TOHOKU UNIVERSITY

Autotuning meets Code Transformations
-- A case study of Xevolver framework --

24th Workshop on Sustained Simulation Performance
December 6, 2016@HLRS

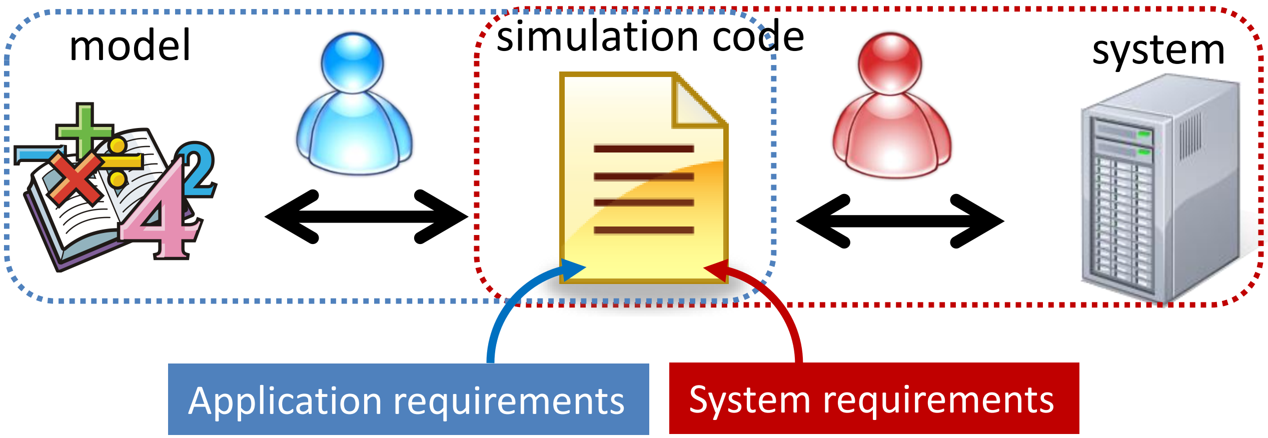
Hiroyuki TAKIZAWA
Tohoku University

Project Background

- HPC application development
  = team work of programmers with different concerns

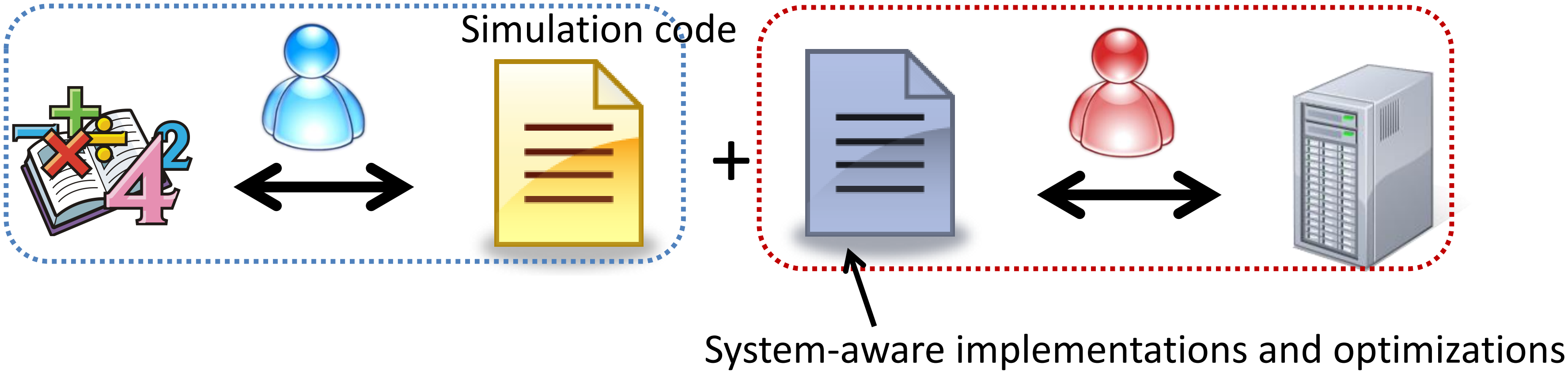
Application developers **(= computational scientists)**
- write a program so as to get correct results
→ Main concern: relationship between **simulation models** and **programs**.

Performance engineers **(= computer scientists/engineers)**
- write a program so as to get high performance
→ Main concern: relationship between **programs** and **computing systems**.



