Hazard Driven Test Generation for SMT Processors

Padmaraj Singh
Nvidia
Portland, OR USA

Vijaykrishnan Narayanan[†]
Department of Computer Science & Engineering
Pennsylvania State University
University Park, PA USA

David L. Landis
Department of Materials Science & Engineering
Carnegie Mellon University
Pittsburgh, PA USA

Abstract— **Multithreaded processors increase throughput by executing multiple independent programs on a single pipeline. Simultaneous Multithreaded (SMT) processors execute multiple threads simultaneously thus add a significant dimension to the design complexity. Dealing with this complexity calls for extended and innovative design verification efforts. This paper develops an analytic model based SMT random test generation technique. SMT analytic model parameters are applied to create random tests with high utilization and increased contention. To demonstrate the methodology, parameters extracted from the PPC ISA and sample processor configurations are simulated on the SMT analytic model. The methodology focuses on exploiting data/control and structural hazards to guide the random test generator to create effective SMT tests.**

Keywords-simultaneous multithreading; superscalar ; analytic model; Markov chains; data hazards; control hazards; structural hazards; random test generation

I. INTRODUCTION

A Multithreaded (MT) processor executes multiple independent programs on a single pipeline. Increased pipeline utilization leads to improvement in overall throughput. Given the increase in performance, several commercial processors implement multithreading [1][2][3]. Multithreading techniques may be broadly classified as interleaved multithreading (IMT), blocked multithreading (BMT) and simultaneous multithreading (SMT) [4]. However, the act of simultaneously tracking multiple threads in the pipeline significantly adds to the overall design complexity [5].

Increased design complexity leads to the need for extended and innovative design verification techniques. Published literature on pre-silicon SMT design verification techniques is sparse. Intel® Hyper-Threading validation is discussed in [6]. SMT test application techniques are presented in [7] and [8]. Intel® Hyper-Threading verification effort augmented the standard verification methodologies by paying special attention to the cross product of architectural state space, logical processor data sharing, logical processor forward progress, atomic operations and self-modifying code.

[†]This work was supported in part by NSF grants 1147388,0916887 & 0903432.

978-3-9810801-8-6/DATE12/©2012 EDAA

IBM's Power 7 SMT verification focused on MP-based SMT verification where the separate scenarios are described for each thread to emphasize collisions between shared resources such as memory, caches and registers. They also introduce thread irritation techniques to facilitate thread coordination and result checking.

Simulation based verification continues to dominate the overall pre-silicon processor design verification effort. Random test generators are indispensable in this endeavor. Functional coverage terms help quantify the stimulus quality and effectiveness but it is impossible to think of and implement coverage terms for all possible corner cases. Completely random tests will require exorbitant amount of compute resources to adequately target all areas of the design. Therefore, it is necessary to develop and apply techniques to effectively focus the random test generator output to exercise the desired design feature. Such a technique to randomly generate effective SMT tests is presented in this paper.

This work is based on the analytic model developed in [9]. We implement the analytic model and derive parameters to drive the random test generator. The resulting process creates tests boasting optimal unit utilization with high inter-instruction dependencies. Thus, verification quality and efficiency is improved by utilizing key design parameters to tailor the random tests. Without such enhancements, the probability of random generators exercising subtle corner cases in the design is significantly reduced. The paper focuses on data/control and structural hazard parameters to drive the test generator.

Section II presents the SMT analytic model. Section III applies the PPC instruction set to three sample SMT processor configurations. Section IV derives the model data/control hazard parameter to feed the random generator. Section V introduces a methodology to adjust the test instruction mix to control the structural hazards. Finally, section VI summarizes the work and presents directions for future effort.

