Chapter 4
Post-silicon Speed-Path Analysis in Modern Microprocessors through Genetic Programming

The incessant progress in manufacturing technology is posing new challenges to microprocessor designers. Nowadays, comprehensive verification of a chip can only be performed after tape-out, when the first silicon prototypes are available. Several activities that were originally supposed to be part of the pre-silicon design phase are migrating to this post-silicon time as well. This chapter describes a post-silicon methodology that can be exploited to devise functional failing tests. Such tests are essential to analyze and debug speed paths during verification, speed-stepping, and other critical activities. The proposed methodology is based on the Genetic Programming paradigm, and exploits a versatile toolkit named μGP. The chapter describes how an evolutionary algorithm can successfully tackle a significant and still open industrial problem. Moreover, it shows how to take into account complex hardware characteristics and architectural details of such complex devices. The experimental evaluation clearly demonstrate the potential of this line of research. Results of this work have been accepted for publication in [137].

4.1 Background

Nowadays, manufacturing technology is advancing at a faster pace than designing capability, posing unprecedented challenges in the arena of integrated circuits. The so-called *verification gap* denotes the inability to fully verify the correctness of devices that could be built, and indeed *are* actually built. Practice surpasses theory: comprehensive verification of a chip can only be performed after tape-out. Once manufacturing is completed and first silicon is produced, the early chips are sent back to their design teams. This process is called *post-silicon verification* to distinguish it from the traditional, pre-silicon, one. More generally, several activities that were originally supposed to be part of the pre-silicon design phase are nowadays migrating to the post-silicon time. The cost of manufacturing prototypical devices is enormous, but this practice is not an option. Designers candidly acknowledge

E. Sanchez et al.: Industrial Applications of Evolutionary Algorithms, ISRL 34, pp. 31–44.
springerlink.com

that "very few chips ever designed function or meet their performance goal the first time" [106].

Microprocessors are a paradigmatic example of the current trend: devices for the desktop market contain billions of transistors, implement complex architectures[1], and operate into the microwave frequency range. To give some examples, in a *pipelined architecture*, assembly instructions are executed as in a production line. Consequently, whereas the single instruction is not sped up, the global throughput is significantly increased. Even more, a *superscalar architecture* exploits duplicated functional units by executing two or more different instructions in parallel. The *branch prediction* unit guesses which way of a conditional branch will be taken, thus the execution may continue without waiting for the actual outcome of the test. Whether the conjecture was mistaken, a mechanism of *speculative execution* enables to efficiently roll back and undo changes.

Since last decade, desktop microprocessors also include hardware support to efficiently execute multiple *threads*, that is, independent flows of instructions. These architectures allow to increase the overall throughput in a multitasking environment, even when it would be impossible to further speed up the single program with the precedent techniques. *Simultaneous multithreading*[2] architectures enable multiple threads to be executed concurrently exploiting superscalar designs. More recently, in a *multicore architecture*, or *chip-level multiprocessor*, two or more independent processing units work side by side packaged in the same chip and sharing the same memory. Indeed, in modern multicore microprocessors each individual core also exploits simultaneous multithreading.

Besides this bewildering complexity, electric signals do propagate inside a microprocessor through different *paths*. To guarantee a correct behavior, all signals must reach a stable value within the current clock cycle, regardless the length or the complexity of their routes. It must be remembered that when a microprocessor is reported to operate at 3 GHz, the time available for signals to stabilize is slightly above 3 x 10-10 seconds. It may be hard to visualize such a frantic activity, for in this interval of time light covers only 10 cm (almost 4 inches).

Non-deterministic effects, such as manufacturing variability, are posing even greater challenges to the designers. It has been long known that several physical defects only appear when the device operates at full speed [152], but nowadays design criticalities also become apparent only at high frequencies. Even worse, they appear only occasionally, and possibly only in a percentage of the manufactured chips. "Finding the root cause of at-speed failures remains one of the biggest challenges in any high-performance design", stated Rob Aitken in his *editor's note* for [83].

