A Generic Micro-Architectural Test Plan Approach for Microprocessor Verification

Allon Adir Hezi Azatchi Eyal Bin Ofer Peled Kirill Shoikhet
IBM Research Laboratory in Haifa
IBM Haifa Labs, Haifa University, Mt Carmel, Haifa 31905, (972)-4-8296211
{adir, hezia, bin, oferp, kirill}@il.ibm.com

ABSTRACT
Modern microprocessors share several common types of micro-architectural building blocks. The rising complexity of the micro-architecture increases the risk of bugs and the difficulty of achieving comprehensive verification. We propose a methodology to exploit the commonality in the different microprocessors to create a design-independent micro-architectural test plan. Our method allows the testing of the huge micro-architectural test space by using systematic partitioning, which offers a high level of comprehensiveness of the tested behaviors. We show how this method was used to find bugs during verification of an actual high-end microprocessor. Our results show the advantages of this approach over the more traditional test methods that use design specific test plans or that use tools with little micro-architectural knowledge for covering micro-architectural aspects of the design.

Categories and Subject Descriptors
B.6.3 [**Logic Design**]: Design Aids – *Verification*

General Terms: Design, Languages, Verification.

Keywords: Micro-architecture, Coverage, Generic Test Plan, Dynamic Verification, Test Generation

1. INTRODUCTION
Microprocessor verification is typically performed at several abstraction levels, ranging from the highest architecture down to the lowest register transfer (RT) level. The mid-range micro-architectural level includes common hardware components, such as pipelines, stages, queues, and so forth, and defines the behavior of instructions flowing through these components. The essence of micro-architecture verification is to verify the location and timing (in cycles) of the instructions during their flow. The focus is on verifying the control, rather than on verifying the data path.

It is especially important to search for bugs at the micro-architectural level, hence the greater effort in this field [2][4][5][6][7]. The importance of verification at this level can be seen from the fact that design bugs and their fixes are commonly explained at the micro-architectural level. Micro-architectural verification can expose performance bugs beyond those that are purely functional. Moreover, micro-architectural verification improves the chances of exposing functional bugs. Specifically, architectural test generation tools [1][3] generally do not offer comprehensive coverage of micro-architectural behaviors (as demonstrated in Section 3.3). In addition, a micro-architectural misbehavior that leads to a functional bug may also exist in many other scenarios that do not expose the bug (Section 3.2).

Micro-architectural verification is done mainly at the unit (cluster) level [1]. Because many bugs exist in the interfaces between the units, it is also important to test the micro-architecture at the chip level. Due to the complexity of this level, the testing is done using simulation methods coupled with micro-architectural coverage measurements [11]. Test generation methods include massive random generation, more intelligent random biased generation [3], coverage-driven generation techniques [8][12], and manual directed test writing. These verification activities are guided by a pre-defined test plan. These test plans are usually the result of an ad-hoc effort for the specific design under test and include many low level design-specific details.

Despite the different microprocessor architectures (ISAs) and the many different micro-architecture implementations, microprocessors today are built using several *common* building blocks. Examples include pipelines, dispatcher, and recurring execution units. This commonality suggests the possibility of using a *generic* framework to verify the different microprocessors. A generic test plan accumulates IP that preserves the verification knowledge assets of the organization. This knowledge can be used directly when a new verification process begins, thus saving much of the effort needed to work out a new design-specific test plan. It also ensures that many aspects that require testing are not overlooked. Designing a generic test plan also encourages thinking about the verification tasks at a higher level of abstraction, thereby capturing their essence rather than their low level details.

The main contribution of this work is a systematic methodology for creating a micro-architectural generic test plan. The method we propose is based on accumulating a collection of generic *cross-product* models [10]. This is done systematically by using the concept of a meta-model that partitions the space of coverage models, as described in Section 2. We also show how to automatically convert the generic test plan into a test plan for a specific micro-architecture under test. In Section 3, based on our industrial experience and on experimental results, we demonstrate the comprehensiveness of the generic approach and the need for micro-architectural models and tools. Section 4 concludes with a summary.

