number, LDEV, Port, PG, CLPR
- The combined list from all the test hosts is "all discovered LUNs"

Vendor-independent LUN attribute discovery

HITACHI
Inspire the Next



Sample attribute values from LUN lister tool

HITACHI
Inspire the Next

```
hostname = cb23
LUN_Name = /dev/sdbu
Hitachi_Product = HM700
HDS_Product = "HUS VM"
Serial_Number = 210030
Port = 1A
LDEV = 00:00
Nickname = ""
LDEV_type = Internal
RAID_level = RAID-1
Parity_Group = 01-01
Pool_ID = ""
CLPR = CLPR0
Max_LBA = 2097151
Size_MB = 1073.741824
Size_MiB = 1024.000000
Size_GB = 1.073742
Size_GiB = 1.000000
Size_TB = 0.001074
Size_TiB = 0.000977
Vendor = HITACHI
Product = OPEN-V
```

LUN attribute matching

HITACHI
Inspire the Next

- The LUN lister tool output csv file header line defines the LUN attribute names:
  - e.g. `"HDS Product,Serial Number, LDEV,..."`
- Internally within the C++ LUN object, ivy takes that column header, trims off any surrounding quote marks and white space, then converts all non-alphabetics/non-digits to underscores _, translates to lower case, and uses that internally within the object as the key.
- Then later on, if you ask if the LUN contains "HDS Product" or you ask for "hds_product", the LUN object's lookup routine does the same thing to the name you ask for before looking it up – either has the same effect.
- Similarly, when you ask if a value matches a LUN, the same normalization of the values is done before deciding if there is a match.

"all discovered LUNs" -> "available test LUNs"

HITACHI
Inspire the Next

- On the `[Hosts]` statement, the `[Select]` clause filters from all discovered LUNs on all the specified test hosts to create the pool of "available test LUNs" upon which you can "[CreateWorkload]".
- Only on the `[hosts]` statement, the `[Select]` clause must specify a non-null value for at least one of "`serial_number`" (which we always have for Hitachi) or "`vendor`" (if we are testing another vendor's equipment).
  - This is designed to prevent accidental annihilation of your boot drive.

[Select] syntax

HITACHI
Inspire the Next

- <expression for attribute name> `is` <expr. for att. value>
  - `"LDEV_type" is "DP-Vol"`
- <expression for attribute name> `is` { <list of attribute value expressions> }
  - `"port" is { "1A", "3A", "5A", "7A" }`
- The Select clause matches against a LUN if that LUN has the specified attribute and the value of that attribute matches.
- Select clauses are parsed by outer programming language, so there are no quotes around the entire `[Select]` expression.

[Select] LDEV, PG

HITACHI
Inspire the Next

- There are a couple of cases of "custom" attribute value matching for Hitachi subsystems. (Other vendors are encouraged to write their own.)
- There is a custom matcher for "`LDEV`" which understands things like "`00:1A-00:3F, 01:FF`"
- There is a custom matcher for "`PG`" which understands
  - "`1-*`" matches PG names starting with 1-
  - "`1-2:4`" matches 1-2, 1-3, 1-4
  - "`1-2:`" matches 1-2, 1-3, …
  - "`1-:2`" matches 1-1, 1-2

[hosts] – use of command devices is automatic

HITACHI
Inspire the Next

- After we have "available test LUNs", (which excludes command devices)
- The [hosts] statement looks through the command devices that were part of "all discovered LUNs", and for each unique subsystem serial number in available test LUNs, for the first command device found that goes to that subsystem, if the Hitachi proprietary command device connector "`ivy_cmddev`" (not part of the ivy open source project) is available, we fire it up remotely on the test host that has the command device, and retrieve the RMLIB API data on the configuration of the subsystem.
- For each available test LUN, if we have RMLIB API configuration data for the LDEV behind that LUN, the RMLIB API LDEV configuration attribute value pairs are merged into the LUN's attributes.
  - That means that if you have a command device, you can select on `drive_type` to create a workload.
- Later, when we run a test step, RMLIB API performance data is collected on the same time boundaries as the test intervals and that data goes in a csv file set for that test step.

Statements - [SetIogeneratorTemplate]

HITACHI
Inspire the Next

```
[SetIogeneratorTemplate] "random_steady"
      [parameters] "IOPS=20, blocksize=4KiB"
                          + ", maxtags=10, fractionread=0.5";
```

- Sets the defaults for the specified I/O generator name.
- If you are going to use multiple [CreateWorkload]s with minor variations, you could use [SetIogeneratorTemplate] to set all the things that are in common, and then when you create each workload you only specify what's unique for that workload.
  - Handy if you are going to create a series of sequential workloads each starting at a different point in the LUN or having coverage of a different portion of the LUN. Then when reading the program, it's more clear what's going on if the [CreateWorkload] only sets what's different each time.
- The ivyscript language parser expects a single character string expression for [Parameters], as the string is passed as a whole to the corresponding underlying ivy engine function, which parses it there.

