Practical Methods in Coverage-Oriented Verification of the Merom Microprocessor

Alon Gluska
Intel MG
MATAM
Haifa, Israel
Alon.gluska@intel.com

ABSTRACT

Functional coverage is a well known means of measuring verification progress. However, approaches to coverage, such as Coverage Driven and Coverage Oriented approaches, are often difficult or impractical to implement. This paper presents the coverage methodology used in the verification of Merom, Intel's first converged-core microprocessor. We describe practical methods and applied techniques which enabled a high return on a significantly reduced investment in coverage measurement and analysis. Given the tight schedule, this approach provided a clear metric for measuring verification progress and for effectively steering resources to improve the quality of the design under test.

Categories and Subject Descriptors

5 [**Verification**]: Functional verification, RTL modeling and verification of hardware designs.

General Terms and Keywords

Logic design. Logic verification. Coverage. Functional coverage.

1. INTRODUCTION

Functional coverage, derived from the explicit functional specification of the design, is widely acknowledged as a central technique for measuring the thoroughness of verification[1][4][7][10]). Functional coverage serves not only to reflect the quality of testing, but also to steer the verification resources toward areas of insufficient testing [5], and to provide a measurable indicator of the progress of the verification.

The definition of coverage tasks is derived from manual interpretation of the specification, micro-architecture and implementation details. Multiple methods to automatically generate coverage spaces have been presented (e.g. [3]). However, none of these methods seems mature enough to adequately produce the desired coverage space. To date, there is no known method to formally or automatically create the full set of coverage cases, therefore its development is primarily manual.

The Coverage-Driven Verification (CDV) approach makes coverage the core engine that drives the entire verification flow [9]. Coverage space is defined a priori, and coverage results are used to measure the quality of the random testing and to steer resources until a satisfactory level of coverage is attained. This, in theory, makes it possible to reach high quality verification in a timely manner.

However, we claim that a pure Coverage-Driven approach is impractical for most designs, as became evident in the course of the verification of Merom, Intel's first converged-core microprocessor. The CDV approach requires the assignment of large number of knowledgeable and experienced engineers at the outset of the verification period. Inevitably, this is done instead of a basic cleanup of the preliminary RTL design. It is reasonable to assume that most projects cannot afford such an approach.

Multiple methods were adopted in Merom verification to make coverage practical and reach optimum return on investment. In particular, we determined that a significant portion of the testing space could be dropped with a minimal impact on quality. We defined targets per coverage monitors which were not necessarily 100%. We focused on interfaces, introduced a counting mechanism, used sliding coverage windows and more. These enabled carrying out the comprehensive coverage program under the tight schedule and staffing constraints of the project.

The paper is organized as follows. We first briefly present the Merom microprocessor and its verification. Then we describe the drawbacks of the pure Coverage-Driven Verification approach, as were evident in Merom. We then present Merom coverage methodology and list some key methods of making functional coverage practical. Finally, we present our results and findings.

2. MEROM AND ITS VERIFICATION

2.1 Merom Microprocessor

Merom is Intel's first converged-core microprocessor manufactured in the 65-nanometer technology. Its architecture combines EM64 and other capabilities of NetBurst with the power-saving features of the Pentium-M platform. The chip, consisting of two CPU cores, embeds security and virtualization, and a long list of advanced micro-architecture enhancements for performance and power saving. Merom's energy-efficient, multi-core design is expected to deliver three times the performance per watt with respect to previous generations and has different configurations for mobile, desktop and server products.

2.2 Merom Verification

Merom design was divided into 6 logical clusters having clear and stable functional boundaries. Consequently, we developed a Cluster Test Environment (CTE) for each of the RTL clusters and multiple Unit-Level Test environments (ULTs) for specific units. Most of the verification was carried out at these levels, and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2006, July 24–28, 2006, San Francisco, California, USA.
Copyright 2006 ACM 1-59593-381-6/06/0007…$5.00.

testbenches were designed to cover included functionality. Special attention was given to accurate modeling of the interfaces, for in term of protocol and in terms of mimicking all possible real-life behaviors of the neighboring functions.

