Design Validation of Multithreaded Processors Using Threads Evolution

D. Ravotto, E. Sanchez, M. Sonza Reorda, G. Squillero

Dipartimento di Automatica e Informatica, Politecnico di Torino, Italy
Contact author e-mail: ernesto.sanchez@polito.it

ABSTRACT

Within the design arena of modern devices based on cutting-edge processor cores, the availability of effective verification, validation and test methodologies able to work on high-level descriptions of processor cores represents an interesting advantage, since it can dramatically reduce the overall time for design and manufacturing, while improving yield and quality. In this paper we propose a semi-automatic test program generation technique able to target modules in modern computer architectures that implement the multithreading paradigm. The methodology starts from high level descriptions of processor cores and using an incremental multi-run approach produces, with very limited manual intervention, a test set able to maximize verification metrics. Experimental results gathered on a couple of real complex designs (the OpenSPARC™ T1 and T2) show the effectiveness of the proposed methodology.

Index Terms: SBST, multithread processors, functional validation, automatic program generation.

1. INTRODUCTION

Most history of multithreaded processors starts in 1964, with the *peripheral processing units* in Seymour Cray's *CDC 6600.* However, multithreading as a technique to further exploit instruction-level parallelism, is much more recent. It becomes widely adopted in the desktop market only in the 2000s, and nowadays represents an effective solution for coping with market requirements of modern off-the-shelf electronic devices. Modern microprocessors exploit a significant quantity of hardware resources to allow multiple threads execution in an overlapping fashion, increasing the global throughput.

A drawback stemming from the inclusion of high-performing microprocessors in electronic devices is the increase in development times. Indeed, verification, validation and testing (VV&T) methodologies have been hardly able to keep the same pace as production methodologies.

The design of electronic devices is commonly carried out through an incremental process delivering a more complex description of the device at every step. The methodology allows designers to progress towards the final product in a gradual manner, decreasing at every step the abstraction level of the description, up to obtain the final model. The first design step produces the most abstract description, which describes the general behavior of the device leaving internal details out, whereas the last steps provide low-level descriptions, with details close to the actual implementation of the circuit. Clearly, the lower the abstraction level, the higher the complexity of the resulting model.

Unfortunately, no mature VV&T technique is able to take advantage of the higher-level models in order to reduce the efforts required to validate the lower-level ones, and hence of the final product.

Considering VV&T in general, it is possible to state that almost all techniques can be classified either as *formal* or *simulation-based*. Formal methodologies exploit mathematical techniques to prove specific properties, such as the absence of deadlocks or the equivalence between two descriptions. Such a proof implicitly considers all possible inputs and all accessible states of the circuit. Differently, simulation-based techniques rely on the simulation of a set of stimuli to unearth misbehaviors. A simulation-based approach may therefore be able to demonstrate the presence of a bug, but will never be able to prove its absence. Indeed, the user may assume that a bug does not exist with level of confidence related to the quality of the simulated test set. The generation of a qualifying test set is the key problem with simulation-based techniques. Different methodologies may be used to add contents to such test sets, ranging from *deterministic* to *pseudo-random*.

Theoretically, formal techniques are able to guarantee their results, while simulation-based approaches can never reach complete confidence. However, the former require an enormous computational power, and therefore may not be able to provide result for a complex design. Moreover, to enable their use, formal methodologies are routinely used on simplified models or with simplified boundaries conditions. Thus, their result is absolutely exact, but the model used to calculate it was not. This introduces a certain amount of uncertainty in the process [2].

Nowadays, simulation-based techniques dominate the VV&T arena for microprocessors, with formal methods bounded to very specific components in the earliest stages of the design. When tackling microprocessors, a set of stimuli has most likely the form of an assembly program. Such test programs, whose introduction dates back to the 80s [4], are executed by the device under test, but their purpose is not to calculate a certain value or to solve a problem, but rather to excite the different supported functions, and to expose the internal information. Test programs have been exploited in the so-called *Software-based Self-test* (SBST) [5], where the goal is to make evident a difference between a faulty device and a working one.

Multithread microprocessors exploit complex architectural paradigms that are both hard to be tackled by hand and greatly limit the effectiveness of random-based approaches. Consequently, the creation of a suitable set of test programs is liable to be a bottleneck in the design process.

Feedback-based techniques are used to iteratively improve a test set in order to maximize a given target measure. The improvement procedure is fully automatic and exploits the result of the simulation to drive the creation of new stimuli. However, simulation of low-level descriptions could require enormous efforts in terms of computational time, memory occupation and hardware.

In this work we propose a simulation-based method able to efficiently exploit high-level descriptions of multithreaded microprocessor cores for generating effective test programs of critical modules. The methodology is mostly automatic and makes use of the feedback from the simulation. The proposed methodology enhances and extends to multithreaded complex microprocessors the idea presented in [3].

The feedback-based generation process is driven by an evolutionary algorithm, and exploits a high-level simulator. The method is focused on the generation of multithreaded assembly test programs targeting the maximization of a set of metrics. The evolutionary algorithm was upgraded to generate, represent and effectively manipulate multithreaded programs. Moreover, in order to minimize the required computational effort, the approach is organized in a series of runs where, at each step, the best test program generated is added to the final test set. Preliminary steps on this work has been presented in [6] and [7].

To demonstrate the suitability and the scalability of the approach, we tackle the *OpenSPARC™ T1* and *OpenSPARC™ T2*, two rather complex microprocessors developed by *SUN Microsystems*. The high-level models of the two microprocessors are publicly available from *SUN Microsystems* [8]. We generated test sets for two different characteristic internal modules.

The experimental results clearly show that the method is effective both in reaching high figures when a given coverage metric is considered, and in generating test programs able to cover very specific corner cases, that could hardly be managed through a random approach. Our method requires a very limited effort in terms of manual intervention, while the required computational power can be traded-off by running it separately for different components of the microprocessor.