2. METHODOLOGY
The Generic-Test-Plan (GTP) usage flow is described in Figure 1, where we first create a GTP and then use it in the verification

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

process of different designs. We begin the GTP creation by enumerating a list of *verification objects* and then creating generic coverage models for each of these objects. Creating these models involves creativity, based on domain knowledge and on a systematic partitioning of the coverage model space, as described in Section 2.2. Once a GTP is available, we convert every generic model (relating to a verification object relevant to the design), for every design, into a concrete model. Part of this concretization can be done automatically, as explained below. The models are then covered using various verification tools.

The first step in creating a GTP (shown in Figure 1) is to enumerate the verification objects; these are the targets of the verification effort. We distinguish two types of micro-architectural verification objects: design components and mechanisms. *Design components* include both high and low level units (e.g., Fixed-Point unit, or a Dispatch unit) and resources (e.g., Issue-Queue, a Store buffer, Register file). *Mechanisms* are features of the micro-architecture that may involve or affect more than a single design component. Examples of mechanisms include Forwarding (Bypassing), Flushing, and Multi-Threading. Our GTP is based on about 20 such verification objects. The list can include verification objects that are related to a limited family of designs (e.g., multi-threading mechanisms). The related sections of the GTP will be skipped in the designs that do not use these objects.

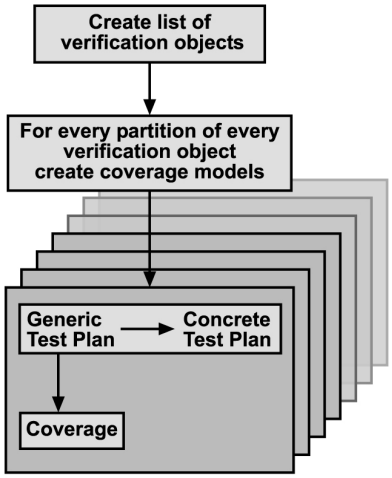

Figure 1: Creation and usage of a GTP

In the next step, we create a set of related coverage models for each verification object. This is done by partitioning the set of coverage models, as shown in Figure 2. The coverage models of the verification object are classified according to a set of independent characteristics defined in the meta-model (Section 2.2). Every partition can include several interesting coverage models—some of which may themselves be of the cross product type, as shown in the figure and in the following section. The final step of the GTP creation is to produce interesting coverage models for every partition (Section 2.3). Once a GTP is available, we can use it in the verification process of specific designs by converting it into a concrete test plan (Section 2.4).

2.1 Cross Product Models

A cross-product model is a parameterized template of a test scenario, in which each of the template's parameters can take values from a pre-defined range. A cross product model is usually based on a desired scenario, which is refined by generalizing several of its aspects. Another source for cross product models is the combination of two or more events from different domains, each with its own set of parameters. Using cross-product models in test plans is better than listing specific scenarios because bugs rarely occur in exactly the scenario that was thought worthy of testing, rather they are found in a neighborhood of similar scenarios. However, the person designing the test plan should be aware of several dangers associated with the cross product approach. A model can include many combinations of parameters that represent unreachable tasks. While covering such a model, it can be difficult to tell which coverage holes should still be targeted and which holes are due to unreachability. Technologies like [8] alleviate this danger by identifying the unreachable tasks of the model. Additionally, it is very tempting to refine a cross product model with too many parameters, or with parameters that have domains so large they render the model non-coverable in practice. The cross product approach also gives equal importance to every combination of parameters. While some combinations may deserve further refinement and other combinations may be unimportant, this can lead to a waste of verification resources. Nevertheless, we chose to use cross product models for our generic test plan because they offer a systematic way to achieve comprehensive high level state coverage of the micro-architecture.

2.2 Meta-model Partition

Our suggested meta-model partitions the space of cross product coverage models for a verification object according to the following characteristics:

Point of view: *Verification-Object-View/Instruction-View*. A coverage model with a Verification-Object-View attempts to cover interesting states of the verification object, which may involve several instructions. A coverage model with an Instruction-View attempts to cover different behaviors of an instruction, while using the verification object.

Example #1: A Verification-Object-View model – two different instructions stall in the same cycle in different stages of the same pipe for all possible pairs of stall causes, and for all pipelines.

