Code Generation and Analysis for the Functional Verification of Microprocessors

Anoosh Hosseini Dimitrios Mavroidis Pavlos Konas
Silicon Graphics Inc.
2011 N. Shoreline Blvd.,
Mountain View, CA 94043
anoosh@sgi.com

Abstract

A collection of code generation tools which assist designers in the functional verification of high performance microprocessors is presented. These tools produce interesting test cases by using a variety of code generation methods including heuristic algorithms, constraint-solving systems, user-provided templates, and pseudo-random selection. Run-time analysis and characterization of the generated programs provide an evaluation of their effectiveness in verifying a microprocessor design, and suggest improvements to the code generation process. An environment combining the code generation tools with the analysis tools has been developed, and it has provided excellent functional coverage for several generations of high-performance microprocessors.

1 Introduction

Functional verification is a vital part in the design and implementation of high performance microprocessors. Both customer confidence and commercial success depend on a defect-free functional product which is introduced into the market in a timely fashion [1]. A design verification team (DVT) presently relies on extensive simulation-based testing of the microprocessor's RTL model to achieve the functional coverage necessary for a design to be released to the manufacturing process. State-of-the-art microprocessors, however, achieve high performance through several advanced execution mechanisms [5]. The increased complexity introduced by these mechanisms forces DVT teams to increasingly depend on advanced code generation tools for the functional verification of microprocessors [1, 2, 3, 6].

Code generation tools create interesting instruction sequences which when simulated on the microprocessor's RTL model can expose flaws and errors in the implementation. Code generation tools are divided into three major categories: user-assisting tools, pseudorandom and heuristic-based code generators.

User-assisting tools simplify and automate tedious tasks such as the permutation, iteration, and interleaving of existing instruction sequences into new sequences with interesting properties. Such tools make the generation of diagnostics for known cases easier and less time consuming. Pseudorandom code generators, on the other hand, focus on producing long sequences of legal instructions assuming that the random interaction of these instructions will produce conditions rarely created by compiler-generated code, or conceived by a programmer. Unfortunately, they usually produce code of poor quality. Finally, heuristic-based code generators combine user-provided attributes and properties with knowledge of the architecture and of the design to produce algorithms targeting the most complicated features of the design. They generate code of high quality by intelligently selecting instructions whose execution will create the proper conditions for an interesting case, which has not been previously covered, to arise.

Isolating a design flaw can be accomplished in two ways. The simplest approach is to generate self-checking code. The test program sets up a combination of conditions and then checks whether the RTL model reacted correctly to the given situation. Unfortunately, the state compare instruction sequence is usually too intrusive at the RTL level; it is coarse grain and, thus, not so accurate; it consumes precious simulation cycles; and it may burden the code generation tool by requiring it to maintain an extensive amount of state. The most efficient approach is to non-intrusively compare the traces generated by the simulation of the RTL model with the simulation traces of an architectural reference model. Such an approach frees the diagnostic program from continuously checking the reactions of the design under testing, it is more accurate, it allows for a more powerful comparison process to be employed, and it relieves the code generation tool from computing the results of all the instructions it generates.

The execution of most tool-generated diagnostic programs results in instruction sequences which the designer can usually neither completely anticipate nor fully evaluate. It is important for the designer, therefore, to analyze the sequence of instructions generated by the tool, to characterize their behavior, and to evaluate their effectiveness using several architectural and microarchitectural metrics. Such metrics relate to utilization across the different units of the microprocessor and include instruction histograms, event coverage, and queue sizes. Furthermore, we can use these metrics in subsequent code generations to improve the quality of the generated programs as well as the efficiency of the generators themselves.