II. THE MODEL

Multithreaded analytic model developed by Yamamoto *et al* [9] is implemented. We implement the model to compute processor Instructions executed Per Cycle (*IPC*) as a function of processor configuration and workload. The resulting data

help generate optimal multithreaded tests for simulation based processor design verification. The model defines *IPC* as:

$$IPC = \sum_{w=1}^{G} P_w * IPC_w \qquad (1)$$

where P_w is the probability of having w instructions in the global window and IPC_w is the expected *IPC* measured for a global window size w. IPC_w models the effects of *structural* hazards. P_w is the performance degradation due to *data* and *control* hazards. When two or more instructions contend for the same hardware resource a structural hazard occurs. Data hazards occur when instructions with data dependency modify data in different stages of the pipeline. Branches cause control hazards. A global issue window consists of all instructions free of data or control hazards ready to be dispatched to the functional units. Vector ***M*** represents the global issue window state:

$$M_w = [m_1, m_2 \cdots m_T], w = \sum_{i=1}^{T} m_i \qquad (2)$$

where m_i is the number of instructions of type i in the global issue window and T is the total number of functional units. The total number of states for global window size w and T functional unit types will be $\binom{w+T-1}{T-1}$. I_m is the total number of instructions that can be dispatched given the global issue window is in state ***M***.

$$I_M = \sum_{x=1}^{T} i_x, i_x = \min(c_x, m_x) \qquad (3)$$

where m_x is the number of instructions of type x in the window and c_x is the number of functional units of type x. The configuration vector is $\boldsymbol{C} = [c_1, c_2, \ldots c_T]$ representing all the functional units in the processor. Then the structural hazard driven *IPC* is:

$$IPC_w = \sum_{M} I_M Q_M \qquad (4)$$

where Q_M is calculated by obtaining the steady state probabilities of a Markov chain involving the states of the global issue window ***M***. All possible next states are determined by the number of instructions that can be executed for a given processor configuration and global window state. The workload instruction mix vector ***V*** determines the state transition probabilities. For example, a workload with 25% integer, 30% floating point, 35% SIMD and 10% branch instructions will be represented as ***V****=[0.25,0.30,0.35,0.1]*. The *IPC* degradation due to data and control hazard is represented in (5).

$$P_w = A \frac{e^{-1}}{w!} \qquad (5)$$

A is the dependency degradation factor. As the instruction count in the issue window increases then the chances of inter-instruction dependencies also increase.

The next section describes the simulation of the PPC instruction set on three processor configurations to compute the *IPC* based on the above SMT processor analytic model.

III. PPC ISA AND SMT ANALYTIC MODEL

The PPC Instruction Set Architecture (ISA) contains 728 unique instructions [10]. If the random generator assigns equal weight to each instruction then the runtime instruction mix vector ***V*** for PPC is constructed as in Table I. For example, there are 128 load-store instructions, thus v_5=128/728=0.18.

TABLE I. PPC RUNTIME INSTRUCTION MIX VECTOR

Functional Units	***V***		**Configurations**		
			C1	***C2***	***C3***
Fixed Point	v_1	0.13	2	2	2
Decimal Floating Point	v_2	0.07	4	1	1
Double Precision Fixed Point	v_3	0.13	1	2	1
Vector	v_4	0.46	1	2	4
Load-Store	v_5	0.18	2	2	2
Branch	v_6	0.01	1	1	1
Condition Register Logic	v_7	0.02	1	1	1

Table I also lists the three sample processor configurations that will be analyzed in this paper. There are 7 functional unit types in each configuration and 12, 11 and 12 total functional units in configurations C1, C2 and C3 respectively.

For the experiment, the number of threads N = 2, the thread issue window size S = 3, and the global issue window $G = N*S$ = 6. To effectively apply the analytic model for test generation, a function to model the data and control hazard degradation (P_w) is introduced in (6). It is similar to (5) in that the probability of dependencies increases as the number of instructions (w) increases.

$$P_w = \frac{1}{w^d} \qquad (6)$$

where d is the degradation parameter. Data and control hazards determine P_w. Inter-instruction dependencies cause data hazards and are categorized as Read-After-Write (RAW), Write-After-Read (WAR) and Write-After-Write (WAW). Inter-instruction dependencies are resolved through various static and dynamic techniques. Methods also exist to analyze and predict hazards in a given program [11]. The ISA, program characteristics and underlying micro-architecture interact to determine the affects of data and control hazards on the overall *IPC*. As the number of instructions in a given window size increase, the probability of dependencies increase. Adjusting the random test generator input parameters can control the rate at which the inter-instruction dependencies increase with the window size.

