

Challenges of analysis

or

Why WCET analysis does not work now
and will not work in the future

Niklas Holsti

Tidorum Ltd

www.tidorum.fi

Outline: the dark prospects

- Definition of WCET analysis: given an application *program*, and some *time-constrained part* of that program, find an upper bound on the *execution time* of this part on a given *processor*.
- Issues:
 - What is a “program”?
 - What is a “time-constrained part” of the program?
 - What is a “processor”?
 - **Who cares?**
- Greed and anomalies
- Some interesting questions that might be solvable
 - flow analysis only
- Summary

Main reason for WCET analysis problems

- As far as is known (!),
 - SW deadline misses *have not killed anyone*
 - SW deadline misses *have not cost anyone millions* of €, \$, ¥
- Consequently,
 - WCET analysis is seldom a critical requirement
 - HW designers target performance, not predictability
 - SW designers target functionality, not analysability
 - System testers target complex cases, not worst cases
- Why have deadline misses not been fatal?
 - real-time systems are usually very **robust**
 - *occasional* deadline misses are easily *tolerated*
 - eg. Apollo 11 lunar landing
 - real-time systems are usually very **periodic**
 - *systematic* deadline misses usually *found in testing*

So why work on WCET analysis?

- “X-by-wire” in aerospace and automotive
 - increased risk of death & damage
 - or extensive and expensive product recalls (of cars)
- prof. R. Wilhelm, father of aiT & AbsInt, re automotive:
“They now [2010] understand that they need something like this, but now they don't have the money for it.”
- I am an anal-retentive control freak
 - no, really...
 - the intellectual challenge: euphemism?
 - basic programmer anxiety: *do I understand my program?*
 - relieved by making an automatic tool to analyse programs
- Well ok, it is really interesting
 - find **practical, partial analysis** for unsolvable problem
 - balancing act

Worst-case analysis in verification

- Verification often needs worst-case performance analysis
 - but not necessarily by means of WCET analysis tools
 - “state of the art” methods are enough
- As WCET tools become available:
 - the “state of the art” advances
 - verifiers/certifiers may start to require WCET analysis
 - chicken and egg...