The rest of the paper is organized as follows. Section II recalls some concepts about design validation and presents the family of OpenSPARC™ microprocessors. Section III describes the proposed approach. Section IV presents the case study and reports some experimental results. Section V concludes the paper.

2. BACKGROUND

A. VV&T approaches

Design VV&T methodologies have been developed in a multi-flavored spectrum. The first sharp distinction that can be made is between *formal* and *simulation-based* techniques.

Formal methodologies use mathematical techniques to prove that a design possesses certain properties, such as conformity to given specifications. Generally speaking, formal verification may always be seen as a form of theorem-proving within a given logic system [9]. But, in practice, research in this field falls within various sub-categories, such as: *automated theorem proving*, *model checking*, *equivalence checking*, and *SAT solver*.

While theoretically formal methodologies are able to verify any property, the computational resources required become significant even for medium-complexity designs. Moreover, when a formal method fails to provide a result, nothing sensible can be said on the examined property.

In order to limit the CPU time and memory requirements, formal methodologies are often applied on simplified models, or sub-parts of the original design. However, an over-simplification in the former case, or a mistake in setting the boundaries conditions in the latter, may impair the reliability of the result.

Simulation-based methodologies aim at uncovering design errors by thoroughly exercising the available model. Briefly speaking, a VV&T process based on simulation requires three basic elements to be performed: the input information (also called set of *stimuli*), the model of the device under evaluation (also called *design* or *device under test*), and finally the *response checker*, which generates the pass/fail information regarding the inspection process based on the comparison between the observed and the expected behavior.

It is clear that depending on the design stage, the audit process could be performed in different ways. For example, depending on the design state, the method could be based on a logic simulator or resort directly to the circuit if the device is already built.

Simulation-based methodologies are strongly dependent on the quality of the set of stimuli used to excite the design. These methodologies are seldom exhaustive and only consider a limited sub-set of possible circuit behaviors. Since the quality of the results depends on the percentage of applied stimuli with respect to the total possible ones, they never achieve 100% confidence of correctness in practical cases. However, a qualifying test set may give a reasonable confidence on the absence of problems.

Producing high-quality set of stimuli is therefore a major issue in the VV&T area, which also requires asserting the quality of a set of stimuli.

Borrowing the idea from software testing [1], it is possible to define a coverage metric as the measure of how thoroughly exercised a device under test is. Code coverage metrics identify which code structures belonging to the circuit description are exercised by the set of stimuli, and whether the control flow graph corresponding to the code description has been thoroughly traversed. The structures exploited by code coverage metrics range from a single line of code to *if-then-else* constructs. Nowadays, commercial logic simulators provide the users with the possibility to extract, during simulation, information regarding coverage metrics; some of the most typically used ones are: *statement coverage*, *branch coverage*, *condition coverage*, *toggle coverage*, among others. The capacity of a given input stimuli to activate specific features of the model may be quantified, and coverage metrics for hardware verification can be defined to assure the adequacy of the set of stimuli, and the collected information about coverage could be exploited as an useful termination test criterion [2]. The higher the coverage values obtained by a given set of stimuli, the higher the confidence in the design it can provide. Intuitively, high coverage values imply high system activation. However, it is worth noting that coverage does not imply that the design fully conforms to the specifications.

Specifically regarding to microprocessor cores, some VV&T techniques based on formal approaches have been exploited to tackle pipelined designs [13][14], as well as superscalar ones [15]. However, the applicability of such techniques within an industrial design flow may need a considerable human effort. Consequently, industry adopts formal methods when facing only single modules or when boundary conditions allow to significantly constraint the targeted task.

On the other hand, constrained-random test generation develops random test sets following constrained generation processes [16][17][18][19]. These techniques usually exploit templates in which the structure of each program fragment to be generated, as well as the parameters to be randomized, are previously defined. Though, such techniques are very challenging when targeting real complex designs.

An evolution of the constrained-random methods is based on the exploitation of feedback information able to drive the generation process. These techniques [20][21][22][23] dynamically and automatically adapt the generation process resorting to coverage feedback results, in order to improve the generated programs. Moreover, they generate and simulate a huge quantity of programs, but the generation process leaves users with only a small set of test programs.

Authors in [24] presented an efficient test generation technique based on decomposition of both design and properties for functional validation of pipelined processors. Results gathered on a multiple issue microprocessor core demonstrate the suitability and efficiency of the proposed approach.

Finally, talking about chip multithread processor cores, some VV&T strategies have been also proposed in order to couple with modern issues. For example, regarding validation techniques, in [7] the authors proposed a feedback based methodology that exploits an automatic generator in order to generate validation programs for a complex processor core. Concerning testing issues, in [25] the authors proposed a novel methodology to speed up the execution of self-test routines in a multithread processor chip.

B. Basics on OpenSPARC™ processor cores

The OpenSPARC™ processor core family is designed exploiting the chip multi-threaded (CMT) processor architecture paradigm. The processor family fully implements Sun's Throughput Computing initiative for the *horizontal system space*. Throughput Computing is a technique that takes advantage of the thread-level parallelism [8].

The basic principle behind Throughput Computing is that exploiting ILP (Instruction Level Parallelism) and deep pipelining has reached the point of diminishing returns, and as a result current microprocessors do not utilize their underlying hardware very efficiently. For many applications, the processor is usually idle most of the time waiting on memory, and even when it is executing, the processor is often able to only utilize a small fraction of its wide execution units. So rather than building a large and complex ILP processor that sits idle most of the time, a number of small, single-issue processors that employ multithreading are built in the same chip area. Combining multiple processors on a single chip with multiple *strands* or threads per processor, allows very high performance for highly threaded applications. This approach is usually called thread-level parallelism (TLP).

