Computing (2011) 93:103–120
DOI 10.1007/s00607-011-0157-9

Artificial evolution in computer aided design: from the optimization of parameters to the creation of assembly programs

Giovanni Squillero

Received: 26 September 2011 / Accepted: 27 September 2011 / Published online: 11 October 2011

Abstract Evolutionary computation has been little, but steadily, used in the CAD community during the past 20 years. Nowadays, due to their overwhelming complexity, significant steps in the validation of microprocessors must be performed on silicon, i.e., running experiments on physical devices after tape-out. The scenario created new space for innovative heuristics. This paper shows a methodology based on an evolutionary algorithm that can be used to devise assembly programs suitable for a range of on-silicon activities. The paper describes how to take into account complex hardware characteristics and architectural details. The experimental evaluation performed on two high-end Intel microprocessors demonstrates the potentiality of this line of research.

Keywords Evolutionary computation · Microprocessors · Post-silicon verification · Speed paths

Mathematics Subject Classification (2000) 00A06

1 Introduction

The 40 years since the appearance of the Intel 4004 deeply changed how microprocessors are designed. Today, essential steps in the validation process are performed relying on physical dices, analyzing the actual behavior under appropriate stimuli. For example, the typical design flow goes through several iterations of frequency pushes prior to final volume production, where the behavior of prototypical devices is checked at increasing operating frequencies, and, as soon as a speed grade is reached with good timing yield, the target is set for an even higher speed grade [1].

G. Squillero (✉)
Politecnico di Torino, Turin, Italy
e-mail: giovanni.squillero@polito.it
URL: http://www.cad.polito.it/

Indeed, most of the timing verification must be performed on silicon. In nanometer processes it is not feasible to consider simultaneously all factors that contribute to the timing behavior during the pre-silicon analysis, and even the analysis algorithms themselves are often approximated or oversimplified [2,3].

This paper describes a methodology based on an evolutionary algorithm that can be used to devise assembly programs suitable for timing verification or other on-silicon activities such as timing verification. The approach exploits the feedback from the microprocessor under examination and does not rely on information about its microarchitecture, nor does it require design-for-debug features.

The first two sections of the paper quickly summarize evolutionary computation and its practical use in the CAD community. Then, some issues about speed debug are introduced, and the proposed approach detailed. An experimental evaluation concludes the paper.

2 Theory of evolutionary computation

Evolution is the biological theory that animals and plants have their origin in other types, and that the distinguishable differences are due to modifications in successive generations [4]. Natural evolution is not a random process. On the contrary, it is based on random variations, but some are rejected while others preserved according to objective evaluations. Only changes that are beneficial to the individuals are likely to spread into subsequent generations. Darwin called this principle "natural selection" [5], a quite simple process where random variations "afford materials".

When natural selection causes variations to be accumulated in one specific direction the result strikingly resembles an optimization process. This process only requires to *assess* the effect of random changes, not the ability to *design* intelligent modifications. Several researchers, independently, tried to replicate such a characteristic to solve difficult problems more efficiently.

Evolutionary computation (EC) is the offshoot of computer science focusing on algorithms loosely inspired by the theory of evolution. The definition is deliberately vague since the boundaries of the field are not, and cannot be, sharply defined. EC is a branch of *computational intelligence*, and it is also included into the broad framework of *bio-inspired heuristics*.

EC does not have a single recognizable origin. Some scholars identify its starting point in 1950, when Alan Turing drew attention to the similarities between learning and evolution [6]. Others pointed out the inspiring ideas that appeared later in the decade, despite the fact that the lack of computational power impaired their diffusion in the broader scientific community [7]. More commonly, the birth of EC is set in the 1960s with the appearance of three independent research lines: John Holland's *genetic algorithms* (GA) [8]; Lawrence Fogel's *evolutionary programming* (EP) [9]; Ingo Rechenberg's and Hans-Paul Schwefel's *evolution strategies* (ES) [10,11]. These three paradigms monopolized the field until the 1990s, when John Koza entered the arena with *genetic programming* (GP) [12]. These four, together with the many other different EC paradigms that have been proposed over the years, can be grouped under the term *evolutionary algorithms* (EAs).

In EAs[1] a single candidate solution is termed *individual*; the set of all candidate solutions that exists at a particular time is called *population*. Evolution proceeds through discrete steps called *generations*. In each of them, the population is first expanded and then collapsed, mimicking the processes of breeding and struggling for survival. Some evolutionary algorithms do not store a collection of distinct individuals, and evolution is depicted through the variation of the statistical parameters that describe the population.

Most of the jargon of evolutionary computation mimics the precise terminology of biology. The ability of an individual to solve the target problem is measured by the *fitness function*, which influences the likelihood of a solution to propagate its characteristics to the next generation. In some approaches individuals may die of old age, while in other they remain in the population until replaced by fitter ones.

The word *genome* denotes the whole genetic material of the organism, although its actual implementation strongly differs from one approach to another. The *gene* is the functional unit of inheritance, or, operatively, the smallest fragment of the genome that may be modified in the evolution process. Genes are positioned in the genome at specific positions called *loci*. The alternative genes that may occur at a given locus are called *alleles*.

Biologists need to distinguish between the *genotype* and the *phenotype*: the former is all the genetic constitution of an organism; the latter is the set of observable properties that are produced by the interaction between the genotype and the environment. In many implementations, EC practitioners do not require such a precise distinction. The single numerical value representing the fitness of an individual is sometimes assimilated to its phenotype.