[1] Some texts emphasize the difference between the specification of the machine language and its implementation, calling the former "instruction set architecture" and the latter "microarchitecture".

[2] Called "hyper-threading" in Intel designs.

4.2 Introduction

To meet today's performance requirements, the design flow of a modern microprocessor goes through several iterations of frequency pushes prior to final volume production. Such a process is called *speed stepping*. A *speed path* (or *speedpath*) is a path that limits the performance of a chip because a faster clock would cause an incorrect behavior. Speed paths may be the location where potential design fixes should be applied, and may indicate places where potential holes in the design methodologies exist.

At design time, the slowest logic path in a circuit is termed the *critical path*, and it can be easily determined. However, for complex high-performance designs, it has been recognized that critical paths reported from the pre-silicon timing analysis tools rarely correlate well to the actual speed paths. The reason is that any pre-silicon analysis tool is only as accurate as the model and the algorithms it uses. Obtaining 100% accurate process models for nanometer processes is difficult, if not nearly impossible. Analysis algorithms are also approximated because of the complexity involved. Moreover, timing behavior on the silicon is a result of several factors mingled together. But in the pre-silicon phase it would not be computationally feasible to consider all these factors simultaneously, and they are analyzed separately [164] [82] [25].

The identification of *failing tests*, i.e., sequences of operations that uncovers incorrect behaviors when run at high frequency, is highly related with speed path identification. Failing tests may be, for example, sequence of inputs to be applied to the microprocessor pins by an automatic test equipment (ATE). Such test are usually crafted with care by engineers starting from the pre-silicon verification test suite; generated by pre-silicon specialized tools, or automatic test pattern generators (ATPGs); or also created post silicon[3], tackling the actual devices [96] [165].

Interestingly, the instruction sets of microprocessors has been successfully exploited to tackle *path-delay faults*, i.e., manufacturing defects that slow down the signals covering a specific path inside the device [93] [35]. The underlying idea of these works is that executing a set of carefully designed programs may uncover timing issues. The main strength of the methodology is that the execution of such *test programs* is per se *at-speed* and requires no additional hardware, or complex and expensive ATEs. No attempts, however, have been reported to devise failing tests directly at the instruction level. No one has yet proposed a post-silicon methodology able to automatically generate a test program that stresses a speed path causing a detectable functional failure.

A *software-based speed-path failing test* is defined as an assembly-language program that produces the correct result only while the microprocessor operating frequency is below a certain threshold. As soon as the frequency is pushed above the threshold, the result yielded by the program becomes incorrect. Let us denote the threshold for a given program as its *functional frequency threshold*, because the incorrect behavior is functionally observable. That is, it can be

[3] The expressions "on silicon" and "silicon based" are also used.

theoretically detected without an ATE or other special equipment, simply by observing the values stored in the main memory and registers. Clearly, the diagnostic capability of a software-based speed-path failing test increases as its functional frequency threshold decreases. A test that produces a failure at a relatively low frequency is preferable to a test that fails only at very high frequencies.

This chapter shows how software-based speed-path failing tests with low functional frequency thresholds can be automatically generated by an evolutionary algorithm. Moreover, it demonstrates that the technologies already available in modern microprocessors can completely cut out the need of external equipments at the expense of a slight decrease in accuracy. The first result advocates for the exploitation of the methodology inside the manufacturer's facility during speed stepping phase. The second calls for coarse-grained, but quite inexpensive, incoming inspection campaigns.

Sections 3 and 4 describe the proposed methodology, detailing the adopted evolutionary algorithm. Sections 5 illustrates the feasibility study and report the obtained results. Section 6 concludes the chapter, sketching the future directions of the research.

4.3 Generation and Evaluation of Test Programs

The proposed approach for generating software-based speed-path failing tests is *pseudo-random* and *simulation-based*, or, more exactly, *feedback-based*. Candidate test programs are created without a rigid scheme, and evaluated on the target microprocessor. The data gathered are fed back to the generator and used to generate a new, enhanced set of candidate solutions. The process is then iterated.