The one disadvantage to employing TLP over ILP is that execution of a single thread will be slower on the TLP processor than an ILP processor. However, current processors running at high frequencies, a strand capable of executing only a single instruction per cycle is fully capable of completing tasks in the time required by the application, making this disadvantage a nonissue for nearly all applications [8].

3. PROPOSED APPROACH

A. Introduction

Multithreaded processor cores (as other modern processor architectures) rely on the efficient distribution of independent instructions among the hardware resources available into the processor; for example, in the case of fine grained multithreading architectures, at every clock cycle it is required to switch the context from the current thread to the next available one. In fact, design engineers are required to include into the processor core complex modules able to handle with all these scheduling requirements.

Test program generation for these complex architectures is really a challenging task: considering multithreaded processors, the search space to be explored is even larger than for other processors, and includes all the possible combinations of sequences of instructions (one for each thread).

The real challenge in a multithreaded environment is to find a thread-oriented test program that runs in all the processor threads, and it is able to fully excite a specifically targeted module. This is not easily performed neither resorting to a random-based approach nor by hand, even though a deep knowledge of the processor core is available. As a matter of facts, facing test program generation by exploiting simulation-based approaches that rely on random or deterministic methods may require enormous efforts in terms of computational time and memory occupation.

Finally, it is worth mentioning the importance of devising methods able to automate as much as possible the generation process, reducing the need for skilled (and expensive) human intervention, and guaranteeing an unbiased coverage of corner cases.

The semi-automatic methodology we propose in this paper faces the above issues: it is built on a suitable feedback-based generation algorithm and a simulation-based program evaluator. The candidate test programs are simulated exploiting a high-level model of the targeted microprocessor. Additionally, a step-by-step schema is implemented in order to increase the efficiency on the generation process.

B. Environment architecture

Figure 1 reports the conceptual view of the proposed methodology. The whole approach is divided in two main parts: the first one is performed by hand, and is related to the general framework configuration; the second one is devoted to automatically generate test programs and it is based on a multi-run process where an evolutionary algorithm is exploited for the automatic generation of assembly programs.

The main sub-phases of the first part are described in the following:

- *Module selection*: the target module is selected and identified by the validation engineer.
- *Metrics selection*: the high-level metric(s) to be maximized are chosen taking into account the description style of the processor model.
- *Environment settings*: the validation engineer must fulfill a series of requirements in order to setup the generation environment: com-

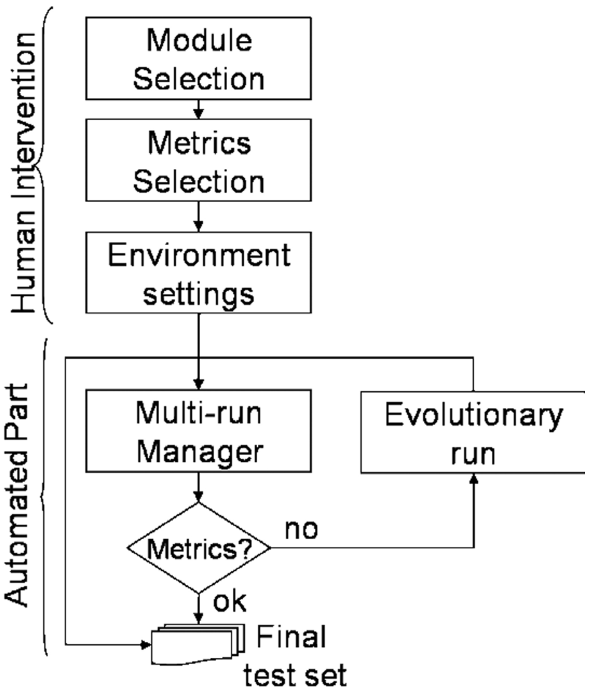


Figure 1. Conceptual view of the proposed approach

pile the *constraint library*, describing the processor assembly syntax; chose the set of instructions among the described ones to be used in every generation run; decide the parameters characterizing every generation run, such as the length of the programs, the programs size, etc; define the termination conditions for both every single run, and the overall multi-run experiment.

The second part of the generation process is divided into a series of automated steps (i.e., evolutionary runs): in every run, a series of test programs are generated, evaluated and eventually improved; at the end, only the best program is preserved in the final test set. It should be noted that since herein the proposed approach targets a multithread processor core, the programs generated are composed by different pieces of assembly code that compile independent threads. In the automated part, the generation process is handled by the *Multi-run Manager* that, using the configuration information previously defined by the test engineer, launches a series of evolutionary runs exploiting the evolutionary algorithm until satisfactory figures are obtained with respect to the chosen metrics, or a given ending condition is matched. Thus, the main goal of the multi-run manager is to activate with the right parameters every evolutionary run, and to collect the final set of test programs.

The rationale for this step-by-step methodology is that in general it is not possible to satisfactorily solve the addressed problem within a single run, i.e., by generating only a single test program with a single set of parameter values, especially when complex modules are targeted. Moreover, this multi-run adaptive execution allows the evolutionary algorithm to better and faster focus on the targeted module, thus reducing the generation time, since long test programs must be only evaluated during the first run.

Once the test engineer provides configuration information to the multi-run manager, the automatic process is ready to start. The first evolutionary run is called a *winnowing run*, since the test program left by this iteration must be able to cover those parts of the targeted module that are relatively easy to excite using for example, long random generated programs. From the second iteration on, the multi-run manager progressively diminishes the dimension of the test programs to be generated, whereas varies the set of instructions, trying to excite specific and not already covered parts of the design. It is important to notice that at the end of each run the covered items are actually marked as covered, so that the generation process focuses only on those parts that are not yet excited, speeding up simulations. Each run is terminated when the evolutionary algorithm reaches a condition in which it is considered that it is not possible to further improve the targeted metrics, or when a maximum number of programs have been simulated.