Verification at the full-chip (FC) level targeted weaknesses in the CTEs and Merom's compliance with its architectural definition. This modular approach, in which FC is used to compensate weaknesses in the CTEs, particularly with regard to interfaces, is a common practice in the industry [4] [8] [9] [11].

Two-thirds of the RTL bugs reported in Merom were found at the cluster or unit level. This number reflects the high effectiveness of the CTEs. Bugs found at the FC level exposed numerous flaws and inconsistencies in the interfaces between clusters that were not fully or accurately modeled by the CTEs.

An additional important activity was formal verification (FV) for selected functions. Most FV was carried out by dynamic verification engineers with the support of FV experts. FV found 3% of the bugs, with high impact where applied.

3. COVERAGE DRIVEN VERIFICATION (CDV) AND ITS DRAWBACKS

3.1 Functional Coverage

Functional coverage is widely known as a means of measuring the quality of verification. The idea is to systematically create a comprehensive list of conditions and to verify that each of them is covered during simulation. Coverage is then used to steer test generation towards conditions that have not been hit yet. In addition, coverage provides a quantitative way of measuring the progress of verification and the extent to which the verification cycle is complete

Progress reports show how coverage progresses over time, and are useful for forecasting the potential coverage value of the existing test suite. Status reports quantitatively show which conditions are covered and, more importantly, which are not. This information helps the verification team direct the testing towards untested or weakly tested areas in the design [7].

3.2 Coverage Driven Verification

It is believed that a large portion of the interesting test cases can easily be hit by random or direct-random testing. If there is no feedback regarding the quality of these tests, a huge number of simulation cycles will be executed, many of which contribute very little to the overall verification. Furthermore, to ensure hitting these cases, engineers often craft directed tests manually.

The Coverage Driven Verification (CDV) approach copes with this problem by using functional coverage as the core engine that drives the verification flow [1] (see Figure 1). In CDV, verification starts with a coverage plan derived directly from studying RTL functionality. The simulation environment, test generator, and coverage tools are designed accordingly to facilitate the implementation of the coverage plan. Random tests are generated and simulated. Coverage results are then used to steer verification resources toward the coverage holes. Once the desired coverage is achieved, the product is ready to tape out.

Note that feedback from coverage analysis is typically used to enhance and modify the simulation environment and the test

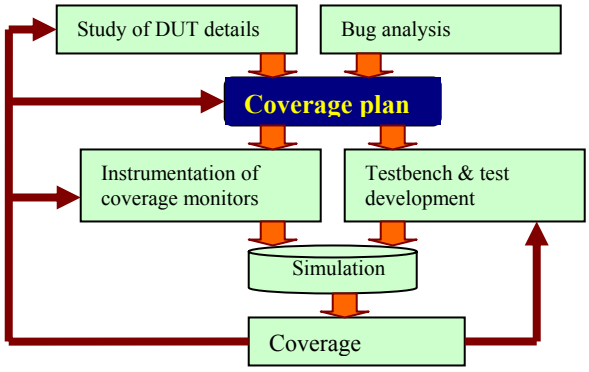

Figure 1: Coverage-Driven Verification flow

generator, as well as to direct updates to the coverage plan. The same coverage monitors and random test templates can subsequently be reused to quickly verify modifications to the RTL or to help in the verification effort of future proliferations.

3.3 Drawbacks of Coverage Driven Verification

CDV optimizes simulation and headcount resources by directing the verification flow according to the pre-defined coverage metric. However, a number of factors render pure CDV difficult, and frequently completely impractical to implement. In particular:

- **Importance of finding bug in early stages**. We cannot emphasize enough the importance of finding bugs as early as possible. The earlier bugs are found, the lower the impact they have on the smooth convergence of the whole design. This is particularly true in advanced designs such as microprocessors or SOCs for which the back-end flows require vast custom implementations. Therefore, even if the resources for the demanding task of coverage are available, it still may be an expensive trade-off to start by assigning them to coverage.
- **Lack of detailed knowledge in early stages.** The definition and the accurate implementation of the coverage plan require fairly detailed knowledge of the low-level details of the DUT. CDV requires that this plan be in place at the outset of the verification cycle, a time when engineers most likely will not have yet acquired the necessary knowledge. Such knowledge is acquired gradually throughout the course of verification from specification documents, reviews, failure analysis and more.
- **Instability of design and uArch spec.** Advanced and complex chips are designed under tight schedule constraints. This makes it necessary to start the design process before the specification and the micro-architecture are completely stable. The specification may be modified afterwards in order to introduce or improve functionalities. The implementation may also change due to learning from bugs or in order to meet power, area or performance requirements. As a result, a detailed coverage plan, and particularly the implementation of coverage monitors crafted early on, will very likely need to be modified, if not completely redesigned.