This paper presents a collection of advanced code generation tools employed in the functional verification of high-performance microprocessors. In section 2 we briefly outline our verification methodology. In sections 3 through 6 we present a few of our sophisticated code generation tools. In section 7 we present an analysis tool which is used in evaluating diagnostic programs. Finally,

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

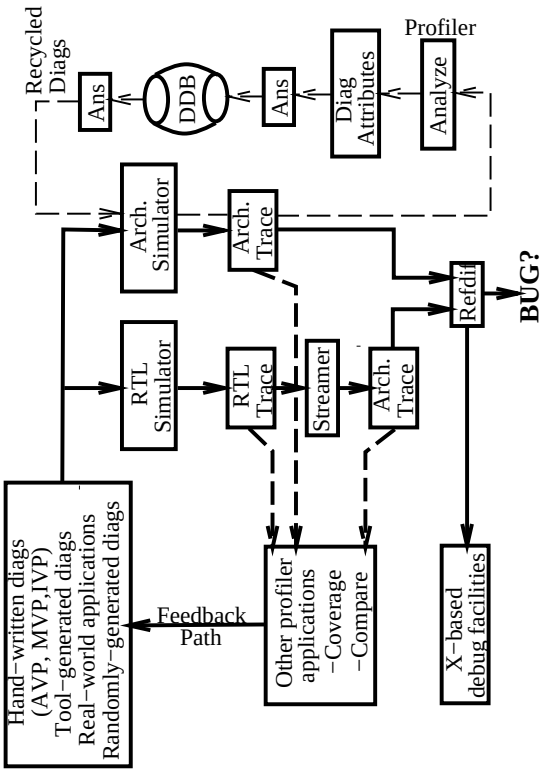

Figure 1: Functional Verification Methodology

section 8 summarizes our approach to simulation-based verification of microprocessor designs.

2 A Functional Verification Methodology

Functional verification aims at isolating design and implementation flaws so that the design released to the manufacturing process is fully operational; that is, the RTL model exhibits the same behavior as an architectural simulator would when executing the same instruction sequence. As the complexity of new high-performance microprocessors increases, as the quality expectations of new products are rising, and as the time-to-market decreases, functional verification becomes a more difficult process and emerges as the bottleneck of the development cycle.

In order to improve the efficiency and the effectiveness of functional verification, we follow the methodology outlined in Figure 1. First, four different sources (verifiers) generate diagnostic programs. Hand-written directed diagnostics are developed by the members of the DVT team and include architectural (AVP), microarchitectural (MVP), and implementation (IVP) verification programs. These diagnostics set up and check conditions deemed interesting by the developer of each test. Second, advanced pseudorandom code generators produce long instruction sequences which aim at creating complicated interaction patterns among the instructions. Such instruction sequences are rarely conceived by a programmer or generated by a compiler. Third, sophisticated tools generate instruction sequences which stress the microprocessor model in ways that cannot be achieved by the first two code generation approaches. Finally, "real world" software applications are used to ensure that the design implements correctly and efficiently the most common operations.

The diagnostic programs generated in any of the above ways are compiled and provided as input into two simulators. The RTL simulator represents the specific microprocessor's implementation. The architectural simulator, on the other hand, describes the behavior of any microprocessor design implementing the given architecture as the latter is specified in the architectural manual. The execution of the object code on the two simulators produces two traces. The architectural trace captures how the architecturally visible state changes as a result of executing the instructions in the diagnostic. The RTL trace, on the other hand, captures how the microprocessor's state changes as a result of executing the same sequence of instructions. However, because of the large number of advanced implementation features contained in state-of-the-art microprocessors the two traces may not be the same. A conversion tool (*streamer*) transforms the RTL trace into a trace representing the changes in the architectural state as they are deduced from the information in the RTL trace. There are several interesting and hard issues involved in such a conversion process, but they are beyond the scope of this paper.

Once we have obtained an architectural trace from the RTL, we compare it with the trace produced by the architectural simulator, using an architectural comparator (*refdif*). If the two traces differ, then the model does not behave correctly, and the diagnostic has identified a flaw in the microprocessor's implementation. A powerful X-based graphical environment which exploits the information provided by the architectural comparator can then be used to debug the identified error.

In addition to identifying flaws in the implementation, traces of diagnostic program executions are also used to analyze the test programs, determine their properties and characteristics, and evaluate their effectiveness (*Profiler*). The results of this analysis and evaluation are stored in a diagnostic database, and they are used subsequently to improve the quality of the generated code as well as the effectiveness of the code generation tools.

In the following sections we take a closer look at the code generation tools as well as at the analyzer and the diagnostic database. These are the most important parts of our approach to code generation for the functional verification of microprocessors.