In every run, the delivered test program, (i.e., the best one) is picked up by the multi-run manager and preserved in the final test set. In addition, the multi-run manager reconfigures the simulation environment removing the covered items, changing the settings, and starting a new evolutionary run (if the termination condition is not yet reached).

It should be noted that once the framework has been set up for a specific module it can be easily used to tackle other modules by only modifying the feedback value provided to the evolutionary engine, and by tuning the parameters for the multi-run execution.

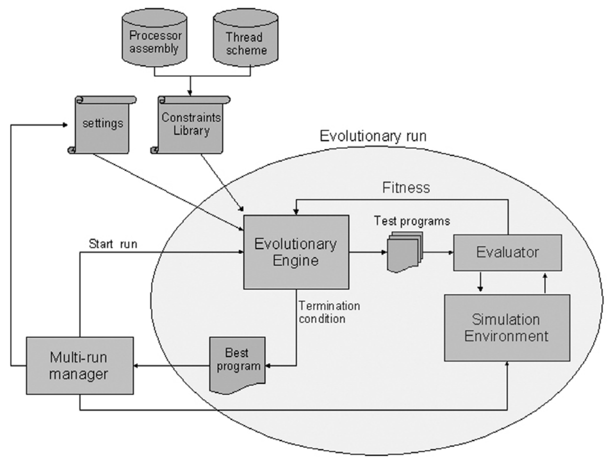


Figure 2. Multi-run architecture

C. Generation loop

Figure 2 presents in more details the environment defined to implement the proposed approach, highlighting the elements that intervene in the generation loop.

Every run launches the evolutionary engine, which plays the role of the automatic test program generator. The engine generates test programs starting from a set of constraints that describe the syntactic appearance of the test programs. This information is organized in the *constraints library*. This library lists all syntactically correct instructions and all the valid operands for each one. Since we are in a multithreaded environment, also the thread scheme and the number of threads to be generated must be encoded in the constraints library. Based on both the constraints library and the settings provided by the multi-run manager, the evolutionary engine generates syntactically correct test programs; each of them is evaluated by the *Evaluator* using a simulation environment, and a measure of how well it solves the given problem is fed back to the evolutionary engine. The evolutionary engine allows to simultaneously maximize all the selected metrics with a defined priority.

D. Evolutionary engine and genetic operators

A preliminary version of the evolutionary tool used here is described in [3]: this evolutionary algorithm exploits a suitable internal representation for programs, based on graphs, and basically works as follows: an initial set of random test programs (called *individuals*) is created and evaluated with respect to an evaluation function; then, the best fitted programs are chosen to be improved by the application of one of the so called genetic operators: mutation and crossover. This process iterates until a stopping condition is reached. The approach has been able to successfully deal with the generation of test programs tackling different goals: processor testing [11], verification [3] and post-silicon verification [12]; however, in order to deal with multithreaded processors, it is necessary to introduce special features that allow the evolutionary optimizer to effectively manipulate and evolve test programs for multithreaded processor cores. In this paper, we present an improved version of the evolutionary algorithm able to effectively generate multithreaded test programs.

The new version of the algorithm generates, manipulates and improves multithreaded test programs; every single program is internally represented as the union of *N* different programs, being N the number of processor threads. Every one of these programs must be enriched and improved contemporarily in order to reach a useful final test program that exploits the actual interdependency among the different instructions on every single thread. In the following, a complete description of the new features of the evolutionary engine that directly regard the automatic generation of multithreaded test programs is reported.

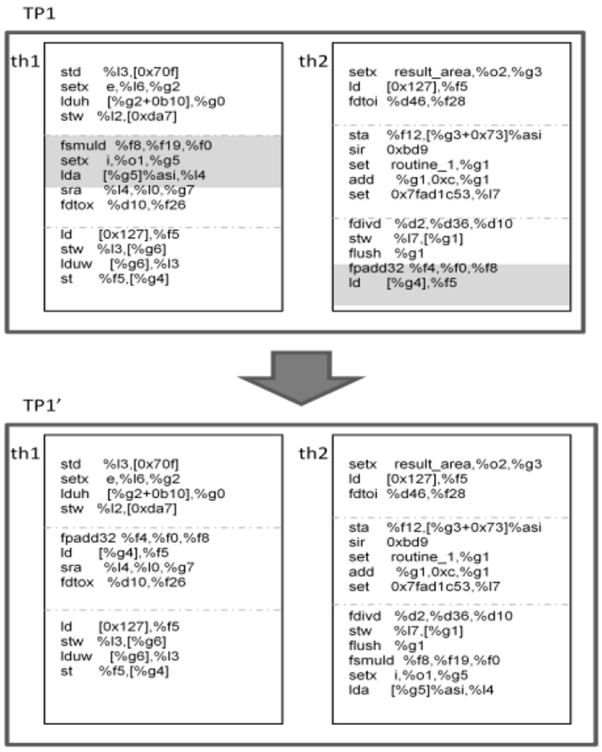
E. New multithread-oriented features

Test programs for SMT processor cores mainly differ from predecessor ones in the composition of every test program: in the SMT case, every single test program is composed of a set of N threads, whereas the effectiveness of the test program is evaluated considering the parallel execution of all the threads. Conceptually, every single thread executes an independent program whereas sharing hardware resources with other threads. However, from the generation point of view, it is important to improve the single thread, as well as to maximize the interaction among the parallel executions of the different threads.

In the version of the tool oriented to SMT processors which is proposed here, the evolutionary engine is able to act separately on both the whole test program and the single thread by exploiting different genetic operators.

The evolutionary tool is able first to create random (but correct) multithreaded test programs and then to independently evolve them by applying suitable thread-oriented evolutionary operators.

In other words, new genetic operators work specifically on independent threads by inserting, deleting and mutating instructions. In addition, the evolutionary engine also exploits an inter-thread crossover operator, allowing code recombination between threads, thus permitting the evolutionary optimizer to exchange valid pieces of code between the different threads of a single program, as well as between threads in different test programs. In this way, the method leverages easy exchange of outstanding pieces of code between different threads.