Example #2: A similar model taken from the Instruction-View partition – the same instruction stalls in two pipeline stages of the same pipeline, for all possible pairs of stall causes and for all pipelines.

Timing: *Snapshot/Scenario*. A coverage model with a Snapshot view attempts to cover combinations of events that occur simultaneously. A coverage model with a Scenario view attempts to cover combinations of events that occur over time. For example, the model described above in Example #1 is of the Snapshot partition, whereas Example #2 is of the Scenario partition.

Example #3: Another model taken from the Scenario partition covers all pairs of different stall causes, for which two consecutive instructions stall in the same pipeline stage.

Isolation: *Isolation/No-Isolation*. This partition is related to models that enumerate several causes for an event. In the Isolation partition, exactly one cause for the event holds. In the No-Isolation partition, the event may occur due to several causes. For example, in all the above examples, in an Isolation type model, the stalling instruction stalls due to a single cause.

Example #4: In a No-Isolation variation of the model, any number of stall causes may hold at once, each of which would justify a stall on its own.

Event-Boundary: *Occurs/Almost-Occurs*. A model of the Almost-Occurs type includes tasks in which an interesting event is not supposed to occur, but would have occurred except for some slight inhibiting reason. These reasons are specified in the model. All the above examples are of the Occurs type, since the related events are supposed to occur.

Example #5: In similar models from the Almost partition, a certain stall occurs only for a particular type of instruction, with particular data and with a resource dependency. In the model, an instruction does not stall, but any two of the above conditions do hold.

Example #6: An instruction stalls if the next pipe stage is occupied by another instruction. In this Almost type model, the stall is avoided because the occupying instruction vacates the stage just in time for the new instruction to enter it.

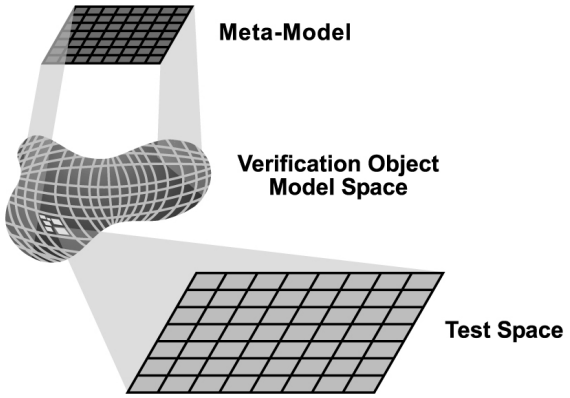

Figure 2: Partitioning the coverage models

Plurality: *Single-Object/Multi-Object* A model can involve several objects (e.g., Example #1 involves several stages) or a single object (e.g., Example #3 involves a single stage).

A concluding example for the model partitioning is the following model for the forwarding mechanism verification object:

Example #7: The model covers all forward paths (data bypassing) between stage pairs. The categorization of this model is: Verification-Object-View (involving two instructions), Snapshot, No-Isolation (because several forward conditions may hold), Occurs, and Single-Object.

Now that this basic model has been defined, all other variations of it can also be systematically added to the generic test plan.

2.3 Creating Micro-architectural Models

Creating a generic coverage model for a particular partition of a verification object mainly involves creativity, but can use the following guidelines:

Generalization: Take a specific scenario and generalize it. The generalization can be done by identifying specific properties of the scenario, which can be tested with different values. Thus the generalized model becomes a cross product type model with a task for every combination of values of these properties. Take for example, the scenario where an instruction is flushed in the same cycle in which it is dispatched. This scenario can be generalized by enumerating the different flush causes and enumerating other interesting timing events that can occur simultaneously with the flushing, such as issue or forwarding. A useful generalization is related to the timing of the scenario. A Snapshot type model can be generalized by considering different flavors of simultaneity, such as whether the events occur in the same cycle or with a delay of a small number of cycles. Events that take up several cycles (e.g., stall) can be interleaved in different manners by considering the relationship of the start and end cycles of the interleaved events.