To exploit such a mechanism it is indispensable to evaluate the goodness of each candidate test. As stated before, a software-based speed-path failing test is as good as it fails at low frequencies, and the key parameter in evaluating a test is its functional frequency threshold. However, it should not be forgotten that variability vexes verification engineers. A failing test may not fail always at the same frequency, even if all controllable parameters are exactly reproduced. The variability of speed paths may be caused by non-deterministic factors, such as noise, die temperature or small fluctuation in the external power. Some design criticalities may appear only under particularly unfavorable conditions. All experiments need to be repeated at least several times, when not on different devices.

Consequently, besides the lowest functional frequency threshold detected amongst the repeated experiments, an additional parameter in evaluating a test is the percentage of runs that actually failed at that frequency. It is intuitively plausible that a test failing half of the times at a certain frequency is more useful that a test that fails only every thousands experiments.

Changing the operating frequency of a microprocessor, however, is not an easy task. To ensure proper synchronization between all the components of the system, only a very limited set of operating clock speeds are available to the end users. While

the microprocessor is connected to an ATE after production, such an evaluation is perfectly feasible. However, outside manufacturer laboratories the large steps in frequencies would likely impair the overall usability. Notably, outside manufacturer laboratories, the final aim would hardly be speed stepping. Conversely, end users may be quite interested in performing an incoming inspection on purchased devices. Tacking this latter goal, this chapter shows how to adapt the methodology in order to require no test equipment and no additional hardware whatsoever.

The architecture of modern microprocessors includes dynamic performance scaling technologies. Intel branded it as *SpeedStep*. Similar mechanisms are available as Advanced Micro Devices *PowerNow!* and *Cool'n'Quiet*, or VIA Technologies *LongHaul*. Such technologies are designed to save power and reduce heat, thus they allow to decrease the operating frequency and the power supply voltage supplied to the microprocessor. Reducing the CPU core voltage is known as *undervolting*.

Roughly speaking, desktop microprocessors are made using the complementary metal-oxide-semiconductor (CMOS) technology, based on field-effect transistors (FETs). In such devices, reducing the voltage increases the time required to switch between logic values [14]. Thus, the effects of reducing voltage may be reasonable related to the effects of increasing the operating frequency. As a matter of fact, whenever a microprocessor is undervolted, its operating frequency is also reduced to guarantee proper functionalities. Manufactures define sets of safe operating states, sometime called *performance states* or *p-states*. While the exact meaning of these p-states is implementation dependent, P0 is always the highest-performance state, with the following *P1* to *Pn* being successively lower-performance and less-consuming states.

Following the discussion, it appears evident that undervolting a microprocessor emphasizes speed-path criticalities. Moreover, reducing the core voltage cannot damage a device. Thus, to stress speed paths the behavior of a microprocessor could be analyzed intentionally outsides the predetermined p-states. Let us define the *functional core voltage* of a failing test as the lower voltage required not to fail the test at a given operating frequency. Conversely to functional core frequency, a failing test is as good as its functional core voltage is high. That is, all tests would fail with a very low core voltage, but only the interesting ones truly require full power.

Thus, an alternative evaluation of a candidate test could be based on its functional core voltage, and on the percentage of runs that actually failed.

4.4 Evolutionary Approach

The proposed test-program generator exploits a versatile evolutionary toolkit called μGP developed at Politecnico di Torino, and available under the *GNU Public License* from *Sourceforge* [146]. Unlike usual genetic programming (GP) implementations, μGP specific target is to produce realistic assembly-language programs. Its original purpose was to assist designers in the generation of programs for the test

and verification of different microprocessors, hence, the Greek letter micro in its name.