TP1, th1:
```
std     %l3,[0x70f]
setx    e,%l6,%g2
lduh    [%g2+0b10],%g0
stw   %l2,[0xda7]

fsmuld  %f8,%f19,%f0
setx    i,%o1,%g5
lda     [%g5]%asi,%l4
sra     %l4,%l0,%g7
fdtox   %d10,%f26

ld      [0x127],%f5
stw   %l3,[%g6]
lduw    [%g6],%l3
st      %f5,[%g4]
```

TP1, th2:
```
setx   result_area,%o2,%g3
ld     [0x127],%f5
fdtoi  %d46,%f28

sta    %f12,[%g3+0x73]%asi
sir    0xbd9
set    routine_1,%g1
add     %g1,0xc,%g1
set    0x7fad1c53,%l7

fdivd   %d2,%d36,%d10
stw     %l7,[%g1]
flush   %g1
fpadd32 %f4,%f0,%f8
ld      [%g4],%f5
```

TP1', th1:
```
std     %l3,[0x70f]
setx    e,%l6,%g2
lduh    [%g2+0b10],%g0
stw   %l2,[0xda7]

fpadd32 %f4,%f0,%f8
ld      [%g4],%f5
sra     %l4,%l0,%g7
fdtox   %d10,%f26

ld      [0x127],%f5
stw   %l3,[%g6]
lduw    [%g6],%l3
st      %f5,[%g4]
```

TP1', th2:
```
setx   result_area,%o2,%g3
ld     [0x127],%f5
fdtoi  %d46,%f28

sta    %f12,[%g3+0x73]%asi
sir    0xbd9
set    routine_1,%g1
add     %g1,0xc,%g1
set    0x7fad1c53,%l7

fdivd   %d2,%d36,%d10
stw     %l7,[%g1]
flush   %g1
fsmuld  %f8,%f19,%f0
setx    i,%o1,%g5
lda     [%g5]%asi,%l4
```

Figure 3. Thread-oriented crossover

Figure 3 graphically represents the application of the crossover operator between two threads of the same program.

Let us suppose that *TP1* (in the upper part of the figure) represents the selected program that undergoes the thread-oriented crossover. *TP1* is composed of two threads (*th 1* and *th 2*), each one of them containing three sections separated by dashed lines. Let us also suppose that in both threads of *TP1* the grayed lines demark the pieces of code selected to be exchanged during the application of the mentioned crossover inter-thread operator. The resulting individual (*TP1'*) is depicted in the bottom part of the figure.

It is interesting to highlight that the evolutionary optimizer is able to safely exchange pieces of code between threads since every test program and in particular every thread is internally represented as an acyclic graph. The graph-based representation of test programs allows the evolutionary tool to copy, transfer, and even eliminate code segments without losing important details regarding the program structure for pieces of code such as subroutines and jumps.

On the other hand, the whole test program is also eventually modified by the application of genetic operators that, as in the previous versions of the evolutionary optimizer, elaborate on the complete program structure.

In order to measure the interaction between threads, the evaluation of every program is performed on the whole multithreaded test program, and the feedback information gathered are used by the evolutionary tool to adapt the generation process.

4. CASE STUDY

We evaluate our approach on two processor cores that exploit the multithreading processor paradigm. In the first case, we deal with the *pick unit* of the OpenSPARC™ T2 processor. Considering the style of the processor description, in this case the coverage metric we selected is *condition coverage*, which measures which possible values each conditional expression takes during simulation, tracking the sensitized vectors (those that change the result of the overall expression) for each conditional expression.

In the second experiment, we exploit the proposed approach for the generation of validation programs for the *switch logic unit* of the OpenSPARC™ T1. In this case, taking in consideration the style of the available description of the processor model, the metric selected is *toggle coverage*, that reports the number of nodes or storage elements that toggle at least once from 0 to 1 and at least once from 1 to 0.

In both cases, the generation environment was developed resorting to the multithread-oriented version of the tool named µGP^3 [26]. The tool counts about 42,000 lines of C++ code.

The multi-run manager, described previously, was developed in Perl as well as some simulation scripts.

The constraint library is written in XML language, counts about 4,500 lines and describes the SPARC V9 instruction set architecture. The constraints are divided in sections, and for each one the number of different instances in the test program generated must be set. In each section we defined some macros, each of them representing a valid instruction or set of instructions. In this case we defined two sections: the first represents a thread (we set up the tool to generate eight different segments of this type) and the second represents subroutines (we set up the tool to generate ten different segments of this type). Writing the constraint library required to the test engineer less than two working days. Clearly, the generation environment settings differ depending on the faced processor core: in the case of T2 processor core, for example, every single generated test program is composed of eight threads, whereas only four are required in the T1 case.

Regarding the multi-run mechanism, we set to 60,000 clock cycles the maximum duration of the first run (excluding privileged instructions), and to 20,000 clock cycles the maximum duration for all the other runs (with all the possible instructions).

In the following, we briefly describe both processor cores as well as the tackled modules. In both the faced experiments, as highlighted by the experimental results, the proposed approach reached outstanding results compared to a pure random-based approach. It is important to note that the tackled modules are specific of the multithreaded architecture, and similar modules can be found in other processors, too. All the reported experiments have been performed on a couple of HP xw 8600 Workstations with 8 GB of RAM running Linux. The simulation environment used is based on the verification suite provided by Sun Microsystems for both the considered processor cores.

A. OpenSPARC™ T2

OpenSPARC™ T2 is a 64 bit open-source microprocessor developed by SUN Microsystems and freely available at [8]. OpenSPARC™ T2 implements the chip multithreading (CMT) paradigm and counts on eight SPARC physical processor cores. Each SPARC core is able to support eight *strands* (i.e., a virtual processor able to run a software thread). Internally, every processor core divides its eight strands in two groups of four strands each. In order to increase the throughput, each SPARC core contains two integer execution pipelines (one per strand group), one floating-point execution pipeline, and one memory pipeline (shared by the eight threads).