- **Staffing of verification teams.** Our experience shows that design flows frequently start when the verification teams are not yet fully staffed. Furthermore, in the early stages of the design, the lead verification engineers are generally busy ramping up the validation team, defining testbenches and the simulation environment, etc. This calls into question the ability of the verification team to develop detailed coverage monitors in a timely manner for the tight project schedule.
- **Completeness of the coverage space**. Coverage plans refer only to the set of conditions identified by the verification engineers. As comprehensive as they may be, they are still incomplete, and corner case bugs may have been overlooked. Moreover, it is risky to assume that coverage monitors themselves are completely free of bugs. They are bug prone, especially when they consist of temporal flows. Use of an incorrect signal or time window is hard to identify, and wrong coverage may be collected and tracked unknowingly.
  In sum, while CDV helps to better balance resources, it still does not eliminate the need to develop comprehensive random testing capabilities and wide stimuli to compensate for human errors and misguidance in coverage.

These considerations make pure CDV hard to use and frequently impractical for most advanced designs.

4. MEROM COVERAGE METHODOLOGY

4.1 Coverage Oriented Verification (COV)

In recent projects we tried to balance the high investment in coverage with practical considerations. The approach used in the previous project referred to Coverage-Oriented, is described in [9]. In Merom, we took this approach a step forward by applying multiple trade-offs in order to significantly reduce the effort invested in coverage, while maintaining the important return.

In our Coverage-Oriented verification (COV) approach, coverage is applied in the second stage [9]. In the first stage, during which most features are integrated into the RTL model, verification aims to ensure their basic functionality at the cluster level and to enable their release to the full-chip model without completely breaking it. This keeps the full-chip model alive and functional, with the ability to identify inter-cluster issues early on in the project.

Application of coverage in the second stage overcomes some of the drawbacks of pure CDV. At this time, most of the basic bugs have already been flushed out, design and uArch specs have stabilized, engineers have acquired a working-level knowledge of the design, and the verification team is adequately staffed.

Our experience with COV [9] shows that the way we use coverage reveals a relatively small number of logic bugs directly. However, coverage has a very important impact beyond the raw number of bugs, such as being an effective means of gaining a good knowledge of low-level design details, which in turn improves the quality of verification and expands the expertise of the engineers. In addition, we expected coverage to provide us with measurable targets that are associated with the quality of the logic.

4.2 Merom Coverage Guidelines

Several fundamental guidelines drove the Merom coverage effort:

Guideline #1: **Coverage is not a goal in itself, but rather a means of improving the quality of verification.**

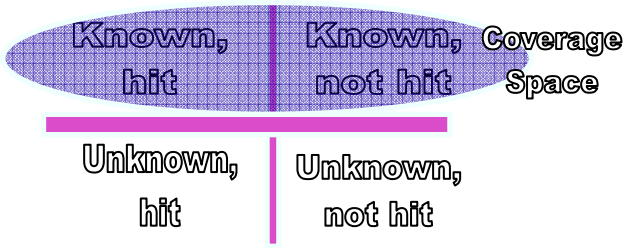

Figure 2: Coverage space represents the Known subset of testing space

The coverage space represents only the subset of testing space which can be considered as Known (see Figure 2), where the complementary subset can be referred to as Unknown. Coverage results then divide the Known coverage space into the areas of 'hit' and 'not hit' entries. The holes, 'Known, not hit' indicate flaws in the verification program, and tuning it needs to be done in a way that reduces the 'Unknown, not-hit' space as well.

Analysis of coverage holes should direct the improvement of random testing capabilities. Specific direct tests written to hit coverage holes have a low chance of hitting other cases that have been missed in the coverage space. The preferred method is to improve the random testing, either by adding random tests directed towards those coverage holes, or by embedding knowledge into the testbench that improves the random testing around them. This idea is illustrated in Figure 3.