3 SBVer: An External Interface Verifier

High-performance microprocessors employ complex external interface units which buffer requests, allow multiple outstanding loads and stores, maintain multi-level caches, and perform cache coherency in multiprocessor configurations. The many states of the external interface combined with an abundance of asynchronous events from other devices, makes the external interface a verification challenge.

For this purpose we have developed SBVer (Store Buffer Verifier), a code generator which focuses on exercising the external interface and the cache management units of the microprocessor. Knowledge about the design of the primary and secondary caches, of the various address spaces, and of the memory management unit have been built into the tool. SBVer, combined with heuristic algorithms, produces sequences of instructions which cause interesting interactions between the processor, the caches, and the main memory. SBVer has also the ability to program external event generators in the system model so that they interact with the processor in a coordinated fashion. For system verification purposes, SBVer may also produce self-checking code based on an internal memory model maintained during code generation. Finally, SBVer has a large number of configuration options in order to provide the user

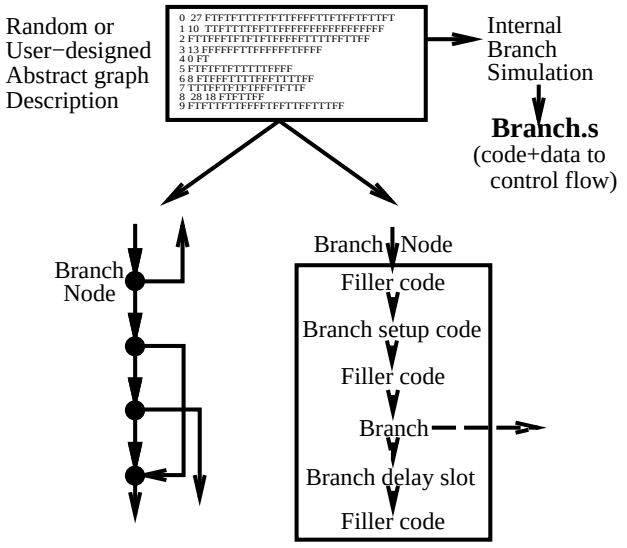

Figure 2: BRVer Design

with control over the tool's behavior. SBVer has been successful in finding flaws in four generations of microprocessors, and in various hardware systems.

4 BRVer: A Branch Verifier

Many pseudorandom code generators avoid complex branching sequences, especially backward jumps, in order to prevent infinite loops. On the other hand, the length of the produced pseudorandom programs results in the verification engineers having limited knowledge of the program flow, and of whether critical sections of the program have been executed. Furthermore, new microprocessors attempt to predict the direction of branches and execute instructions beyond a branch speculatively. The result of speculative execution is a significant increase in the number of branch related cases which need to be examined. In order to address these issues in a systematic way, we have developed BRVer. Figure 2 shows the various components of BRVer and how the branches are modeled.

BRVer accepts as input a large number of configuration parameters and an Abstract Graph Description (AGD) which is either provided by the user or it is generated heuristically. The input AGD contains the number of nodes (effectively branches) in the graph, how the nodes are connected to one another, and for each branch the action to be performed (fall through or take the branch) upon successive arrivals. BRVer "compiles" the AGD input producing an instruction stream whose run time behavior correctly represents the flow described.

BRVer also accepts user provided input streams as filler code in between branches. This proves to be a convenient way to apply the branch management mechanisms to code produced by other tools such as SBVer and Theo.

5 Multiprocessor Verification

Over the last few years, most manufacturers develop multiprocessor ready microprocessors [7, 8]. As a result, it is essential that the DVT team verifies the microprocessor's mechanisms facilitating the sharing of information across the processors of a multiprocessor (MP) machine. Such a verification process entails two important issues. First, we need to verify the microprocessor's correct operation under stressful conditions, which rarely, if at all, happen during its operation in a deliverable MP system. Second, we need to verify its functionality and performance when the multiprocessor is running "real world" parallel applications.