μGP was designed to support assembly peculiarities, like various conditional branches, different addressing modes, or instruction asymmetries. Generated programs take advantage of syntactic structures as global and local variables, subroutines and interrupts. Since its creation, the tool underwent three main revisions [40], [149] and [134]. The latest version internally encodes individuals as directed multigraphs, and this enable the handling of a quite wide range of problems.

μGP is asked to devise an assembly program to be used as a software-based speed-path failing test. Following the previous discussion, a population of candidate test programs is evolved, and the evaluation of their goodness is used as fitness function to drive the process. However, the specificity of the task calls for several different problem-specific knacks.

4.4.1 Fitness Function

During experiment the system frequency is first increased using the so-called overclocking features of modern main boards. An excessive increase of the frequency may cause overheating or otherwise irreparably damage the microprocessor, but increasing it slightly is usually perfectly safe. Then the evaluation is performed by reducing the core voltage, only.

Similarly to *software-based self test* [132], candidate test programs include a mechanism that help checking their own correctness: all the results of the calculations performed by the test program are compacted in a single signature using a hash function. The evaluator runs the test program in safe conditions, i.e., at full power, and store the signature. Then it runs the program again at decreasing CPU core voltages, checking that the signature is not modified. As soon a difference is detected, the functional voltage threshold is recorded. The whole process is repeated R times to tackle variability.

In μGP the fitness function may be specified as a vector of positive numbers. The components of the vector are strictly hierarchical, with the first being the most important. The first component of the fitness value is simply the functional voltage threshold. The second is the number of failures detected over the R repetitions at the maximum voltage. It must be stressed out that the actual result of the calculations is of no interest, the only relevant detail is that it changes when the test is executed undervolting the CPU below the functional core voltage.

4.4.2 Individual Evaluation

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. At the end of the loop, before the next iteration, the values in the registers are used to update the signature.

In the proposed methodology the very same microprocessor is used both for generating candidate tests, i.e., for running μGP , and for their evaluation. Using the same processing unit to evolve individuals and calculate their fitness is quite a standard procedure in GP. In most μGP application, conversely, the interesting data is not the *result* of the computation, but *how* the test program is actually computed by the specific device. And the evaluation of the assembly-language test programs is usually carried out on an different unit, physically, by emulation, or by simulation. Extracting information from the microprocessor currently executing μGP may be quite tricky. It has been first attempted 2004, during a collaboration with Intel [97].

When it is required to calculate the fitness of the newly generated offspring, individuals are compiled to stand-alone executable and run. The manager also takes care of invoking the evolved fragment of code while varying the CPU core voltage, and creating a text file with the results. Eventually, the execution of μGP is resumed.

4.4.3 Evolution Start

Evolution advances *through the accumulation of slight but useful variations* [47]. Thus, if all individuals in the initial population are indistinguishable, it is hard for the process to start. Unfortunately, this is not an uncommon situation. The computer used for generating and evaluating the test programs is almost completely working. It is able to perform nearly all operations, and indeed finding an incorrect behavior requires elaborate sequences of instructions. Thus, in the first step it is not infrequent to have a population of test programs not able to fail at any voltage, with exactly the same fitness value.

To overcome this problem the first population is significantly larger than the usual ones. μGP uses the parameter ν (the Greek letter nu) to control the number of randomly generated individual in the beginning of the evolution.

4.4.4 Internal Representation, Multithreading and Multicore

In μGP , the individual is internally encoded as a directed multigraph. With the adopted scheme, disregarding all the details, each node encodes a line of the assembly program. Edges represent syntactic or semantic relationship. For instance one edge connects every two adjacent lines; an additional edge connects a branch instruction with its target; another edge connects a node referring to a global variable with the line defining the data.

Modern processors may implement a multithreaded design; or they can exploit a multicore architecture; or even both. From the perspective of the test-program

generator details are not relevant, but it is vital to create multiple independent instruction flows.

A single individual is composed of different independent functions. The manager activates them as different threads on different cores using appropriate operating system calls, or directly whether no operating system is used. Such blocks, in the individual, are represented as disjoint subgraphs. Notably, different blocks may be forced to have different structural characteristics, or use different subsets of instructions.

