The Cost of SoC Bugs

Ken Albin
Hardware Advanced Development
Oracle Labs
Austin, TX 78727
Email: ken.albin@oracle.com

***Abstract*- Decisions about hardware design flow investments in tools and methodology are using models based on 40 year old software development data from the era of programming with punch cards. Even though the supporting data for these models is lacking, the trends depicted do have some similarity to what is observed on real projects. A new modeling framework for the cost of bugs is proposed, along with a discussion of how to use the resulting models to make effective methodology and resource allocation decisions.**

I. INTRODUCTION

Enabled by Moore's law and driven by market forces, System-on-Chip design projects continue to grow is size and scope. While platform-oriented approaches have made SoC software development somewhat more tractable, hardware development is still an open problem.

Verification is a major part of hardware development efforts - 57% of ASIC/IC projects according to the 2014 Wilson Research Group study sponsored by Mentor Graphics [1]. The same study reports that verification engineers spend 37% of their time in debug activities.

A key to successful hardware development is selecting appropriate verification tools, methodologies, and resource allocation. Understanding the cost of bugs provides a basis for making those tradeoffs. These topics are also applicable to SoC software, but this paper will focus on the hardware development aspects.
.

A. Sources of Bug Costs

Four major cost categories associated with design defects are:

1. Missed market windows.
2. Liability for safety or security defects.
3. Damage to a company's reputation.
4. Engineering costs of finding and repairing defects during development.

The first three cost sources are potentially much larger than the fourth, but they are largely out of direct control of the engineering staff and out of scope for this paper. Eliminating bugs during development and improvements in quality, however, positively impact the other cost categories.

B. The Power of Ten

Fig. 1 below was presented at DAC 2004 as part of a verification panel and expresses the widely accepted notion that bugs cost more to fix when found later in the design process. The costs in this case are attributed to the rework in the stage where the defect was found.

Clearly bugs can cost more to fix in some hardware development stages than in others, but the ten times multiplier for very different stages seems unlikely. Similar estimates are common in industry literature (especially in vendor marketing presentations) but there don't appear to be any public studies supporting these ratios.

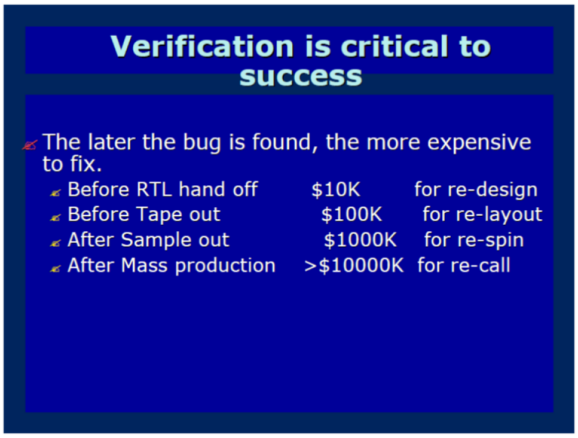

Figure 1. Typical hardware bug cost table. [2]

Further, defects are not always found in the stage in which they are created, which can lead to rework in multiple stages. A more accurate model of cost components would allow us to make more reasoned decisions about tool and methodology choices to reduce verification cost.

The table in Fig. 1 and many similar estimates for hardware development appear to have taken their inspiration from software metrics. This seems reasonable - with synthesizable hardware description languages, behavioral testbenches, and increasing use of software models, hardware design projects are looking more and more like large, complex software development projects.

II. Software Defects Cost-to-fix

A quick search of the software engineering literature turns up many charts illustrating the idea that defects become exponentially more expensive to fix in later design phases. Often they show the same power of ten per development phase multiplier found in hardware development literature as seen in Fig. 2.