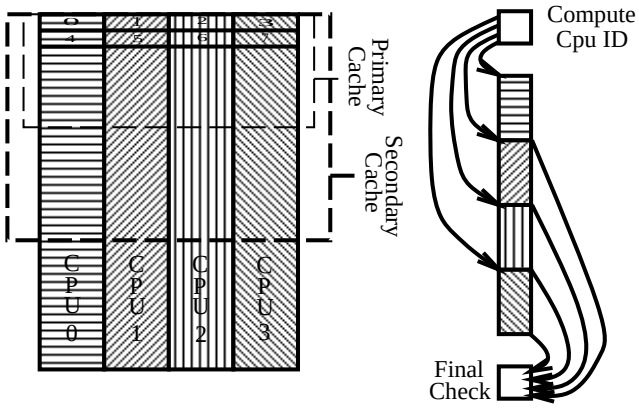

Figure 3: False Sharing in MPVer

5.1 MPVer: A Multiprocessor Verifier

The verification of multiprocessing features is complicated by the interaction between multiple code streams; the unpredictable nature of MP arbitration; and the limited number of MP test suites available to the verification engineer. In order to address these issues, we have used an abundance of asynchronous external events in a uniprocessor environment, as well as developed an MP code generator.

In general, MP verification necessitates the testing of cache coherency protocols and of the correct operation of MP primitives. Generating MP test cases requires the sharing of data between processors combined with locking mechanisms which manage accesses to shared data structures, and which synchronize concurrently executing instruction streams. Computing the expected results of MP test programs is challenging and it is not easily accomplished with a traditional reference machine. MPVer successfully addresses these issues by generating multiple code streams which interact with each other, and yet they are able to verify the produced results with fine granularity. The runtime flow and relationship between the code streams is shown in Figure 3.

A novel approach is used to exploit the important issue of false sharing. Through this approach we are able to achieve high processor interaction and provide full coverage of the cache coherency mechanisms without using expensive locking and synchronization operations, which interfere with the MP program flow and which even limit the number of interesting situations.

True data sharing is supported and tested through the use of locks. However, because intermediate values are unpredictable, results are checked after all MP operations are guaranteed to have finished. For the verification of a microprocessor in a distributed shared memory system, we have parameterized MPVer with the frequency with which each CPU is to access the different memory segments. Such a parameterization is important because we are able to program different traffic patterns, to stress routing algorithms, and to observe MP system stability.

MPVer produces portable code which can run on either a simulation model or a true MP system. In both environments, MPVer

has been very successful in finding MP related microprocessor and system hardware flaws.

5.2 MPApplicationVerifier

MPApplicationVerifier (MPAV) is an environment for the development and execution of "real world" parallel applications as diagnostics in the MP verification of a microprocessor. The environment supports thread-based parallel execution, and it can be considered as a user-level, bare-minimum operating system [4].

The user of the environment writes a single C program, augmented with directives which support its parallel execution. The C program is compiled into two executables which facilitate three execution modes. In the first mode, the user executes the application natively on a workstation or on an MP system. In that way the user is able to debug the application code, and improve its performance and efficiency. In the other two modes of execution, the parallel program is simulated by an architectural simulator and by the microprocessor's RTL model. The purpose of these two execution modes is to test the hardware under construction both at the microprocessor level and at the system level. These modes of execution allow us not only to isolate implementation flaws, but also to pinpoint performance problems.

So far we have ported onto this environment several "real world" parallel applications including the SPLASH-2 benchmarks [9]. Other parallel applications including chaotic algorithms and branch-and-bound algorithms are currently being ported. Incorporating a new application into the MPAV environment is simple. The user only needs to write three "interface functions." Two of these functions perform the initializations of the data structures of the parallel program, whereas the third function provides the environment with the "starting points" of the parallel program's execution. In addition, we can easily incorporate sequential applications into the MPAV environment, such as the diagnostics programs created by other code generation tools.

A powerful, yet flexible, X-based user interface makes MPAV an easy to use MP code generation and execution environment. The user selects the applications to be included in a particular execution, sets the corresponding input parameters for each included application, and then compiles and executes the resulting suite. MPAV's user interface makes the construction and execution of MP test programs a simple exercise for the user.

6 Theo: A Sophisticated Code Generator

State-of-the-art microprocessors employ several advanced techniques in order to improve their performance. At any given time several partially executed instructions are active (i.e. at some stage of their execution) in the processor. Instructions move between different units as resources become available. In order to reduce interruptions in the execution pipeline, which result in lost performance, computed results are bypassed to previous pipeline stages, and state is committed to registers or to memory many cycles after the instruction was issued. Historically, most design flaws have been attributed to the implementation of these complex features. The design flaws typically exhibit themselves when sequences of dependent instructions activate a combination of conditions within the design.