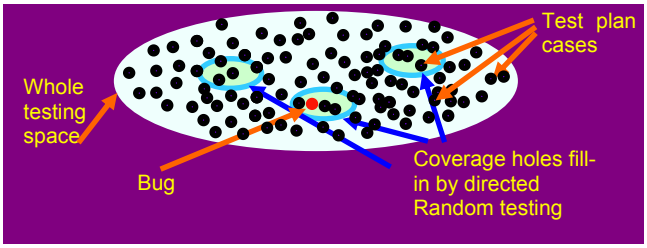

Figure 3: Role of Directed Random testing for coverage holes

Guideline #2: **Make test plans coverage friendly.**

Experience shows that in many cases, test plans that serve very well for the development of high-quality tests turn out to be vague and incomplete for coverage. In order to be coverage-aware, test plans should:

- **Formally specify the coverage space**. Formal definition makes it easier to turn a test plan entry into a coverage monitor. This need is magnified further considering that engineers who write the test plans are often not the same engineers who write coverage monitors at a later stage.
- **Refer to well-defined events**. Indicate specific RTL signals when they are stable. Use of internal signals must be coordinated with the design team (see also in [1]).
- **Explicitly define the expected coverage target**. As will be explained later, the target need not always be 100%.
- **Define the relative importance** of each coverage monitor in terms of its contribution to the overall quality of the verification process, as will be discussed later.
- **Indicate the total number of legal cases per entry.** Indicating this helps ensure the accurate interpretation of the coverage space as derived directly from its definition. This

turned out to be very useful in test plan reviews and coverage instrumentation in later stages.

Guideline #3: **A hierarchy of coverage facilitates reuse.**

Experience shows that the same key events are used repeatedly by numerous coverage monitors. To simplify the reuse of these events, we defined a hierarchy of libraries for coverage events that can be shared at the cluster, unit or function level (see Figure 4). All events used a lower-level layer that referred to the events and specific signals in the DUT. This also facilitates the reuse of the very same monitors at both cluster and FC levels.

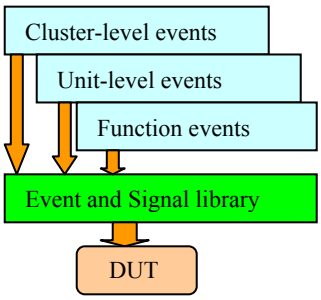

Figure 4: Hierarchy of coverage elements

5. MEROM COVERAGE METHODS

The full implementation of a comprehensive coverage program was constrained in Merom by schedule and limited staffing. It is reasonable to assume that these constraints hold for many design projects in the industry. Therefore, we introduced a series of practical trade-offs that kept us on schedule by significantly reducing the coverage effort, while maintaining high quality of verification. We will now elaborate on the steps taken.

5.1 Prioritized Coverage

A coverage-oriented test plan typically lists the full set of coverage tasks. However, not all tasks are of similar importance and risk. Careful analysis typically shows that a significant portion of the testing space can be identified as covered even without explicit monitoring. In Merom we prioritized all entries in the test plans in order to drop low-priority monitors from the coverage space. Prioritization took into account factors such as:

- **Controllability**. The specific function is highly controllable from the boundaries of the test bench.
  Example #1: The IA (Intel Architecture) restricts adding prefixes to an instruction such that the total length of the instruction must not exceed 15 bytes. Violations yield a General Protection Fault (GPF) indication. The test plan may specify combinations of prefixes which produce instructions that are too long. However, it is a straight forward task to exhaustively generate all instructions listed in the test plan.
  Example #2: It is relatively easy to generate all the micro-instructions listed in the test plan and directly feed them to the execution units in the cluster testbench. This reduces the need to develop corresponding coverage monitors.
- **Intensity of testing.** The logic is tested very intensively to provide high confidence that all cases considered in the coverage plan are adequately tested.
  Example #3: The Instruction Fetch Unit needs to identify instructions of all legal lengths and locations within a cache line. Because this logic is implicitly tested by any ISA-level test and such tests are intensively run, we can expect that all desired cases are adequately hit implicitly.
- **Complexity and risk.** Logic functions can be subjectively classified according to the risk associated with them. Higher risk may be the result of complex design, lack of detailed knowledge or even a history of late bugs.
  Example #4: Merom holds a completely new Cache-to-Cache mechanism for snoops between the caches of its two cores. This mechanism was considered of medium priority, but was upgraded to High following late bugs and changes.
- **Use of other verification techniques** such as Formal Verification (FV), applied to the same function.
  Example #5: FV was applied to the execution units in Merom, covering almost the full set of micro-instructions and proving the correctness of their functionality. This significantly lowered the need for coverage in these units.