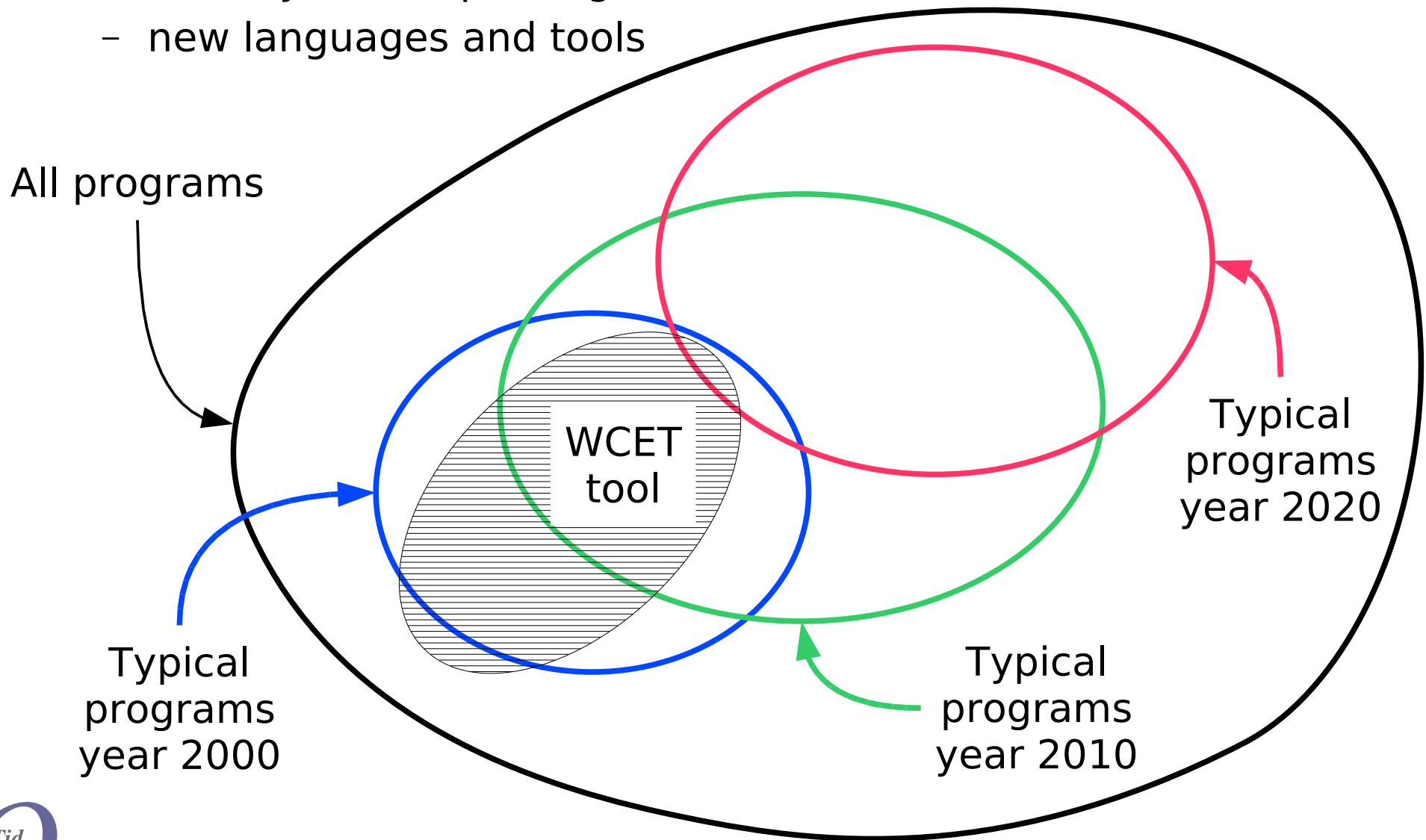


Would WCET analysis have helped?

- Helicopter (Chinook?) crash kills about thirty
 - push-button switch **toggles** engine mode
 - **present mode** indicated by **light in button**
 - sometimes light changes **a few seconds** after button press
 - pilot thinks button not pressed, or did not work
 - pilot presses button **again**, changing mode again
- Therac-25 radiotherapy machine kills three, injures many
 - **timing errors and race conditions** in user interface lead to wrong machine configurations, giving overdoses
- JAS Gripen crashes, two planes lost, pilots survive
 - **pilot-induced oscillation** (PIO)
 - slow response to pilot stick commands
 - pilot increases command, more stick deflection
 - airplane responds much more than pilot intended

Evolution in programs

- Program architecture evolves
 - new styles and paradigms
 - new languages and tools



What is a program?

- Historically:
 - machine code compiled and linked from source code
 - burned into the (EEP)ROM, same in all units
 - invariant during execution, not self-modifying
 - **understood by the programmers**, at least on the source-code level, often on the machine code level too
- Now becoming:
 - a “model” in Matlab/Simulink, UML, or whatever
 - created by 5-10-100-... programmers
 - who **do not understand** how the model is converted into machine code for execution, via C or Java, bytecode, JIT, DLLs, etc, etc.
 - the final machine code may be different depending on the unit, the external and internal conditions, and the phase of the moon, and may change during the execution

Consequences 1: Hiding global control flow

- Only *local* control-flow is visible in C/machine code
 - *global* control-flow only in the model (FSM)
 - code for FSM is an eternal loop with a case statement
 - WCET analysis finds the worst “case” in the loop
 - sequences of FSM states are hidden from flow analysis
- Does it matter?
 - *no*, if the required deadline concerns each FSM step
 - WCET for worst “case” is WCET for any FSM step
 - *yes*, for WCET of a “transaction” with several FSM steps
- Solution?
 - identify the FSM “state” var and its changes in the code
 - import or reconstruct the FSM state graph
 - include state graph in IPET, with connections to CFG
- Analysis of a VM + bytecode: same problem

Consequences 2: More data-dependent flow

- In several ways:
 - virtual function calls depend on object class
 - table-driven routines depend on table contents
 - call-backs depend on call-back pointers
- Present value analysis in WCET tools unsuitable
 - **interval domain poor** for object class, pointer, enum
 - ditto polyhedron domain
- Solution?
 - for static (constant) data: see consequences 4
 - for dynamic (variable) data: see consequences 1?
 - apply “shape analysis” to the data?