To generate the offspring, EAs implement sexual and asexual reproduction. The former is named *recombination*; it involves two or more participants, and implies the possibility for the offspring to inherit different characteristics from different parents. When recombination is achieved through an exchange of genetic material between the parents, it often takes the name of *crossover*. Asexual reproduction may be named *replication*, to indicate that a copy of an individual is created, or, more commonly, *mutation*, to stress that the copy is not exact. All operators exploited during reproduction can be cumulatively called *evolutionary operators*, or *genetic operators* because they act at the genotypical level. Almost no evolutionary algorithm takes gender into account; hence, individuals do not have distinct reproductive roles.

3 Practical use of evolutionary computation

An EA performs better than a pure random approach. This rather simple consideration is probably the main reason why EAs are sometimes exploited outside the EC community. They provides an effective methodology for trying random modifications, where no preconceived idea about the optimal solution is required. Being based on a population, EAs are more robust than pure hill climbing. Both small and large modifications

[1] This short introduction tries to emphasize the common aspects in EC, unifying different paradigms under the same umbrella. A comprehensive survey that highlights the distinctive traits is [48].

are possible, but with different probabilities. Sexual recombination allows merging useful characteristics from different solutions, exploring efficiently the search space. Furthermore, EAs are quite simple to set up, and require no human intervention when running. They are inherently parallel, and a nearly-linear speed-up may be easily achieved on *multiple instruction/multiple data* (MIMD) architectures. Finally, it's easy to trade-off between computational resources and quality of the results.

Unfortunately, several hidden and rather obscure details may significantly impair EAs' efficacy. This may explain the relative slow acceptance, compared to other bio-inspired heuristics, such as *simulate annealing* (SA).

3.1 Common pitfalls and open issues

Several details may impair the efficacy and performance of an EA. First, failing to define an appropriate fitness function may easily spoil the evolution process. The fitness function must be able to discriminate between almost all individuals. That is, different solutions must receive different fitness values, because small variations must trigger differential survival in the artificial population.

Moreover, since the fitness is a synthetic measure, it is necessary to carefully consider which information is removed. For instance, if an EA is used to implement an *automatic test-pattern generator* (ATPG) the fitness cannot be simply defined as the attained *fault coverage* (FC%), because two individuals detecting 10% of the faults would be considered absolutely equivalent. However, if the first one is able to detect some random-resistant, hard-to-test faults while the second is an uninteresting random pattern, only the characteristics of the first one must be preserved in subsequent generations.

The choice of the representation for the individual is also important. Individuals must encode some structural information about the desired solution. However, too much information may bias the process preventing the discovery of optimal solutions. On the contrary, too few information may increase the dimension of the search space slowing down the evolution process. In some case, the encoding follows naturally from the problem. For instance, a fixed number of real parameters when optimizing filter coefficients, or a variable number of bit vectors when implementing a sequential ATPG. Conversely, different problems, like the design of circuits or the creation of assembly-language programs, require less obvious encoding. The individual representation also influences the behavior of genetic operators. Evolution calls for the offspring to preserve most of the parents' characteristics, however they are defined by the current application.

Besides, the main problem with EA is definitely *premature convergence*. Generation after generation individuals in the population tend to be all alike, the population converges to a single point in the search space, all recombination operators become quite ineffective and the evolution process almost stops. In such a condition, an EA behaves like an overloaded, inefficient random-mutation hillclimber.

Premature convergence is an endemic problem of EC. In nature, "natural selection, also, leads to divergence of character; for more living beings can be supported on the same area the more they diverge in structure, habits, and constitution" [5]. However, in artificial evolution there is no explicit environment since its effects are modeled through the fitness function.Consequently, the *divergence of character*, i.e., one of the pillar of the Darwinian theory of evolution, is completely missing.

Several contributions describe methodologies to tackle this problem [13–15]. Authors proposed to preserve the diversity of the population by aging individuals, limiting interactions or other quite complex mechanisms. However, none of these artificial mechanisms can be called a panacea, and practitioners still need to tackle premature convergence on a problem-specific basis.

3.2 Evolutionary algorithms in computer aided design

The first works exploiting EC techniques in the CAD field appeared in the early 1980s. In the beginning, the term GA was often used to denote a generic EA, and artificial evolution was mainly used to optimize numeric coefficients[2] (e.g., [16,17]).

In the 1990s, EAs eventually gathered some recognition in the CAD community [18]. Several methodologies were unable to scale with the technology and handling the complexity of the circuits became a serious issue. Evolutionary heuristics were seen as promising alternatives to classic approaches. Researchers proposed EA-based methodologies for placement, floorplanning and routing (all NP-hard problems). Compared to the early works, such applications required more complex encoding for the individuals and the design of ad-hoc genetic operators (e.g., [19,20]). Optimization and mapping for FPGA based on *configurable logic block* (CLBs), logic synthesis, and decision-diagrams optimization (e.g., BDD, OBDD, KFDD) also offered a fertile ground for EAs. Figure 1 shows the number of papers exploiting EAs that appeared in some key CAD conferences from the 1990.

Holland's GAs operated on strings of bits, and a test sequence for a sequential circuit is precisely a variable-length string of bits. As the complexity of the circuit started preventing the use of the traditional methodologies for sequential ATPGs, such as the *D algorithm* or *path-oriented decision making* (PODEM), some authors proposed to exploit EC. In the 1990s, several EA-based ATPGs were proposed, such as *GATTO* [21] and *STRATEGATE* [22]. Most of the research focused on exploiting the evolutionary core in an effective way, while the standard GAs' genetic operators were used. Driven by the first successes, researchers quickly moved into close areas, such as ATPG for fault diagnosis (e.g., [23]), ATPG for delay faults (e.g., [24]), ATPG for RT-level circuits (e.g., [25]), and test-sequence compaction (e.g., [26]). Works on EA-based ATPG steadily appeared in the scientific literature in the following 20 years.[3] In the 2000s, the idea of high-level ATPG was further extended and EAs were proposed to generate sequences of assembly instructions to test a microprocessors [27].