[iogenerator] some common [parameters]

HITACHI
Inspire the Next

- VolumeCoverageFractionStart default "0.0" same as "0%"
  VolumeCoverageFractionEnd default "1.0" same as "100%"
  - Establishes the "coverage zone" within the LUN. You can layer different workloads in different parts of the same LUN.
- blocksize default "4 KiB" same as "4096" – also supports "MiB" units.
- maxTags default "1".
  - The maximum number of I/Os that this workload on this LUN is allowed to ***try*** to issue at one time.
  - OS call to start I/Os may block if underlying HBA/device driver is out of tags. Workloads share LUNs and share the underlying HBA/device driver.
- IOPS default "5"
  - IOPS = "max" - keep starting I/Os trying to keep queue depth at "maxTags".
- fractionRead default "1." same as "100%".

`[iogenerator]` random – two types

HITACHI
Inspire the Next

- `random_steady`
  - I/Os are issued to random locations on a steady drumbeat in time.
- `random_independent`
  - I/Os occur at random times as well as to random locations
  - Random independent distributions are easier to model mathematically.
  - The lower the IOPS rate or the shorter the observation period, the more erratic `random_independent` IOPS will appear.
  - In general, `random_independent` I/O patterns will have a slightly higher service time compared to `random_steady` workloads, because scheduled I/O start times are independent and in general can collide (bursty), whereas `random_steady` workloads space out I/O scheduled start times evenly.

[iogenerator] "sequential"

HITACHI
Inspire the Next

- In ivy, a sequential workload must be all reads (`fractionRead=1.0` / `fractionRead="100%"`) or all writes (`fractionRead=0`).
- But, you can use a for loop to create a series of sequential threads starting at different points along the LUN, where each of the threads is either a read thread or a write thread
  - `SeqStartFractionOfVolumeCoverage = 0.23`
  - Range is from 0.0 to less than 1.0 - this is relative to the volume coverage zone defined from `VolumeCoverageFractionStart` to `VolumeCoverageFractionEnd`.
  - More commonly use the volume coverage parameters to have sequential threads wrap around in their own areas.

Statements - [CreateWorkload]

HITACHI
Inspire the Next

- [CreateWorkload] "r_steady"
  [select] "LDEV" is "00:04"
  [iogenerator] "random_steady"
  [parameters] "fractionRead = 75%"

- Apply a [select] filter matching against "available test LUN" attribute values.

- On each selected LUN, create an identical workload thread with the specified workload name, running the specified [iogenerator] plug-in, and supplying the [iogenerator] with a [parameters] text string that the iogenerator will parse and apply.
  - Each type of iogenerator has its own set of valid parameter names.

[CreateWorkload]

HITACHI
Inspire the Next

- Attributes for selection are those of the underlying LUN plus several special built-int attributes
  - `workload`, set to the specified workload name, and
  - `host` which is an alias for `ivyscript_hostname`, the name used on the `[hosts]` statement which might be an alias or a dotted quad.
- The newly created workload threads will be in "waiting for command" state.

Statements - [DeleteWorkload]

HITACHI
Inspire the Next

- `[DeleteWorkload] "r_steady" [select] "LDEV" is "00:04";`
- `[DeleteWorkload] "r_steady" ;`
  - Deletes all instances of the `r_steady` workload on all test hosts / all LUNS.
- `[DeleteWorkload] ;`
  - Deletes all workloads.

Statements – [CreateRollup]

HITACHI
Inspire the Next

- `[CreateRollup] "Serial_Number+Port"`
- `[CreateRollup] "host"`
  `[nocsv] [quantity] 8 [MaxDroopMaxToMinIOPS] "25%";`
- A rollup is a partition of all workload threads.
- Every workload thread belongs to exactly one instance of each rollup.
- There is always an "`all`" workload which only has one instance "`all`".
- `[nocsv],[quantity],[MaxDroopMaxToMinIOPS]` are optional, but if they appear they must be in that order
- To get individual data for each workload thread, say "`workloadID` " which is comprised of "`host+LUN_name+workload`".