Consequences 3: More function pointers

- Reasons for it
 - object-oriented designs (virtual function calls)
 - call-backs to compose “SW components”
 - or to specialize “SW frameworks”
- Problems
 - **call-graph hard to recover** from machine code
 - but the design tool probably knows it very well !
- Why are function pointers so hard to analyse?
 - they are **initialised far away** from their uses
 - they are held in memory, subject to **aliasing**
 - over-estimation has **drastic effects** on the analysis
- Solutions?
 - convince code generators not to use function pointers
 - or generate also the annotations to help WCET tools

Consequences 4: More initialization code

- Running at SW boot:
 - crt0, of course, but also:
 - object constructors
 - registry calls, call-back set-ups
 - HW presence checks & adaptations
- The linked memory image is no longer a good description of the state of the program at execution time
 - analysis of a subprogram/thread must consider the **global state** set up by the boot/init code
- Solution?
 - simulate or execute the boot/init code
 - dump an “execution-ready” memory image for analysis
 - the value-analysis of a WCET tool is almost a simulator

Consequences 5: Inhuman code

- Example: “Averest” model (“synchronous” language)
 - model as concurrent FSMs
 - construct product automaton, generate C code
- Result: single C function with
 - ~ 200,000 instructions, including
 - ~ 20,000 branch instructions
 - Bound-T fails (stack overflow) while building the CFG
- Solutions?
 - ~~shoot~~ educate the translator programmers?
 - develop intra-procedural division into components?
 - one loop
 - one case of a switch
 - one branch of a conditional
 - ugh...

What is a “time-constrained part”?

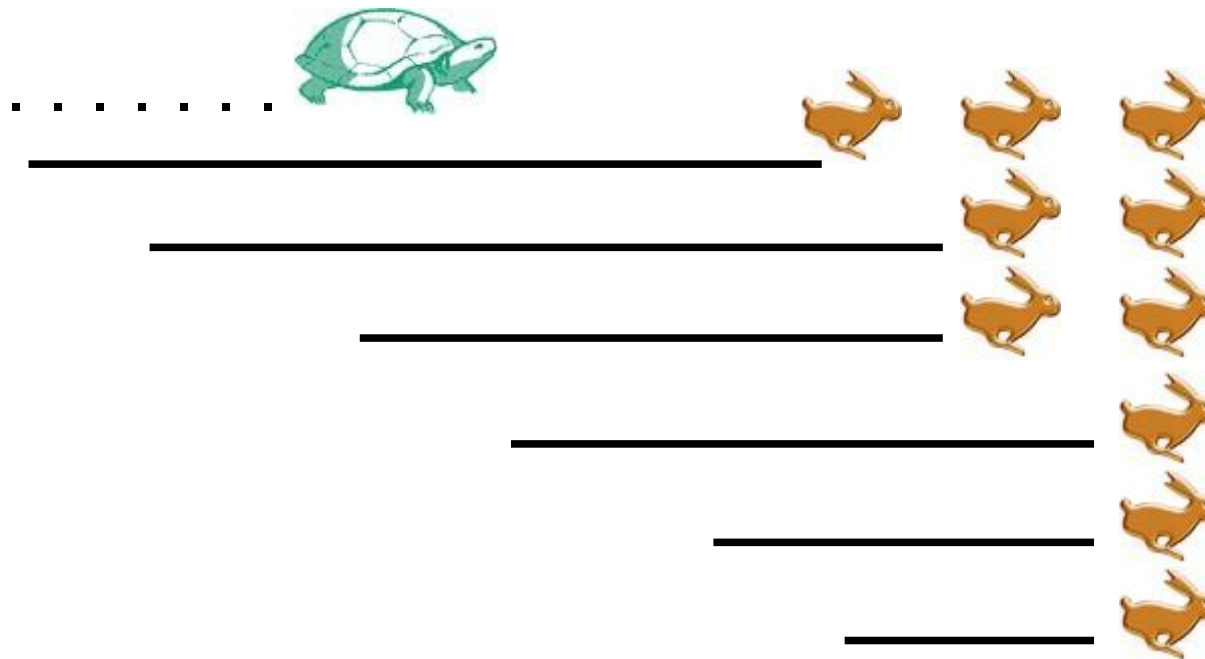
- Historically for WCET analysis
 - one *subprogram* (function)
 - the main function of a thread
 - an interrupt handler
 - a critical (blocking) operation or region
 - anyway, a piece of sequentially executing code
- Now becoming:
 - a *transaction* from input event to response, involving
 - some computations, perhaps on one or more cores
 - some communications over buses/channels
 - some waiting for the above
 - thus, many small pieces of sequential code
 - where does WCET analysis end and schedulability begin?