[2] EAs do not approximate gradients, nor presume their existence, thus can be used where other methods, like *quasi-Newton*, *conjugate gradient* or *Broyden–Fletcher–Goldfarb–Shanno* (BFGS), fail due to discontinuities, sharp bends, noise or local optima. EA-based numerical optimizers sharply improved in the 2000s. The most effective techniques available today are: *covariance matrix adaptation evolution strategy* (CMA-ES), developed by Nikolaus Hansen [45]; *differential evolution* (DE) devised by Kenneth Price and Rainer Storm [46]; *particle swarm optimization* (PSO), proposed by Russell Eberhart, James Kennedy and Yuhui Shi [47].

[3] A remarkable recent contribution is [49].

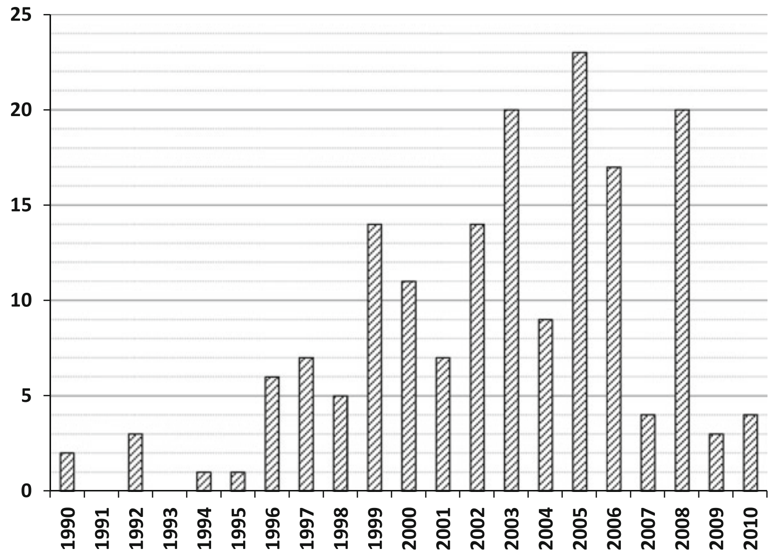

Fig. 1 Number of papers exploiting EAs published in key CAD conferences: *Design automation conference* (DAC); *International test conference* (ITC); *European design automation conference* (EURO-DAC, 1990–1996); *Design, automation and test in Europe* (DATE, 1997–2010)

EAs were also used for simulation-based design validation [28]. The generation of sequence of stimuli resemble the ATPG problem, however even more problems arise from the design the fitness function. A bug is either caught or silent, with no intermediate values, while evolution requires most of the individuals in the population to have different fitness values. EAs have been proposed as stimuli generators for *semi-formal verification*. In such contexts, fitness functions can be based on the coverage-metric figures obtained when simulating the individuals (e.g., [29]). Since 2000, the possibility to evolve assembly programs was also exploited for microprocessor design validation [30] and microprocessor post-silicon validation [31].

Since the very first works, EAs have been used as a design aid, to find the optimal values for parameters and constants, or to solve placement and routing problems. In addition, they were exploited to optimize analog cells for synthesis (e.g., [32,33]). In later works, to design more complex elements, like non-standard *built-in self-test* (BIST) structures (e.g., [34,35]). All these contributions are sometimes incorporated into a broader framework named *evolvable hardware* (EH or EHW). The recent availability of programmable devices, like *field-programmable gate array* (FPGA), abetted the experimentation in this area [36]. Despite a large amount of research and some interesting successes [37], however, EHW still has difficulty in being accepted by the CAD community.

4 Speed debug and on-silicon timing validation

Speed debug is the identification of paths that actually limit the performance of a chip, i.e., the locations where design or technological fixes are required. It is a critical activity, for instance, during the speed stepping of a microprocessor when a prototype is tested at increasing clock frequencies until a misbehavior is detected.

A *failing test* is a pattern of operations that uncovers an incorrect behavior. The availability of failing tests is essential for performing an effective speed debug. Unfortunately, the development of failing tests can be very expensive and time consuming. A *software-based functional failing test* is an assembly-language program whose result is functionally incorrect. That is, the misbehavior may be detected simply checking the values in the registers at the end of the execution.

Roughly speaking, speed paths can be highlighted in two complementary scenarios: low core voltage and high clock frequency. In a technology based on *field-effect transistors* (FETs), reducing the voltage increases the time required to switch between logic values [13] However, increasing frequency and reducing voltage involves significantly different phenomena, especially if paths have different voltage sensitivities or are interconnect dominated.

The following describes an EA-based methodology for the automatic generation of software-based functional failing tests suitable for speed debug or similar on-silicon activities. Both extreme cases are considered: in a first set of experiments, the core voltage is varied while the operating frequency is fixed. Conversely, in a second set of experiments, the operating frequency is varied while the core voltage is fixed.

5 Proposed architectures

Candidate failing tests are created without a rigid scheme, and evaluated on the target microprocessor. The data gathered are fed back to the generator and used to calculate their fitness. Then, a new and enhanced set of candidate tests is then generated. The whole process is iterated while some improvement is achieved. A feasibility study of the methodology has been presented in a poster at VLSI-SoC 2011 [38].

Two computers can be used: the *master* computer runs the EA-based failing-test generator; the *slave* executes them. (Fig. 2, left). Alternatively, only one computer may act both as *generator* and *evaluator*. There are both practical advantages and disadvantages in this second option (Fig. 2, right).