You make rollups for four reasons

HITACHI
Inspire the Next

1. To get an output csv file with a csv folder by rollup type (e.g. Port+CLPR) and csv files by rollup instance (e.g. Port+CLPR = 1A+CLPR0)
   - This is how you get custom "sliced & diced" data.
2. To perform dynamic feedback control (`dfc=PID`) at the granularity of the rollup instance.
3. To identify a valid measurement period at the granularity of the rollup instance using `measure=on`.
4. To validate the test configuration as operating correctly
   - E.g. test that the number of ports reporting was what you expected
   - E.g. validate that no one port had an IOPS too far below the highest IOPS seen on any port.

Rollups are key to how the ivy engine works

HITACHI
Inspire the Next

- **`[CreateRollup] "Serial_Number+Port"`** (no spaces are permitted around the + sign)
  - Both `Serial_Number` and `Port` must be valid LUN-lister column header attributes, or built-in layers on top of those attributes
- Then for all existing WorkloadIDs, we build a data structure that looks like this

```
"Serial_Number+Port"
    "410123+1A"
        "sun159+/dev/sdd+workload_name", "cb28+/dev/sdd+workload_name",
    "410321+1A"
        "sun159+/dev/sdf+workload_name", "cb28+/dev/sdf+workload_name"
"host"
    "sun159"
        "sun159+/dev/sdd+workload_name", "sun159+/dev/sdf+workload_name"
    "cb28"
        "cb28+/dev/sdd+workload_name", "cb28+/dev/sdf+workload_name"
```

The rollup type

The rollup instance. It's the rollup instance that has the rolled up data.

List of WorkloadIDs that "landed" on this rollup instance

- Every WorkloadID appears exactly once in each rollup type

[CreateRollup] combines LUN attribute names

HITACHI
Inspire the Next

- If you would like to see a rollup instance for each unique LDEV across two or more subsystems, make "serial_number+LDEV".
- [nocsv] – prevents csv files from being created.
- The [quantity] <int expression> clause can enforce that the right number of distinct rollup instances are reporting in this rollup.
  - If not, even if the DFC reports "success" and designates a subinterval subsequence representing a successful measurement, no measurement rollup csv data will be produced – instead error msg.
  - Make a rollup by "port" or "host+scsi_bus_number__hba_" and use the [quantity] rollup to validate you have the number of paths you think you have.
- [MaxDroop] <double expression>
  - "25%" means invalidate test if any one rollup instance has an average IOPS more then 25% below the highest average IOPS over all rollup instances.

[DeleteRollup]

HITACHI
Inspire the Next

- `[DeleteRollup] "serial_number+Port";`
- `[DeleteRollup] ;`
  - Deletes all rollups except the "all" rollup.

[EditRollup]

HITACHI
Inspire the Next

- `[EditRollup]`
  `"serial_number+Port = { 410123+1A, 410123+2A }"`
  `[parameters] "fractionRead = 100%";`

- The `[EditRollup]` statement operates in between test steps while the workload threads are in "waiting for command" state.

- It gives you access to the same mechanism used by a Dynamic Feedback Controller to send out real time parameter updates to running workload threads.

- You specify a set of rollup instances to send to, such as "`all=all`", or "`Port={1A,3A,5A,7A}`" and you specify the text `[parameters]` string to send to the remote iogenerator to parse and apply.

[EditRollup]

HITACHI
Inspire the Next

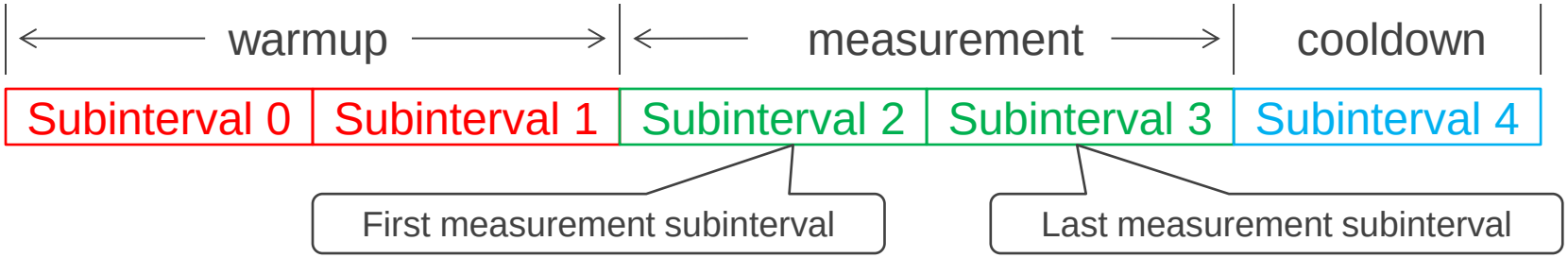
- `[EditRollup]` is typically used at the top of a do-loop, to change whatever parameters vary by loop pass.
- Use `[EditRollup] "all=all"` to send to all workload threads.
- There is a special parameter name that is only recognized by [EditRollup] - `total_IOPS` - where the numeric value you specify is divided by the number of workload threads comprising a workload instance, before being sent out as `IOPS=`.
  - `[EditRollup] "all=all" [parameters] "total_IOPS = 1000000";`