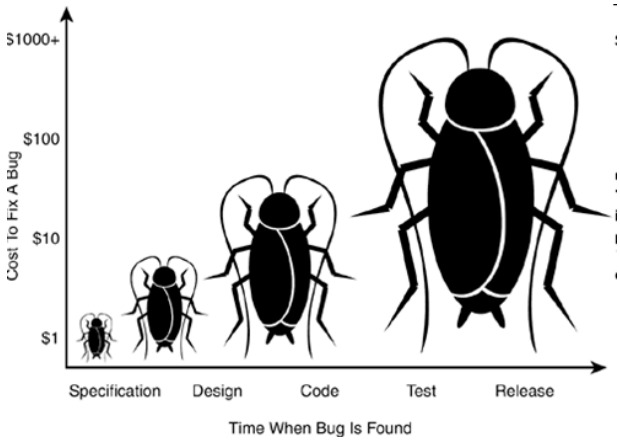

Figure 2. Cost per defect in a more whimsical form. [3]

Often the idea is expressed in a less whimsical but no more accurate chart or table and used to make serious arguments for process improvements ranging from more code reviews to agile methods. What is missing from nearly all of these charts and tables are any description of what is actually being counted as the cost of fixing the defects, and in what phase the defects were introduced.

A. Software Metrics Archaeology

If you trace through the tangle of references and attempt to find original sources as Graham Lee did in [4], you eventually end up with what appears to be the most substantial source - the chart in Fig. 3 with the results of studies from the late 1970s.

Despite some confusion over what is actually being counted as cost and the studies having been conducted at a time when punch cards were the norm, these results live on in many forms. The data has been interpreted in a variety of ways, generalized, and extrapolated first for new software development methodologies and now hardware development.

The original interpretation was that errors found in later phases of development are more expensive to fix since they have more documentation and other collateral that would have to be revised.

A more recent interpretation is that the "longer a defect is latent in the design," the more expensive it is to fix. This is often illustrated with a simple exponential curve on a linear scale chart as shown in Fig. 4. The rationale here is that the amount of rework increases exponentially with time.

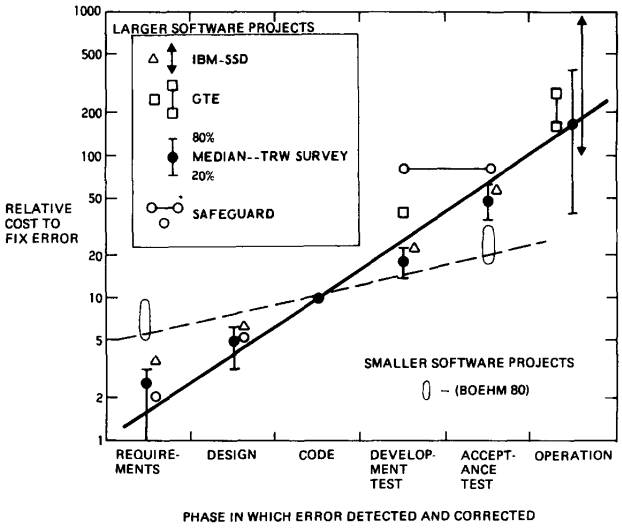

Figure 3. Early Cost-to-fix studies. [5]

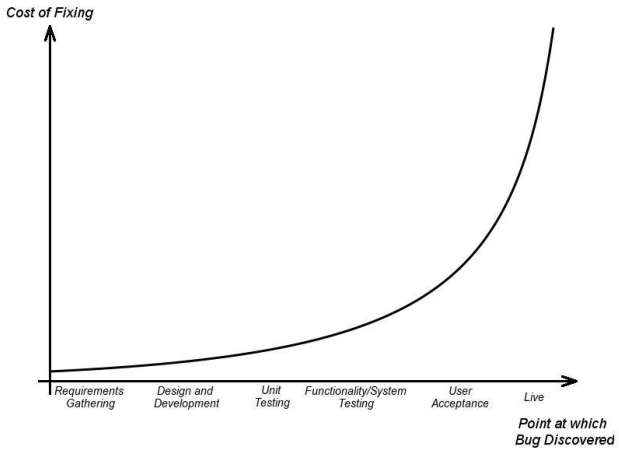

Figure 4. "…graph showing the rough relationship between the cost of fixing a bug or defect, and what stage of the development process that bug or defect was found. [6]

B. Phase Containment

"Time latent in the design" was later refined to reflect the idea that a defect is cheapest to find and repair in the project phase in which it was introduced, as illustrated in Fig. 5. It seems reasonable that an error might be easier to identify and fix in the stage it was created since the team is familiar with the design details and has an environment that provides visibility at the correct level of abstraction. Additionally, the overhead for communication and releases between groups would be less.

