CovVise:
How We Stopped Throwing Away Interesting Coverage Data

Wilson Snyder,
Robert Woods-Corwin

SiCortex
Maynard, MA

http://www.veripool.org/covvise

ABSTRACT

We have build CovVise, a new open source coverage system (http://www.veripool.org/covvise). Verifiers can write coverage statements in SystemC we extended with SystemVerilog-ish coverage statements, and designers can write one-line coverage statements that expand to SystemVerilog coverage statements. Data is collected on both passing **and failing tests** to aid spotting interesting trends. We connected this to a memory database that scales above 10 billion inserts a day, and a web front-end for visualizing the results.

Table of Contents

1. Introduction

At SiCortex, our last project had a fairly traditional coverage methodology, and we were left wanting more for our successor project.

Our previous project had a fairly typical coverage methodology; we waited until near the end of the project to add coverage. We then added simple counters to our SystemC testbenches to collect items of interest ("bins"), and summed together all hits on each bin across all passing tests. This data then was saved in a simple file-based database.

Each verification person then wrote a custom Perl script to extract his device-under-test's data as an HTML document. The process worked, but was fairly tedious, and required an ugly connection between multiple languages all exposed to the users: coverage statements in Verilog, SystemC, internals in C++, and reporting scripts in Perl and HTML.

In parallel and completely independent from the above scheme we used the commercial simulator coverage system for expression coverage.

This year we demolished most of what we had, and revised the traditional assumptions. We decided to collect data on failing tests, and separate out low-hit-tests (more about this later.)

To enter the coverage bins, we kept SystemVerilog for RTL coverage, but added to the SystemC language an extension to support SystemVerilog-like crosses and bins.

Next came the database. We planned to execute more than 10 billion coverage inserts a day, so we came up with a fully distributed Memcached-MySQL solution that uses parallel servers to both allow high insert rates, and medium term storage. The data collected is still fairly large (>10 GB per day), so we also automatically prune the data and archive to slower per-month databases.

Finally, the coverage viewing user interface was implemented as a web 2.0 application (using AJAX[1]). This allows users to interactively traverse a hierarchy of coverage bins, making it easy to see the coverage tables of interest. The presentation makes it easy to see what crosses still have holes, and a single click identifies which tests are the best candidates to run for covering those bins.

This process enables us to see the coverage holes, and close on what random stimulus improvements were required much sooner than in our previous project.

[1] AJAX stands for Asynchronous JavaScript And XML. AJAX is the web programming technique first popularized by Google Maps to implement what looks like a normal non-web application in the browser; users can click on items in their browser and have them updated by the server without loading a whole new page.

2. Tradition Dictates...

- Coverage is added by verification team members
- Coverage is added near the end of the verification effort
- Coverage is collected only on passing tests
- Coverage "hit count sums" are all created equal

Before starting CovVise, we looked at how we were doing coverage, and what assumptions we would like to revisit.

The first tradition is that verification team members add all the coverage. Yes, designers have line coverage to worry about, but our verification team traditionally generated and analyzed the line coverage reports. Instead, we wanted a process whereby the designers would be encouraged to insert coverage statements and view reports. The designers needed an easy coverage language, and the designers had to be able to easily see the data – as an HTML report that was generated automatically on every test run. We will cover the SystemVerilog cover statements and process in Chapter 4.

The second tradition is that coverage is added near the end of the verification effort, when it's believed verification is almost complete, and it's just a matter of adding coverage to detect holes. Instead, we tried to learn from "test driven development" which suggests the tests for code precedes the source code itself. (Coverage being the "test" of the verification code.) Knowing what buckets were easily getting covered, or were huge holes, helped direct the verification effort; next time we probably would have started coverage even earlier. For example, what was thought in the verification plan to be a rare case requiring a focused test turned out based on the coverage data to be releatively easily generated psudo-randomly, thus we knew apriori we didn't need to write the focused test. Another reason to start early is that design and verification engineers often have passing thoughts of "we should remember to cover these cases or these conditions," and such thoughts are easy to implement now but are hard to remember later (or are forgotten altogether).

The third tradition is that coverage is collected only on passing tests, indeed, the authors could find no suggestions that there was any other way to do it. Why is this? Obviously only passing tests should contribute to the coverage metrics, however consider a coverage bin with a "0" in it – IE no passing tests have ever hit it. It may be illegal, something not yet in being stimulated, or something waived from testing. What if you additionally know that a failing test is hitting it? It may still be illegal, but you're certainly going to do a lot more investigations before waiving it! Furthermore, by making a list of all failing tests and grading them on the number of bins that they hit that no passing test also hits, you can obtain a quick estimate of what tests are most valuable to work on to reach the coverage goals.

The final tradition is that all coverage bin "hit counts" are equal. Not equal in the weight sense[2] but equal in what a count of "20" hits summed across all tests means. Unfortunately, there's a

[2] As for weighting, bins should probably be weighted such that large cross-product tables in total have similar weights to small tables. Otherwise a cross with 10,000 bins will swap your

difference between twenty tests each causing an event once, and one test causing an event 20 times. While both sum to 20 hits, the former may easily be triggered by a test initialization step, but 20 in one test most likely indicates a test hits that bin fairly well.[3] Thus, CovVise creates splits covered bins into “ok,” “low,” and “zero”. “Ok” indicates a single test hit that bin more than 10 (programmable-per-bin) times, while “low” indicates no test hit that bin more than that number of times. Again as another example, the summed count could be 50,000 but still considered “low” coverage, because every test in a large regression of 50,000 tests hit that bin just once. We see that a lot on control registers, indicating there’s no test that toggles the CSR a few times, indicating we’re probably missing the focused read-write test for that CSR.

