GATTO*: a PVM-based Distributed Test Pattern Generation Algorithm for Large Sequential Circuits

F. Corno P. Prinetto M. Rebaudengo M. Sonza Reorda

Dipartimento di Automatica e Informatica
Politecnico di Torino

E. Veiluva

Centro Supercalcolo Piemonte
Torino

Goal

To develop an ATPG for large synchronous sequential circuits.

Outline

- Introduction
- Genetic Algorithms (GAs)
- The ATPG Algorithm
- Distributed Version
- Experimental Results
- Conclusions.

Introduction

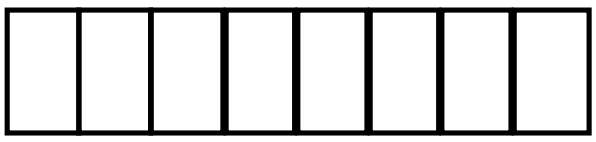
State-of-the-art approaches:

- ***topologic***
- ***symbolic***
- ***simulation-based*** **(e.g., GA-based).**

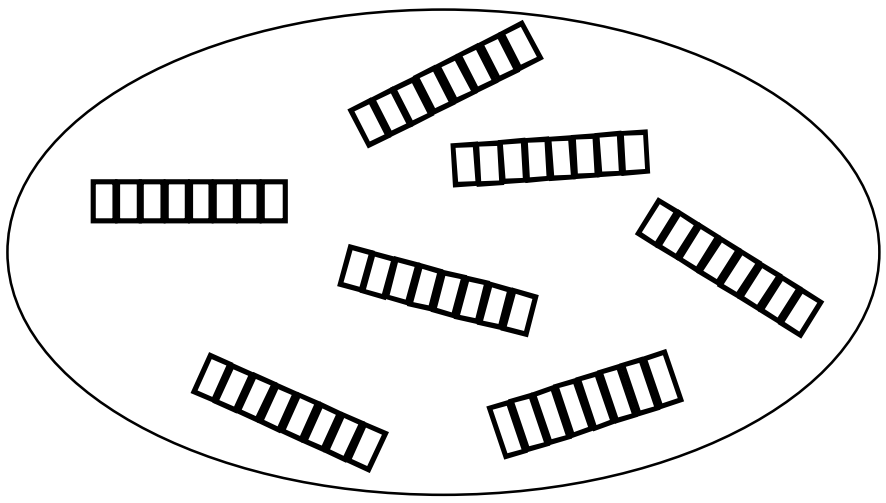
Genetic Algorithms

- **Mimic the evolution process in nature**
- **Suitable for optimization and search problems**
- **Already applied to many real problems.**

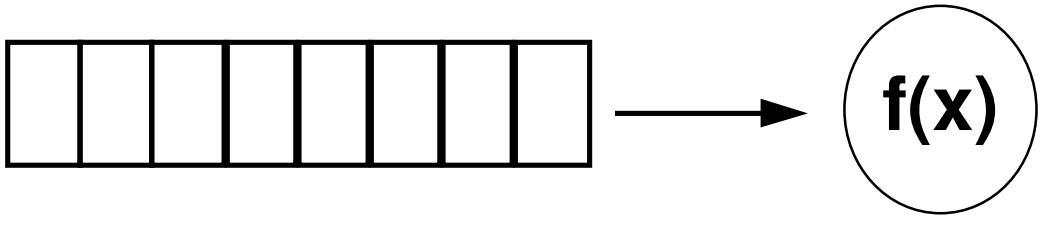
GAs basics



1 individual = 1 solution = 1 string

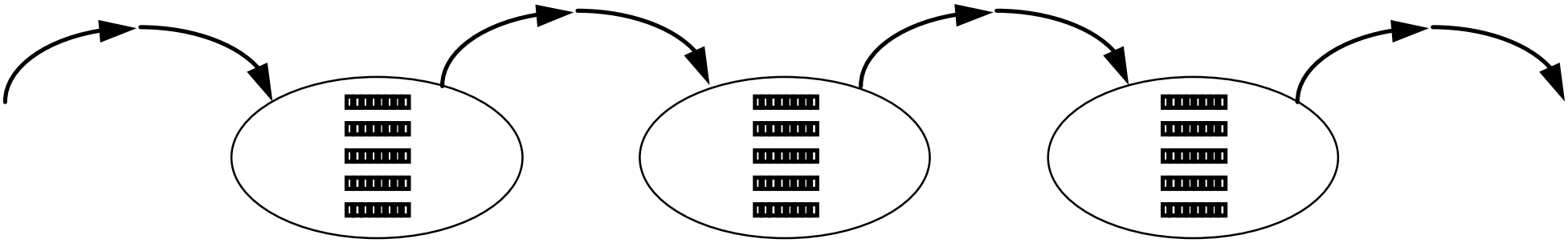


Population = set of individuals



A fitness function evaluates how *close* each individual is to the optimum. ePVMug'94