But note that even with the segmenting of development phases, the exponential curve is still present in Fig. 5.

This "time latent in the design" interpretation gave rise to the idea of "phase containment" of errors, with associated methodologies and metrics [7]. This phase-based focus makes sense in "waterfall development" where each stage depends on previous stages.

While non-waterfall development methods are generally favored in software development, hardware development has to deal with dependencies arising from step-wise physical implementation steps, often performed by specialized teams at different sites.

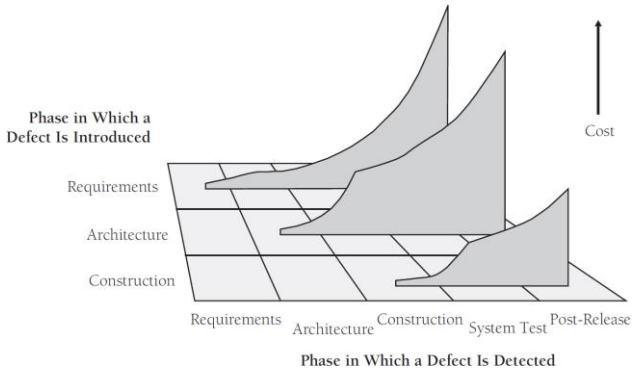

Figure 5. Cost of defects by phase created and found. This figure is from Code Complete, 2nd Edition, by Steve McConnell, © 2004. All Rights Reserved. Used with permission of the author. [7]

III. A Proposed Modeling Framework for Bug Costs in Hardware Development

A more accurate model for the cost of SoC bugs must account for 1) cost components within a development stage, and 2) the interaction overhead between stages. The specifics of the cost components vary from one stage to another and in the interactions between stages, but a framework can be parameterized with the specifics.

A. Costs of preparing and accepting deliverables

In order to model a design stage, the hand-off of deliverables from one stage to the next must be accounted for. Depending on the stages involved, the deliverables could be a written set of requirements, a C++ reference model, IP RTL, verification IP, netlists, etc.

There is a cost incurred by a stage accepting a deliverable from a previous stage: unpacking, integration work, updating testbenches, regression testing, updating documentation, etc. This is represented by the yellow arrow in Fig. 6.

There is also a cost incurred by a stage when preparing to release a deliverable for a subsequent stage: more extensive testing, documentation, checklist reviews, tagging, packaging, etc. This is represented by the green arrow in Fig. 6.

In both cases, the cost could be large or small depending on the specifics of the deliverables and the amount of automation.

A stage can receive deliverables from multiple sources, but only one source is shown in the diagram for simplicity. In particular, project-level requirements and specifications need to be communicated throughout the design flow, and implementation details need to be passed on from prior stages.

B. The cost of a bug-free design

If given a design that had no defects, what would need to be done to confirm it was correct?

A normal verification plan would be created and executed: building testbenches, creating tests, measuring coverage, holding code reviews, proving properties, etc. until the plan was complete.

For now, assume the bug-free design is complemented by a bug-free verification environment that is constructed as part of the verification plan.

The activity needed to confirm a defect-free design defines the stage's activity (and cost) baseline, shown in Fig. 7 as the purple circle in the stage. This is work that always has to be done, and errors in the design or verification environment represent additional incremental costs above this baseline.