Refinement: Base a model on an existing model and identify properties that have enumerated domains. For example, the above model can be refined by considering the instruction that is involved in the scenario and enumerating the types of instructions that need to be covered. A single basic model can be refined in this manner into several different derivative models. Another way that a model can be refined is by combining it with a different model. For example, a model covering all forward paths can be combined with a model covering types of flush events. Note that while the refined model covers the original model by construction, there may still be reason to maintain the original model in the GTP. The original unrefined model can be used in early stages of verification or when there are not enough resources to cover the more refined derivatives.

Structure: Follow the micro-architectural structure of the verification object. For example, queue related models can target states of the queue (full/empty) or entry and exit patterns. Steering units (such as an instruction dispatcher or issue unit) should be tested with various types of instructions and steering destinations.

Bugs: Study actual bug escapes and use the scenarios that were involved. Generalize the scenarios and see if they can also be relevant for different verification objects.

Existing test plans: Analyze legacy models or specific scenarios, generally saved as organized documents or distributed knowledge, for commonality among different designs, and generalize the knowledge into a GTP.

Rare cases: Try to focus on scenarios that have a low probability of being covered by simple random massive test program generation. The more trivial models will be covered “incidentally" during the effort of covering of the harder models in the test plan. The trivial models can still be placed in the GTP and used for measurements.

Feasibility: The size of the model should allow full coverage. For cross product coverage models, it is especially important to avoid the temptation of adding too many parameters, or parameters with very large domains. The available means of test generation (various tools or manual test writing) must also be taken into account. Our micro-architectural GTP includes coverage models with an order of several thousand tasks. The model must also be such that its coverage can be measured. In spite of the above claims, unfeasible models may still be retained in the GTP, for the purpose of partial coverage, monitoring the coverage process or with the hope that future technologies will allow its full coverage.

2.4 Concretization of the GTP

When planning a verification process for a new design based on a GTP, we must adjust the generic test plan to fit the specific processor and the available verification resources. In addition, we try to prioritize and order the coverage of the valid models. As the design matures and the flood of basic bugs is reduced, we can start covering the more complex models. These can involve progressive

refinements of the basic models and those models that relate to suspicious verification objects at the particular stage of verification.

When covering a specific model from the GTP as part of the verification of a specific design, we must first define the design-specific form of that generic model, as it is applied to the design under test. This includes setting the domains of the model's parameters and determining the legal combinations of these parameters. This step can be done either manually or automatically, as shown in Figure 3, with the generic components at the top and the design specific components below.

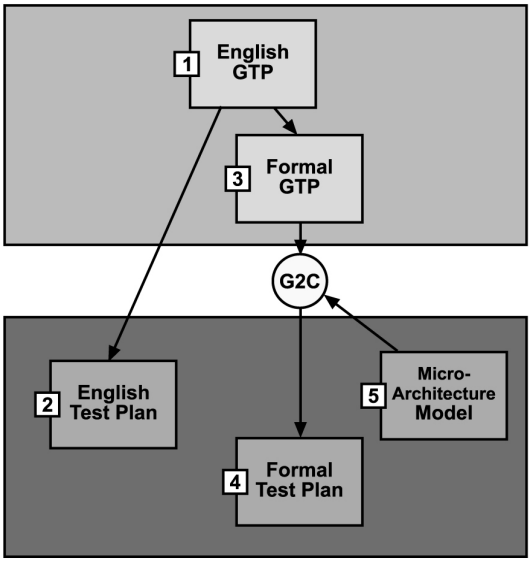

Figure 3: Converting a GTP into a concrete test plan

For example, consider the coverage model in Figure 4 (part of block 1 in Figure 3). This model can be translated manually into a design-specific model, as shown in Figure 5 (part of block 2 in Figure 3). In this case, setting the domains of the model's parameters is a relatively straightforward step; however, listing the feasible durations of each stall case can be difficult even for the designer. Alternatively, our approach was to write the *generic* model in a formal language, as shown in Figure 6 (part of block 3 in Figure 3). The language that was used comes from the Piparazzi test program generator [8]. This generic model is converted into its design-specific representation by the addition of the model parameters' domains (Figure 7). This process is done automatically (G2C in Figure 3) by referring to Piparazzi's formal model of the processor's micro-architecture. For example, to compute the domains of the parameters, Piparazzi refers to the processor's micro-architecture model for the list of the processor's pipelines, stages, and valid stalls in each stage. The design-specific formal language parameter domain description (Figure 7) does not in itself indicate which of all the combinations of parameters are actually possible under the design. However, Piparazzi can be used to produce an actual test case for every possible combination of the coverage model and to report which combinations are unreachable. It does this by referring to the processor's micro-architecture model.