Consequently, each entry in the test plan was rated for its importance as a coverage item. Overall, about 30% of the test plan entries were rated as High-Priority and were completely implemented, as were some of those rated as Medium Priority. Low-Priority entries were completely dropped from the coverage space. Overall, about 60% of the test plan entries were dropped from the measured coverage space.

5.2 Hit Count of Basic Events

Random testing is expected to hit events in a balanced manner. Imbalance in test generation may result from various causes, such as bugs in the test generator itself or incorrect directives. Such flaws are usually easy to detect. However, this imbalance may also be the result of changes introduced to functions of the design or the testbench over time. Such changes are the most problematic. Identical directives and constraints applied previously to the testbench to produce balanced stimuli may later yield a significantly different characteristic of testing without any indication. This in turn may hide bugs in the RTL.

An example of such a newly introduced imbalance was evident during the verification of Yonah, the predecessor of Merom. A fatal bug, discovered very close to tape-out, was the result of a late change to the mechanism that handles a full micro-instruction queue. The change disrupted a basic behavior and made reaching this condition extremely rare. The verification team didn't suspect that this condition was not being hit any more, and the bug was only found several months after its insertion into the RTL.

Standard coverage, as defined in test plans, is not optimized to track balancing in test generation because it is mainly focused on checking whether events are hit at least once. To better cope with the need to track balancing in test generation, we added a simple, quick counting mechanism and applied it mainly to basic events. Such events were typically based on a combination of a few signals (e.g. a request arrives when a queue is full), or simple temporal flows (e.g. a queue is full for exactly 1 cycle). Each basic event was expected to be hit at least a certain number of times during regression, or else a visible alert was issued.

The ROI of Count Coverage proved to be high. While easy to define and implement, these very simple coverage monitors facilitated finding multiple bugs. In particular, they improved the quality of the stress testing, where regressions were biased towards intensive stimulation of selected functions in the logic.

5.3 Focus on Functional Boundaries

Functional boundaries between logic modules are considered a weak spot for verification. As a means of improving visibility of traffic on the boundaries, we developed a flow that automatically generates coverage monitors on the interface signals of every selected block during RTL model build. The monitors measure the toggling of every RTL interface signal. These automatically produced monitors enabled finding flaws in test generation, especially during the period after they were first introduced.

5.4 Clock Gating Coverage

The design of Merom aggressively turns off clocks in almost all modules when they are not in use. This clock-gating mechanism is supported by the RTL library. We developed a flow that automatically extracts all gated clocks and produces coverage monitors that measure their toggling when they are expected to be gated. This completely eliminated the need to explicitly verify clock gating mechanisms.

5.5 Merge of Similar Events

Another useful technique we developed allowed coverage from different entities that were identified as similar to be merged automatically. This technique simplified coverage definition and reduced hole-analysis overhead. For example, Merom has some identical decoders for instructions. The merge mechanism was used to group together coverage collected from all decoders into a single decoder entity. This concept has also been described in [4].

5.6 Sliding Coverage Window

Coverage collection means merging new coverage data with previous data to produce the updated status. This means that once a coverage item is identified as hit, it will keep on being presented as such unless coverage is reset and recollected from scratch. Such reset is necessary to remove stale data stemming from changes to the design or testbench.

In Merom, we introduced a sliding time window which counted and merged hits during a pre-defined period (e.g. one month). A complementary data-base maintained the events that were hit along with the test parameters, in order to reproduce hits from past models. This eliminated the need for reset and enforced the introduction of stable improvements to maintain coverage results over time.