Goal = Appropriate Division of Labor

Separation of system-awareness from application programs



There are many approaches to abstraction of system-awareness

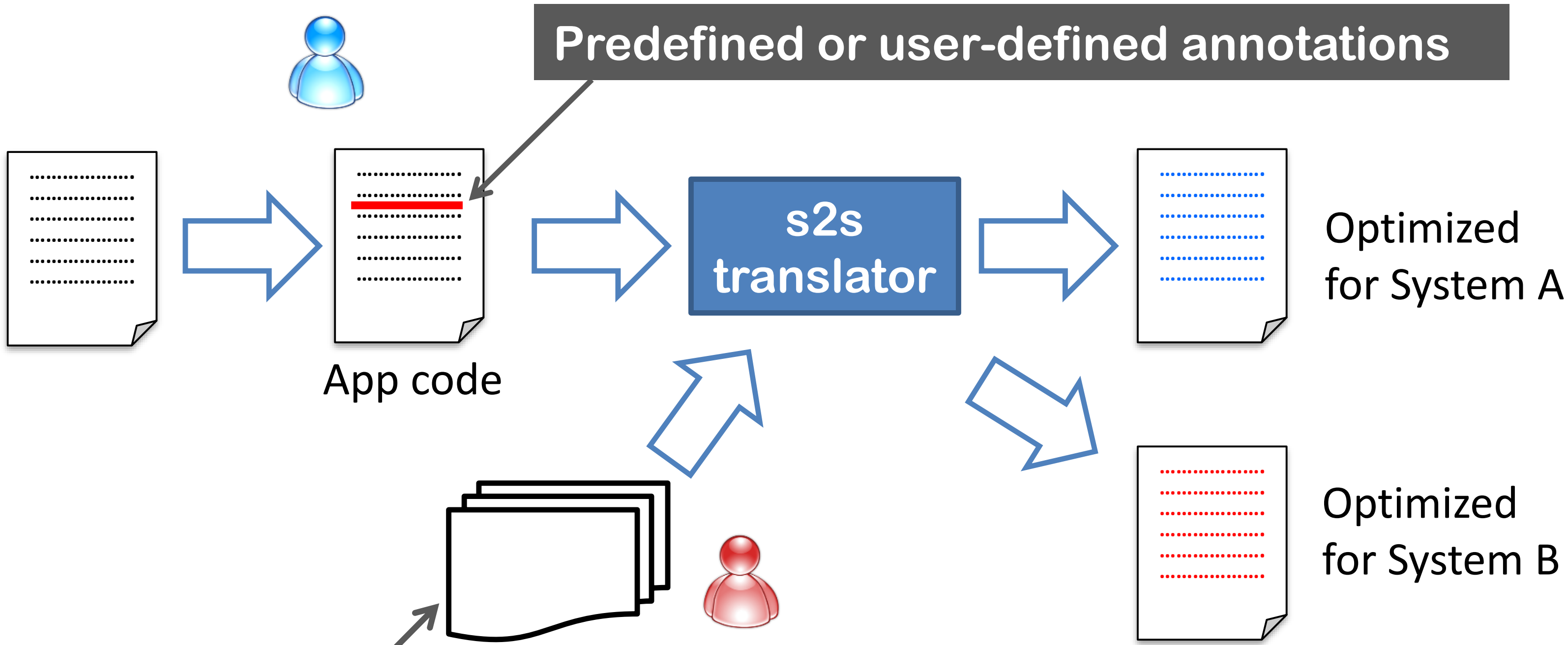
- System-aware implementations with a common interface = Numerical libraries
- Standardized programming models and languages = MPI, OpenMP, OpenACC …

In reality, we still need to modify a code to achieve high performance for application-specific and/or system-specific reasons.
→ How can we abstract such code modifications?

Xevolver Framework

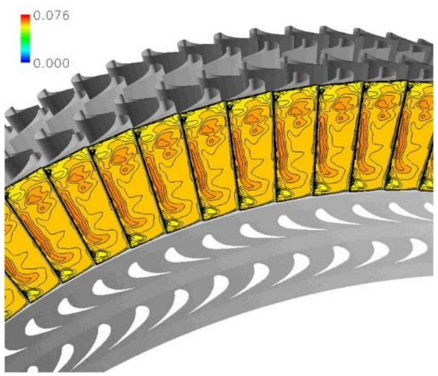
TOHOKU UNIVERSITY

Various transformations are required for replacing arbitrary code modifications.
= cannot be expressed by combining predefined transformations.
→ Xevolver : a framework for custom code transformations



Translation rules
- Define the code transformation of each annotation
- Different systems can use different rules
- Users can define their own code transformations

How to Describe Transformation Rules



Numerical Turbine (Yamamoto et al.)