Cover every stall cause in every stage of every pipeline for any duration between 1 and 4 cycles.

Figure 4: Generic English description of the model

Cover the tasks where an instruction stalls for all possible durations between 1 and 4 cycles in the following cases:

stage 1 of FXU1 pipe because of data not ready.

stage 1 of FXU2 pipe because of data not ready.

stage 2 of FXU2 pipe because the instruction is multi-cycle.

stage 4 of FPU pipe because of normalization delay.

Figure 5: Design specific English description of the model

```
#include <DesignSpecificDomains>
<ForEach iterator="p" set="pipelines">
 <ForEach iterator="s" set="pipe_stages[p]">
  <ForEach iterator="i" set="stall_cases[p][s]">
   <ForEach iterator="delay" set="{1,2,3,4}">
    <Task description="Stall type">
     <Rule expression="iop[0].PIPELINE = p" />
     <Rule expression="iop[0].stall[s][i].AMOUNT = delay" />
    </Task>
   </ForEach>
  </ForEach>
 </ForEach>
</ForEach>
```

Figure 6: Generic model description in a formal language

```
DesignSpecificDomains:
pipelines={FXU1, FXU2, FPU}
pipe_stages[FXU1]={STAGE1, STAGE2}
pipe_stages[FXU2]={STAGE1, STAGE2}
pipe_stages[FPU]={STAGE1, STAGE2, STAGE3, STAGE4, STAGE5}
stall_cases[FXU1][STAGE1] = DATA_NOT_READY
stall_cases[FXU2][STAGE1] = DATA_NOT_READY
stall_cases[FXU2][STAGE2] = MULTI_CYCLE
stall_cases[FPU][STAGE4] = NORMALIZATION
```

Figure 7: Design specific domains in a formal language

3. RESULTS

Based on the methodology presented, we developed a micro-architectural GTP and used it during the verification process of a high-end PowerPC microprocessor [13]. This GTP includes a total of about 100 models for the following verification objects:

Design components: decoder, dispatcher, issue queue, generic pipeline.

Mechanisms: flushing, forwarding (bypassing), multi-threading, reject, branch prediction.

We show first in Section 3.1 that the generic nature of the test plan results in higher comprehensiveness of its coverage by comparing it with design specific test plans. We then demonstrate in Section 3.2 the importance of targeting micro-architectural models by presenting several bugs that were found while using our GTP. Finally, Section 3.3 shows that having micro-architectural knowledge in the test

generation tool is essential when covering these micro-architectural models.

3.1 Genericness Leads to Comprehensiveness

To demonstrate the generality of our GTP, we compared it to the micro-architectural verification test plans of three mid to high end PowerPC processors, developed by different design groups. In Table 1, we present the results of this comparison for the Dispatch unit of the processors. The GTP includes 26 models for the unit. The table shows how many of these models were targeted by the specific test plans.

Table 1: Coverage of the GTP Dispatch unit models by design specific test plans

	Processors		
	I	**II**	**III**
Fully covered	2	5	10
Partly covered	11	8	9
Partly covered on architectural level	2	2	1
Not covered	9	8	5
Not applicable for the specific design	2	3	1

The table shows that at least half of the models are either not covered at all, or covered only partially. Table 2 shows how many of the models related to the Dispatch unit of the three design-specific test plans are covered by the GTP. As can be seen, most non-covered models relate to mechanisms that are unique to the specific processors. These may be considered for a further extension of the GTP if they are thought to be relevant for future designs.