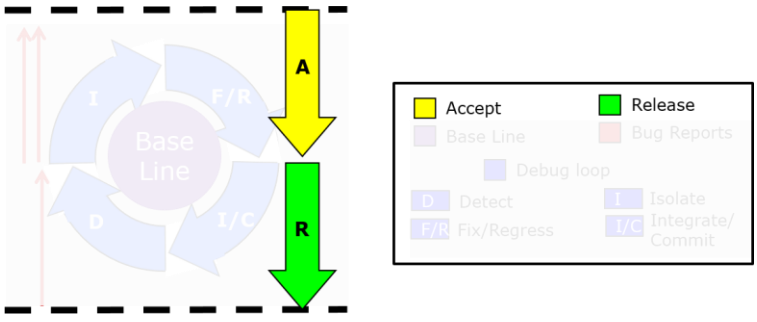

Figure 6. Deliverable cost components.

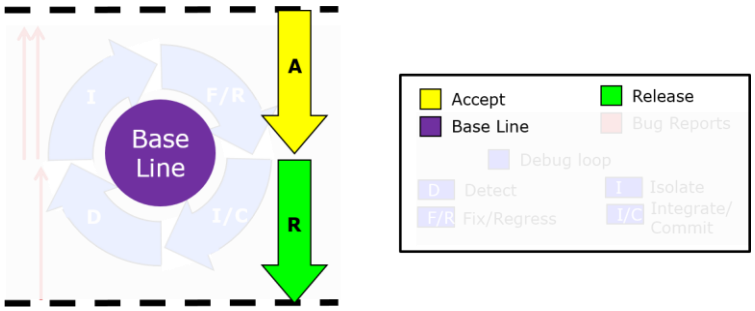

Figure 7. Baseline work cost component.

C. The incremental costs of bugs

A common idea in industry is the "debug loop," shown in Fig. 8 as the blue arrows.

The debug loop is divided into four segments:
1) Detection (linting, random testing, code inspections, property proofs, etc.)
2) Isolation (manual debug effort)
3) Fix and Regression testing
4) fix Integration and Commit

Note that there are likely Fix/Regress or other iterations within the overall debug loop, but they won't be separated out for this top-level model. Iterations within the debug loop would be a logical place to look for efficiency improvements by adding the next level of detail.

The ratio of resources used for Detection vs. Isolation will change over the course of the project: with an immature design it is easy to detect more bugs than engineers have time to isolate and fix; towards the end of the project much more resources are needed to detect bugs for engineers to isolate and fix.

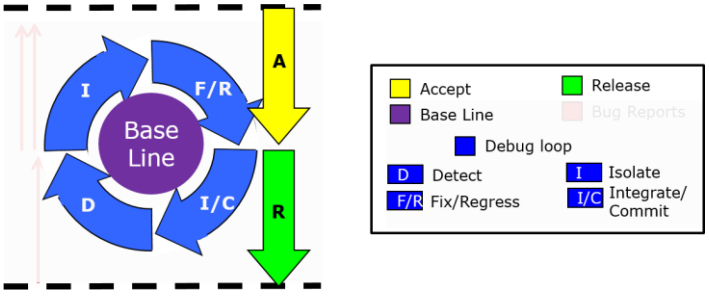

Figure 8. Debug cycle cost component.

D. *Variations on the Debug Cycle and Interaction with Other Stages*

Not all bugs use the entire debug cycle:

- A bug that was created in the current stage may be detected and isolated but not fixed due to resource constraints or other planned changes.
- A bug in a deliverable from a previous stage can be detected and at least partially isolated in the current stage, but will generally be reported to the stage where it originated (shown as red arrows in in Fig. 9) where it will be fixed and included in a future release.
- A bug introduced in the current stage may go undetected until after it is delivered to a subsequent stage. While the downstream stage pays the price of detection, the current stage has to execute the rest of the debug loop steps and possibly initiate a delivery with the fix.

E. Putting It All Together

The model described above provides a framework to account for most of the activities performed in a single development stage. Using this framework, project or organization-specific cost-to-fix models can be constructed.

Rather than generic curves, stage models can be composed to model the entire SoC design flow. The per-stage costs in waterfall development can be understood as a function of cost components:

- the increased difficulty in detecting and isolating bugs in a downstream context
- required rework (additional deliveries of designs, testbenches, documentation, etc., possibly through multiple stages)
- communication overhead across team and function boundaries

For each additional downstream stage, the difficulty of debug and amount of rework increase in measurable ways.