- A real-world application written in Fortran
  - Long history of development
- Optimized for NEC SX-9 system
  - Maximizing innermost loop parallelism
- 44 kernel loops have almost the same structure
  - OpenACC compiler cannot exploit the loop parallelism

→ **44 loops must be modified in the same way.**

```
program nt_opt
!$xev tgen var(i1,i2,i3,i4,i5,i6,if) stmt
!$xev tgen list(body) stmt
!$xev tgen var(lstart,lend,II2,IIF) exp
!$xev tgen condef(has_doi) contains stmt begin
 DO I=II2,IIF
!$xev tgen stmt(if)
!$xev tgen stmt(body)
 END DO
!$xev tgen end
!$xev tgen list(stmt_with_doi) stmt cond(has_doi)
!$xev tgen src begin
 DO L=lstart,lend
!$xev tgen stmt(stmt_with_doi)
 END DO
!$xev end tgen src
!$xev tgen dst begin
 DO I=1,inum
   DO L = lstart, lend
     IF (I .GE. IS(L) .AND. I .LE. IT(L)) THEN
       EXIT
     END IF
!$xev tgen stmt(if)
!$xev tgen stmt(body)
  END DO
 END DO
!$xev end tgen dst
end program nt_opt
```

Code Pattern before Transformation

Code Pattern after Transformation

OpenACC-friendly version

Your feedbacks are welcome!

TOHOKU UNIVERSITY

- Xevolver is online available
  - Visit http://xev.arch.is.tohoku.ac.jp for more details.

Legacy Code

- We have a lot of **legacy** HPC applications…

  Those applications may or may not work on a future HPC system.
  Anyway, **we will be unable to expect high performance** of them.

  – Low-level languages (e.g. C and Fortran)
    - The code has mostly been written by application developers.
    - There is no chance for performance engineers to select languages unless the code is rewritten.

  – Long development history
    - A lot of programmers have been involved in the development.
      – No one has a holistic understanding of the code.
      – Performance-sensitive code fragments are scattered over the whole code.

  – "Legacy" means "important"! -- reliable and useful apps
    - This is why the code has been maintained for a long time.
    - It has been proven to produce correct results.

  → Application developers want to avoid drastic modifications.

Importance of Autotuning

- **Performance Tuning for Future HPC Systems**
  - The complexity and diversity of **HPC system architectures** are increasing.
    - Individual systems may potentially require different parallelization methods, programming models, languages, etc.
  - The complexity and scale of **practical applications** are also increasing.
    - Individual applications may potentially require different algorithms, performance tuning and/or maintenance strategies, etc.

It is difficult to estimate performance without executing the code.
= various options need to be examined in a try-and-error fashion.
→ Automation of such an empirical tuning process = **Autotuning**

The idea of AT is simple

1. Assume the target code has some parameters
   - ✓ The performance of the code changes by adjusting the parameters.
2. Tune the parameters
3. Evaluate the performance
4. Repeat Steps 2 and 3 until an acceptable parameter configuration is found
   - ✓ A key is how to adjust parameters so as to quickly reach an optimal or suboptimal configuration

Can we assume a legacy code has such parameters? No, at all…
We have to make a legacy code auto-tunable for auto-tuning the code.

Auto-tunable Code

maintain the original code.

```
DO i=0,n
  DO j=0,n
    sum = c(j,i)
    DO k=0,n
      sum = sum+a(k,i)*b(j,k)
    END DO
    c(j,i) = sum
  END DO
END DO
```

Optimize!

```
DO i1=0,n,BLOCK_SIZE1
  DO j1=0,n,BLOCK_SIZE2
    DO k1=0,n,BLOCK_SIZE3
      DO i=i1,n+BLOCK_SIZE1
        DO j=j1,j1+BLOCK_SIZE2
          sum = c(j,i)
          DO k=k1,k1+BLOCK_SIZE3
            sum = sum+a(k,i)*b(j,k)
          DO END
          c(j,i) = sum
        END DO
      END DO
    END DO
  END DO
END DO
```

Auto-tuning is used to efficiently determine BLOCK_SIZE*.

- ✓ Application developers need to maintain the complex auto-tunable version.
- ✓ A custom code modification **on a case-by-case basis** is needed because there is no universal way to make a code auto-tunable.

AT meets Code Transformations

[1] Ansel et al.@PACT2014
[2] Takizawa et al.@HiPC2014

- **OpenTuner[1] = Autotuning framework**
  - Performance engineers can efficiently explore a huge parameter space, and quickly find an appropriate parameter configuration, **only if the target code is auto-tunable**.
- **Xevolver[2] = Code transformation framework**
  - Performance engineers can make a legacy code auto-tunable **without messing it up**.