Table 2: Coverage of the existing models by GTP

Fully covered	25
Partly covered	4
Partly covered processor unique features	3
Not covered	2
Not covered processor unique features	7

3.2 Micro-architectural Models Reveal Bugs

From the GTP, we selected a set of models to be covered as part of the actual verification process. The coverage was done using Piparazzi's test generation environment. Piparazzi handled the automatic concretization of the coverage models so they fit the actual design under test. Piparazzi was also used to generate test cases for the models and identify the unreachable tasks. Piparazzi reports the expected micro-architectural behavior of the generated tests. This data was used to validate that the targeted event was hit, thus removing the need for a trace analyzer. In addition, this data was compared with the actual simulation trace in order to check for design bugs.

During this work, we covered three selected models from the GTP using two test generators: Genesys-Pro [9] and Piparazzi [8]. We found eight design bugs in the process, two of which are described below. The first bug was revealed by the model checking all the cases in which two simultaneous dispatch hold conditions apply. When this happened for a specific pair of such conditions, the thread priority mechanism was not updated properly, resulting in an unnecessary stall of instructions in another thread. While the bug caused only slight performance degradation, it uncovered a hole in the thread priority mechanism and possibly prevented more serious bugs in the future.

The second bug was found while covering all resource dependency conditions that lead to a dispatcher stall. We found that one of the conditions written in the specification was not implemented. The fact that the condition was missing, causes a functional bug in only very rare cases. However it causes micro-architectural discrepancies more often. Such a discrepancy was found by Piparazzi while targeting a related micro-architectural model from the GTP.

These bugs demonstrate the importance of checking micro-architectural consistency and not relying entirely on architectural (functional) checks. Functional checks cannot find performance problems and, in addition, some very rare architectural bugs can be readily extrapolated from more frequent micro-architectural fault behavior.

3.3 Micro-architectural Coverage

In this section, we compare an architectural level tool, based on fast massive test generation, with a test generator that is capable of precise targeting of micro-architectural events based on micro-architectural knowledge. For one micro-architectural model, we compared the coverage progress given by the two test generators: the architectural random biased Genesys-Pro and the micro-architectural Piparazzi. In the tested processor, the Dispatcher can dispatch a group of up to five instructions in every cycle. The model covers all combinations of resource dependencies among a dispatch group. Further micro-architectural aspects render some of the tasks in the model unreachable, but it is hard to determine manually what these tasks are. The results are given in Table 3.

Table 3: Covering a model using different approaches

Tool	**Genesys-Pro**						**Piparazzi**
Approach	**Lab**	**I**	**II**	**III**	**IV**	**I-IV**	**I**
Hits	17	8	10	17	26	27	60
Proved illegal	-	-	-	-	-	-	963
Instructions	62k	20k	30k	25k	30k	105k	0.7k
Run time	154h	50h	80h	80h	80h	300h	220h
Total tasks	1024						

The "Hits" row indicates the number of different tasks of the model that were hit at least once. The next row gives the number of tasks that were proved to be unreachable. The next two rows give the number of instructions that were generated and the amount of time it took to generate and simulate these instructions. The "Lab" column gives the results measured during six days of an actual verification process for a PowerPC design. During this period the design was massively tested with Genesys-Pro using 194 different general directives that target various aspects of the processor that were unrelated to our model. Columns I – IV in the table show the results of running Genesys-Pro on four respective directives. The directives were written to specifically target the tasks of the model and were progressively amended in order to improve the coverage. The "I-IV" column summarizes the usage of all four directives. The final column shows that Piparazzi covered practically all of the reachable

tasks and proved that 963 tasks are actually unreachable in the design.

Figure 8 shows the number of tasks covered by the two test generators and the total number of reachable tasks in relation to the number of resource dependencies in the task. As the figure indicates, it was hard for Genesys-Pro to generate the tasks that had more than one dependency and it did not hit *any* task with more than two dependencies.

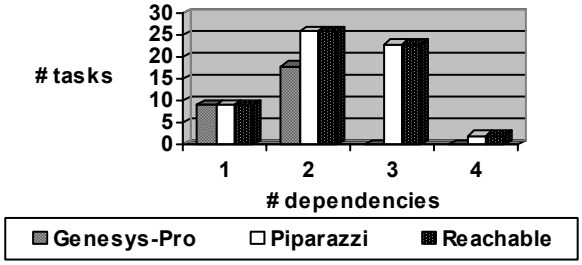

Figure 8: Task coverage by number of dependencies

These results demonstrate that it is important to target micro-architectural models directly and not rely on massive random generation, even for seemingly trivial coverage models. Additionally, targeting an architectural "variation" of the model gives a non-uniform distribution of the hit tasks. The "easy" tasks were hit very frequently, whereas the "harder" tasks were never hit at all.