When two computers are used, the master needs to control the operating conditions of the slave, modifying the core voltage or the operating frequency. A significant

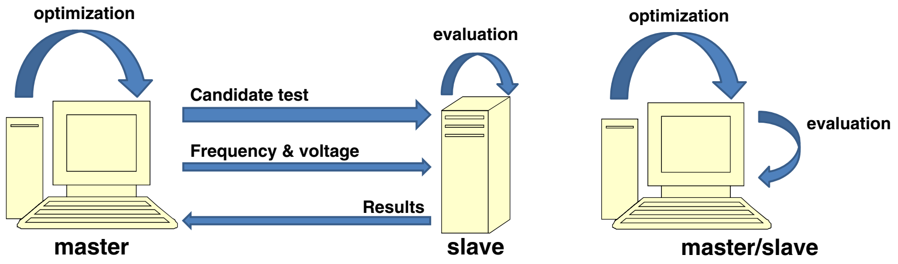

Fig. 2 System architecture: master and slave (*left*), and single computer (*right*)

overhead is added to the evaluation process. Furthermore, additional hardware may be required, although modern motherboards include several features to this end.

When only one computer is used the overhead is reduced, but the process is more intricate. The microprocessor must stop executing the evolutionary core and modify its own operating parameters before executing the candidate test. To this end it is possible to exploit the dynamic performance scaling technologies. Intel branded them as *SpeedStep*; Advanced Micro Devices as *PowerNow!* and *Cool'n'Quiet*; VIA Technologies as *LongHaul*. Since such technologies are intended to save power and reduce heat, they only allow decreasing the operating frequency and the power supply voltage supplied to the microprocessor. Thus, in the beginning of the experiments it may be required to increase the operating frequency in a different way, such as modifying motherboard settings.

In both scenario, it is required to handle serious crashes. When operating outside standard parameters, the microprocessor can hang or freeze unexpectedly.

5.1 EA-based failing-test generator

The proposed methodology is based on an evolutionary core able to create suitable candidate test programs. The automatic generation of programs for solving a generic user-defined problem has always been one of the goal of EC since its very beginning. GP is probably the branch of EC more close to this ambitious goal, indeed, the name *genetic programming* reflects this aspiration. Nevertheless, most of the actual research focused on the evolution of mathematical expressions, rather than full programs.

A versatile toolkit, named µGP, was developed at Politecnico di Torino in the early 2000s and it is now available under the *GNU Public License* from *Sourceforge*.[4] µGP does not aim at *creating* a program to solve generic problems, but rather to *optimize* realistic assembly-language programs for very specific purposes. Its original use was to assist microprocessors' designers in the generation of programs for test and verification. Hence, the Greek letter *mu*, pronounced "micro", in its name.

µGP is designed to support all assembly peculiarities, like various conditional branches, different addressing modes, and instruction asymmetries. Generated programs may take advantage of all syntactic structures, such as global and local variables, subroutines and interrupts. Since its creation, the tool underwent three main revisions [39,40] and [41].

The latest version internally encodes individuals as directed multigraphs. During the evolution, one or more instructions can be added; one or more instructions may be removed; the operands of certain instructions can be modified. µGP also includes crossover operators, thus new programs may be obtained by mixing, in different ways, existing ones. The multigraph representation helps ensuring that the result of the crossover is still a sensible program, resembling to both parents and, thus, inheriting potentially good characteristics from both of them.

Finally, in µGP, the fitness is not a single value but a vector of positive coefficients. The individual A is considered to be fitter than the individual B if the first j elements

[4] http://ugp3.sf.net/.

of the two fitness vectors are equals, and the $(j + 1)$th element of the A's fitness is greater than the $(j + 1)$th element of the B's fitness.

5.2 Individuals

The internal representation is a key aspect of every EA. µGP requires a description of the assembly language to be used. For the generation of failing test it is essential to test all possible instructions, and especially the newest. The assembly instructions made available to µGP can be divided in three main classes.

Integer instructions include all usual instructions, such as logical and arithmetical ones. They operate on internal registers or memory. In the adopted scheme, only two registers are employable, while the others are used by the manager. However, this restriction should not impair the global result. Comparisons, tests and branches are also included in this class. To avoid endless loops, µGP was forced to create only forward branches in the generated code.

x87 instructions are the subset of the Intel 32-bit architecture (IA32) related to the floating point unit (FPU). The name stems from the old separate floating point coprocessors, like 80287 and 80387. They provide single precision, double precision and 80-bit double-extended precision binary floating-point arithmetic according to the IEEE 754-1985 standard. x87 instructions operates on a stack of eight 80-bit wide registers, but some instruction modifiers allow the use of the stack as a set of registers. In the actual version, µGP uses x87 instructions in only one thread.

The third class of instructions requires a slightly longer introduction. In 1996, Intel introduced *single-instruction/multiple-data* (SIMD) *instructions* in the *Pentium* microprocessor, its first superscalar implementation of the x86 instruction set architecture. In a SIMD instruction, multiple processing elements perform the very same operation simultaneously on different data. Matter-of-factly, the technique is called *data-level parallelism*. Pentium's SIMD instructions were originally branded as *MMX extension*, then Intel changed their name to *Streaming SIMD Extensions* (SSE). Advanced Micro Devices offered its own version as *3DNow!*

Not surprisingly, SIMD instructions are particularly critical during speed stepping. The complex calculations involved by these instructions cause data to go through several functional units, and the resulting datapaths are prone to be source of problems when the operating frequency is increased.

Cache memories must be taken into account as well, since there may be a significant difference in performance and power consumption between a L1 cache hit and a L1 cache miss. In order to give µGP the possibility to generate cache hits and cache misses, a special set of C variables was defined. The variables are carefully spaced so that all their memory locations will be cached in the very same cache location. If the microprocessor uses a k-way set associative L1 cache and $C > k$, a shrewd sequence of read and write operations on such variables may generate the desired cache activity.

It must be noted that the goal of adding such variables is to let the evolutionary core to control the cache activity, but no suggestions are given on how nor when to exploit them. µGP would find autonomously which sequence of operations is more useful to generate a failing test.