Statement – [Go]

HITACHI
Inspire the Next

- [Go] "stepname=random4K, subinterval_seconds=5, …"
- The [Go] statement starts the workload threads running a "test step", which is a sequence of "subintervals" each of a duration specified in the subinterval_seconds parameter, defaulting to 5 seconds.
  - If you have a case for using ivy to measure a restricted set of things much more frequently, we can talk about putting in support.
  - Most of the time 5 seconds is plenty short and if you are going to be doing any tests that will run for hours you may want to consider a longer subinterval just to mercifully cut down on the size of the csv files by subinterval.
  - Sometimes when you say you want an answer to +/- 1% and the behaviour is a bit noisy, it can take time to see enough to say you are sufficiently confident statistically. (Did you say you wanted "valid" data?)

Test step = warmup, measure, cooldown



- There must be at least one warmup, one measurement, and one cooldown subinterval.
- Parameter defaults
  - `warmup_seconds = 5` - this number is divided by subinterval_seconds, and rounded up to get the (minimum) number of warmup subintervals.
  - `measure_seconds = 60` - also rounded up to the minimum number of measurement subintervals.
  - `cooldown_by_wp = on` - If a command device is available for the subsystem under test, the cooldown period is extended until write pending is empty.

For each test step you get:

HITACHI
Inspire the Next

- A subfolder of the overall test output folder that contains the csv files with one line for each subinterval in that test step.
  - Nested subfolders for each workload data rollup
    - Containing a csv file for each rollup instance, with one line per subinterval.
  - A nested subfolder with raw RAID_subsystem RMLIB API data.
    - Collected time-synchronized "just before" the end of each subinterval.
- A single line in the overall test results "summary.csv" files.
  - In ivy terminology, this is called a "measurement" line, which represents the rollup from the first to last measurement subintervals.
    - Unless "`measure=on`" with specified accuracy timed out – then you get an error message line

cooldown_by_wp

HITACHI
Inspire the Next

- Default: `cooldown_by_wp = on`
- Set `cooldown_by_wp = off`
  - When it is valid to carry forward dirty data in cache (Write Pending) from one test step to the next.
  - This can speed up the next test step tremendously if
    - the next step doesn't stabilize until WP is full,
    - AND if both steps place the SAME things into WP.

The default `[Go]` statement

HITACHI
Inspire the Next

- [go];
  - Default `warmup_seconds = 5`
  - Default `measure_seconds = 60`
  - Default `subinterval_seconds = 5`
  - Default `cooldown_by_wp = "on"`
    - Runs at least one cooldown subinterval
    - If you have a command device and the proprietary command device connector software, continuing more cooldown subintervals until WP is empty.
  - Useful when you are developing an ivyscript workflow and you just want to see quick sample csv files.

stepname

HITACHI
Inspire the Next

- On the [Go] statement to start a test step, you can optionally specify `"stepname="`, which defaults to "step" followed by a four digit step number starting with 0000, so the default name for the first step is `step0000`.

- Giving a test step a meaningful name is useful when looking at overall measurement summary csv files, where you get one csv line for each test step.

- Those labels are handy when making Excel charts, as you can use the stepname column as the series name on a chart.

[Go] "focus metric"

HITACHI
Inspire the Next

- If you want to run a fixed workload for a fixed number of subintervals, all you need is `warmup_seconds` and `measure_seconds`.

- Otherwise, we need to specify the "focus metric".

  1. The focus metric is what we are making a valid measurement of using "`measure=on`", the "seen enough and stop" feature.
     - Measure the focus metric to a required plus/minus accuracy with a specified % confidence level.
  2. When dynamically adjusting `total_IOPS` using the PID loop dynamic feedback controller (`dfc=pid`), the focus metric is the "feedback" in dynamic feedback control.

Granularity of the "focus metric"

HITACHI
Inspire the Next

- When `measure=on` and/or `dfc = pid` are used, measurement or PID loop DFC is performed at the granularity of each instance of the `focus_rollup`.
- For the default, `focus_rollup = all`, the measurement or DFC is at the overall level.
- When a `focus_rollup` is used that has multiple rollup instances,
  - With `measure=on`, a successful measurement identifies a subsequence of subintervals where for every rollup instance within the `focus_rollup`, the measurement is valid for that rollup instance.
  - When `dfc=pid` is used, dynamic feedback control is performed independently for each rollup instance in the `focus_rollup`.