5.7 Coverage Targets and Grading

A known practice ([4][6]) is to define coverage spaces derived from a cross-product of vectors, usually within a small time window. Such spaces are easy to define and produce large coverage spaces which are usually not distinct. In Merom, approximately 30% of the coverage monitors were derived from a cross-product, and held over 80% of the total coverage space. Hitting 100% of this space is neither practical nor necessary

We defined the *Distribution Target* of such spaces to be the sub-level of the cross-product for which all possible combinations are hit. For example, a *Distribution Target* of 3 in a 5-domain space means that all possible triplets of values out of the 5 vectors need to be hit.

A *Density Target* was defined to be the desired raw number of events out of the total number. For small or simple cases, such as those easily controlled by testbench tuning, this target was 100%.

Finally, we defined a *Weight* metric to identify the relative impact of coverage monitors on the progress of the verification effort. This metric allowed a 5-fold difference between monitors having the same level of coverage. For example, if two monitors had the same level of coverage, but had weights of 10 and 2, the impact of the higher one was 5 times greater than that of the lower one.

The test plan identified the weight and target distribution and density for each coverage monitor. Similarly to Banias [9]) we used the following formula to grade the coverage of a given set of coverage monitors (e.g., all monitors of a single functional unit):

$$G_i = W_i \frac{e_i}{E_i} \frac{\min(p_i, P_i)}{P_i}$$

Where:

- W_i is the weight of the specific monitor
- E_i and e_i are the target and actual number of coverage elements in the desired level of distribution
- P_i and p_i are the target and actual percentages of covered events (raw coverage)

Additional set of weights, e.g. for unit level, is used to formulate the coverage indicator G_s for a set of coverage monitors:

$$G_s = \frac{\sum_{i=1}^{N} W_i G_i}{\sum_{i=1}^{N} W_i}$$

Where N is the number of monitors belonging to the set

Relative grading was made visible to verification engineers in the coverage analysis tool in a bar chart where the height of each bar corresponds to its weight, as shown in Figure 5. This emphasizes the impact of progress in each of the coverage spaces on the total quality indicator of coverage. For example, the 30% hole of monitor 2 is of higher risk than the 100% hole of monitor 5.

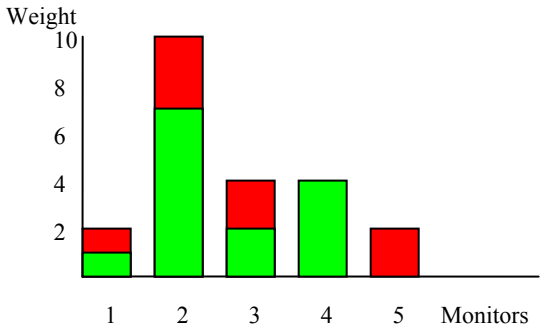

Figure 5: Visualtion of coverage grading

5.8 Coverage Tracking

Figure 6 shows coverage indicators of Merom clusters in the 9 months before tape-out. As we can see, the indicators climbed quickly because of the initial intensive test base developed in the first stage of the project, when gaps were mostly the result of

inaccuracies in monitor definitions or instrumentations. The graph for each of the clusters rises steadily with glitches usually stemming from major changes to the monitors that reset their coverage collection.

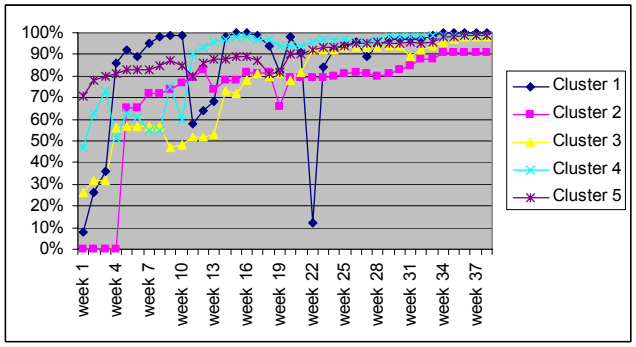

Figure 6: Progress of coverage in Merom clusters

6. SUMMARY AND RESULTS

6.1 Impact of Coverage Usage in Merom Verification

In Merom, we applied the Coverage-Oriented Verification approach along with a number of practical methods that significantly improved the return-on-investment. We started coverage at the second stage of verification when we focused on in-depth clean up of the RTL design. Coverage tasks were prioritized according to reviewed criteria, and low priority tasks were dropped from the measured coverage space. We gave extra focus to measuring the frequency of basic events and to covering the toggling of interface signals. We collected coverage data within a sliding coverage time window, and provided a tool-supported environment for coverage analysis and indicators.