Evolution through Reproduction



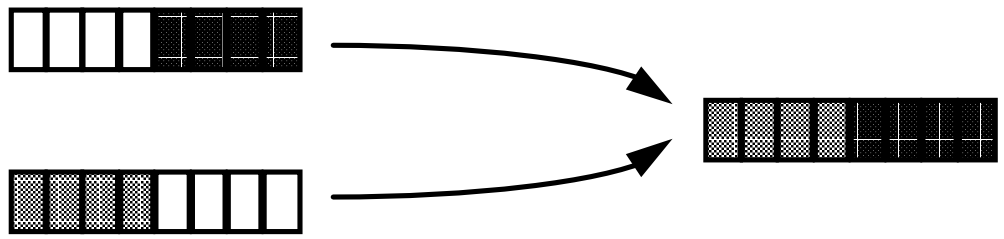
generation *i-1* generation *i* generation *i+1*

fitness improves

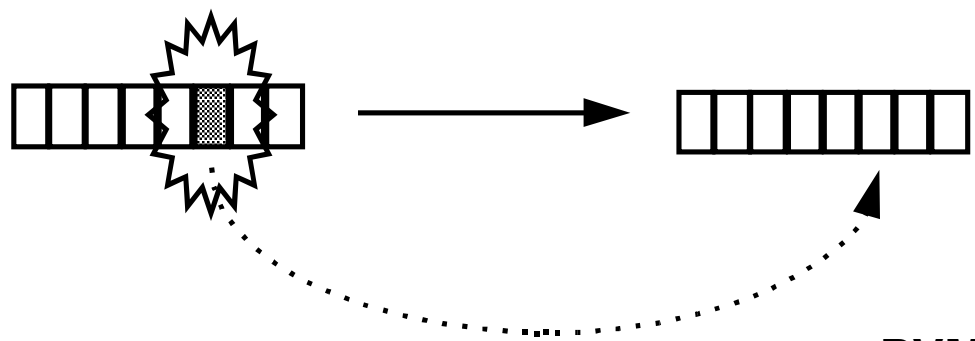
Reproduction Operators

New individuals are created through

- ***cross-over***



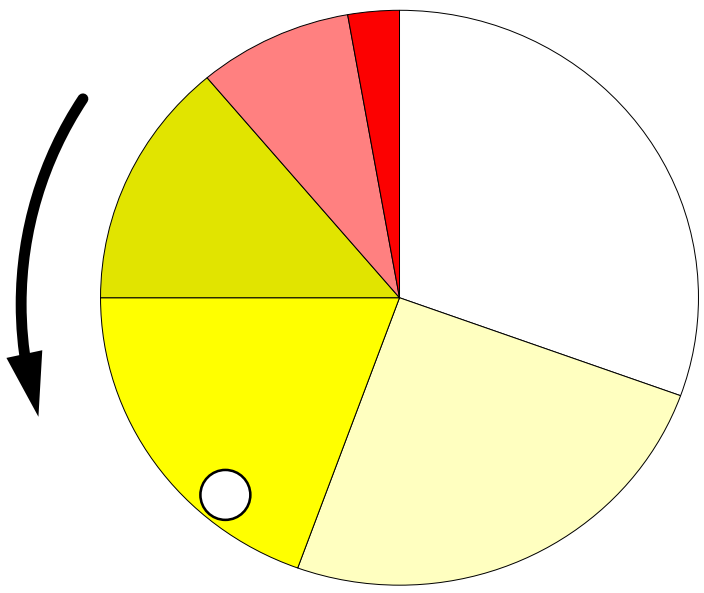
- ***mutation*.**



Individual Selection

The probability of the individuals to be selected for cross-over is proportional to their fitness.

The adopted mechanism is the *roulette-wheel*.



The ATPG Algorithm

A *Genetic Algorithm* is used to generate a test sequence for a single fault.

The whole algorithm loops through 3 phases.