The OpenSPARC™ T2 can execute up to 64 simultaneous threads, but in every SPARC core at most two strands can be active at the same clock cycle. The strands are switched at each clock cycle choosing the least recently issued available strand within each group. A strand is marked as unavailable if it encounters a long latency event, such as a cache miss, a floating point operation, etc.; no instructions are picked from an inactive strand until the long-latency event is terminated. The execution of the others strands continues normally while the long latency operation is executed.

Each SPARC processor core has a 16 KB, 8-way associative instruction cache, an 8 Kbytes, 4-way associative data cache, an 64-entry fully associative instruction TLB, and an 128-entry fully associative data TLB, that are shared by the eight strands. The eight processor cores are connected through a crossbar to an on-chip unified 4 Mbytes, 16-way associative L2 cache (64-byte lines) banked eight ways. The L2 cache connects to four on-chip DRAM controllers.

The main components of the OpenSPARC™ T2 core are:

- The Instruction Fetch Unit (IFU), which distributes instructions to the rest of the core, while updating both the program counters (PC) for each thread, and the instruction caches. The IFU is composed of three subunits: the fetch unit, the pick unit, and the decode unit.
- The Execution Unit (EXU), which executes almost all the integer operations, with the exception of the multiply and division instructions.
- The Load and Store Unit (LSU), which manages memory accesses between the processor core and the data caches (L1 and L2).
- The Cache Crossbar (CCX), which connects the 8 SPARC cores to the 8 banks of the L2 cache. A maximum of 8 operations on the memory can be processed simultaneously.
- The Floating-Point and Graphics Unit (FGU), which implements the SPARC V9 floating-point instruction set, integer multiply, division, and population count (POPC) instructions.
- The Trap Logic Unit (TLU), which handles exceptions, trap requests, and traps for the SPARC core. A trap is a vectored transfer of control to supervisor software through a trap table. The TLU maintains the processor state related to traps as well as the Program Counter (PC) and the Next Program Counter (NPC).
- The Memory Management Unit (MMU), which handles all memory accesses validating the permission to access to the requested address.

B. Module and metric selection on the T2 processor

We evaluated our approach on the pick unit of the OpenSPARC™ T2 processor, which is the module mainly responsible for the management of the different threads to be executed in parallel. In addition, this module is distinctive of a multithreaded processor and is not present in other types of processors. Targeting the pick unit is particularly critical, since a suitable test program must properly combine the execution of several threads, making this unit the most suitable part of the OpenSPARC™ T2 where to evaluate the approach. The module is located within the instruction fetch unit (IFU) and at each cycle is in charge of selecting two threads out of eight for the execution. Its correct behavior is essential in order to guarantee both proper functioning and maximum performance of the processor.

The threads are organized in two thread groups of four threads. The pick unit contains a state machine for each thread with two possible states: READY or WAIT. A thread can be picked if it is in a READY state. A thread is in a WAIT state if any wait condition exists for the thread and it remains in a WAIT state until the condition or the conditions that caused the transition to WAIT are resolved or the thread is flushed. The details about the functioning of the pick unit can be found in [8].

The RT level description of the pick unit is written in Verilog and counts 3,688 lines. As mentioned before, at each clock cycle the pick unit needs to check a number of thread conditions. The RT-level description is thus organized in a significant number of signal conditional assignments that check all the possible WAIT / READY conditions. For this reason, the obvious choice for the main metric to drive the generation process is Condition Coverage. This metric is very meaningful for the pick unit; in fact, as mentioned before, a thread could transition to the wait state due to several conditions and we need to ensure that all of them are covered by the test programs in order to fully excite the unit. Moreover, in the T2 pick unit description other coverage metrics like Statement Coverage, Branch Coverage, FSM coverage have a number of items to be monitored negligible, or null (as mentioned before, the RT-Level is organized with a significant amount of signal conditional assignments that are always all executed). On the other hand, the number of conditions is 29,394.

C. Experimental results

Table I shows the results obtained on the pick unit when applying the multi-run proposed methodology. The table reports for every generated test program the percentage of the attained condition coverage, the code size, and the duration (in terms of clock cycles) at the end of every evolutionary run. The condition coverage reported is the cumulative value obtained by the set of programs collected until the referred run and not by the single program; the code size and the test program duration, on the other hand, are related to the best test program produced in each run.

Table I. Results obtained on the OpenSPARC™ T2 processor.

Run #	Condition Coverage [%]	Code size [KB]	Test program duration [CC]
1	90.40	53.376	25,165
2	93.79	11.168	13,680
3	95.90	10.680	9,568
4	96.37	6.976	4,712

The methodology is able to achieve 96.37% of condition coverage by executing the complete test set, which lasts 53,125 clock cycles. The amount of computational time required for generating the complete test set is about 5.5 days: 2.5 days for the first winnowing run and about 1 day for each of the following runs. The required time for performing the simulation of each generated program corresponds to about 6 to 8 minutes for the first winnowing run and about 2 to 3 minutes for the following runs, since these are shorter programs. The generation of the complete test set requires the simulation of about 2,000 programs; it is interesting to note that the additional computational effort required by the evolutionary engine is negligible.