Finally, since modern processors implement multithreaded designs, exploit multicore architectures, or both, the candidate test is required to contain multiple independent instruction flows. Such independent instruction flows are encoded in a µGP individual as disjoint subgraphs.

5.3 Fitness function

A functional failing test for speed debug is an assembly-language program that produces the correct result only while the operating frequency is below a certain threshold and the core voltage is above a certain other threshold. Let denote both these values as *functional thresholds*, because the incorrect behavior is functionally observable.

The most relevant aspect of a candidate test is its functional threshold. In the first scenario lower frequency values are better: a test that produces a failure at a relatively low frequency is preferable to a test that fails only at very high frequencies. Conversely, in the second scenario, higher voltage values are better: all tests would fail with a very low core voltage, but only the interesting ones require full power. Thus, the first element in the fitness vector represent directly the functional frequency threshold, or it is inversely proportional to the functional voltage threshold.

µGP creates assembly functions that are assembled and linked with a *manager* module. These functions contain a loop that execute L times a set of instructions. The instructions themselves are devised by the evolutionary core, while the framework is fixed. Similarly to *software-based self test* (SBST) [42], candidate test programs include a mechanism for checking their correctness: all the results of the calculations performed by the test program during the loops are compacted in a single signature using a hash function. The program is first executed in safe conditions and the signature recorded. Then, operating conditions are varied checking that the signature is not modified. As soon a difference is detected, the functional threshold is recorded.

Non-deterministic effects pose additional challenges: design criticalities may appear only occasionally, thus, the test is repeated several times. The practical usefulness of a functional failing test increases with its predictability: a test failing half of the times is more useful than a test that produces a single failure every thousand runs. Thus, each test is repeated R times, and the second element of the fitness is the number of failures recorded at the functional threshold.

6 Experimental evaluation

Devising a comparison for the proposed methodology is not an easy task: there are no publicly-available test suites for assessing results on functional failing-test generation, and very few results have been disclosed in the scientific literature. However, a closely related problem is frequently faced by the *overclockers*, a community of computer enthusiasts. Overclockers enjoy themselves pushing the performance of their microprocessors by increasing the operating frequency far beyond the nominal specification

[43]. For instance, an *Intel Celeron D 352* has been reported running with a clock above 8.3 GHz,[5] 160% higher than the nominal 3.2 GHz.

Overclockers need to assess the stability of their systems. The whole community is actively seeking *stability tests* able to quickly and reliably discriminate a working system from one that have been pushed too far. Such test suites are used to stress the systems and highlight criticalities, thus they may be regarded as generic functional fail tests not focused on a specific microprocessor. They have been used as a baseline to evaluate the performances of the proposed methodology.

While all the stability tests are quite different, a common point is that modern ones do extensive SIMD calculation. Another common point is their ability to increase the temperature of the microprocessor. It is well known that high temperature may cause both reversible and irreversible effects on electronic devices. Heating may increase the skew of the clock net and alter hold/setup constraints, causing design criticalities to become manifest and the circuit to operate incorrectly [44].

However, while such an effect is sensible when assessing the stability of a system, it may not be desirable when the goal is to find a failing test during speed stepping. The main reason is that the failing test should be as repeatable as possible, while increasing the temperature also increases non-deterministic phenomena. Nevertheless, since no other comparison is possible, the proposed approach was tested against the state-of-the-art stress tests used by the overclocking community.

6.1 Overclockers' stress tests

Most of the information about stability stress tests is available through forums and web sites on the internet, with few or none official sources. However, there is quite a generalized agreement in the overclockers community on these tools.

Prime95 is the name of an application written by George Woltman and used by a project for finding *Mersenne prime numbers*.[6] It makes extensive use of the fast Fourier transform, or FFT, with a highly efficient implementation that exploits SIMD instructions. Over the years, it has become extremely popular among overclockers as a stability test. It includes a "Torture Test" mode designed specifically to test systems and highlight problems. In the overclocking community, the rule of thumb is to run it for some 10 h.

LINPACK is a software library for performing numerical linear algebra on digital computers. It was originally written in Fortran in the 1970s and early 1980s. Newer implementations of LINPACK exploit SIMD instructions and are highly optimized. Significantly, Intel includes a benchmark based on an optimized version of LINPACK in its *Math Kernel Library*.[7] Different applications exploited such benchmark to assess

[5] http://valid.canardpc.com/records.php.

[6] A *Mersenne number* is a positive integer that is one less than a power of two: $M = 2^p - 1$. As of September 2011, the largest known prime number is a Mersenne number: $N = 2^{43,112,609} - 1$.

[7] http://software.intel.com/en-us/intel-mkl/.

the stability. The most common are *LinX*,[8] *IntelBurnTest*,[9] and *OCCT*.[10] The last one also includes a proprietary stress test.

6.2 Target microprocessors

Experiments were run on two different microprocessors, namely the *Intel Pentium Core 2 Duo E2180* and the *Intel Pentium Core i7-950*. These microprocessors were mounted on standard motherboard and equipped with in-house manufactured water cooling systems.

The E2180 is a dual-core microprocessor based on the *Core* architecture. It does not exploit simultaneous multithreading. Each core has two separate 32 KiB L1 caches for data and instructions, both implementing an 8-way set associative architecture. Each core has also a L2 cache is 1 MiB, 4-way set associative that is used for both data and instructions. While the default clock was 2 GHz, for the purpose of the experiments the system was overclocked to 2.93 GHz.