3. SystemC extensions for Coverage

SystemVerilog provides a fairly good syntax for specifying coverpoints, bins, and crosses. But most of our verification code used SystemC, which does not have coverage conveniently built into the language.

We decided to created extensions to SystemC to provide a similar set of features as available in SystemVerilog, and added this to the SystemPerl pre-processor, which already sat on top of our SystemC simulator.

Much like in SystemVerilog, the user specifies one or more covergroups inside a SystemC module. Each covergroup contains one or more coverpoints, all of which are sampled at the same time. Each coverpoint is associated with a single variable, and the values of the variable are sampled into a number of bins, which can be specified in a number of ways:

```
SC_MODULE(myModule) { // A SystemC Module
  ...
  SP_COVERGROUP myGroup (   // Define the covergroup
    coverpoint myCoverPoint{
      bins seven = 7;              // a single value
      bins three_to_five = [3:5];  // a bin for a range
      bins many[] = {6,9,[21:25]}; // seven bins
    };
  );
  ...
  void process() {  // a method of the class
    ...
    if (sampling_signal) {         // when to sample
       SP_COVER_SAMPLE(myGroup);   // bins incremented here
    }...
  };
};
```

coverage metrics, which seems wrong as that large table probably isn’t a lot more important in aggregate than 10 well thought out single bins. SystemPerl/CovVise implements this automatically.

[3] Assuming a count of greater than one indicates the test has escaped initialization only works if there’s only one initialization sequence per test. Generally we don’t put subtests each with their own initialization into a single test run, for ease of debug and triage.

Multidimensional crosses can also be specified from up to 8 different coverpoints in a given covergroup (more than 3 or 4 tends to get unwieldy):

```
SC_MODULE(myModule) {  // A SystemC Module
  ...
  SP_COVERGROUP myGroup (
    coverpoint first {
      bins seven = 7;               // a single value
      bins three_to_five = [3:5];  // a bin for a range
      bins many[] = {6,9,[21:25]}; // three explicit bins
    };
    coverpoint second[8] = [0:7];  // 8 bins, 0 through 7
    cross myCross {
      rows = {first};
      cols = {second};
    };
  );
  ...
};
```

Once we had this basic level of functionality, we quickly discovered that there were many instances of a common pattern, a coverpoint with one bin corresponding to each value of an enumerated type. So, we added a shortcut for SystemPerl automatic enumerated types:

```
SC_MODULE(myModule) {  // A SystemC Module
  CmdType command;  // a SystemPerl AUTOENUM

  SP_COVERGROUP Cbox_cmd (
    coverpoint command {
      auto_enum_bins = CmdType;
    };
  );
};
```

Another common pattern, especially with crosses, is that certain combinations of values are unreachable and should be ignored or assert an error. In either case, the generated HTML table shows grey boxes without numbers, indicating that this combination should never be hit. We added a mechanism to specify these procedurally:

```
SC_MODULE(myModule) {  // A SystemC Module
  CmdType command;  // a SystemPerl AUTOENUM

  SP_COVERGROUP mygroup (
    ...
    cross myCross {
      rows = {first};
      cols = {second};
      illegal_bins_func = myCross_illegal()
      ignore_bins_func = myCross_ignore()
    };
  );
  ...
  // function returns true if the argument pair is illegal
  // and should cause test failure
  bool myCross_illegal(uint64_t first, uint64_t second);

  // function returns true if the argument pair should
  // be ignored, but should not cause test failure
  bool myCross_ignore(uint64_t first, uint64_t second);
};
```

We also added functions to waive certain bins, this is used to indicate a particular bin isn't impossible, but need not be completely tested before tapeout. This information is fed to the database, and a little "W" appears when coverage has been waived, so the user knows there's something to look at post-tapeout.

4.SystemVerilog extensions for RTL Coverage

- Designers should add coverage just as they add assertions
- Simple preprocessing allows coverage inside normal RTL always blocks

We believe some coverage belongs in the RTL in the same way that some assertions belong in the RTL. For example, a low level RTL detail that makes the coincidence of two rare conditions interesting is something the RTL designer would add coverage for, preferably when writing that RTL code. However, functional coverage of a specified major cross-block interface remains the domain of the verifier.

At the start of the current project we viewed RTL designer coverage points as an extension of line coverage – something the designer adds, but not something added to the test plan. Over the course of this project we learned that many coverage points written in the verification plan were already or better implemented in the RTL, or vice versa. As time goes on we are learning which coverage is easier where, and hope to better decide who implements which coverage point, and also update the testplans based on coverage ideas the RTL designer added to cover low level RTL details.

The catch with RTL coverage was when inserting such coverage points our designers found SystemVerilog coverage and assertion statements to be quite unwieldy. Coverage statements do not fit in well into standard “always block” code, and thus often require half a dozen extra lines of code to accomplish what should take one line. Alas, as it has often been noted, the SystemVerilog coverage and assertion syntaxes simply seem like "a different language" when compared to the RTL modelling constructs.

Thus to assist RTL modellers we added several language features. These were implemented in the VPPreproc preprocessor that is a component of the Verilog-Perl package. VPPreproc reads in the RTL code containing macros and writes out standard SystemVerilog code with the macros expanded into assertion and coverage statements. The elegance of this approach is it works with all SystemVerilog simulators and formal tools.

The first construct we added was to collect a coverpoint at the current control-flow point in an always statement. This is used in an always blocks as follows:

```
always @* begin
  ...
  if (...) begin
    if (req & ack) begin
       // Format: $ucover_clk(clock, label)
       $ucover_clk(myclk, req_and_ack_label);
       // other non-coverage statements.
```

The preprocessor converts the $ucover_clk to a temporary signal and makes a clocked assertion at the specified edge, roughly as follows:

```
reg _tempsignal;
always @* begin
  ...
  _tempsignal = 0;
  if (...) begin
    if (req & ack) begin
       _tempsignal = 1;
  end
end

req_and_ack_label: cover property (@(posedge myclk) _tempsignal)
```

Note the work this saves the designer; their other alternatives were to replicate the expression in the cover statement, or create the temporary signal. Either approach is more effort, and error prone.