IV. Applying the Cost Model

The insights that led to this modeling framework have come from successful application of partial models in different contexts:

- creating a spreadsheet to do impact analysis of IP defect reduction for an SoC
- analyzing costs of finding bugs in early ("debug saturated") and late stage ("bug hunting") verification
- reversing a multi-million dollar business decision on a debug tool based on data collected on the debug loop

Some approaches that proved useful in applying cost models are described below.

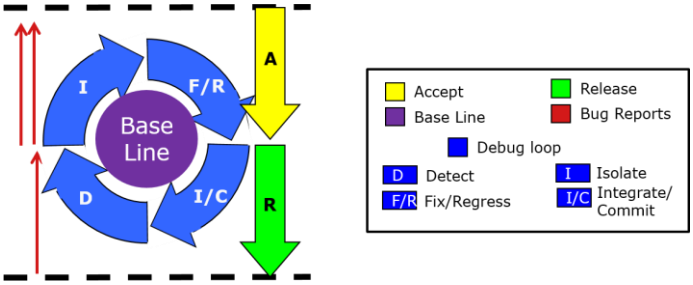

Figure 9. Bug reporting cost components.

A. Estimating to fill in gaps in collected data

While all of the data needed for the model is generally available, it is not always tracked or may be tracked but not split into the necessary categories. Estimates or ratios of available data are enough to get started.

For example, an estimate that 5% of bugs came from upstream deliveries could be used. Based on estimates of what it costs to detect and isolate a bug, the amount of investment that would be worthwhile to reduce or eliminate these upstream bugs can be computed.

B. Conversion to dollars

The data to be collected is in the form of engineering time, compute resources, and license usage plus counts of deliveries, bugs found, etc. These quantities can be converted into dollars in a spreadsheet and used for evaluating automation trade-offs, and cost-benefit analysis of tool or capital purchases.

C. Evaluating Proposed Methodology and Tool Changes within a Single Stage

Single stage improvements are interesting because they are generally within the scope of the team owning the stage to identify and implement. Within a stage, specific cost components were identified: accepting and preparing deliveries, performing baseline work, debug loop iterations, and bug reporting. In each case, to improve efficiency you can:

- reduce the cost of an action, or
- reduce the number of times you perform the action.

Here are some examples of ways to improve each of the single stage cost components:

- Reduce costs in accepting and preparing deliveries:
  - Automating release/acceptance procedures shrinks the per hand-off cost.
  - Improving IP quality will reduce the need for additional unscheduled releases.

- Reducing baseline costs:
  - Approaches that work at a higher level of abstraction such as behavioral models or testbenches also offer possible gains, explaining the popularity of SystemC, and SystemVerilog TestBench (SVTB) language features.
  - The single biggest way of reducing the baseline cost is through reuse: existing designs, testbenches, infrastructure, tests, verification components, etc. Universal Verification Methodology (UVM) and other libraries are intended to enable increased reuse.

- Faster debug loop:
  - linting and static analysis can detect many bugs more quickly than simulation
  - assertions are very effective at localizing an error, speeding isolation
  - more compute resources could speed the fix/regress component.
  - improvements on loops such as fix/regress within the overall debug loop.

- Reduce the number of debug loop cycles:
  - The comments on base-line efficiency apply here: reuse and higher-level languages create fewer bugs.
  - Early use of techniques like linting and code reviews can allow errors to be found and fixed without the debug loop being invoked or the bug logged.
  - Errors passed from upstream stages can be reduced by better upstream quality and more thorough acceptance criteria.

Adding cost to acceptance testing vs. the cost of debugging upstream bugs is a good example of the type of tradeoffs that can be made within a stage.