4. CONCLUSIONS

Our work presents a systematic method for creating generic test plans at the micro-architectural level. We showed that using a test plan of a *generic* nature with systematically generalized *cross product* models leads to a comprehensiveness of the covered aspects of the design that is greater than any achieved with the more common approach of using design specific test plans. Our results also show that the *micro-architectural* nature of the test plan and of the generation tools used to cover them lead to the coverage of events (some revealing design bugs) that are much harder to cover with an alternative more architectural approach.

Along with these encouraging results, we wish to emphasize that using a GTP should not be the sole means guiding the verification process. The GTP should be complemented with massive random and biased random test generation, which may hit events that were missed during the construction of the test plan. In addition, there will always be aspects of the processor under test that are not to be found in the GTP because they relate only to the particular design under test.

As part of our continuing work on the GTP, we expect to extend it to cover more verification objects, such as Load-Store and Floating Point units, and synchronization.

5. REFERENCES

[1] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek. *Test program generation for functional verification of PowerPC processors in IBM.* In Proceedings of the 32nd Design Au-tomation Conference, pages 279–285, June 1995.

[2] D. V. Campenhout, T. Mudge, and J. P. Hayes. *High-level test generation for design verification of pipelined micropro-cessors.* In Proceedings of the 36th Design Automation Con-ference, pages 185–188, June 1999.

[3] L. Fournier, Y. Arbetman, and M. Levinger. *Functional veri-fication methodology for microprocessors using the Genesys test-program generator.* In Proceedings of the 1999 Design, Automation and Test in Europe Conference (DATE), pages 434–441, March 1999.

[4] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose. *Auto-matic test program generation for pipelined processors.* In Proceedings of the International Conference on Computer Aided Design, pages 580–583, November 1994.

[5] K. Kohno and N. Matsumoto. *A new verification method-ology for complex pipeline behavior.* In Proceedings of the 38th Design Automation Conference, pages 816–821, 2001.

[6] S. Ur and Y. Yadin. *Micro-architecture coverage directed generation of test programs.* In Proceedings of the 36th De-sign Automation Conference, pages 175–180, June 1999.

[7] J.-T. Yen and Q. R. Yin. *Multiprocessing design verification methodology for Motorola MPC74XX PowerPC micropro-cessor.* In Proceedings of the 37th Design Automation Con-ference, pages 718–723, June 2000.

[8] A. Adir, E Bin, O. Peled, and A. Ziv. *A Test Program Generator for Micro-Architecture Flow Verification*. In Proceedings of the HLDVT 2003, pagse 23-28.

[9] Adir, A., Almog, E., Fournier, L., Marcus, E., Rimon, M., Vinov, M. and Ziv, A. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. *IEEE Design & Test of Computers*, Mar-Apr. 2004, 84-93.

[10] A. Ziv . *Cross-Product Functional Coverage Measurement with Temporal Properties-based Assertions*. In Proceedings of the 2003 Design Automation and Test in Europe (DATE'03)

[11] R. Grinwald, E. Harel, M. Orgad, S. Ur and A. Ziv. *User Defined Coverage - A Tool Supported Methodology for Design Verification.* In Proceedings of the 35th Design Automation Conference 1998. pages 158-165

[12] M. Aharoni, S. Asaf, L Fournier, A. Koyfman and R. Nagel. *FPgen A Deep-Knowledge Test Generator for Floating Point Verification*, HLDVT 2003, pages 17-22

[13] C. May, E. Silha, R. Simpson and H. Warren. *The PowerPC Architecture.* In Morgan Kaufmann, 1994.

[14] Julia Dushina, Mike Benjamin, Daniel Geist. *Semi-Formal Test Generation with Genevieve.* DAC 2001.