The ATPG Algorithm

Phase 1

Choice of a target fault

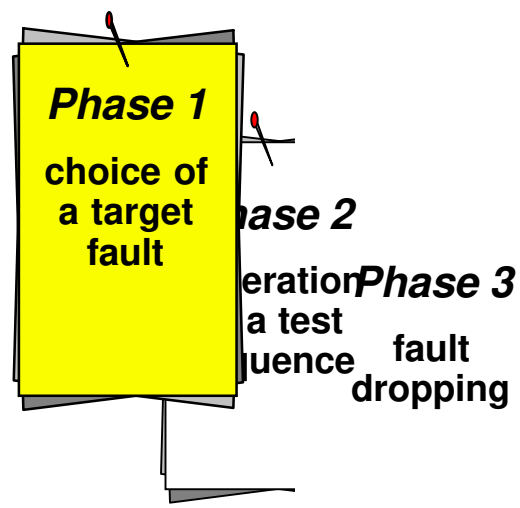
Phase 2

Generation of a test sequence for the target fault

Phase 3

Fault dropping

Phase 1



Main Goal:

- **Chose a *Target Fault***

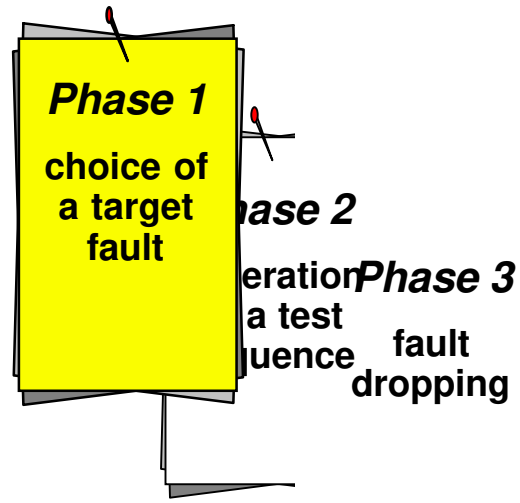
How:

- **Randomly generating sequences**
- **Stopping when the evaluation function for a fault reaches a given threshold**

By-product:

- **Dropping faults.**

Phase 1



Main Goal:

- **Chose a *Target Fault***

How:

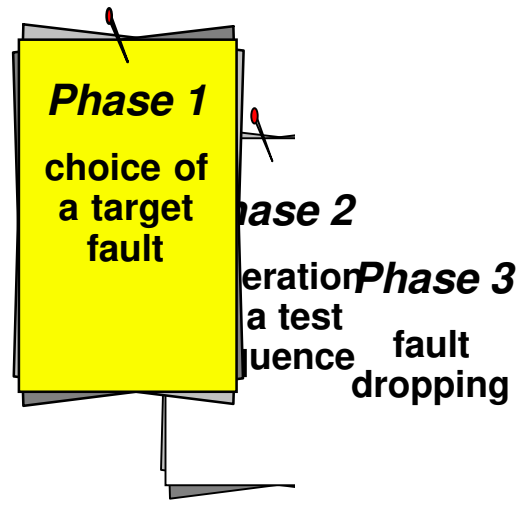
- **Randomly genera ng sequences**
- **Stopping w**
  for a fault r

By-product:

- **Dropping fa**

The fault for which phase 2 is most likely to be able to generate a test sequence.

Phase 1



***Main Goal*:**

- **Chose a *Target Fault***

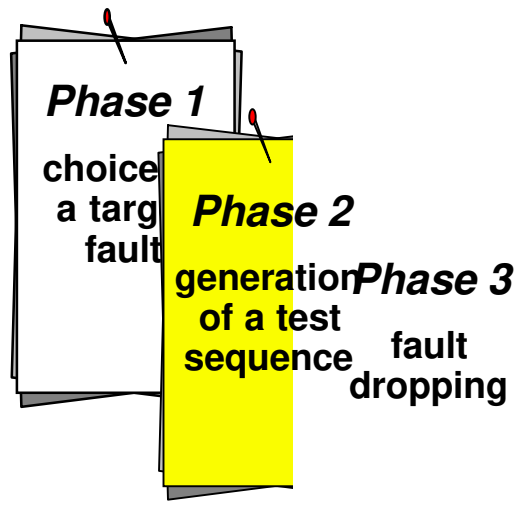
***How*:**

- **Randomly generating sequences**
- **Stopping when the evaluation function for a fault reaches a given threshold**

It evaluates how *close* a sequence is to the detection of a fault:

$$f = k_1 * \Delta FF + k_2 * \Delta gates$$

Phase 2

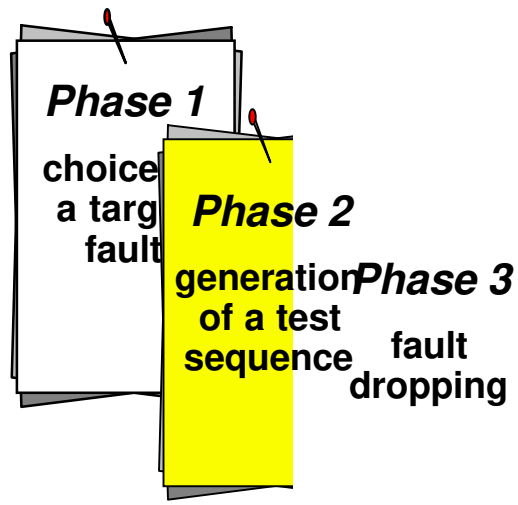


It aims at generating a test sequence for the target fault.