D. Evaluating Proposed Methodology and Tool Changes over Multiple Stages

While many efficiency improvements can be made within a stage by the team owning the stage, some significant improvements can only be done with benefit of the big picture and may require encouragement from management for cross-team cooperation:

- shift resources to an earlier stage to improve quality on deliveries
- when planning, focus more on goals than specific tool flows (e.g., X analysis vs. gate-level simulation)
- deliberately plan which environments should be tightly coupled (e.g., to enable faster emulation builds or more VIP reuse) and which should be loosely coupled (e.g., to accommodate separation by organization or site)
- identify and address "stage containment" problems with bugs being discovered downstream, especially more than one stage downstream
- balance the benefit of pushing checks to upstream stages with the impact on the upstream stage.

V. Some Practical Considerations of Measuring Cost Components

The stage model framework requires some data that not every project tracks, but much of it is available (e.g., license usage logs of debug tools, regression resource usage identified by naming conventions, etc.).

Estimates of activities (e.g., 10% of bugs come from prior stages) are enough for quick early analysis. Values estimated initially can be collected directly later, and areas needing closer scrutiny (e.g., fix/regress loops in the debug cycle) can be modeled in more detail.

Data needs to be collected from a number of different sources such as:

- bug counts from issue tracking system(s)
- compute usage statistics from the batching system
- license usage statistics, especially for interactive tools
- staffing estimates or headcount tracking
- project phase, number of deliverables, etc. from project management

Bugs created and fixed in the same stage must be distinguished from bugs passed from one stage to another. This can be done with attributes in the bug record ("stage detected", "stage introduced", and possibly "stage fixed") or through commit comments or other issue tracking system mechanisms.

The cost for detecting a bug may include stimulus, coverage, simulations, triage, property checking, etc. Because it is common to be working on more than one bug at a time, it is reasonable to compute a cost average: amount of work on a particular release or over a period of time divided by the number of bugs.

VI. A Note of Caution

Capers Jones argues in [8] that software cost-to-fix is not directly related to the phase itself, but rather to the number of bugs being found and fixed in that phase. The net effect, Jones asserts, is to make quality improvements appear more expensive (overall costs amortized over fewer bugs leads to a higher apparent cost per bug).

Along the way he builds a simple cost model of the software development process and uses it to illustrate how different accounting of costs can lead to entirely different conclusions. While there are differences between software and hardware development, his cautions about drawing bad conclusions from data are valid.

VII. Conclusion

In order to make important decisions about hardware design flow investments in tools and methodology, 40 year old software development data from the era of programming with punch cards is being used which has been misinterpreted, generalized, and extrapolated first for new software development methodologies and now hardware development.

Even though the supporting data for these charts is lacking, the trends depicted have some similarity to what has been observed on real projects. A more accurate model of the hardware development process based on cost components should allow us to better evaluate tool and methodology tradeoffs and better allocate resources. A development stage modeling framework was proposed that can be parameterized with project data.

Future work will focus on creating stage models for different activities in the hardware development process and composing them to provide a basis for identifying and evaluating overall design flow improvements.

Acknowledgment

An early version of this work was done while the author was at AMD [9] and has been significantly improved through discussions with many people in the design verification community. Special thanks to Oracle Labs for supporting the development and presentation of this paper.

References

[1] 2014 Wilson Research Group Functional Verification Study sponsored by Mentor Graphics https://blogs.mentor.com/verificationhorizons/blog/2015/07/13/part-8-the-2014-wilson-research-group-functional-verification-study/
[2] M. Ishii, DAC 2004 Verification Panel, LSI Design Division1, SoC Solution Center, SSNC, Sony.
[3] R. Patton (2005). Software Testing (2nd ed.).
[4] G. Lee. http://www.sicpers.info/2012/09/an-apology-to-readers-of-test-driven-ios-development
[5] B.W. Boehm, Software Engineering Economics, Prentice-Hall, Englewood Cliffs, NJ (1981).
[6] G. Coates, https://www.chromosphere.co.uk/2009/03/10/testing-fixing-and-costs/
[7] S. McConnell, Code Complete, 2nd Edition (2004).
[8] C. Jones, http://namcookanalytics.com/wp-content/uploads/2013/07/COST-PER-DEFECT-2013.pdf
[9] K. Albin Navigating the Rapids and Eddies of Your Design Flow, IEEE Technology Management Council, Austin and Central Texas Chapter, November 2009.