Theo is based on the idea that if we focus on instruction sequences to which a particular implementation may be sensitive, then we can reduce the number of test cases examined, as well as improve the quality of the verification code generated. The overall architecture of Theo is shown in Figure 4.

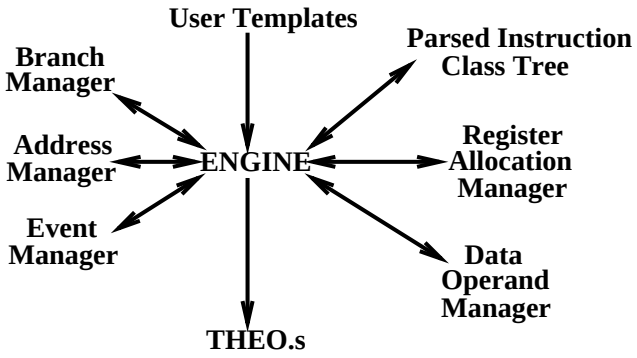

Figure 4: Theo Architecture

The input to Theo is a collection of templates written in a superset of the assembly language, which permits instruction specification at any level of detail, and, at the same time, allows the use of symbolic notation for operands. These templates define sequences of instructions representing "constraints." Theo allows the users to focus on developing sequences for their own area of interest, while Theo's engine searches for their "optimal" placement which satisfies the specified constraints. A typical hand-written diagnostic only stresses a particular unit, while other sections of the microprocessor remain idle. Theo, on the other hand, attempts to combine templates so that all units of the microprocessor are active simultaneously.

Theo uses a constraint solving engine to produce Intermediate Code Representation (ICR) through repetitive application of template instances. Subsequently, it performs instruction assignment, global resource allocation, and condition setup to produce an assembly program ready for simulation [2].

Templates only use symbolic names for registers. The actual register assignment is performed by Theo during one of the last phases in the code generation process. The use of symbolic instruction class names, register names, and operands in templates is encouraged, since this allows Theo to select the actual assembly instructions and operands using sophisticated heuristic algorithms. At the same time, such a notation permits the verification engineer to express the conditions of interest in the most generic way.

Code generation starts with an uninstantiated ICR. Each element in this ICR is a place holder for an instruction which initially has no particular attribute or property. Subsequently, Theo selects one of the user provided templates and applies it to the ICR; that is, the template instruction sequence, its properties, and its constraints are transferred into the ICR. Theo's template placement algorithm avoids placing templates one after the other. Rather, it strives to achieve overlap between templates while maintaining the requirements of each template. This is accomplished by checking for subset properties, by constraint solving, and by temporary unification in order to verify that an overlap can occur. If all resource requirements are met, then the unification becomes permanent. Successive application of the input templates to the ICR results in the further refinement and growth of the code.

Template placement stops when the code size requirement is met. Theo goes through the ICR assigning actual instructions for any instruction class references that may exist. Then, the engine consults the register allocation manager, the address manager, the branch manager, the operand manager, and the external event man-

ager in order to allocate resources and insert condition setups. Finally, the ICR is translated into assembly code.

Though this technique for code generation is complex, it has the unique property that it can create new test sequences from previously independent blocks which now interact with each other. By overlapping templates, we are also able to activate multiple units of the microprocessor while still maintaining the sequence and conditions represented by each template. The various managers utilized by Theo encapsulate heuristic and formal algorithms which may be applied across the entire code stream and which can be tuned with user biasing.

7 Diagnostic Programs Evaluation

7.1 Code Analysis and Diagnostics Retrieval

In their effort to cover as many interesting cases of the given architecture as possible, the code generators presented so far tend to create a large number of lengthy diagnostic programs. This abundance of test programs forces us to seek a systematic and automated way of analyzing the run time behavior of these diagnostics, and post processing this information into concise and meaningful metrics.

Several reasons warrant such an evaluation. First, the code generation tools could use the information from the analysis tool as a feedback in order to improve their effectiveness. Given the pseudorandom nature of the code generation tools, such an analysis has been proven extremely useful in creating diagnostic programs which cover in depth specific sets of interesting cases.