`source` of the `focus_metric`

HITACHI
Inspire the Next

- `source = workload`
  - Specifies that we are selecting a focus metric from data collected by ivy workload threads on test hosts.
  - We always have rollup data from test host workload threads (*more next page*)
- `source = RAID_subsystem`
  - Requires the proprietary command device connector that is not part of the ivy open source project.
  - Specifies that we are selecting the focus metric from real time performance data collected from a command device.
  - There's a small list of subsystem metrics specified in an ivy source code table that are filtered and rolled up from the raw bulk RMLIB data by rollup instance, and from which you select the focus metric. (*more even later after we explain* `source=workload`)

Selecting a "source=workload" metric

HITACHI
Inspire the Next

- category =
  - overall, read, write, random, sequential, random_read, random_write, sequential_read, sequential_write
- accumulator_type =
  - bytes_transferred, service_time, response_time
- accessor =
  - avg, count, min, max, sum, variance, standardDeviation
- It will be easier to explain first accessor then category, then accumulator_type

Accumulators and "accessor"

HITACHI
Inspire the Next

- An accumulator is an object that you push numbers into in order to be able to compute summary values.
- Every time that an I/O completes, ivy posts the service time into one accumulator, the bytes transferred into another accumulator, and if we are not running IOPS=max, it posts the response time into another accumulator.
- The selectable values for "`accessor`" are the names of the methods that you can use to retrieve something from an accumulator
  - `avg, count, min, max, sum, variance, standardDeviation`
  - `avg` gives you the average of the numbers that were pushed in the accumulator
  - `count` gives you how many numbers were pushed in.
  - Et cetera.

Attributes of individual I/Os:

HITACHI
Inspire the Next

- read vs. write
- blocksize
- LBA
  - Logical Block Address = sector number from 0 within LUN
- service_time (in seconds)
  - The duration from when ivy launched an I/O until ivy received the notification that the I/O was complete.
- response_time* (in seconds) (analogue to application-level response time)
  - The duration from the scheduled start time of an I/O until the time the I/O is complete.
  - An I/O may not be started at the scheduled time if there are no idle asynchronous I/O "slots" (~tags) available.
  - ***only I/Os with a non-zero scheduled start time will have a response_time attribute.**
  - When running iops=max, all I/Os have a scheduled start time of zero, meaning you don't get response_time

Ivy uses the Linux nanosecond resolution clock for all timing

How ivy posts results of each I/O

HITACHI
Inspire the Next

- Based on the attributes of each I/O, an accumulator category is selected.
  - Then the I/O is posted into the selected category "bucket" (into two or three accumulators in that bucket – more in a moment.)
- Currently, the breakdown for the array of categories for which there are accumulators are
  - read vs. write
  - random vs. sequential (The I/O sequencer tells you if it's a random or sequential sequencer.)
  - For each of those 4 there is a further breakdown as a histogram by service time and by response time
    - You see the histograms in the csv files.
    - Ivy doesn't currently expose the histogram in the PID loop, but if there is interest it can be added.

Other category breakdowns could be defined

HITACHI
Inspire the Next

- The rollup mechanism operates on a view of the categories as an array, and is blind to the significance of each position in the array.
  - It is easy to define a different mapping from the attributes of an individual I/O to the category bucket the I/O will be recorded in.

- Future:
  - We could just as easily define a histogram of a 100 buckets by LBA range - we could break out the data by each 1% of the LBA range across the volume.
    - If we had an I/O sequencer that was playing back a customer I/O trace, we could show if workload characteristics were different in different areas of the LUN.
    - If we simply run sequential transfers across the LUN, we could see the sustained data rate "staircase" showing the zones in underlying HDDs.

During rollups, the categories are preserved

HITACHI
Inspire the Next

- For the `all=all` instance, you still have all the category breakdowns.
- Then in addition to the category bucket array, there are virtual categories, implemented as functions, which rollup underlying category buckets.
  - `overall` – sum over all categories in the bucket array
  - `read, write`
  - `random, sequential`
  - `random_read, random_write, sequential_read, sequential_write`
- You can see these virtual category rollups in column groups in ivy csv files.

source=workload available category values

HITACHI
Inspire the Next

- overall
  read, write
  random, sequential
  random_read, random_write, sequential_read, sequential_write

- These are actually the virtual categories, representing the rollup over the underlying service time / response time bucket arrays (histograms).
  - If there is a need, we could provide access to the more fine-grained underlying category bucket array, or we could define other virtual categories as aggregations of the buckets.