It is based on a Genetic Algorithm:

- **what is an individual ?**
- **what is the fitness function ?**
- **what does the mutation operator do ?**
- **what does the cross-over operator do ?**

Phase 2

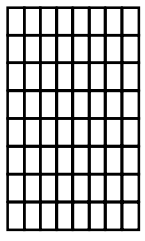


It aims at generating a test sequence for the target fault.

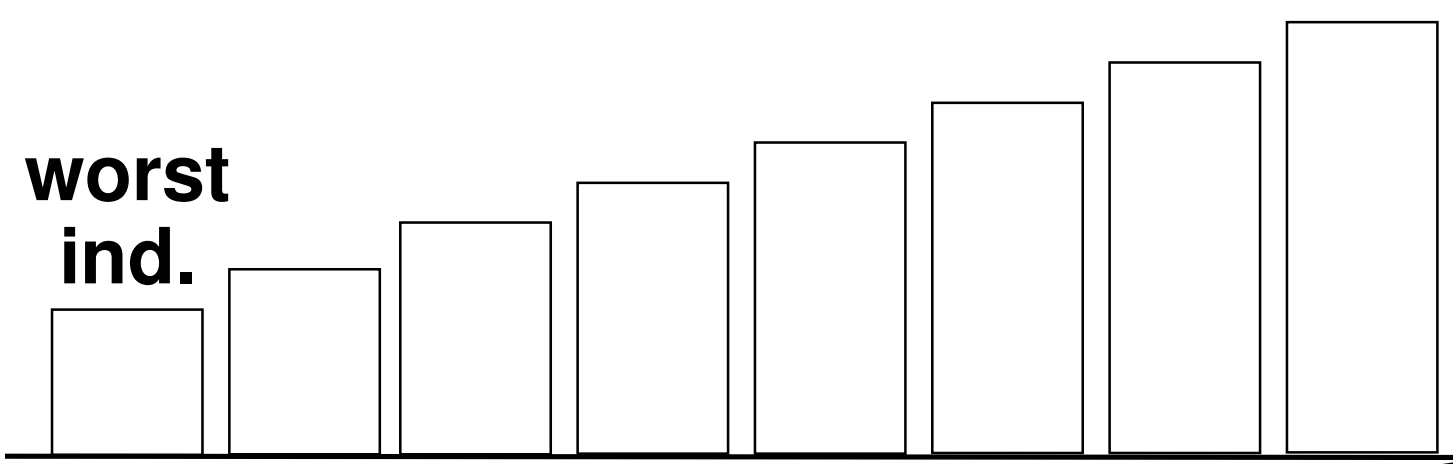
It is based on a Genetic Algorithm:

- **what is an individual ?**
- **what is the fitness function ?**

Each individual is a test sequence of variable length, which starts from the reset state.



It is the *linearized* evaluation function w.r.t. the target fault.

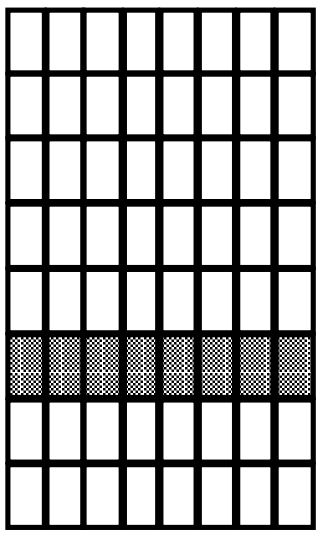


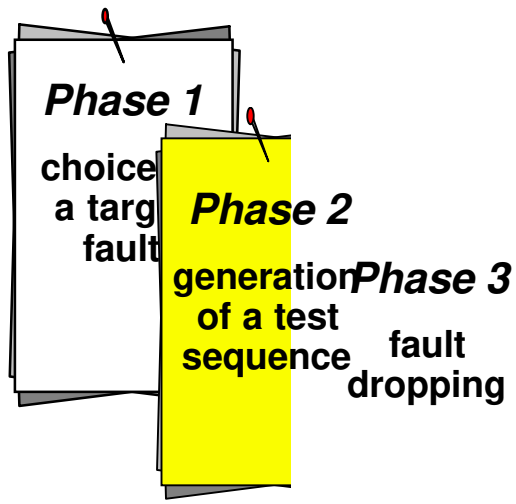
e 3

t
ing

– what is an individual ?
– what is the fitness function ?
– what does the mutation operator do ?
– what does the cross-over operator do ?