In all configurations, the overall *IPC* is constrained by the small window size *S*. A value of 3 is selected to keep the number of states in the Markov chain model manageable. The

SIMD instructions dominate the PPC ISA with runtime instruction mix probability v_4 of 0.46 for the *vector* unit. The number of *vector* units in configurations C1, C2 and C3 are set to 1, 2 and 4 respectively. Consequently, the throughput of random tests increases with *IPC*=3.463, 4.064 and 4.264 for C1, C2 and C3 respectively when d=0.001. The results demonstrate the improvements in *IPC* as number of *vector* units are increased. Section IV develops steps to utilize the *IPC* results for effective SMT test generation.

IV. Optimized Degradation Parameter Derivation

Average functional unit utilization is highest when *IPC* is at it's maximum. When runtime mix vector $\boldsymbol{V}$ and processor configuration vector $\boldsymbol{C}$ are held constant then data and control hazard degradation probability P_w is the key determinant in the *IPC* computation. A random test generator may bias the inter-instruction dependencies while preserving the random selection of instructions. Degradation parameter d may be used as the input into the random test generator to generate high utilization tests with optimal data and control hazards. Objective is to find the degradation parameter d resulting in maximum *IPC* with maximum data and control hazards. Thus, keeping the functional units maximally occupied within a stressful instruction stream with maximal data and control hazards.

Degradation parameter d in a completely random test is set to around 0.5. This will rarely result in maximum possible *IPC* in a processor. Thus, the tests will exercise the part inadequately. When d=0.5 *IPC* drops by 23, 16 and 10 percent from maximum *IPC* for C1, C2 and C3 respectively. However, by tailoring the biasing of the data and control hazards based on the Markov chain model enables superior SMT test generation.

To further bias the test generator towards creating effective SMT tests, degradation d is modulated during the duration of the test. Consequently, the functional units experience a range of utilization while working through various levels of inter-instruction dependencies. A logical choice is to modulate d in a sinusoidal manner as represented in (7).

$$d = \beta \sin(\mu * g) \qquad (7)$$

where d is degradation parameter, β is d with maximum *IPC*, μ is the modulation frequency and g is determined by instructions per test. In the example configurations, *IPC* start settling to a constant around d=3, thus β=3. For example, in configuration C1, to modulate *IPC* a single cycle within a 500-instruction test, $\mu \approx 0.012$.

Two additional variations in d will further improve the test quality:

- Constrain data/control hazards on thread basis.
- Constrain data/control hazards on functional unit basis.

The global issue window distributes the instructions to the functional units. The instructions in the global issue window are free of data and control hazards. Thus, beyond this point the data/control at thread or functional unit level granularity are irrelevant. However, the hardware leading up to the global issue window must sort through these dependencies. Skewed dependency burdens may introduce unusual backlogs leading to rare corner cases. Thus, during test generation degradation parameter may be broken down and varied according to (8) and (9).

$$d = f(d_0, d_1, d_2 \ldots d_N) \qquad (8)$$

where d_0-d_N are the degradation parameters for each thread. Typically, d will be the average degradation across all threads but sophisticated variations to the function in (8) may be introduced. The degradation on functional unit basis will be a multidimensional function:

$$d = f\begin{pmatrix} d_{00} & \cdots & d_{0T} \\ \vdots & \ddots & \vdots \\ d_{N0} & \cdots & d_{NT} \end{pmatrix} \qquad (9)$$

where d_{NT} is the degradation in thread N due to instruction type T. In the current example, the degradation is uniform across all threads. The runtime instruction mix vector in Table I influences the instruction type degradation. A random test is dominated by SIMD instructions (v_4=0.46). Under these conditions the dependencies within the SIMD instruction data and registers drive the effective degradation parameter. However, it is possible to create potent data/control hazard driven SMT tests by selectively manipulating d_{00}-d_{NT} in (9).

Calculating Q_M in (4) is the most compute intensive step. The number of Markov chain states increase significantly as the thread count, window size and functional unit count increase. But for a given configuration it has to be computed once, then Q_M may be used for various optimization computations. This section introduced effective ways of deriving degradation parameter d to drive the SMT test generator. Next section proposes a methodology to adjust runtime instruction mix vector V to generate effective SMT tests.