There is no known method for measuring the true effectiveness of coverage. A common metric is to count the raw number of bugs found by an activity with respect to the effort invested, and to compare that to alternate activities in the same period. However, coverage has benefits beyond the raw number of bugs uncovered. It sets clear goals for verification activities. By providing on-going feedback regarding quality and progress, coverage results significantly contribute to the confidence necessary for bringing the product to tape-out.

To better evaluate the impact of coverage, we held a survey among verification engineers a short while before tape-out. The survey revealed a strong belief that coverage enforced the study of low-level uArch and implementation details. Another strong belief was that coverage improves the quality of testing. These results were somewhat surprising since most of the engineers said they found no bugs as a direct result of the analysis of coverage holes.

6.2 Bugs found by Coverage activities

We counted a total of 42 RTL bugs uncovered through direct analysis of coverage holes. These bugs were not uniformly distributed, and as mentioned, most coverage activities didn't reveal any bugs in the RTL directly. We estimate that at least as many bugs were found indirectly as a result of coverage, in particular due to improvements and bug fixes to testbenches and tests, and also due to knowledge acquired by instrumentation and analysis of coverage. The ~80 bugs correspond to 8.2% of the total number of bugs found in the relevant period.

Over half of these bugs were found in two clusters, and particularly in very specific functions. The common denominator for most of the bugs was that they involved specific temporal behaviors of signals originating in neighboring clusters. Count coverage revealed 9 RTL bugs which can be seen as a high number consider the low effort in its implementation.

In addition to the RTL bugs an even higher number of bugs and issues were found in the verification environment. These bugs led to coverage holes or the wrong balancing of random testing.

Despite its advanced and complex features and the relatively small verification team, Merom logic was exceptionally clean with only a very few logic bugs detected in silicon. Most of these bugs were due to holes in verification plans, where approximately half of these were on cross-cluster features. However, it is strongly evident that the quality of verification was exceptionally high. This suggests that the Merom practical Coverage-Oriented methods presented in this paper achieved their goal of enabling high quality verification with a significantly lower effort.

7. REFERENCES

[1] A. Piziali. *Functional Verification Coverage Measurement and Analysis*. Kluwer Academic Publishers, 2004.

[2] A. Allan, D. Edenfeld, J. William H. Joyner, A. B. Kahng,M. Rodgers, and Y. Zorian. *2001 technology roadmap forsemiconductors*. IEEE Computer, pages 42–53, Jan. 2002.

[3] Fine S., Ziv A., *Coverage directed test generation for functional verification using Bayesian networks*, DAC 2003, pages 286-291

[4] O Lachish, E. Marcus, S. Ur, A. Ziv. *Hole Analysis of Coverage Data*, 39th DAC

[5] Paul Gingras, *"Panel: Functional Verification – Real Users, Real Problems, Real Opportunities"*, DAC 1999, 260-261.

[6] A. Ziv. *Cross-Producr Functional Coverage Measurement with Temporal Properties-Based Assertions*. DATE 2003, pages 834-839.

[7] R. Grinwald, E. Harel, M. Orgad, S. Ur, A. Ziv. *User Defined Coverage – A Tool Supported Methodology for Design Verification*. DAC 98, 158-163.

[8] A. Adir, H. Azatchi, E. Bin, O. Peled, K Shiokhet, *A Generic Micro-Architectural Test Plan Approach for Microprocessor Verification.* DAC 2005, pages 769-774.

[9] Gluska, A. *Coverage-Oriented Verification of Banias*. In Proceedings of the 40th Design Automation Conference, June 2003.

[10] Wagner, I., Bertacco, V., Austin, T. StressTest: *An Automatic Approach to Test Generation Via Activity Monitors*. DAC 2005, pages 783-788.

[11] Bob Bentley. Validating the Intel Pentium 4 Microprocessor. DAC 20