It randomly changes one test vector in the sequence.

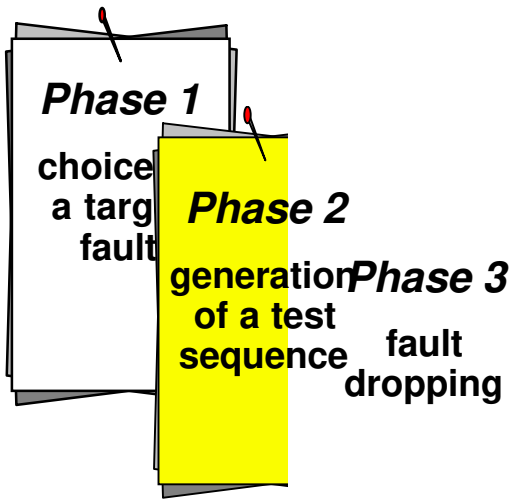




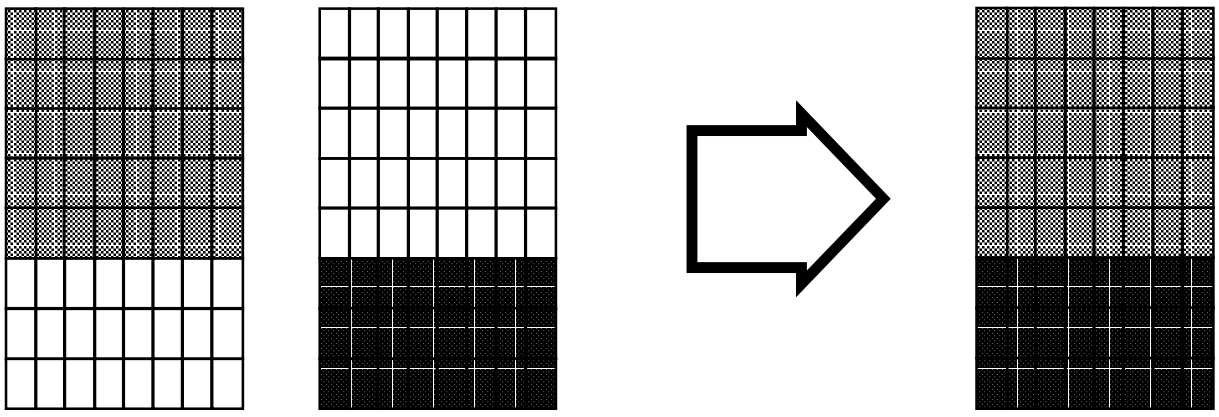
ence for the

thm:

- **what is an individual ?**
- **what is the fitness function ?**
- **what does the mutation operator do ?**
- **what does the cross-over operator do ?**



It combines two test sequences.



ce for the

- what is the fitness fun on ?
- what does the mutation operator do ?
- what does the cross-over operator do ?