The above approach proved valuable, so we added an extension for covering busses:

```
always @* begin
   if (...) begin
      // Format: $ucover_foreach_clk(clock, label, "msb:lsb", (...$ui...))
      $ucover_foreach_clk(myclk, "my_label", "27:3", (i[$ui]));
```

This is requesting coverpoints for i[27], i[26]... i[3]. There's triple magic in this expansion. The first magic is the same conversion to a temporary-signal discussed with $ucover_clk. Second is the expansion of bit ranges "msb:lsb" to indicate a range from msb downto lsb inclusive. Finally

$ui is a magic variable replaced by the pre-processor with the loop index. This expands to approximately:

```
   reg [27:3] _tempsignal;
   always @* begin
      _tempsignal = 0;
      if (...) begin
         for (ui=27; ui>=3; ui=ui-1) begin
           if (i[ui]) _tempsignal[temp-index] = 1;
         end
      end
   end

   my_label_27: cover property (@(posedge myclk) _tempsignal[27])
   my_label_26: cover property (@(posedge myclk) _tempsignal[26])
   ...
   my_label_3:  cover property (@(posedge myclk) _tempsignal[3])
```

Coding this in pure SystemVerilog is not a one-line exercise. The one weakness with the current expansion however is that it creates N independent coverage points rather than a covergroup, but that's a fairly easy future improvement.

5. The CovVise Database

- Coverage databases are harder than you might think
- Consider how to archive the data
- Expect and support high insert rates

We started off CovVise knowing that keeping up with storing the data would be hard, but scaling up was more difficult than expected. The first problem was sheer magnitude of data. On this project we had ~150,000 coverage bins, and ~50,000 tests per day, but we wanted the system to be architected to support another ten times that level.

The first decision what data to keep. Keeping a full matrix of the count (say 4 bytes[4]) of each bin for every test would require nearly 30GB of data per day, which while not impossible is difficult to search and wouldn’t scale up by 10x or more. Thus we instead keep the best 4 (programmable) tests that hit each bin, which is sufficient for test grading and to answer questions such as “what test is hitting this low-coverage bin” that our previous system could not answer. We also allowed for collecting more data on “important” bins, and less on easy to hit bins, but did not need this feature in the end.

[4] Many other coverage systems allow you to optionally choose a smaller number of bytes of count per bin, perhaps even only one bit. This does reduce the data load, but doesn’t allow the user to see the relative hit rates between bins, which is important to determine when addresses or other distributions aren’t as uniform as expected.

The second decision was to allow for multiple database servers for collection, presentation, and archival.[5] This was greatly simplified by using MD5 hashes of test names and coverage points instead of an integer database row key. While hashes are longer, they have the nice property that you can lookup data in O(1) time. They also allow data to be pushed or pulled from one database to another quickly, without needing to search or re-index the data.

The third decision was how to deal with high insert rates; again, with 150,000 bins and 50,000 tests, we need to do 8 billion updates per day, or nearly 100,000 per second, which would swamp most affordable database systems. Thus we selected "Memcached", which is basically a hash table distributed across multiple servers. Using Memcached allowed the tests to push data into memory striped across N distributed servers, rather than needing to wait for a single database to insert data. The database was then updated with the aggregate data only when an entire regression series completed.[6]

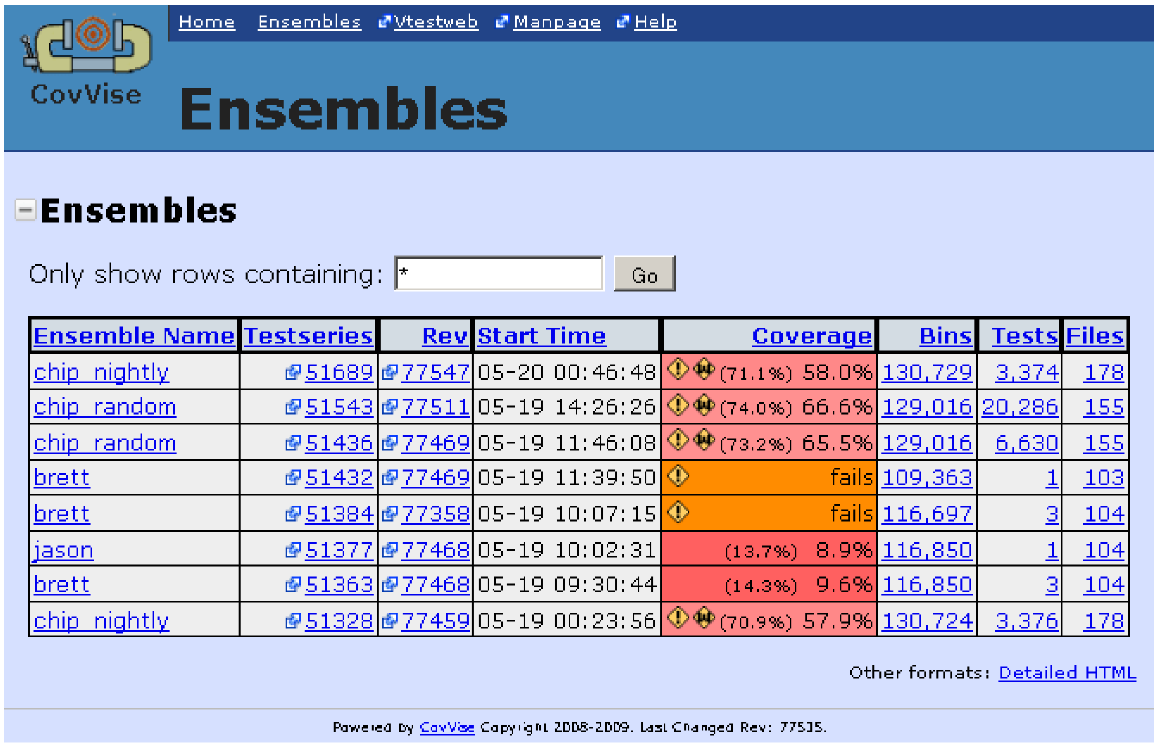
6. The CovVise Web Interface

- Users interface to CovVise data through the web
- Data is presented as hierarchy of coverage "pages"
- Coverage can be annotated onto source code listings
- Hits can be tracked across tests, and across historical runs