The i7-950 is a quad-core microprocessor based on *Nehalem* architecture, the successor of the *Core* architecture. It is able to run up to 8 threads with simultaneous multithreading. Each core has two separate 32 KiB L1 caches for data and instructions, both implementing an 8-way set associative architecture. Each core has also an L2 cache of 1 MiB, 8-way set associative that is used for both data and instructions. There is an additional 8 MiB L3 cache, 16-way set associative that is shared by the 4 cores using a design branded as *Intel smart cache*. The default clock ranges between 3.06 and 3.48 GHz, thanks to the so-called *Intel turbo boost technology 2.0* that automatically allows processor cores to run faster than their base operating frequency.

6.3 Experimental results

The parameters used in the experiments are summarized in Table 1. All μGP ones are standard. R and L impact on the length of the experiments. C must be equal or larger compared to cache parallelism.

The failing test devised by the proposed approach on the target system was compared with the state-of-the-art stress tools used by the overclocking community. Columns are labeled with the name of the program used to test the system. The last column reports data of the test generated by μGP. Rows indicate the CPU core voltage at which the experiments were run. Cells shows the time required for the given stress test to report a failure. To reduce overheating effects, all tests were stopped after 10 minutes. The infinity sign "∞" means that no failure has been detected in the allowed time. All experiments have been repeated 10 times.

Tables 2 and 3 report the comparison against overclockers' stress when the criticalities where caused by a reduction of the core voltage. The first table shows results

[8] Originally posted on http://forums.overclockers.ru/.

[9] http://www.ultimate-filez.com/.

[10] http://www.ocbase.com/perestroika_en/.

Table 1 μGP parameters and failing-test constants

Parameter	Meaning	Value
μ	Size of the population	30
ν	Size of the initial (random) population	100
λ	Operators applied in each generation	20
R	Repetitions of each test to tackle variability	10
L	Repetitions inside each test	5,000,000
C	Variables to exploit cache hit/miss	16
	Stopping condition	Steady state

Table 2 Time required to detect an incorrect behavior on the E2180 (system clock set to 2.93 GHz)

CORE [V]	Prime95	IBT	LinX	OCCT	μGP
1.2625	$1''$	$2'$	$2'$	$3''$	$\leq 1''$
1.2750	$6''$	$2'$	$2'$	$4''$	$2''$
1.2875	$4'$	$4'$	$2'$	$7'$	$2''$
1.3000	∞	$7'$	$7'$	∞	$10''$
1.3125	∞	∞	∞	∞	$8'$
1.3250	∞	∞	∞	∞	∞

Table 3 Time required to detect an incorrect behavior on the i7-950 (system clock set to 3.82 GHz)

CORE [V]	Prime95	IBT	LinX	OCCT	μGP
1.21250	$6'$	$1'$	$2'$	$4'$	$< 1''$
1.21875	∞	∞	$4'$	$5'$	$< 1''$
1.22500	∞	∞	∞	∞	$< 1''$
1.23125	∞	∞	∞	∞	$< 1''$
1.23750	∞	∞	∞	∞	$< 1''$
1.24375	∞	∞	∞	∞	$< 1''$
1.25000	∞	∞	∞	∞	$< 1''$
1.25625	∞	∞	∞	∞	$1''$
1.26250	∞	∞	∞	∞	$1''$
1.26875	∞	∞	∞	∞	$3''$
1.27500	∞	∞	∞	∞	$3''$
1.28125	∞	∞	∞	∞	$3''$
1.28750	∞	∞	∞	∞	$5''$
1.29375	∞	∞	∞	∞	$30''$
1.30000	∞	∞	∞	∞	$2'$
1.30625	∞	∞	∞	∞	$5'$
1.31875	∞	∞	∞	∞	∞
1.32500	∞	∞	∞	∞	∞

Table 4 Time required to detect an incorrect behavior on the i7-950 (V-core set to 1.24375 V)

CPU Freq. (GHz)	Prime95	IBT	LinX	OCCT	μGP
3.827	30″	2′	3′	5′	29″
3.803	9′	10′	4′	∞	29″
3.783	∞	∞	5′	∞	29″
3.758	∞	∞	6’	∞	29″
3.737	∞	∞	∞	∞	30″
3.721	∞	∞	∞	∞	30″
3.691	∞	∞	∞	∞	30″
3.666	∞	∞	∞	∞	30″
3.645	∞	∞	∞	∞	77″
3.622	∞	∞	∞	∞	∞

Table 5 Time required to detect an incorrect behavior on the i7-950 (V-core set to 1.2500 V)

CPU Freq. (GHz)	Prime95	IBT	LinX	OCCT	μGP
3.827	3′	6′	6′	8′	28″
3.803	∞	∞	6′	∞	28″
3.783	∞	∞	∞	∞	29″
3.758	∞	∞	∞	∞	29″
3.737	∞	∞	∞	∞	29″
3.721	∞	∞	∞	∞	30″
3.691	∞	∞	∞	∞	30″
3.666	∞	∞	∞	∞	30″
3.645	∞	∞	∞	∞	∞
3.622	∞	∞	∞	∞	∞

gathered on the E2180. All programs use two threads, that is, one for each core. The second table shows results gathered on the i7-950. All programs use eight threads, that is, two for each core. The i7-950 system clock was fixed to 3.82 GHz (about 10% faster than the nominal frequency).

μGP required about 5 h to generate the best failing test for the E2180, and 40 h for the i7-950. The difference in time can be explained taking into account the greater number available steps, and the length of the test itself. Experiments where run using a single computer.

Tables 4 and 5 report the comparison against overclockers' stress when the criticalities where caused by an increase of the operating frequency. Since it was not easy to tamper with the frequency of the E2180, experiments were run twice on the i7-950 with two core voltage, namely 1.24375 and 1.2500 V. μGP required about 100 h to generate each failing test. Experiments where run using a master-slave architecture, and this added a significant overhead to the evaluations.