4.4.5 Assembly Language

For the generation of failing test is performed during speed stepping or an incoming inspection, it is essential to test all possible instructions, and especially the newest. The assembly instructions made available to μGP can be divide 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 provides 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 SIMD instructions were originally branded as *MMX extension*, and operate on eight 64-bit wide registers. Advanced Micro Devices offered its own enhanced version of the SIMD instructions two years later, marketing them as *3DNow!*. In 1999, Intel outbid with the so-called Streaming *SIMD Extensions*, or *SSE*. Followed in 2001 by *SSE2*, in 2004 by *SSE3*, and finally in 2006 by *SSE4*. Not mentioning the *Supplemental Streaming SIMD Extensions 3* (SSSE3, with three "S") included in Intel microprocessors from 2006. Advanced Micro Devices is planning to include *SSE5* in its *Bulldozer* processor core in 2011.

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.

4.4.6 Cache

Cache memories are small, expensive and fast memories placed near the processor core. The rationale is to read and write the most frequently accessed data as efficiently as possible. Modern microprocessors exploit a hierarchy of cache memories, or *multi-level caches*, with the level-1 (L1) cache being the smallest, more expensive and fastest. And, indeed, the closest to the central processing unit.

When the memory is accessed, the L1 cache checks whether the data is *cached*, i.e., if it contains the specified location. In this case, called *cache hit*, the L1 swiftly replies to the request. If the data is not present, termed *cache miss*, the L1 cache delivers the request to the L2 cache and so on. Considering only the first level, there is a significant difference in performance and power consumption between a L1 cache hit and a L1 cache miss. Such effects may be significant for the generation of a failing test, and must be taken into account.

The internal design of a cache is complex, and the policies for determining which data to store and which to discard are different. In a *fully-associative* cache, every memory location may be cached in every location of the cache. However, such a design is too complex and slow if the size of the cache increases. Thus, usually, the architecture imposes that a specific memory location may be stored only in a subset of cache locations. In a *direct-mapped cache* each memory location can be cached in only one location, while in a *k-way set associative cache*, in k alternative locations.

In order to give the 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 to exploit them. μGP would devise which sequence of operations is more useful to generate a failing test.

4.5 Experimental Evaluation

While no working attempts of functional failing-test generation has been reported in the specialized literature, a related problem is faced by a community of computer enthusiasts. *Overclockers* try to push the performance by increasing the operating frequencies of their microprocessors and the CPU core voltages [38]. However, after pushing their computers to astonishing frequencies, they need to assess the stability

of their systems. The test suites that are used to stress the systems and highlight criticalities may be regarded as generic fail tests not focused on a specific microprocessor. Thus, they can be 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 [27].

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 increase 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.

4.5.1 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.

SuperPI is a version of the program used by Yasumasa Kanada in 1995 to compute π to 232 digits. It is based on the Gauss-Legendre algorithm. SuperPI implementation makes use of x87 instructions only, it exploits no SIMD instructions, and it is strictly single threaded. CPU BurnIn is a stress test developed by Michal Mienik in the beginning of 2000s. Like SuperPI it uses no SIMD instructions and is single threaded. These two programs are rather old, but have been included for the sake of comparison.

Prime95 is the name of an application written by George Woltman and used by a project for finding *Mersenne prime numbers*[4] [1]. 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 for testing systems and highlight problems. In the overclocking community, the rule of thumb is to run it for some tens of hours.

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.

[4] A *Mersenne number* is a positive integer that is one less than a power of two: $M = 2^p - 1$. The name came from the French theologian, philosopher, mathematician and music theorist Marin Mersenne, sometimes referred to as the "father of acoustics". As of August 2010, only 47 Mersenne prime numbers are known. Remarkably, the largest known prime number is also a Mersenne number: $N = 2^{43,112,609} - 1$.