Experimental Results

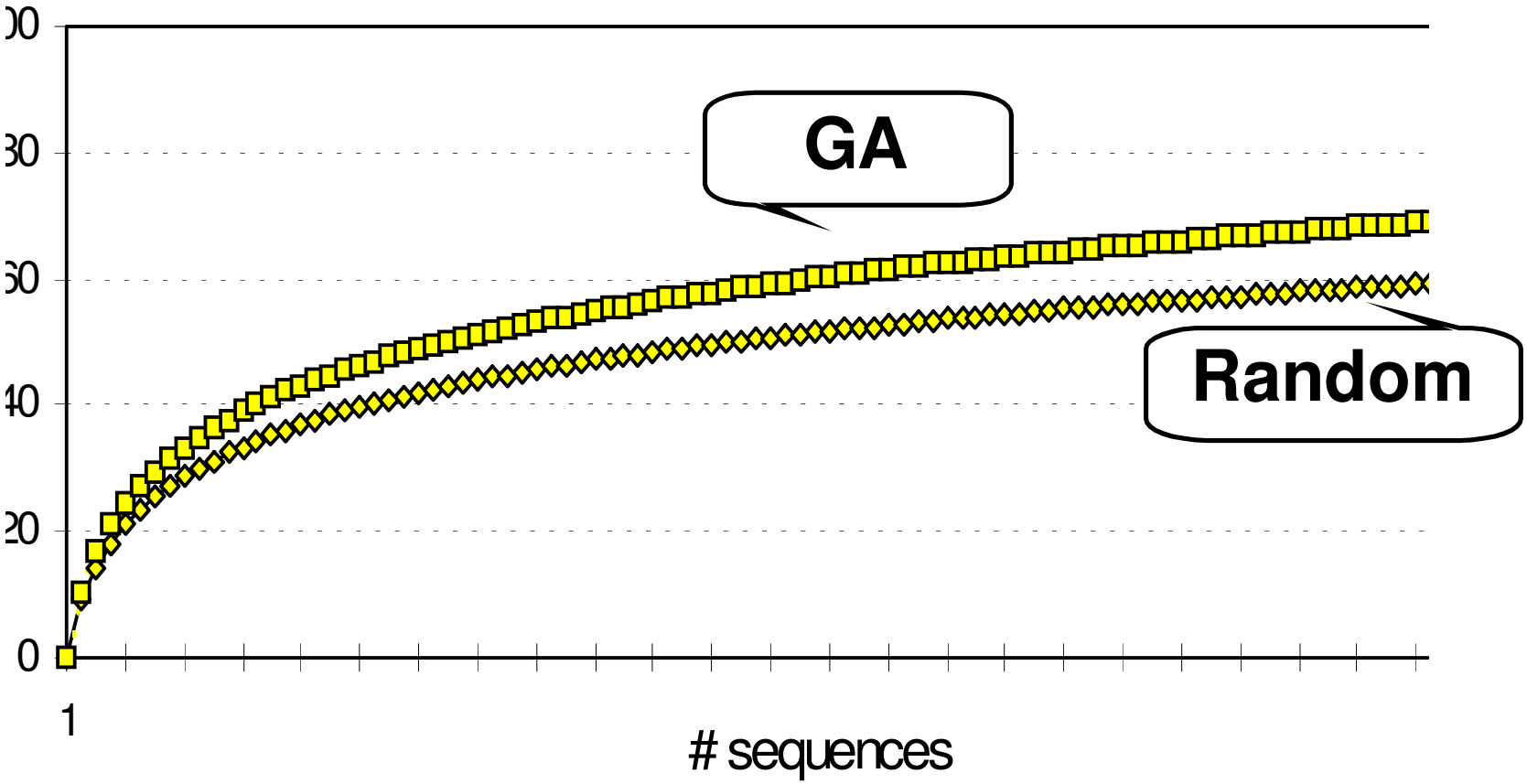
Prototype (named *GATTO*):

- **3,000 lines of ANSI-C code**
- **DEC Alpha AXP 3000/500 workstation with a 32 Mbyte memory**
- **PROOFS-like fault simulator.**

Monoprocessor Version

- All the ISCAS'89 circuits can be handled.
- Results comparable with those produced by the other methods in terms of:
  - CPU time
  - fault coverage
  - test length.

GA vs. Random



Conclusions

GATTO:

- **handles all the largest ISCAS'89 circuits**
- **compares well with the other methods**
- **has reduced memory requirements**
- **allows the user to trade-off CPU time and fault coverage**
- **is flexible.**

Distributed Version

- The mono-processor version spends most of its time in *fault simulation*.
- Simulation can be easily *distributed* among many processors

Distributed Version

- Workstation Network
- Message passing
- Master-Slave Architecture
- PVM environment