What is a processor?

- Historically:
 - a machine that executes one sequence of instructions
 - from a standard instruction set for this architecture
 - using a well-defined, stable sequence of cycles / stages
 - fetch, decode, execute, ...
 - same for many applications
- Now becoming:
 - a system of communicating, parallel functional units
 - each with its internal history-dependent state
 - executing several instruction streams
 - in parallel, with dynamic scheduling and ordering
 - with wildly varying execution time per instruction
 - depending also on the implementation of the architecture
 - eg. ARM chips from various manufacturers

The processor race

- The **turtle of analysis** falls behind the **rabbits of processor cores**
- Who also multiply to create multicores...



- Unfortunately, these rabbits will not fall asleep

Can it be analysed statically?

- My impression:
 - static-analysis models exist for many “features”
 - caches, pipelines, branch predictors, ...
 - but not, in practice, for their complex combinations
- State of the art: aiT from AbsInt
 - models the processor as communicating units (FSMs)
 - abstracts *only*:
 - the *cache* (to eg. LRU “ages”)
 - the *values of addresses* (to intervals)
 - no other real abstractions of the whole processor state
 - aiT must *simulate* most possible executions in a BB
 - does not scale to really complex processors (my opinion)
- Solutions? to analysis of such processors
 - none, I believe :-)

Timing anomalies, why?

- Trying to keep all HW units busy at all times = *greed*
 - a delay in one unit (eg. cache miss) delays this and other units, *but also*
 - the state of other units changes in different ways depending on the delay/no delay
 - this changes execution times *later* in unobvious ways
- Hippocratic Oath: “*never do harm to anyone*”
 - if all HW units obey this oath: no anomalies
 - Q: if the memory bus is free, why not use it to prefetch code or data that may be needed later?
 - A: because this could evict other data from the cache
 - use a separate prefetch cache? same problem again?
 - hard to implement
 - *greedy schedulers are sub-optimal* (anomalous)

On greed

- Without timing anomalies, the analysis *can* be greedy:
 - analysis considers only worst case at each choice
 - cache miss worse than cache hit
 - both locally and in total
- If the processor is greedy, the analysis *cannot* be greedy:
 - greed in processor causes timing anomalies
 - analysis must consider all choices
 - both cache miss and cache hit
 - and all their future effects

Sad example

- Photolithography machine (ASML, Netherlands)
- Rapid and accurate motion of large, heavy parts
 - to project chip circuitry from mask to semiconductor die
 - many (100s) identical chips per die
- About 10 high-end processors control the machine
 - much attention to speed, monitoring, etc.
 - **cache warming**
- BUT **still timing problems**
 - on deadline overrun:
 - activate recovery code
 - **lose (destroy) only the current chip**, not the whole die
- **“Worst-case analyses useless...overestimation...”**

Final insult...