Newer implementation of LINPACK exploits SIMD and are highly optimized. Significantly, Intel includes a benchmark based on an optimized version of LINPACK in its Math Kernel Library [2]. Different applications exploited such benchmark to assess the stability. The most common are *LinX*[5], *IntelBurnTest*[6], and *OCCT*[7]. The last one, also includes a proprietary stress test.

4.5.2 Target System

Experiments were run on an Intel Pentium Core 2 Duo E2180, MSI motherboard NEO2-FR with the Intel chipset P35. The system was equipped with 3 GiB RAM memory DDR2-800, and a Sparkle Nvidia 8800GT graphic card. While the default clock was 2GHz, for the purpose of the experiments the system was overclocked to 2.93GHz. The only non-standard device was an in-house manufactured water cooling system (Fig. 4.1).

The E2180 is a dual-core microprocessor. It has a 32 KiB L1 cache for data implementing an 8-way set associative architecture. An identical cache is for instructions. The L2 cache is 1 MiB, 4-way set associative, and it is used for both data and instructions. The *Core* architecture can be traced back to the *P6*, introduced in 1995 with the *Pentium PRO* and revived in 2000 with the *Pentium M* line. It supports SIMD instructions up to SSE3 and SSSE3, and the *Enhanced Intel SpeedStep* (EIST) technology. Unlike its predecessor *NetBurst* and its successor *Nehalem*, the Core 2 Duo architecture does not exploit simultaneous multithreading.

Given the goal of the feasibility study, the difference between multicore and multithread may be regarded as a marginal detail. From the perspective of μGP there is no difference whether the different threads are evaluated on the same core or on multiple cores.

4.5.3 Experimental Results

The failing test devised by the proposed approach on the target system was compared with the state-of-the-art stress tools used by overclocking community. Results are reported in Table 4.2 and Table 4.3. 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. Thus “more than

[5] Originally posted on `http://forums.overclockers.ru/`
[6] `http://www.ultimate-filez.com/`
[7] `http://www.ocbase.com/perestroika_en/`

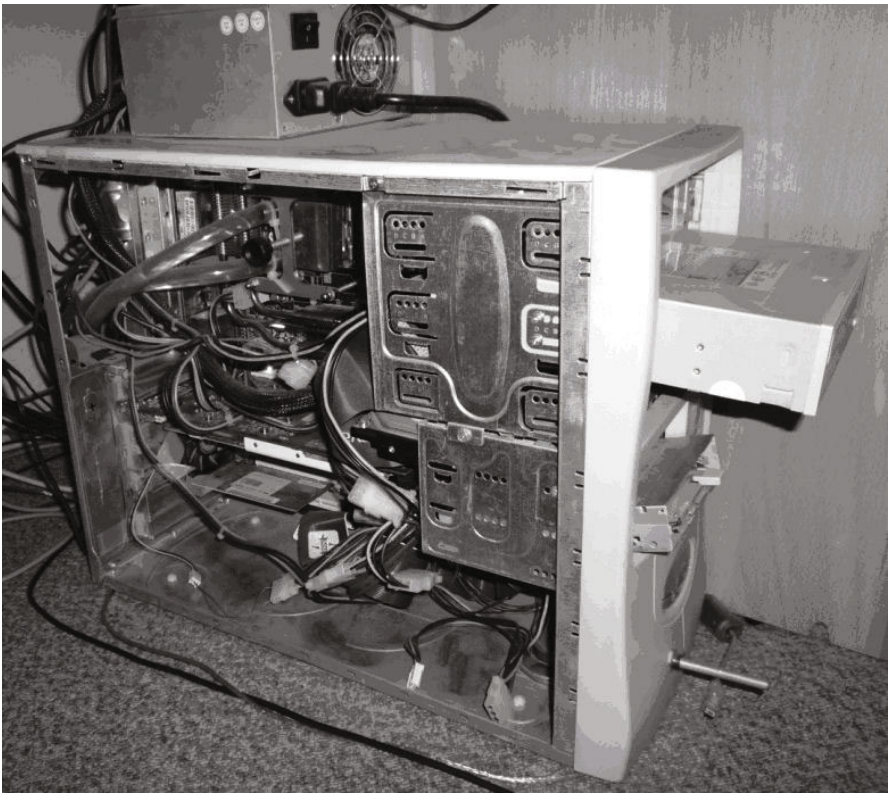

Fig. 4.1 The system used for the experiments.