source=workload - selecting accumulator

HITACHI
Inspire the Next

- Category buckets have 3 accumulators
- accumulator_type = bytes_transferred
  - For every I/O, the blocksize is posted to bytes_transferred.
  - Use sum attribute and divide by elapsed seconds to get bytes per second. Use count instead and get IOPS.
- accumulator_type = service_time
  - For every I/O the duration from when ivy started it to when it completed.
  - service_time and response_time values for I/Os are posted in units of seconds, with nanosecond resolution.
  - Use "avg" and multiply by 1000 to get average service time in ms.
- accumulator_type = response_time (~ application response time)
  - Only posted for those I/Os that have a non-zero "scheduled time".
  - Duration from scheduled time to I/O completion time.
  - The I/O sequencer computes the scheduled time, and when that time is reached, the I/O is started if there is an idle Asynchronous I/O "slot" (~tag) available. If not, it waits.
  - For IOPS=max, I/Os have a scheduled time of 0 (zero), so then you don't get any response_time events.

Summary: source=workload

HITACHI
Inspire the Next

- category =
  - overall, read, write, random, sequential, random_read, random_write, sequential_read, sequential_write
- accumulator_type =
  - bytes_transferred, service_time, response_time
- accessor =
  - avg, count, min, max, sum, variance, standardDeviation

source = RAID_subsystem

HITACHI
Inspire the Next

- Subsystem performance data is collected from a command device, and for each subsystem with a command device, there is a subfolder within the test step folder, where each csv file has one line per subinterval within that test step.
  - You cannot select the focus metric from this raw, bulk subsystem performance data.

- A small subset of metrics are extracted from the bulk subsystem data, and filtered and summarized by rollup instance
  1. To serve as candidates for selection as the focus metric
  2. To be printed as columns in rollup instance csv files side-by-side with the columns of host-workload data.

- This is controlled by a table in ivy source code, which has two levels that you pick from
  - `subsystem_element`, and within that, `element_metric`.

- For each metric in the table, you can optionally set a flag to have the value inserted a column side by side with the normal workload data for each rollup instance.

Subsystem metrics by rollup instance 2015-11-19

HITACHI
Inspire the Next

- MP_core
  - busy_percent, io_buffers
- CLPR
  - WP_percent
- PG
  - busy_percent,
    random_read_busy_percent, random_write_busy_percent, seq_read_busy_percent,
    seq_write_busy_percent
- LDEV
  - read_service_time_ms, write_service_time_ms,
    random_blocksize_KiB, sequential_blocksize_KiB,
    random_read_IOPS, random_read_decimal_MB_per_second , random_read_blocksize_KiB,
    random_read_hit_percent,
    random_write_IOPS, random_write_decimal_MB_per_second, random_write_blocksize_KiB,
    sequential_read_IOPS, sequential_read_decimal_MB_per_second, sequential_read_blocksize_KiB,
    sequential_write_IOPS, sequential_write_decimal_MB_per_second,
    sequential_write_blocksize_KiB,

Subsystem data filtered by rollup instance

HITACHI
Inspire the Next

- The way this works is via a "config filter" that is prepared in advance before a subinterval sequence starts.
- For each thing you get data for, such as PG, or LDEV, or MPU, etc., the config filter has the set of instances of PG or LDEV or MPU names that were either
  - directly observed as a SCSI Inquiry attribute of the LUNs underlying the workloads in the rollup instance, or
  - observed as an attribute of an underlying LDEV obtained via the RMLIB API, or
  - which were inferred from static tables of relationships for the particular subsystem model.

Subsystem data by rollup instance – csv columns

HITACHI
Inspire the Next

We know how many drives underlie the each workload rollup

Shows you if the OS / device driver are breaking up your large block application-level I/O into smaller pieces

Matching subsystem vs. application data validates that both host-workload rollups and subsystem data rollups are working correctly

Shows you if there is delay between when the application issues the I/O and when the device driver issues the I/O.

Shows you the amount of that delay in ms.

subsystem avg LDEV sequential_blocksize_KiB	subsystem avg LDEV read_service_time_ms	subsystem avg LDEV write_service_time_ms	host IOPS per drive	host decimal MB/s per drive	Subsystem IOPS as % of application IOPS	Subsystem MB/s as % of application MB/s	Subsystem service time as % of application service time	Path latency = application minus subsystem service time (ms)	Overall IOPS	Overall Decimal MB/s	Overall Average Blocksize (KiB)	Overall Little's Law Avg Q
0	6.82666	0	1.90	-	98.22%	98.22%	98.69%	0.091	30	0.12288	4	0.207528
4	0	60.5501	98.30	0.40	99.99%	99.99%	99.08%	0.563	1572.9	6.4426	4	96.1246
4	50.0142	49.0471	120.70	0.50	100.43%	100.43%	99.27%	0.364	1931.98	7.91339	4	95.9223
4	41.4719	32.018	162.00	0.70	99.99%	99.99%	99.24%	0.283	2592.74	10.6199	4	95.9846
4	39.3228	3.90259	194.90	0.80	99.68%	99.68%	99.02%	0.303	3118.22	12.7722	4	96.0234
0	20.9364	0	283.10	1.20	100.51%	100.51%	98.79%	0.257	4529.76	18.5539	4	95.9988