Distributed Algorithm

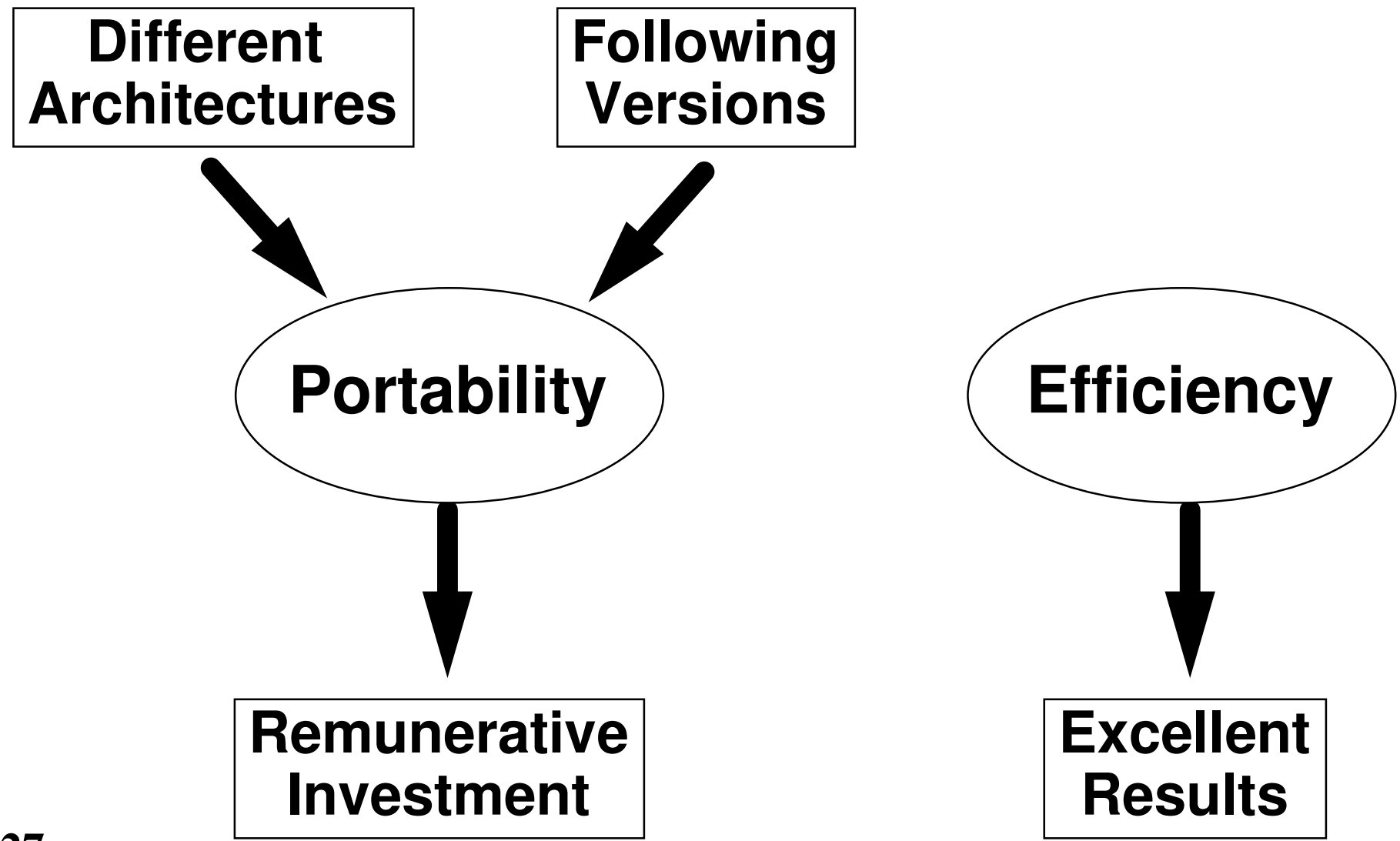
Master

- I/O
- Input Sequence Distribution
- Result receipt
- Global data structures (fault list and test pattern).