Second, even though the tools can generate a large number of diagnostics relatively fast, only a limited number of them can actually be simulated daily on the RTL model, because this model is complex and, thus, expensive to run. Code analysis is valuable when trying to decide the subset of the created diagnostics that should be simulated on the RTL model.

Third, as the design evolves the number of accumulated diagnostics continuously increases, and the selection of the diagnostics that cover a specific case hardens. One way to address this issue is to build a diagnostic database (DDB) containing all the test programs, along with some information characterizing their run-time behavior. This information can later be used to retrieve a set of diagnostics with particular characteristics from the DDB.

In the following two sections we describe ther two major parts of the evaluation process: the code analysis, which for each diagnostic deduces a set of attribute values, and the systematic storage and retrieval of this information into and from the database. The entire process is outlined in Figure 5.

7.2 Code Analysis - The Profiler

Each generated diagnostic program is currently executed on two simulators. The first one is an architectural simulator which is used as a reference machine. This simulator is fast and inexpensive to use. The second one is the RTL simulator, representing the particular microprocessor implementation. This simulator is much slower than the architectural one, and much more expensive to use.

Whenever a diagnostic is run on any of the two simulators, a trace file containing information about each execution cycle of the diagnostic is created. The current model "passes" the specific diagnostic when the RTL and the architectural traces match under the architectural comparator (refdif in Figure 1).

In order to analyze the execution of a diagnostic, we postprocess the trace file created during the execution of the code on either of the simulators. By doing so, we can deduce information

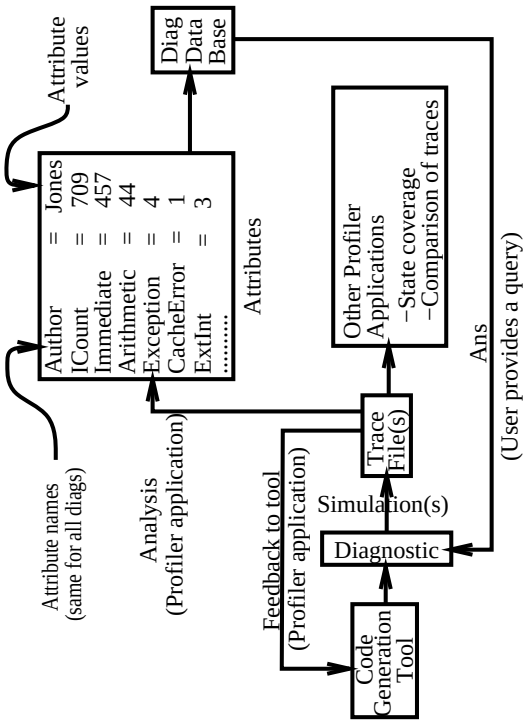

Figure 5: Code Analysis Methodology

about the interesting cases covered during the particular simulation. Examples of interesting cases include cache hits and misses, types of exceptions, and queue sizes. This information is later stored in the DDB.

In order to probe into the trace files systematically and extract interesting information quickly, we have developed the Profiler library which is used as an interface between the analysis code and the trace files. It provides the user with a mechanism for "stepping" through the simulation cycles recorded in a trace file, including going forward and backwards in simulation time. At any given "step" (simulation cycle) the user can retrieve the value of any one of the variables which constitute the machine state.

The library approach was chosen mainly because of the flexibility it provides. Due to its object-oriented design, the interface remains the same irrespectively of the type or format of the trace file being processed. This interface allows the user to write C++ programs that are guaranteed to work in current and future simulation environments.

In addition to diagnostic evaluation, the Profiler library has also been used in a number of other tasks. We have used it to check transition coverage in the RTL model; to compare traces from different models; and to verify that certain (illegal) conditions never arise during the simulation of the model.

7.3 Storage and Retrieval of the Results - The Diagnostic Database

Every time a diagnostic program is simulated, a Profiler-based analysis code is executed on the trace file which represents the particular simulation. The results of this analysis are typically expressed as a set of values for a prespecified, common for all diagnostics, set of attributes. Example of attributes generated during the analysis include the number of instructions executed, the number of cache hits and misses, and the lengths of various queues in

the microprocessor.