Failing tests devised with the proposed methodology clearly outperform all other approaches, forcing the processor to fail in conditions significantly close to nominal

Table 6 Feedback from the overclockers community

CPU	Frequency		V-Core	IBT	LinX	OCCT	μGP
	N	A					
i7-860	2.80	4.25	1.4	–	FAIL	–	PASS
i7-860	2.80	4.30	1.4	–	FAIL	–	FAIL
i7-920	2.66	4.02	1.27	–	PASS	–	FAIL
i7-920	2.67	2.65	1.27	–	–	–	PASS
i7-920	2.67	3.20	1.0	PASS	–	–	FAIL
i7-920	2.67	3.20	1.044	PASS	–	–	PASS
i7-920	2.67	3.20	1.0312	FAIL	–	–	FAIL
i7-920	2.67	3.20	1.0375	FAIL	–	–	FAIL
i7-920	2.67	4.20	1.35	–	–	–	PASS
i7-920	2.67	4.33	1.385	–	PASS	–	FAIL
i7-920	2.67	4.40	1.45	–	–	–	PASS
i7-930	2.80	3.80	1.2	–	–	–	PASS
i7-950	3.06	4.03	1.31	PASS	–	–	PASS
i7-950	3.06	4.03	1.28	FAIL	–	–	FAIL
i7-950	3.06	4.03	1.328	PASS	–	PASS	FAIL
i7-950	3.06	4.20	1.34	PASS	PASS	–	FAIL
i7-950	3.06	4.20	1.31	PASS	PASS	–	FAIL
i7-965	3.20	3.46	1.21	–	–	–	PASS

ones. Remarkably, μGP was asked to find a very fast failing test for a specific microprocessor, and therefore it is highly improbable that the devised program would fail on a different model.

Moreover, the test was required to be very short, to avoid heating effects. On the contrary, stress tests intentionally exploit overheating and are designed to work with different architectures. The temperature of the microprocessor during the experiments never exceeded 50°C, while it was significantly higher while running LINPACK-based stress tests, even with the liquid cooling.

6.4 Feedback from the overclockers' community

The generated tests for the i7-950 were made available to the overclockers community as *ultra-fast stability test*.[11] The feedback is summarized in Table 6. The column CPU shows the CPU model used in the experiments. The two columns labeled with *Frequency* report the nominal (N) frequency of the CPU and the one actually used by the overclocker (A). The next column shows the actual core voltage. The following columns report the results of the various stability test: IBT, LinX, OCCT and the one

[11] http://www.cad.polito.it/research/Microprocessors_Test_and_Verification/Speedpath_and_Overclocking/.

generated by µGP. All the programs were considered *stability tests*, thus “FAIL” is a positive result, meaning that the test was able to uncover the instability. On the other hand, “PASS” means that the test was unable to pinpoint any problem.

Some overclockers did not run comparison tests with IBT, LinX or OCCT. Nevertheless, the fact that they try the µGP one implies that they were considering their system fully reliable.

Although not systematic, the feedback fully confirmed our claims: results on i7-950 microprocessors show the superiority of the µGP test. Similar results are achieved on all i7-9xx units. As expected, the failing test is not effective on units from the i7-860 family. Thus, it sounds plausible that the test stresses specific microarchitectural features present in the former families but not in the i7-8xx one.

7 Conclusions

The paper proposed an efficient post-silicon methodology for devising software-based functional failing tests. Such failing test may be exploited during speed debug or other on-silicon activities, like timing verification.

Experimental results clearly demonstrate that tests are able to highlight criticalities very specific of the target microarchitecture. More interestingly, it is able to do it without any information about the design. The methodology was successfully tested on an both an Intel Pentium Core 2 Duo E2180 and an Intel Pentium Core i7-950, and it could be very easily applied to different devices.

The proposed methodology could be easily exploited by microprocessor manufacturers during timing verification, speed debug or other post-silicon activities.

Acknowledgments I need to acknowledge my friends Ernesto Sanchez and Alberto Tonda for their help and insightful comments.

References

1. Zeng J, Guo R, Cheng W-T, Mateja M, Wang J (2011) Scan-based speed-path debug for a microprocessor. IEEE Des Test Comput PP(99). doi:10.1109/MDT.2011.73 (accepted)
2. Killpack K, Kashyap C, Chiprout E (2007) Silicon Speedpath Measurement and Feedback into EDA flows. In: 44th design automation conference, pp 390–395
3. Callegari N, Wang L-C, Bastani P (2009) Speedpath analysis based on hypothesis pruning and ranking. In: 46th ACM/IEEE design automation conference, pp 346–351
4. Encyclopædia Britannica. (2011) Encyclopædia Britannica Online. Online. http://www.britannica.com/EBchecked/topic/197367/evolution
5. Darwin C (1859) On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life. Murray, London
6. Turing AM (1950) Computing machinery and intelligence. Mind, no 9, pp 433–360
7. Fogel D (ed) (1998) Evolutionary Computation: the fossil record. IEEE Press, New York
8. Holland JH (1975) Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control and artificial intelligence. The University of Michigan Press, Ann Arbor
9. Fogel LJ (1962) Autonomous automata. Ind Res 4:14–19
10. Rechenberg I (1971) Evolutionsstrategie—Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Ph.D. thesis
11. Schwefel H-P (1974) Numerische Optimierung von Computer-Modellen. Ph.D. thesis