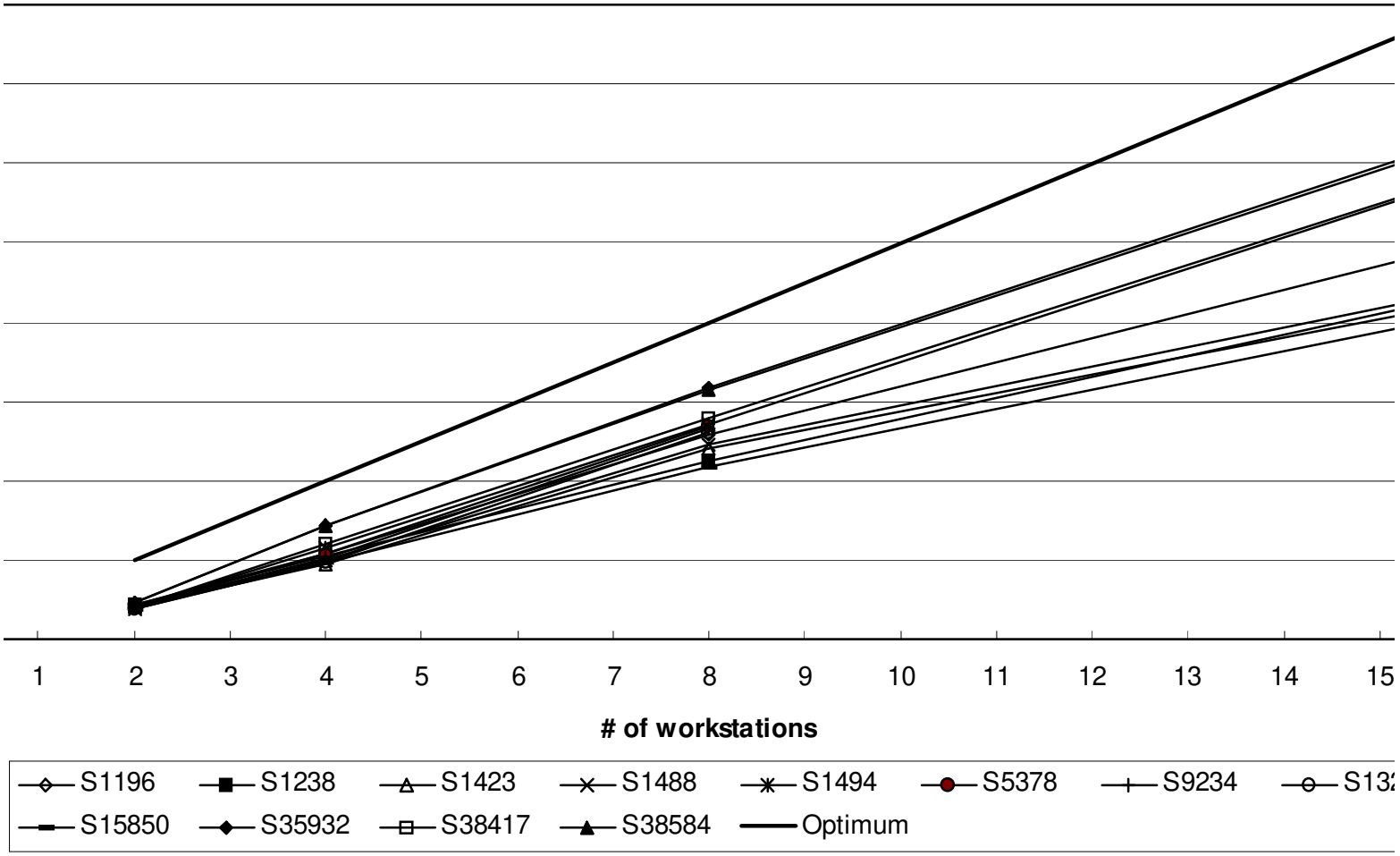
Slave

- Input Sequence Simulation

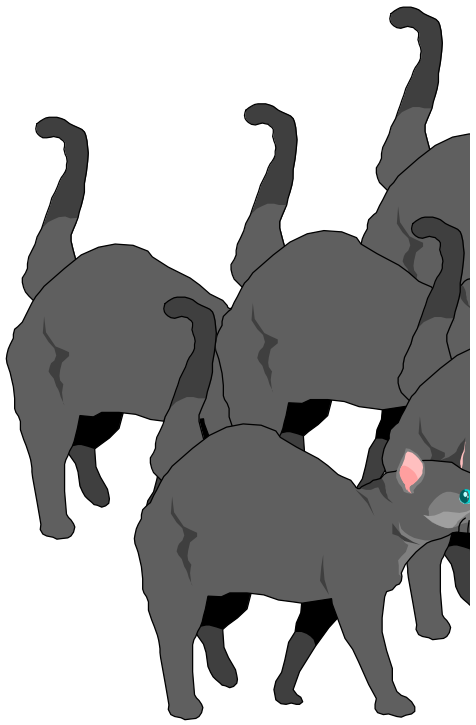
Why PVM ?


Different Architectures
Following Versions
Portability
Efficiency
Remunerative Investment
Excellent Results

Speed-up


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
# of workstations
S1196
S1238
S1423
S1488
S1494
S5378
S9234
S132
S15850
S35932
S38417
S38584
Optimum

GATTO*



– 3,500 lines of ANSI-C code
– PVM library
– Workstation network composed of 16 DEC Alpha AXP 3000/500

Conclusions

GATTO*:

- **portable over a wide range of architectures**
- **it can easily incorporate improvements developed for the mono-processor algorithm**
- **reduced memory requirements**
- **good load balancing**
- **good scalability.**

On-going Work

- Improving the Fitness Function
- Dynamic tuning of GA parameters
- Parallel Version on a CM-5.

Thanks

- **CNR (*Parallel Processing Project*)**
- **CASPUR (Roma)**
- **USTL (Lille)**
- **DEC (Maynard)**