Users typically view CovVise data through an AJAX web interface. This begins with the CovVise home page, which is a simple list of "ensembles." Ensembles are our term for aggregations of coverage data. Most ensembles directly correspond to a "series" which is a collection of test runs created by a single script, such as a nightly regression. You can merge series together to reach coverage goals.

[5] SiCortex ended up only needing a single MySQL database server with 32GB of memory for collection and presentation, and archived per-month databases on a second 16GB server. We held ourselves to fit on this hardware by limiting ourselves to 150,000 coverage bins.

[6] The catch with Memcached is that a power failure will loose any coverage data collected since the last save. Losing up to 12ish hours of coverage data wasn't seen as a big deal, since we have only one unplanned power failure exceeding our UPS battery life every two-ish years.



Home Ensembles Vtestweb Manpage Help

CovVise

Ensembles

Ensembles

Only show rows containing: * Go

Ensemble Name	Testseries	Rev	Start Time	Coverage	Bins	Tests	Files
chip_nightly	51689	77547	05-20 00:46:48	! W (71.1%) 58.0%	130,729	3,374	178
chip_random	51543	77511	05-19 14:26:26	! W (74.0%) 66.6%	129,016	20,286	155
chip_random	51436	77469	05-19 11:46:08	! W (73.2%) 65.5%	129,016	6,630	155
brett	51432	77469	05-19 11:39:50	! fails	109,363	1	103
brett	51384	77358	05-19 10:07:15	! fails	116,697	3	104
jason	51377	77468	05-19 10:02:31	(13.7%) 8.9%	116,850	1	104
brett	51363	77468	05-19 09:30:44	(14.3%) 9.6%	116,850	3	104
chip_nightly	51328	77459	05-19 00:23:56	! W (70.9%) 57.9%	130,724	3,376	178

Other formats: Detailed HTML

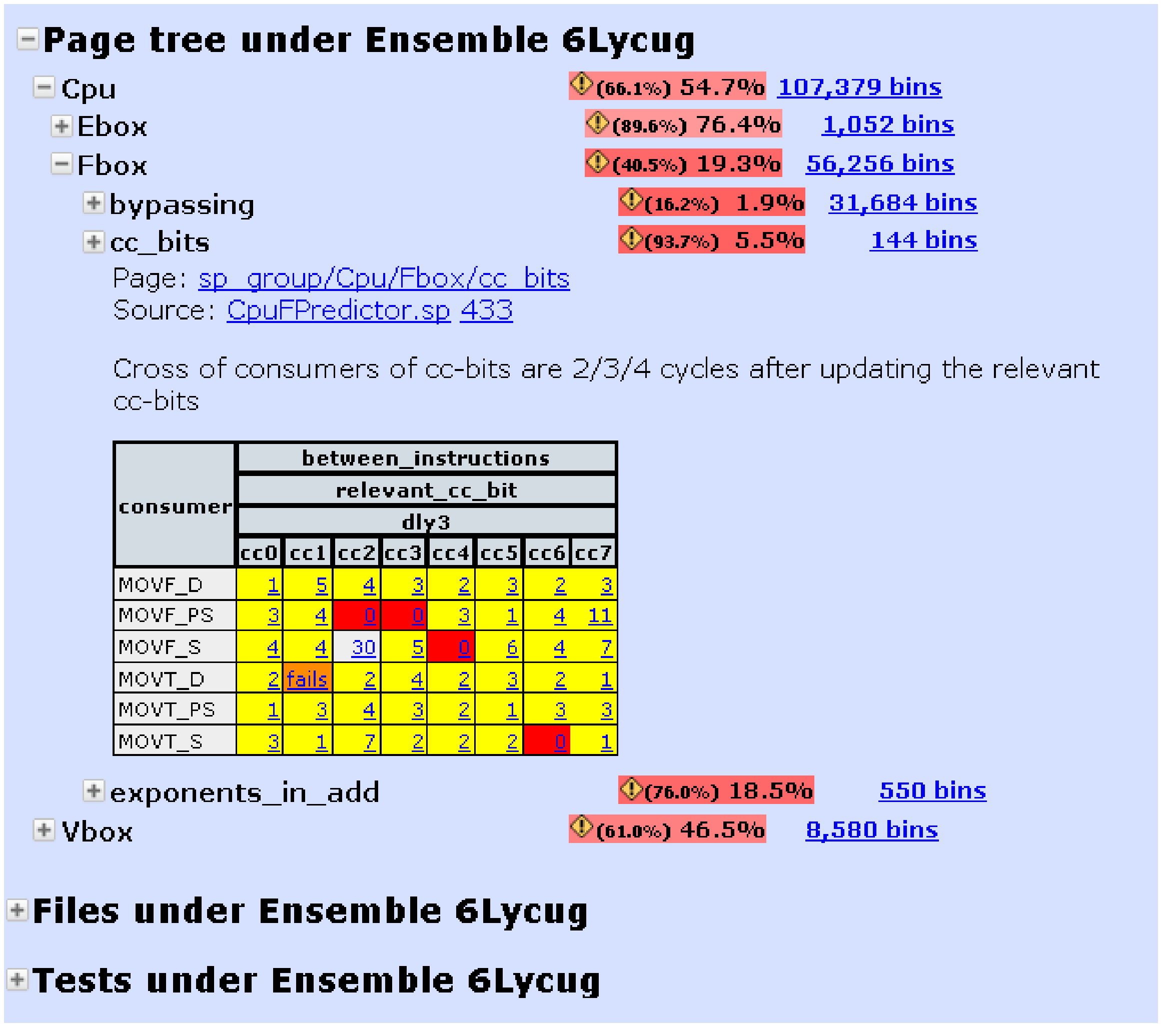
Powered by CovVise Copyright 2008-2009. Last Changed Rev: 77535.