12. Koza John (1992) Genetic programming: on the programming of computers by means of natural selection. The MIT Press, Cambridge
13. Burke EK, Gustafson S, Kendall G (2004) Diversity in genetic programming: an analysis of measures and correlation with fitness. IEEE Trans Evol Comput 8(1):47–62
14. De Jong K, SarmaJ (1996) An analysis of the effects of neighborhood size and shape on local selection algorithms. In: International conference on parallel problem solving from nature. Springer, Berlin, pp 236–244
15. Squillero G, Tonda A (2008) A novel methodology for diversity preservation in evolutionary algorithms. In: GECCO conference
16. Etter D, Masukawa M (1981) A comparison of algorithms for adaptive estimation of the time delay between sampled signals. In: IEEE international conference on acoustics, speech, and signal processing, pp 1253–1256
17. Etter D, Hicks M, Cho K (1982) Recursive adaptive filter design using an adaptive genetic algorithm. In: IEEE acoustics, speech, and signal processing, pp 635–638
18. Drechsler Rolf (1998) Evolutionary algorithms for VLSI CAD. Springer, Berlin
19. Cohoon JP, Hegde SU, Martin WN, Richards D (1988) Floorplan design using distributed genetic algorithms. In: IEEE international conference on computer-aided design, pp 452–455
20. Cohoon JP, Paris WD (1987) Genetic placement. IEEE Trans Comput Aided Des Integr Circuits Syst 6(6):956–964
21. Corno F, Prinetto P, Rebaudengo M, Sonza Reorda M (1996) GATTO: a genetic algorithm for automatic test pattern generation for large synchronous sequential circuits. IEEE Trans Comput Aided Des Integr Circuits Syst 15(8):991–1000
22. Hsiao MS, Rudnick EM, Patel JH (2000) Dynamic state traversal for sequential circuit test generation. ACM Trans Des Autom Electron Syst 5(3):548–565
23. Girard P, Landrault C, Pravossoudovitch S, Rodriguez B (1996) A diagnostic ATPG for delay faults based on genetic algorithms. In: International test conference, pp 286–293
24. Heragu K, Patel JH, Agrawal VD (1999) A test generator for segment delay faults. In: International conference on VLSI design, pp 484–491
25. Corno F, Prinetto P, Sonza Reorda M (1997) Testability analysis and ATPG on behavioral RT-level VHDL. In: International test conference, pp 753–759
26. Rudnick EM, Patel JH (1996) Simulation-based techniques for dynamic test sequence compaction. In: International conference on computer-aided design, pp 67–73
27. Corno F, Sonza Reorda M, Squillero G, Violante M (2001) On the test of microprocessor IP cores. In: Design, automation, and test in Europe, pp 02–09
28. Sonza Reorda M, Squillero G, Corno F (1999) Approximate equivalence verification of sequential circuits via genetic algorithms. In: Automation and test in Europe, pp 754–755
29. Corno F, Sonza Reorda M, Squillero G, Manzone A, Pincetti A (2000) Automatic test bench generation for validation of RT-level descriptions: an industrial experience. In: Design, automation and test in Europe, pp 385–389
30. Sanchez E, Squillero G (2007) Evolutionary techniques applied to hardware optimization problems: test and verification of advanced processors. In: Palade V, Srinivasan D, Jain LC (eds) Studies on computational intelligence, vol 66. Advances in Evolutionary computing for system design. Springer, Berlin, pp 83–106
31. Sanchez E, Sonza Reorda M, Squillero G, Lindsay W (2004) Automatic test programs generation driven by internal performance counters. In: Microprocessor test and verification, pp 8–13
32. Kruiskamp W, Leenaerts D (1995) DARWIN: CMOS opamp synthesis by means of a genetic algorithm. In: Design automation conference
33. Iskander R et al (2003) Synthesis of CMOS analog cells using AMIGO. In: Design, automation and test in Europe, pp 297–302
34. Cataldo S, Chiusano S, Prinetto P, Wunderlich H-J (2000) Optimal hardware pattern generation for functional BIST. In: Design, automation and test in Europe, pp 292–297
35. Polian I, Becker B, Reddy SM (2003) Evolutionary optimization of Markov sources for pseudo random scan BIST. In: Design, automation and test in Europe, pp 1184–1185
36. Sekanina L (2004) Evolvable components: from theory to hardware implementations. Springer, Berlin
37. Higuchi T, Yao X (eds) (2006) Evolvable hardware. Springer, Berlin
38. Sanchez E, Squillero G, Tonda A (2011) Post-silicon failing-test generation through evolutionary computation. In: 19th IFIP/IEEE international conference on very large scale integration, Hong Kong

39. Corno F, Cumani G, Sonza Reorda M, Squillero G (2002) Efficient machine-code test-program induction. In: Proceedings of the 2002 congress on evolutionary computation, pp 1486–1491
40. Squillero G (2005) MicroGP—an evolutionary assembly program generator. Genetic Program Evol Mach VI(3):247–263
41. Sanchez E, Schillaci M, Squillero G (2010) Evolutionary optimization: the μGP toolkit. Springer, Berlin
42. Chen Li, Dey S (2001) Software-based self-testing methodology for processor cores. IEEE Trans Comput Aided Des Integr Circuits Syst, pp 369–380
43. Colwell B (2004) The zen of overclocking. Computer 37(3):9–12
44. Chakraborty A et al (2008) Dynamic thermal clock skew compensation using tunable delay buffers. IEEE Trans Very Large Scale Integr (VLSI) Syst 16(6):639–649
45. Hansen N (2006) The CMA evolution strategy: a comparing review. In: Larrañga P, Inza I, Bengoetxea E, Lozano JA (eds) Towards a new evolutionary computation. Advances in estimation of distribution algorithms. Springer, Berlin pp 75–102
46. Price Kenneth, Storn Rainer, Lampinen Jouni (2005) Differential evolution: a practical approach to global optimization. Springer, Berlin
47. Poli R (2008) Analysis of the publications on the applications of particle swarm optimisation. J Artif Evol Appl 1–10
48. Fogel D (1995) Evolutionary computation: toward a new philosophy of machine intelligence. IEEE Press, Piscataway
49. Li Min, Hsiao MS (2009) An ant colony optimization technique for abstraction-guided state justification. In: International test conference, pp 1–10