V. Runtime Instruction Mix Vector $\boldsymbol{V}$

Average functional unit utilization is highest when *IPC* is at it's maximum. Degradation parameter in an unconstrained randomly generated test is set to 0.5. The runtime mix vector for a randomly generated PPC SMT test is $\boldsymbol{V}$=[0.13,0.07,0.13,0.46,0.18,0.01,0.02]. The resulting *IPC* are 2.65, 3.41 and 3.82 for configurations C1, C2 and C3 respectively. The random $\boldsymbol{V}$ does not produce optimal utilization of the functional units. To balance the load evenly onto all functional units vectors, $\boldsymbol{V}$ is derived from $\boldsymbol{C}$ as in (10).

$$\boldsymbol{V_{opt}} = \frac{\boldsymbol{C}}{f_{total}} \qquad (10)$$

where f_{total} is the total number of functional units. Thus, (10) sets the instruction probability according to the number of functional units of a particular type. For example, in configuration C1, total number of functional units (f_{total}) is 12.

The number of decimal floating-point units (c_4) is 4. The resulting probability of picking a decimal floating-point instruction (v_4) is set to $c_4/f_{total} = 4/12 = 0.333$.

On redistributing $\boldsymbol{V}$ according to (10) with data/control hazards remaining random (d=0.5) then the *IPC* are 3.8, 4.5 and 5.6 for configurations C1, C2 and C3 respectively. These *IPC* numbers represent the optimal utilization of functional units with data/control hazards left unconstrained. $\boldsymbol{V_{opt}}$ will keep the functional units busy but two related factors remain to be addressed:

- Fair distribution of instructions in the tests
- Backlog on functional units to stress the design

In a test with unbiased distribution of instructions, each instruction in the ISA has equal probability of being selected. This provides greater variations in the test with all instructions experiencing equal exposure. Runtime instruction mix vector $\boldsymbol{V}$ according to Table I represents a fair distribution of PPC instructions and henceforth will be referred to as $\boldsymbol{V_{ISA}}$.

To induce extreme backlog the runtime instruction mix vector is skewed to 1 for each functional unit type. For example, to create a backlog on the fixed-point unit we set $\boldsymbol{V}$=[1,0,0,0,0,0,0]. Such vectors are termed $\boldsymbol{V_{bklg0}}$ through $\boldsymbol{V_{bklgT}}$. Therefore, to optimally exercise the SMT processor the tests must oscillate among the vectors $\boldsymbol{V_{opt}}$, $\boldsymbol{V_{ISA}}$ and $\boldsymbol{V_{bklg0}}$-$\boldsymbol{V_{bklgT}}$.

Rather than abruptly changing instruction probabilities v_0 through v_T while transitioning between $\boldsymbol{V_{opt}}$, $\boldsymbol{V_{ISA}}$ and $\boldsymbol{V_{bklgT}}$, their values are changed gradually. One approach is incrementing or decrementing v_0-v_T by some value Δv till the current $\boldsymbol{V}$ matches the subsequent instruction vector mix. Another approach is to assure that the *IPC* does not change abruptly. Smooth transition between IPC_{opt}, IPC_{ISA} and IPC_{bklg} can be created by increasing or decreasing *IPC* by a predetermined amount ΔIPC. The matching $\boldsymbol{V_{transition}}$ for the ΔIPC is derived from the SMT analytic model. With the aid of a preliminary high-level model of the processor, the SMT test generator will create an effective mix of instructions to strategically modulate the overall *IPC* during the test.

VI. Summary and Future Work

Standard verification methodologies continue to serve in validating SMT processor correctness. Targeted random stimulus is indispensable in leading the design into difficult to reach corner cases.

We implemented the analytic model from [9] and developed a hazard-based methodology to drive the SMT test generator. Three sample processor configurations and the PPC ISA profile helped demonstrate the model. Data/control hazards degrade *IPC* in each configuration. The *IPC* vs. degradation parameter d profile varies from configuration to configuration. This paper developed a generalized method to vary d as a function of test length for a given configuration ($\boldsymbol{C}$) and runtime instruction mix vector ($\boldsymbol{V}$). Simple random SMT tests will hover around sub-optimal *IPC* (23, 16 and 10 percent below the maximum *IPC*). Thus, exercising the functional units at low utilization. In contrast, the methods developed here significantly increase the SMT test potency by strategically varying degradation parameter d across the test to alternate between maximum and minimum *IPC* in a sinusoidal fashion. The paper also started to explore degradation as a function of thread count and functional unit type.