Figure 1: CovVise Ensembles

CovVise presents all coverage percentages as two numbers, for example "(71.1%) 58.0%." This indicates that including "low coverage bins," those bins with < 10 hits per test as described previously, the coverage is 71.1%, while "ok coverage" which excludes those low coverage bins has a coverage of 58.0%. "Fails" means all tests failed so the passing coverage is zero. These coverage numbers may also be annotated with two little icons. A "!" icon indicate a bin exists under this ensemble which was hit only in a failing test; the user may "follow" the "!"s down to find out the bins that failed. A "W" icon indicates some coverage was waived under this ensemble, and may be likewise followed.

Ensemble Page Tree

Clicking on one of the ensembles brings up the ensemble page. This lists a series of coverage "pages". These pages are hierarchical and may be either created by a program (line coverage) or by the user when defining coverage points. The browsing of the tree is dynamic AJAX HTML; clicking on a "+" contacts the database and updates the page in real time.



Page tree under Ensemble 6Lycug

- Cpu — (66.1%) 54.7% — 107,379 bins
  - Ebox — (89.6%) 76.4% — 1,052 bins
  - Fbox — (40.5%) 19.3% — 56,256 bins
    - bypassing — (16.2%) 1.9% — 31,684 bins
    - cc_bits — (93.7%) 5.5% — 144 bins

      Page: sp_group/Cpu/Fbox/cc_bits
      Source: CpuFPredictor.sp 433

      Cross of consumers of cc-bits are 2/3/4 cycles after updating the relevant cc-bits

consumer	between_instructions							
	relevant_cc_bit							
	dly3							
	cc0	cc1	cc2	cc3	cc4	cc5	cc6	cc7
MOVF_D	1	5	4	3	2	3	2	3
MOVF_PS	3	4	0	0	3	1	4	11
MOVF_S	4	4	30	5	0	6	4	7
MOVT_D	2	fails	2	4	2	3	2	1
MOVT_PS	1	3	4	3	2	1	3	3
MOVT_S	3	1	7	2	2	2	0	1

    - exponents_in_add — (76.0%) 18.5% — 550 bins
  - Vbox — (61.0%) 46.5% — 8,580 bins

Files under Ensemble 6Lycug

Tests under Ensemble 6Lycug

Figure 2: CovVise Ensemble Page Tree

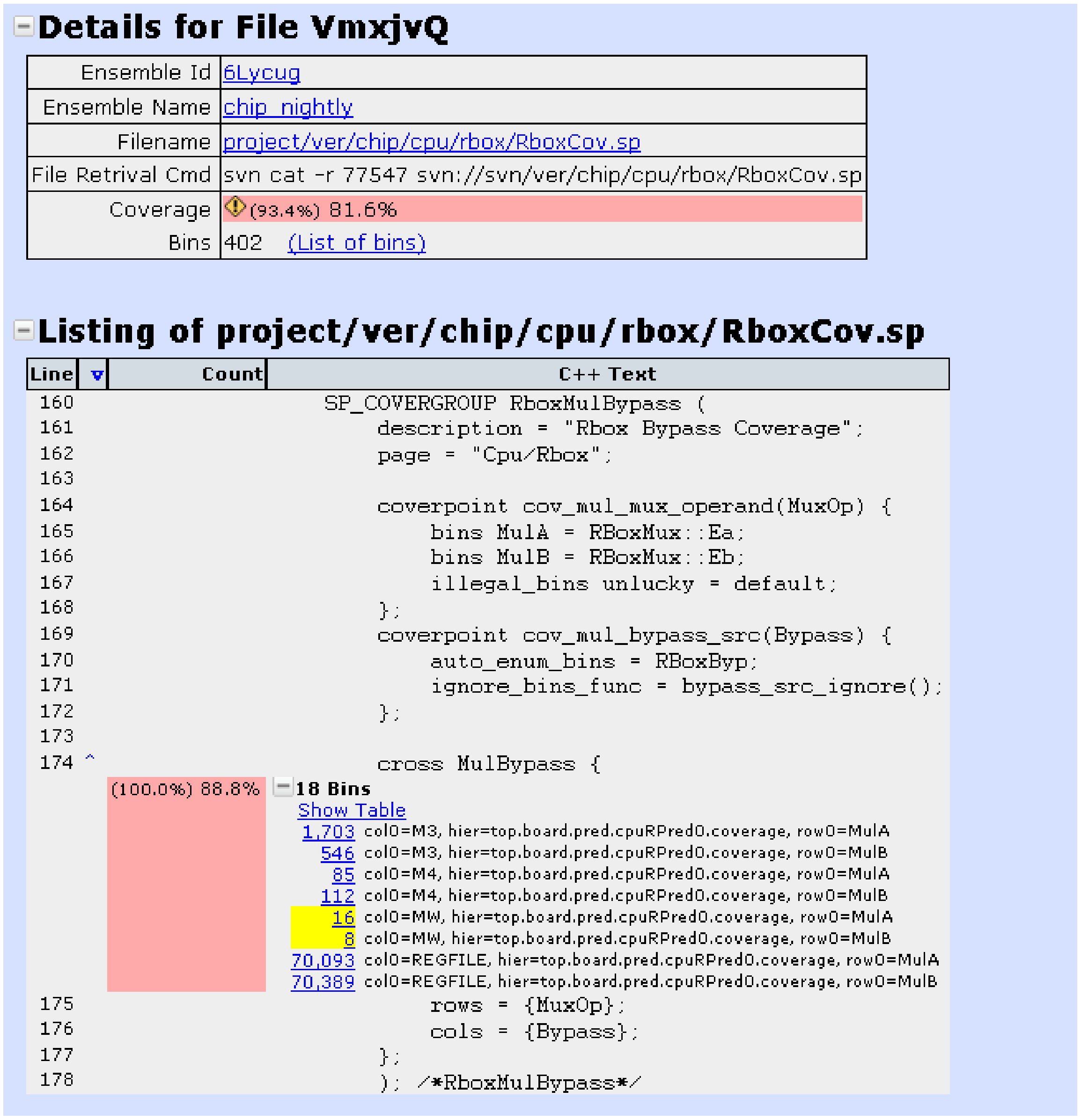
The page tree shows a coverage table. This two-dimensional table was created by a plugin that understands how to display the SystemPerl covergroup statements described above.