10 minutes" means that no failure has been detected. All experiments have been repeated 10 times. μGP parameters are shown in Table 4.1.

Table 4.1 μGP parameters

Parameter	Meaning	Value
μ	Size of the population	30
ν	Size of the initial (random) population	100
λ	Genetic operators applied in each generation	20
R	Repetitions of each test to tackle variability	10
L	Repetitions inside each test	5,000,000

Table 4.2 compares the proposed methodology with older stress tests. Since multiple threads are not supported by SuperPI and CPU BurnIn they were disabled in μGP as well. It can be noted that the critical functional voltages are quite low, thus the microprocessor needs to be undervolted significantly in order to originate a

problem. Table 4.3, on the other side, reports the comparison against newer stress tests. All these programs uses two threads, that is, one for each core.

Table 4.2 Failing-test duration for single thread

CORE V	SuperPI	CPU BurnIn	μGP
1.2625	...	5′	1"
1.2750	10′	$>$ 10′	1"
1.2875	$>$ 10′	$>$ 10′	...
1.3000	$>$ 10′	$>$ 10′	...
1.3125	$>$ 10′	$>$ 10′	...
1.3250	$>$ 10′	$>$ 10′	...

Failing tests devised with the proposed methodology clearly outperform all the other approaches. However, it must be noted that the comparison is not completely fair, since the goal of the programs were different. μGP was asked to find a very fast failing test for a specific microprocessor, and there is no guarantee that they would fail on different models. Moreover, the test was required to be very short, to avoid heating effects. On the contrary, the adopted stress tests intentionally exploit overheating and are designed to work with different architectures.

Table 4.3 Failing-test duration for multiple threads

CORE V	Prime95	IntelBurnTest	LinX	OCCT	μGP
1.2625					2"
1.2750					2"
1.2875	4′			7′	2"
1.3000	>10′	7′	7′	>10′	10"
1.3125	>10′	>10′	>10′	>10′	8′
1.3250	>10′	>10′	>10′	>10′	

The final failing test is 614 line long. The two functions executed by the two cores are respectively 280 and 235 line long. The remaining lines are mainly used to define and initialize variables or other program parts. It should also be noted that μGP requires about 50' to generate a test failing at a core voltage of 1.2625V; 6h to find a test failing at a core voltage of 1.2750V; additional 5h for 1.2875V; and additional 5h for the 1.3000V. For the sake of experimentation, the failing test devised for 1.3000V was run at a core voltage of 1.3125V and consistently failed in about 8'. Interestingly, the temperature of the microprocessor during this last experiments never exceeded 40°C, while running LINPACK-based stress tests it is permanently above 45°C.

4.6 Conclusions and Future Works

An efficient post-silicon methodology for devising functional failing tests is proposed. The result of the chapter is twofold: first, it demonstrates the possibility for an evolutionary algorithm to generate assembly-level failing tests, tackling the most advanced microprocessor designs; second, it shows that the methodology can produce interesting results with negligible, or even nil, hardware overhead.

The proposed methodology could be exploited by microprocessor manufacturers, during verification or speed stepping. Or it could be used to generate a fast test able to check the reliability of a system. The latter can be important for the incoming inspection of a set of purchased devices.

Future works include enhancing the evolutionary algorithm, letting it tuning the number of repetitions in each test L. The interaction between x87 and SIMD instructions also deserves a closer examination. A customized version of the μGP requiring no operating systems can be devised in order to more easily run experiments on the microprocessor. Also, the signature could be improved by including more information on the state of the execution, such as the internal performance monitor.