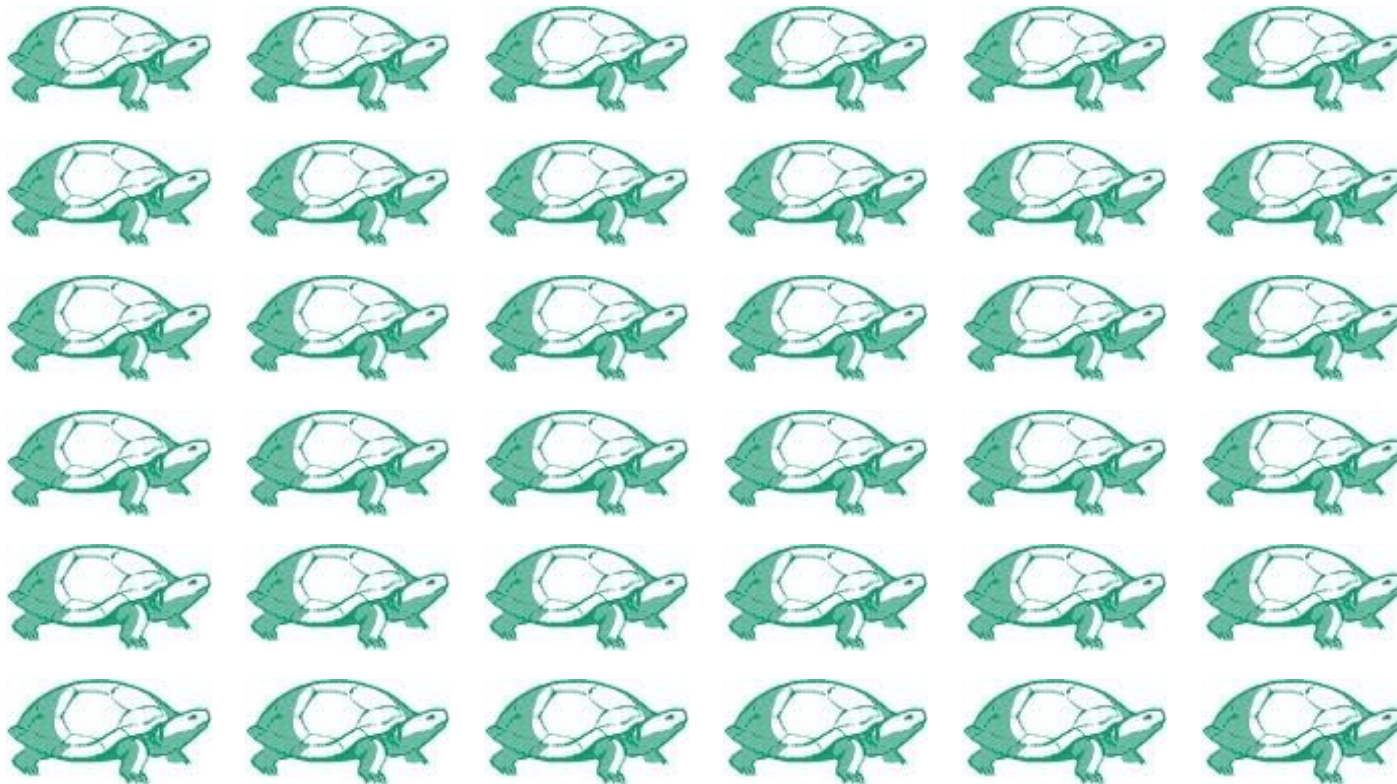
- Asynchronous processors
 - **no clock !**
 - each logic signal comes with a handshake
 - “relay race”, computations go as fast as possible
 - ET depends on voltage and temperature, etc.
 - ET depends on data values
- Advantages:
 - low-noise operation
 - no clock / power transients on signals
 - perhaps low-energy operation
 - only those FF's change that need to
- WCET analysis?
 - static analysis unsafe without large over-estimates

Special processors for hard RT?

- To be predictable and analysable
- Scratchpads, lockable caches, ...
 - static allocation limits size of fast memory
 - especially difficult for multi-threaded systems
 - easiest to analyse if different instructions for fast memory
 - eg. Intel 8051 “internal” and “external” memory space
 - but complex to program (eg. pointers to either space)
- Suggestions:
 - multicore with predictable cores (thus rather slow)
 - perhaps a bit of VLIW for compile-time scheduling
 - plenty of local memory per core
 - all memory accesses can be analysed as fast
 - no shared caches
 - all use of shared or off-chip resources analysed as I/O
 - in the schedulability analysis (“not my problem” :-)

Turtles all the way

- An array of mostly independent, analysable turtles



XMOS? www.xmos.com ... they have a WCET tool, too ...

Measurement-based methods

- End-to-end, ad-hoc or existing tests: traditional “method”
 - **unknown & unknowable underestimation** (if black-box)
- End-to-end, automatic black-box test generation
 - heuristic maximum-finding of unknown function... ditto
- End-to-end, coverage-controlled tests (glass box)
 - can find ET of program parts (with almost no probe effect)
 - hard to find ET variation of program parts
- Detailed (BB) measurement, coverage-controlled tests
 - can measure ET variation of program parts
 - **no theory for “sufficient” coverage** (my opinion)
- IPET with (worst) observed BB times (“hybrid method”)
 - **best of the measurement-based methods**
 - loop bounds still a problem (and other control flow, too)
 - **no theory for error distribution / risk** (my opinion)

How bad can the cache be?