Clicking on the "Source" link will bring you to the source code that created these bins; see the File Annotation example below.

Not shown is a pop-up that appears when you hover over each bucket, showing the number of tests that hit the bin (or missed it) and the distribution of bucket hits.

File Annotations

Any file with coverage statements can be line-by-line annotated with the coverage inserted by that line, in this case a SystemPerl file. This started as a way to show line coverage, but proved a valuable debug tool for anyone writing coverage statements.



Details for File VmxjvQ

Ensemble Id	6Lycug
Ensemble Name	chip_nightly
Filename	project/ver/chip/cpu/rbox/RboxCov.sp
File Retrival Cmd	svn cat -r 77547 svn://svn/ver/chip/cpu/rbox/RboxCov.sp
Coverage	(93.4%) 81.6%
Bins	402 (List of bins)

Listing of project/ver/chip/cpu/rbox/RboxCov.sp

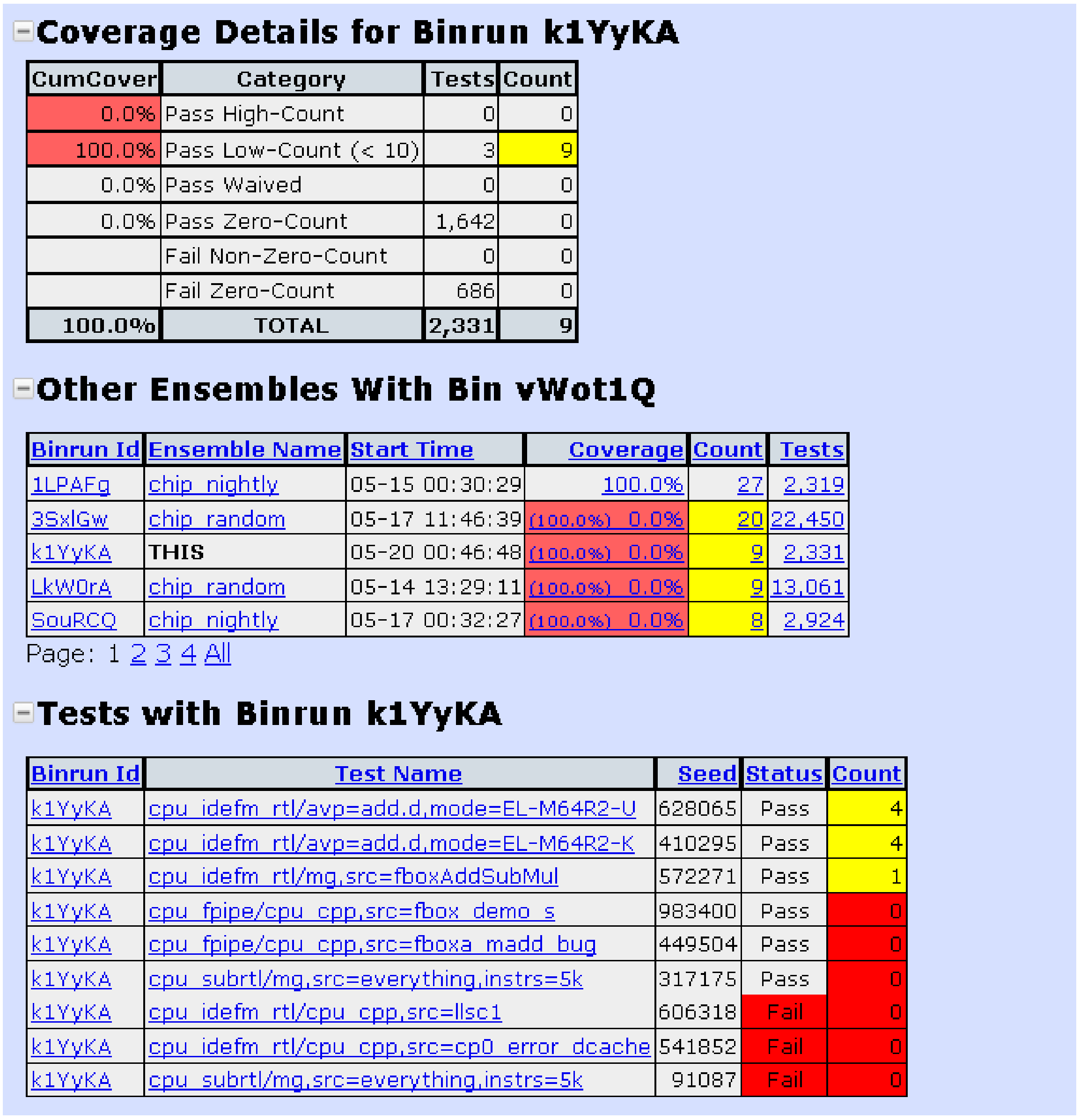
```
Line       Count           C++ Text
160                        SP_COVERGROUP RboxMulBypass (
161                            description = "Rbox Bypass Coverage";
162                            page = "Cpu/Rbox";
163
164                            coverpoint cov_mul_mux_operand(MuxOp) {
165                                bins MulA = RBoxMux::Ea;
166                                bins MulB = RBoxMux::Eb;
167                                illegal_bins unlucky = default;
168                            };
169                            coverpoint cov_mul_bypass_src(Bypass) {
170                                auto_enum_bins = RBoxByp;
171                                ignore_bins_func = bypass_src_ignore();
172                            };
173
174 ^                          cross MulBypass {
       (100.0%) 88.8%   18 Bins
                        Show Table
                         1,703 col0=M3, hier=top.board.pred.cpuRPred0.coverage, row0=MulA
                           546 col0=M3, hier=top.board.pred.cpuRPred0.coverage, row0=MulB
                            85 col0=M4, hier=top.board.pred.cpuRPred0.coverage, row0=MulA
                           112 col0=M4, hier=top.board.pred.cpuRPred0.coverage, row0=MulB
                            16 col0=MW, hier=top.board.pred.cpuRPred0.coverage, row0=MulA
                             8 col0=MW, hier=top.board.pred.cpuRPred0.coverage, row0=MulB
                        70,093 col0=REGFILE, hier=top.board.pred.cpuRPred0.coverage, row0=MulA
                        70,389 col0=REGFILE, hier=top.board.pred.cpuRPred0.coverage, row0=MulB
175                                rows = {MuxOp};
176                                cols = {Bypass};
177                            };
178                        ); /*RboxMulBypass*/
```