In order to further assess the proposed methodology we performed a comparison against a purely random method. We randomly generated several test programs counting in average about the same number of instructions of the test programs generated in the first run of the proposed approach. Every random program is evaluated, and those programs able to improve the addressed condition coverage in the pick unit are retained in the final test set. Similarly to the proposed approach, 10 different experiments were simulated considering about 2,000 test programs in every launch. In this case, however, the generation process for every run lasts about 7 days. It is interesting to note that the generation time is higher since all the randomly generated programs count with a similar length of the test programs generated in the first run of the proposed approach, avoiding the gradual decrease in the length of the test programs presented in the proposed approach. Table II reports a comparison between test sets of the best run obtained exploiting the purely random approach and of the proposed approach, highlighting the cumulative condition coverage obtained by the test set and the cumulative number of clock cycles required to execute the complete set.

Table II. Comparison between the random and the proposed approach.

Strategy	Number of programs	Condition Coverage	Test set duration
		[%]	[CC]
Best random	11	91.04	926,400
Proposed	4	96.37	53,125

It can be seen from table II that the proposed approach is able to achieve a significantly higher condition coverage figure than the purely random one, while the test length of the set of test programs is significantly lower. This difference can partly be attributed to the ability of the evolutionary engine of guiding the search towards the most promising areas, and partly to the multi-run mechanism, which allows boosting up the coverage at every run while reducing the program length (with respect to the first run).

D. Covered Corner Cases

An analysis performed on the random resistant conditions left uncovered by the random experiments (while covered by the evolutionary-based approach) shows that the random approach is not able to cover some interesting corner case, such as the store buffer full wait condition. The load and store unit has, for each thread, a buffer with eight entries containing all outstanding store instructions. The pick logic, on its side, maintains a four bit speculative store counter per thread, which is incremented each time a store is picked. When this counter reaches the value eight and a new valid store is detected, the pick unit transitions the thread to the WAIT state. This condition is not easily reachable using random approaches: in fact, it needs a very specific sequence of memory operations. On the other side, the proposed approach is able to excite this condition, mainly due to the generation process, where the best test programs are combined and/or manipulated.

E. OpenSPARC^{TM} T1

OpenSPARC^{TM} T1 is a 64 bit open-source microprocessor core developed by SUN Microsystems and freely available at [8]. The OpenSPARC^{TM} T1 implements the 64-bit SPARC V9 architecture. If instantiated in the full version, the processor contains eight SPARC processor cores, able to support four threads. Each SPARC core has an instruction cache, a data cache, and a fully associative instruction and data translation lookaside buffers (TLB). The eight SPARC cores are connected through a crossbar to an on-chip unified level 2 cache (L2-cache).

Each SPARC core has single issue and six stages pipeline. The six stages are: *Fetch*, *Thread Selection*, *Decode*, *Execute*, *Memory* and *Write Back*. Additionally, every SPARC has the following units:

- Instruction fetch unit (IFU) includes the following pipeline stages – fetch, thread selection, and decode. The IFU also includes an instruction cache complex.
- Execution unit (EXU) includes the execute stage of the pipeline.
- Load/store unit (LSU) includes memory and writeback stages, and a data cache complex.
- Trap logic unit (TLU) includes trap logic and trap program counters.
- Stream processing unit (SPU) is used for modular arithmetic functions for crypto.
- Memory management unit (MMU).
- Floating-point front-end unit (FFU) interfaces to the FPU.

Every single thread, on the other hand, is supported by a register file (with eight register windows), with most of the general purpose and special purpose registers replicated per thread. The four threads share the instruction, the data caches, and the TLBs. Each instruction cache is 16 Kbytes with a 32-byte line size. The data caches are write through, 8 Kbytes, and have a 16-byte line size.

F. Module and metric selection on the T1 processor

The OpenSPARCTM T1 processor core is described at RT-level in Verilog; in this case, the chosen module is the *switch logic unit* that belongs to the instruction fetch unit (IFU) and is mainly responsible for managing the four threads that are executed inside the processor core by performing the processor thread selection policy. The thread selection policy is as follows: the available threads are selected at every clock cycle giving priority to the least recently executed thread. The threads become unavailable due to the long latency operations like loads, branch, MUL, and DIV, as well as to the pipeline stalls like cache misses, traps, and resource conflicts. The loads are speculated as cache hits, and the thread is switched-in with lower priority.

The switch logic unit is described at RT-level in four different files and counts 1,428 lines, and as in the first experiment, this is a distinctive module in a multithreaded processor core, making this module crucial to the correct behavior of the processor core. The selected coverage metric regards with toggling the 7,387 store elements presented in the switch logic module. Once again, considering the processor description, this is the most important coverage metric available in the RT-level model, other verification metrics are negligible or not available for the considered module.

G. Experimental results

Table III shows the results gathered on the switch logic unit of the OpenSPARCTM T1 processor core. Every line of Table III reports the results obtained by the best programs obtained in every experiment step (*Run* #). In particular, the toggle coverage reported is the accumulated value of the test set up to the considered run. Additionally, for every program the code size, and the duration (in terms of clock cycles) is also reported.

The complete test set lasts 19,615 clock cycles to be executed, requiring about 96KB. The complete test set achieves about 86% on the targeted metric. The complete experiment required about 4 days on the described workstation, requiring the simulation of about 4,000 programs. Compared with the pure random strategy, the proposed approach improves the coverage results in about 8%.

Table III. Results obtained on the OpenSPARCTM T1 processor.

Run #	Toggle Coverage	Code size	Test program duration
	[%]	**[KB]**	**[CC]**
1	75.12	9.03	3,672
2	80.92	25.61	4,410
3	82.37	27.07	4,067
4	84.25	26.18	3,810
5	85.67	8.89	3,656

5. CONCLUSIONS

In this paper a semi-automatic methodology able to generate effective test programs for the validation of complex modules in multithreaded processors has been presented. The methodology requires a limited and low-skilled manual intervention, basically to describe the assembly syntax of the targeted processor core, and can easily focus on specific critical targets (thus reducing the required computational effort), while still maintaining the ability of generating test programs for the whole processor. Despite the complexity of generating effective multithreaded test programs, the results gathered on a couple of real complex designs, the OpenSPARCTM T1, and OpenSPARCTM T2 processor cores, show the effectiveness and the scalability of the proposed method, in terms of both achieved coverage figures, and ability to cover very specific corner cases. The method is also shown to significantly outperform a random approach in terms of achieved results when a comparable number of test programs is evaluated.