Section V introduced a formula to compute optimal runtime instruction mix vector $\boldsymbol{V_{opt}}$. Functional unit utilization is significantly higher with $\boldsymbol{V_{opt}}$, thus exposing the machine to a more realistic and stressful traffic than a completely random SMT test generated with $\boldsymbol{V_{ISA}}$. $\boldsymbol{V_{bklg}}$ stresses individual functional unit types to the extreme. $\boldsymbol{V_{transition}}$ facilitates transitions between different instruction mix vector $\boldsymbol{V}$ types. $\boldsymbol{V_{ISA}}$ leads to sub-optimal IPC_{ISA}, which result in low quality tests. Strategically varying $\boldsymbol{V_{opt}}$, $\boldsymbol{V_{ISA}}$, $\boldsymbol{V_{bklg}}$ and $\boldsymbol{V_{transition}}$ across a test lead to potent SMT tests.

The techniques presented in this paper may be expanded into several directions. Optimally determining degradation parameter sub components in (8) and (9) is an interesting topic. Optimally selecting $\boldsymbol{V_{opt}}$, $\boldsymbol{V_{ISA}}$, $\boldsymbol{V_{bklg}}$ and $\boldsymbol{V_{transition}}$ based on test size will further improve the test quality. Future experiments may also explore simultaneous variation of d and $\boldsymbol{V}$.

References

[1] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd, "Power7: IBM's next-generation server processor," IEEE Micro, vol. 30, issue 2, pp. 7-15, March 2010.

[2] D. Koufaty and D.T. Marr, "Hyperthreading technology in the netburst microarchitecture," IEEE Micro, vol. 23, issue 2, pp. 56-65, March 2003.

[3] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded Sparc processor," IEEE Micro, vol. 25, issue 2, pp. 21-29, March 2005.

[4] T. Ungerer, B. Robic, J. Silc, "A survey of processors with explicit multithreading," ACM Computing Surveys, vol. 35, no. 1, pp. 29-63, March 2003.

[5] J.L.Lo, J.S.Emer, H.M.Levy, R.L.Stamm, D.M.Tullsen and S.J.Eggers, "Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading," ACM Trans. On Comp. Sys. (TOCS), vol. 15, issue 3, pp. 322-354, Aug. 1997.

[6] D.Burns, "Pre-Silicon Validation of Hyper-Threading Technology," Intel Technology Journal, vol. 6, issue 1, Feb. 2002.

[7] K.D. Schubert, "POWER7: verification challenge of a multi-core processsor," ICCAD '09 Proc. Int. Conf. on Comp. Aided Design, pp. 809-812, Nov. 2009.

[8] J.M.Ludden, M.Rimon, B.G.Hickerson, and A.Adir, "Advances in simultaneous multithreading testcase generation methods," HVC'10 Proc. Int. Conf. on HW & SW Verif. & Testing, Oct. 2010. (http://www.research.ibm.com/haifa/conferences/hvc2010/present/Advances_in_Simultaneous_Multithreading_Testcase_Generation_Methods.pdf)

[9] W.Yamamoto, M.J.Serrano, A.R.Talcott, R.C.Wood, and M.Nemirovsky, "Performance estimation of multistreamed, superscalar processors," In Proc. 27th Hawaii Int. Conf. on Sys. Sc., IEEE Comp. Soc., pp 95-204, Jan. 1994.

[10] *Power ISA™ Version 2.06*, IBM 2009. (https://www.power.org/resources/downloads/PowerISA_V2.06B_V2_PUBLIC.pdf)

[11] T.M. Austin, and G.S. Sohi, "Dynamic Dependency Analysis of Ordinary Programs," In Proc. 19th Int. Symp. on Comp. Arch., IEEE and ACM,pp342-351,May1992.