Figure 3: CovVise File Listing

Figure 3 shows multiple coverage bins were recorded for a single line of the file. Clicking on the "show table" link would show the multi-dimensional table similar to the one shown in the Page Tree above.

Bin-Run data

Clicking on any coverage bin brings you to the bin-run page. This shows the results for one coverage bin as tested under one ensemble (the run). The number of hits on the bin is divided into six categories based on the pass/fail status and number of hits. A test that passes and gets over 10 (programmable) as described previously goes into the Hi Count category. A test that hits a few times (1-10) goes into the Low Count category. A test that does not hit the bin goes into the zero count category.



Coverage Details for Binrun k1YyKA

CumCover	Category	Tests	Count
0.0%	Pass High-Count	0	0
100.0%	Pass Low-Count (< 10)	3	9
0.0%	Pass Waived	0	0
0.0%	Pass Zero-Count	1,642	0
	Fail Non-Zero-Count	0	0
	Fail Zero-Count	686	0
100.0%	TOTAL	2,331	9

Other Ensembles With Bin vWot1Q

Binrun Id	Ensemble Name	Start Time	Coverage	Count	Tests
1LPAFg	chip_nightly	05-15 00:30:29	100.0%	27	2,319
3SxlGw	chip_random	05-17 11:46:39	(100.0%) 0.0%	20	22,450
k1YyKA	THIS	05-20 00:46:48	(100.0%) 0.0%	9	2,331
LkW0rA	chip_random	05-14 13:29:11	(100.0%) 0.0%	9	13,061
SouRCQ	chip_nightly	05-17 00:32:27	(100.0%) 0.0%	8	2,924

Page: 1 2 3 4 All

Tests with Binrun k1YyKA

Binrun Id	Test Name	Seed	Status	Count
k1YyKA	cpu_idefm_rtl/avp=add.d,mode=EL-M64R2-U	628065	Pass	4
k1YyKA	cpu_idefm_rtl/avp=add.d,mode=EL-M64R2-K	410295	Pass	4
k1YyKA	cpu_idefm_rtl/mg,src=fboxAddSubMul	572271	Pass	1
k1YyKA	cpu_fpipe/cpu_cpp,src=fbox_demo_s	983400	Pass	0
k1YyKA	cpu_fpipe/cpu_cpp,src=fboxa_madd_bug	449504	Pass	0
k1YyKA	cpu_subrtl/mg,src=everything,instrs=5k	317175	Pass	0
k1YyKA	cpu_idefm_rtl/cpu_cpp,src=llsc1	606318	Fail	0
k1YyKA	cpu_idefm_rtl/cpu_cpp,src=cp0_error_dcache	541852	Fail	0
k1YyKA	cpu_subrtl/mg,src=everything,instrs=5k	91087	Fail	0

Figure 4: CovVise Bin Data

Coverage Graphs and Metrigator

We wanted to get graphs of the coverage data not just as filler for management meetings, but to help us identify trends. Rather than integrate graphing directly into CovVise, we made a new tool, Metrigator, which accepted data from a plugin. Metrigator was our common collection database for recording verification metrics and design metrics over time. This included:

- CovVise coverage (percent low coverage, ok coverage, number of bins)
- Headcount (total, by role)
- Bug count (total, closed, per-component, by priority, etc)
- Bug closure rate (total, per-component, by priority, etc)
- Source code commits (size, number of edits)
- Verification test success (number of tests, failures)

Integrating this into a single tool makes it easy to spot correlations. It also spared users from learning multiple graphing controls, and saved us effort from writing multiple graph interfaces.