RMLIB API candidates flagged to display

HITACHI
Inspire the Next

maxTags	IOPS input parameter setting	fractionRead	test host cores	test host avg core CPU % busy	subsystem MP_core count	subsystem avg MP_core busy_percent	subsystem CLPR count	subsystem avg CLPR WP_percent	subsystem PG count	subsystem avg PG busy_percent	subsystem LDEV count	subsystem avg LDEV random_read_IOPS	subsystem avg LDEV random_write_IOPS	subsystem avg LDEV random_blocksize_KiB
16	5	1	16	0.10%	12	2.52%	1	0%	4	1.12%	6	4.91	-	4.00
16	max	0	16	0.00%	12	5.39%	1	68.78%	4	99.83%	6	-	261.42	4.00
16	max	0.25	16	0.00%	12	5.63%	1	68.84%	4	99.82%	6	80.11	242.82	4.00
16	max	0.5	16	0.00%	12	6.06%	1	68.27%	4	99.85%	6	215.46	216.43	4.00
16	max	0.75	16	0.10%	12	6.23%	1	62.59%	4	99.80%	6	388.87	128.99	4.00
16	max	1	16	0.10%	12	4.34%	1	0.20%	4	95.16%	6	758.77	-	4.00

- The “subsystem” columns are automatically generated according to the focus metric RMLIB API candidate table.
- As raw data comes in for each MP_core, CLPR, PG, LDEV, etc., ivy filters the data to aggregate for each rollup only the data for the MP_cores, etc. that map to LDEVs/LUNs underlying workloads in that rollup.
- Make a rollup by MPU, and each MPU rollup instance will show data for 4 MP_cores.

Examples of data for each rollup – drive / PG type

HITACHI
Inspire the Next

Rollup Type	Rollup Instance	drive type	drive quantity	RAID level	PG layout	iogenerator type	blocksize	maxTags	IOPS input parameter setting	fraction Read
all	all	DKR2E-H4R0SS+DKR5D-J600SS	8+8=16	RAID-5	3+1	random_steady	4 KiB	16	5	1
all	all	DKR2E-H4R0SS+DKR5D-J600SS	8+8=16	RAID-5	3+1	random_steady	4 KiB	16	max	0
all	all	DKR2E-H4R0SS+DKR5D-J600SS	8+8=16	RAID-5	3+1	random_steady	4 KiB	16	max	0.25
all	all	DKR2E-H4R0SS+DKR5D-J600SS	8+8=16	RAID-5	3+1	random_steady	4 KiB	16	max	0.5
all	all	DKR2E-H4R0SS+DKR5D-J600SS	8+8=16	RAID-5	3+1	random_steady	4 KiB	16	max	0.75
all	all	DKR2E-H4R0SS+DKR5D-J600SS	8+8=16	RAID-5	3+1	random_steady	4 KiB	16	max	1

- Information comes from RMLIB API configuration data, filtered / aggregated for each rollup instance.
- (There are also dedicated csv folders that contain detailed RMLIB API subsystem configuration and performance data csv files.)

Now that we know how to specify the focus metric

HITACHI
Inspire the Next

- We will look at
  - The [`go`] statement `measure=on` option and its subparameters
    - Specifying `measure=on` on a Go statement means "watch the focus metric and when you have seen enough to make a measurement of the specified accuracy, stop. Timeout if it takes too long."
  - The [`go`] statement `dfc=pid` option and its subparameters
    - If you don't specify a `dfc`, the workload settings remain constant through the test.
    - If you specify a `dfc` (dynamic feedback controller), it gets called at the end of every subinterval once all the rollups are done.
    - The DFC looks at what has happened so far, looking at all workload data and all subsystem data, and then may use the ivy engine real time edit rollup mechanism to send out parameter updates to rollup instances (to the workload threads belonging to the rollup instance).