REFERENCES

[1] Tasiran S., Keutzer K., “Coverage metrics for functional validation of hardware designs”, *IEEE Design & Test of Computers,* vol.18, no.4, Jul/Aug 2001, pp.36-45.

[2] Pradhan Dhiraj K., Harris Ian G., *Practical Design Verification*, Cambridge University Press. June 2009, ISBN: 9780521859721.

[3] Corno F., Sanchez E., Sonza Reorda M., Squillero G., “Automatic test program generation: a case study”, *IEEE Design & Test of Computers*, vol.21, no.2, Mar-Apr 2004, pp. 102-109.

[4] Thatte S., Abraham J., “Test Generation for Microprocessors”, *IEEE Transactions on Computers*, Vol. C-29, 1980, pp 429-441.

[5] Kranitis N., Paschalis A., Gizopoulos D., Xenoulis G., “Software-based self-testing of embedded processors”, *IEEE Transactions on Computers*, Vol 54, issue 4, April 2005, pp 461-475.

[6] Ravotto D., Sanchez E., Sonza Reorda M., Squillero G., “On the Generation of Test Programs for Chip Multi-thread Computer Architectures”, *IEEE Test Conference, 2008. ITC 2008*, ISSN: 1089-3539.

[7] Ravotto D., Sanchez E., Sonza Reorda M., Squillero G., “Design validation of multithreaded architectures using concurrent threads evolution”, *IEEE 22nd Annual Symposium on Integrated Circuits and System Design, 2009*. ISBN: 978-1-60558-705-9.

[8] OpenSparcTM processor cores: http://www.opensparc.net/

[9] Kurshan R., *Computer-Aided Verification of Coordinating Processes*, Princeton University Press, 1994, pp 272.

[10] Goodenough J. B., Gerhart S. L., “Toward a Theory of

Testing: Data Selection Criteria", *Current trends in programming methodology*, vol. 2 R. T. Yeh, Ed. Prentice-Hall. Englewood Cliffs, NJ, 1977, pp. 44 – 79.

[11] Sanchez, E., Sonza Reorda M., Squillero, G., "Test Program Generation From High-level Microprocessor Descriptions", *Test and validation of hardware/software systems starting from system-level descriptions*, Edited by M. Sonza Reorda, M. Violante, Z. Peng, Springer publisher, 2005, 179 p, ISBN: 1-85233-899-7, pp. 83-106.

[12] Lindsay, W., Sanchez, E., Sonza Reorda, M., Squillero, G., "Automatic test programs generation driven by internal performance counters", *MTV'04: 5th International Workshop on Microprocessor Test and Verification*, pp. 8-13.

[13] Harman N.A., "Verifying a Simple Pipelined Microprocessor Using Maude", *Lecture Notes in Computer Science 2267*, Springer-Verlag, 2001, pp. 128-142.

[14] Van Campenhout D., Mudge T.N., Hayes J.P., "High-Level Test Generation for Design Verification of Pipelined Microprocessors", *36th Design Automation Conference* (DAC 99), 1999, ACM Press, pp. 185-188.

[15] Velev, M.N., Bryant R.E., "Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exception, and Branch Prediction", *37th Design Automation Conference* (DAC 00), 2000, ACM Press, pp. 112-117.

[16] Jun Y., Pixley, C., Aziz A., Albin K., "A framework for constrained functional verification", *International Conference on Computer Aided Design*, 2003. ICCAD-2003., vol., no., pp. 142-145.

[17] Adir A., Almog E., Fournier L., Marcus E., Rimon M., Vinov M., Ziv A., "Genesys-Pro: innovations in test program generation for functional processor verification", *IEEE Design & Test of Computers*, vol.21, no.2, Mar-Apr 2004, pp. 84-93.

[18] Behm M., Ludden J., Lichtenstein Y., Rimon M., Vinov M., "Industrial experience with test generation languages for processor verification", *41st Design Automation Conference* (DAC 04), 2004, ACM Press., pp. 36-40.

[19] Emek R., Jaeger I., Naveh Y., Bergman G., Aloni G., Katz Y., Farkash M., Dozoretz I., Goldin A., "X-Gen: a random test-case generator for systems and SoCs", *Seventh IEEE International High-Level Design Validation and Test Workshop*, 2002., pp. 145-150.

[20] Guzey O., Wang L.-C., "Coverage-directed test generation through automatic constraint extraction", *IEEE International High Level Design Validation and Test Workshop*, 2007, HLVDT 2007, pp.151-158.

[21] Saxena N., Abraham J.A., Saha A., "Causality based generation of directed test cases", *Asia and South Pacific Design Automation Conference*, 2000, pp. 503-508.

[22] Mishra P., Dutt N., "Functional coverage driven test generation for validation of pipelined processors", *Design, Automation and Test in Europe*, 2005, Vol. 2, pp. 678-683.

[23] Benjamin M., Geist D., Hartman A., Wolfsthal Y., Mas G., Smeets R., "A study in coverage-driven test generation", *36th Design Automation Conference* (DAC 99), 1999, pp.970-975.

[24] Heon-Mo Koo, Prabhat Mishra, "Functional Test Generation using Property Decompositions for Validation of Pipelined Processors", *IEEE Design, Automation and Test in Europe, 2006. DATE '06*, pp. 1-6.

[25] Apostolakis A., Psarakis M., Gizopoulos D., Paschalis A., Parulkar I., "Exploiting Thread-Level Parallelism in Functional Self-Testing of CMT Processors", *14th IEEE European Test Symposium*, 2009, pp. 33 – 38. http://ugp3.sourceforge.net/