- Example: assume 4-way associative LRU code cache

```
loop
  proc1;
  if cond2 then proc2a; else proc2b; end if;
  proc3;
  if cond4 then proc4a; else proc4b; endif;
  proc5;
end loop;
```

- Assume no loops or calls in proc1..proc5
- A change in a *single bit* (cond2 or cond4) can change code cache hit rate from *100% to 0%*
 - if five called procedures all map to the same cache lines
- Testing can cover all calls and all branches without testing the (single) path that gives 0% hits
- All other paths can give 100% hits

Come on, that is very unlikely

- Admitted (for the 100% to 0% case)
- But:
 - a cache miss can take ~ 100 cycles or more
 - **one % point increase in miss rate can ~double the ET**
 - eg. increase from 1% misses to 2% misses
 - good-bye and thanks for all the fish...
 - unless we do something useful while waiting for cache fill
 - which leads to the complex processors with anomalies
 - and not always possible even for them
- How can we possibly compute the risk?
 - risk estimates (eg. for RapiTime) are based on *assumed stochastic independence* of ETs of different BBs
 - how can one know if they really are independent?
 - this loop is a counterexample

Flow analysis: work to do

- Loop bounds for single loops
 - several methods, some good ones, none perfect
- Correlations between different loops
 - some methods for nested loops, eg. “triangular” loops
 - Stefan's “census” method, for example
 - no (?) methods for **correlated separate loops**
- Example: insert element in sorted vector
 - loop to find the insertion point;
 - insert;
 - loop to shift the rest of the elements up;
- If N elements in vector:
 - both loops iterate at most N times, so $2N$ in total
 - but in fact sum of loop iterations is at most N .
- Can be annotated, of course. Analysis?



Post-context for calls

- Many WCET tools use “context” to analyse calls
 - variable values **before** a call can **influence** loop bounds and paths in the callee, thus the WCET for the call
- Sometimes we could use a “post-context”
 - variable values **after** a call can **report** what happened in the callee, give post-facto bounds on the WCET for the call

```
procedure Try_It (Done : out Boolean) is
begin
  if <??> then Done := False;
  else          Compute_A_Lot;
               Done := True;   end if;
end Try_It;

...
Try_It (Done);
if not Done then <did NOT Compute_A_Lot>...
```

Infeasible paths in general

- Unstructured problem
 - little work on classification of types of infeasible paths
 - attempt (A. Holsti):
 - **local** (intra-procedural) path
 - **non-local** (inter-procedural) path
 - **over-iteration** path (loop cannot repeat so many times)
 - **intra-repeat** path (within one iteration of loop)
 - **inter-repeat** path (over one or more iterations of loop)
 - **loop-entering** path
 - **loop-exiting** path
- Practical importance not well known
 - easy to construct examples with huge effects
 - my experience: sometimes very important, sometimes not

Time-introspective programs

- Conditional branches that depend on execution time

```
if (ET of this thread so far) > 100 ms then
    use_fast_sloppy_method;
else
    use_slow_precise_method;
end if;
```

- This does happen in some programs
 - time-outs
 - detecting risk of overrun (as above)
 - application-defined scheduling, time slices, ...
- **Ties the present WCET-analysis-flow into knots**
 - estimated “ET so far” influences control flow
 - seems impossible to model in IPET

Summary

- **WCET analysis is practical now only for relatively simple programs on relatively simple microcontrollers**
 - “simple” does not imply “small”
 - highly critical systems: aerospace, automotive, nuclear
- Static analysis of worst-case processor behaviour seems **hopeless** for high-end, general processors
 - open: are predictable but powerful processors possible?
- Msmt-based analysis is **unreliable** for the same reasons
 - but more reliable than end-to-end measurements
- **Flow analysis** has promising problems to work on
- Increased use of static analysis for bug-finding etc.
 - may push programs to be more analysable
- **Existence of WCET tools pushes the “state of the art”**
 - may make WCET analysis required for critical SW