measure = on

HITACHI
Inspire the Next

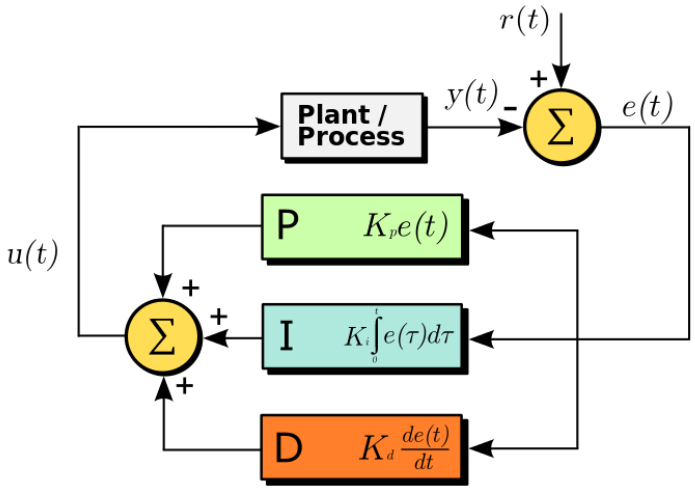
- `accuracy_plus_minus = "2%"`
  - Any numeric value with an optional trailing % sign maybe specified.
- `confidence = "95%"`
  - How confident you need to be that your measurement falls within the specified plus or minus range around the long term average that you would get measuring forever.
  - Default is 95%
  - Ivy has a menu of 11 specific pre-loaded confidence values that you pick from.
    - 50%, 60%, 70%, 80%, 90%, 95%, 98%, 99%, 99.5%, 99.8%, and 99.9%
    - http://en.wikipedia.org/wiki/Student%27s_t-distribution

`measure` Write Pending-based stability criteria

HITACHI
Inspire the Next

- `max_wp = "2%"` - default “100%”
  - A subinterval sequence will be rejected if WP is above the limit at any point in the sequence.
  - Set this to "1%' or so for read tests to ensure WP is empty during the test.
- `min_wp = "67%"` - default “0%”
  - A subinterval sequence will be rejected if WP is below the limit at any point in the sequence.
  - Use this for write tests to ensure WP is full during the test.
- `max_wp_change = "3%"` - default “3%”
  - A subinterval sequence will be rejected if WP varies up and down by more than the specified (absolute) amount at any point in the sequence. `max_wp_range="3%"` matches from 0% to 3% Write Pending, as well as from 67% to 70% Write Pending. (not a percent OF the WP value)
  - Use this in general all the time so you reject periods with major movement in Write Pending.

`dfc=pid` dynamically adjusts total IOPS



- General purpose DFC – see http://en.wikipedia.org/wiki/PID_controller
- The feedback is the value of the focus metric
  1. `source=workload`
     - E.g. host-view service time, response time
  2. `source=RAID_subsystem`
     - e.g. `subsystem_element="PG", subsystem_metric="busy_percent".`
- User specifies “p”, “i”, and “d” constants.

PID loop basics

HITACHI
Inspire the Next

- See "PID loop" on wikipedia https://en.wikipedia.org/wiki/PID_controller
- ivy's PID loop dynamically adjusts IOPS up and down to hit a target value for the focus metric.
- The "error signal" is the difference between the measured focus metric value and the target value.

PID loop – computing new IOPS setting

HITACHI
Inspire the Next

- The user provides 3 multiplier factor constants: p, i, d.
- The new total_IOPS setting is
  - "p" times the error signal (**p**roportional factor)
  + "i" times the sum of the error signal since the start of the test (**i**ntegral factor)
  + "d" time the rate of change of the error signal (**d**erivative factor)
- The ivy engine "edit rollup" mechanism sends out the new `total_IOPS` setting to the focus metric's rollup instance (usually "`all=all`"), where it takes effect in real time.

[Go] parameter summary

HITACHI
Inspire the Next

- Overall
  - stepname = stepNNNN
  - subintervalseconds = 5
  - warmup_count = 1
  - measure_count = 1
  - cooldown_by_wp = on
- For dfc = pid
  - p = 0
  - i = 0
  - d = 0
  - target_value = 0
- For measure = on
  - accuracy_plus_minus = "2%"
  - confidence = "95%"
    - 50%, 60%, 70%, 80%, 90%, 95%, 98%, 99%, 99.5%, 99.8%, or 99.9%
  - max_wp = "100%"
  - min_wp = "0%"
  - max_wp_change = "3%"
- Focus metric
  - focus_rollup = all
  - source = ""
    - or workload / RAID_subsystem
  - subsystem_element = ""
  - element_metric = ""
  - category = overall
    - or read, write, random, sequential, random_read, random_write, sequential_read, sequential_write
  - accumulator_type = ""
    - or bytes_transferred, service_time, response_time
  - accessor = ""
    - avg, count, min, max, sum, variance, standardDeviation

HITACHI
Inspire the Next

<end>
Thank You

Hitachi Data Systems