Below shows one graph of the SiCortex coverage percentage over time. In this example there are four coverage percentages displayed. From the top line down to the bottom line these are: low coverage for a random regression and low coverage for the nightly regressions, followed by ok coverage for random and nightly (although sometimes the nightly regression gets better coverage so is sometimes above). Also note the overview of several months on the panel at the left, complemented by the zoom in on the right panel.

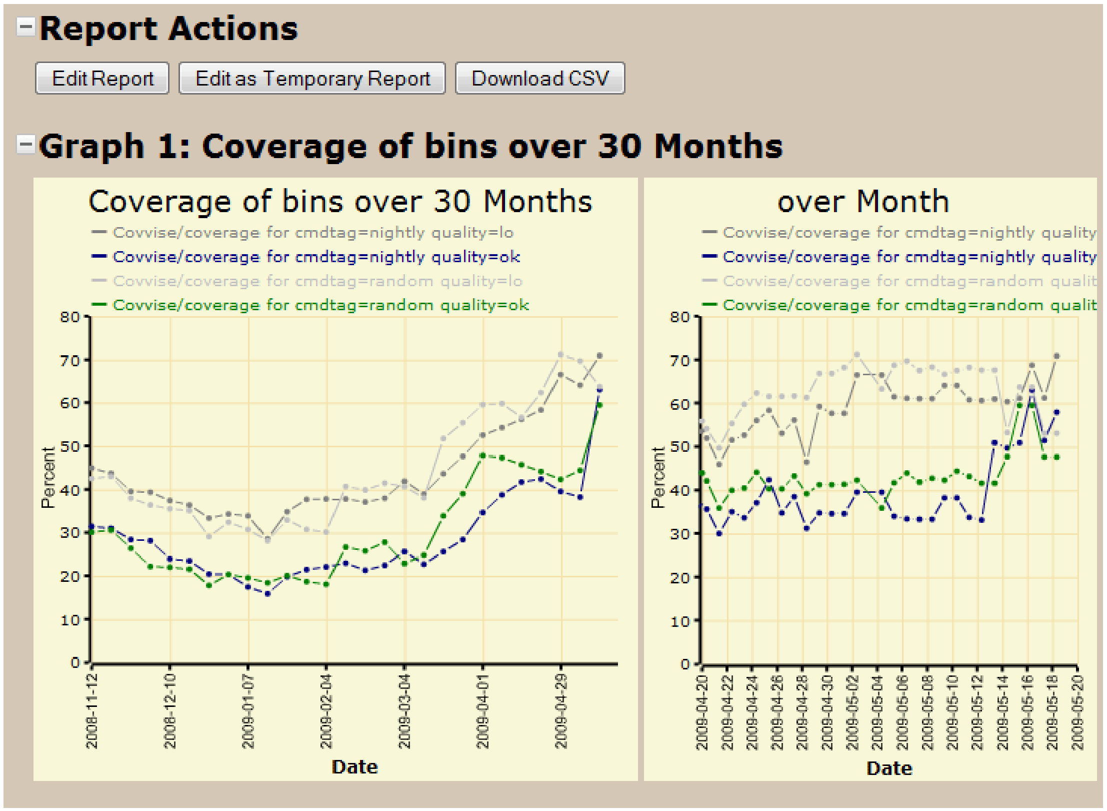


Figure 5: Coverage Percentage Graph

The trend towards coverage completion is obvious here, as is that at the time of writing we had a good ways to go to plateau and meet our coverage targets. Note also how the low coverage is approaching the ok coverage; in addition to normal progress on better tests, this also represents an effort around the date of the graph to mark some large tables with lower coverage limits.

7. Futures

- Additional coverage tools
- Coverage crosses with time
- Grand loop and automated test selection

Although our project is not complete, CovVise was working well for us. We however see some obvious areas for additional improvements. First, coverage data isn't yet integrated with the data from all of our commercial simulators, but we know how to write the plugins to do so.

Second, we found that many verification engineers were spending lots of effort on specifying "timing window" sequence coverage to try to measure pipeline cases like "did we see an instruction consume the output of an older instruction each of 0,1,2,3,4,5,6 cycles ago?" This code tends to involve ad-hoc circular arrays to store previous values of a sampled structure, and there is a lot of commonality that could be abstracted away. We thus desire to add new declarative syntax to make it easier to specify coverage of timing windows, including crossing windowed coverpoints with other coverpoints. A single declaration should suffice to measure coverage of a complete bypass network.

CovVise provides us with a database of coverage results, including lists of tests that are hitting new bins, and more importantly tests hitting "low" or "failed only" coverage bins.[7] We occasionally run these reports and did minor tuning of our regressions, but the hope is to get to a fully automated system where every night's runs would be tuned based on this data.

8. Conclusions

By throwing away an old coverage system, and revisiting some traditional assumptions, SiCortex was able to greatly improve our coverage techniques. First, verification engineers could use powerful SystemC language extensions to specify their coverage measurements. Second, by adding coverage earlier in the project, we could better track what work remained. Third, by enlisting RTL designers to add coverage, we found interesting internal cases that were important to hit. Fourth, by collecting data on failing tests we could see what bins were hittable if we were just to clear those failures. Finally, by separating out "low-hits" we could tell what fields were randomized versus just set up at initialization time.

This was implemented using a robust combination of SystemC and SystemVerilog for entering bins, plus advanced database techniques using MySQL and Memcached, all visible from a web browser.

[7] Again, both "low" and "failed" bins are important because we know they aren't impossible and haven't been stressed much, so are well worth more machine time.

9. References

[1] http://www.veripool.org/covvise

[2] http://www.veripool.org/systemperl

[3] http://www.veripool.org/verilog-perl

[4] http://www.sicortex.com

[5] http://en.wikipedia.org/wiki/Test-driven_development

[6] http://danga.com/memcached