We developed a tool called Ans which compresses all this information into a highly efficient, object-based diagnostic database (ODDB). Ans both modifies the database and queries it to retrieve a set of objects (diagnostics) that satisfy a given set of criteria. Ans is a general tool, designed to handle any object-based collection of data, in a highly sophisticated and user friendly way.

When retrieving objects from a database, Ans uses an input set of criteria to select and return a set of objects which satisfy the given criteria. These criteria are usually expressed in some form of equalities or inequalities on the attribute values of the objects stored in the database. For example, the user can ask for all the diagnostics that contain less than 2000 instructions (i.e. attribute 'ICount' is less than 2000), and take at least 10 floating point exceptions (i.e. attribute 'FP_Exception' is greater than or equal to 10). The following attribute form describes these two constraints: $((ICount <= 2000)\&\&(FP\_Exception >= 10))$. Given this form, Ans would select a subset of the diagnostics currently stored in the given DDB whose attribute values satisfy the given constraints and would return these diagnostics to the user.

8 Summary

In this paper we have presented a collection of advanced code generation tools employed in the simulation-based verification of high-performance microprocessor designs. Each of the presented tools addresses a unit of the microprocessor which historically has been a significant source of hard to find flaws. We presented SBVer, a code generator which focuses on exercising the external interface and cache management units of the microprocessor. Then we described BRVer which targets the branch mechanisms of the design; these mechanisms become increasingly more complicated as designers attempt to improve the performance of the chip through speculative execution. We addressed MP verification by presenting two tools with complementary roles. MPVer targets the sharing of information across the processor of an MP system as well as the communication between processors. MPAV, on the other hand, provides an environment for the development and execution of "real world" parallel applications on our simulators. Finally, Theo provides a state-of-the-art environment for the generation of diagnostics based on user provided templates, constraint solving systems, and knowledge of the microprocessor design.

We have also presented the Profiler and Diagnostic Database which comprise a set of tools for the analysis of diagnostics and their efficient storage and retrieval. These tools provide us with efficient ways to evaluate the code produced by the generators and to propagate this information back to the tools so that we can improve their effectiveness.

We are currently working on expanding our tool-set with highly specialized code generators as well as powerful generic ones. Furthermore, we are extending our sophisticated heuristic algorithms to cover areas that have not yet been addressed. Finally, we incorporate all our verification tools in an integrated environment which supports the easy and efficient production of high quality diagnostic programs.

Design verification is an important part of the development of a microprocessor. As time-to-market decreases and the complexity of the high-performance microprocessors increases, design verification becomes the bottleneck of the development cycle. Good verification tools become vital to the success of any microprocessor design, and their significance will continue to increase as we move to even higher performance microprocessors.

References

[1] M. Bass, T.W. Blanchard, D.D. Josephson, D. Weir, and D.L. Halperin. Design Methodologies for the PA 7100LC Microprocessor. *Hewlett-Packard Journal*, 46(2):23–35, April 1995.

[2] A. Chandra et al. AVPGEN –A Test Generator for Architecture Verification. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(2):188–200, June 1995.

[3] B. Turumella et al. Design Verification of a Super-Scalar RISC Processor. In *Twentyfifth International Symposium on Fault Tolerant Computing*, pages 472–477, June 1995.

[4] I. Foster. *Designing and Building Parallel Programs*. Addison Wesley, 1995.

[5] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[6] M. Kantrowitz and L.M. Noack. Functional Verification of a Multi-issue, Pipelined, Superscalar Alpha-Processor – the Alpha 21164 CPU Chip. *Digital Technical Journal*, 7(1):136–144, August 1995.

[7] D. Marr, S. Thakkar, and R. Zucker. Multiprocessor Validation of the Pentium Pro Microprocessor. In *Proceedings of COMPCON '96*, pages 395–400, January 1996.

[8] B. O'Krafka, S. Mandyam, J. Kreulen, R. Raghavan, A. Saha, and N. Malik. MTPG: A Portable Test Generator for Cache-Coherent Multiprocessors. In *Fourteenth Annual Phoenix Conference on Computers and Communications*, pages 38–44, March 1995.

[9] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd ISCA*, pages 24–36, June 1995.