Their combination enables auto-tuning of a legacy code while keeping it maintainable.

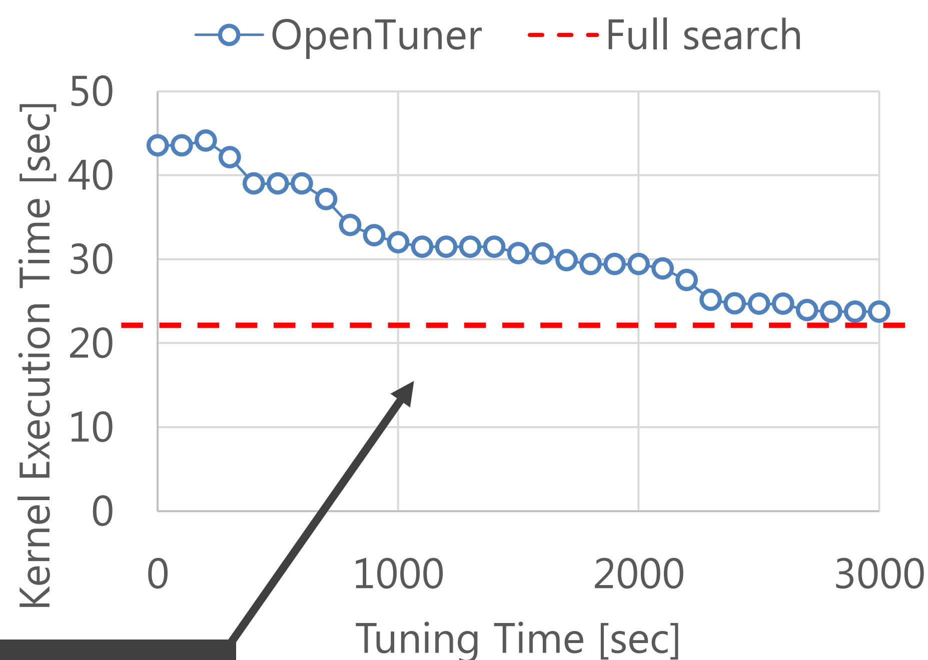
Reduction in Tuning Time

- The benefit of auto-tuning is **clear**
  - Auto-tuning Himeno benchmark

Experimental Setup

System
- CPU : Intel(R) Xeon(R) CPU E5-2630@2.30GHz
- Mem : 8 Gbytes
- OS : CentOS 6.4
- Compiler: GNU Fortran 4.7

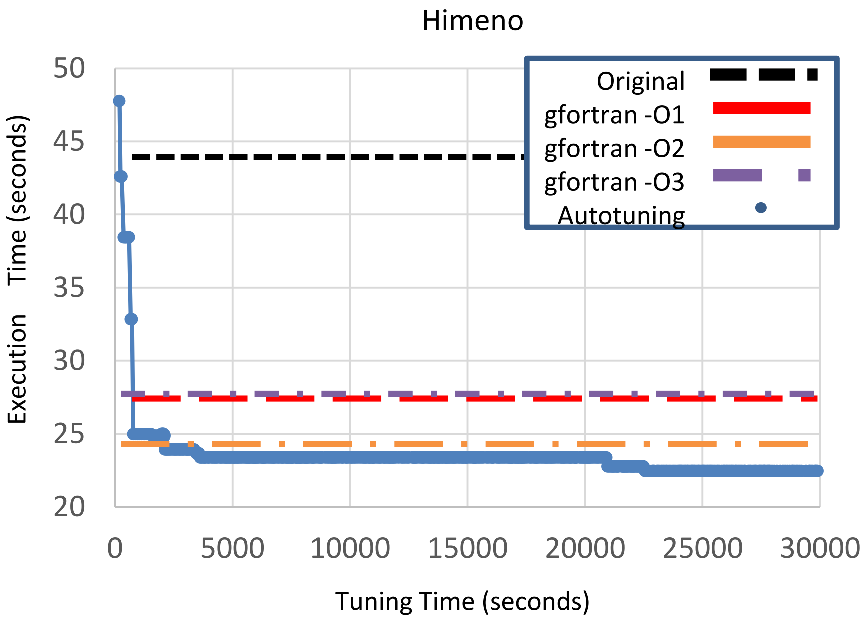
Tuning parameters
- Loop blocking, loop collapse, or no loop optimization (original).
  - ✓ For loop blocking, the block size is also determined.
- Discrete arrays (original) or an array of structures.
- 426 compiler options
  - ✓ -O0/-O1/-O2/-O3
  - ✓ -fexpections, -fwrapv, -funsafe-math-optimizations
  - ✓ -funroll-loops, … etc

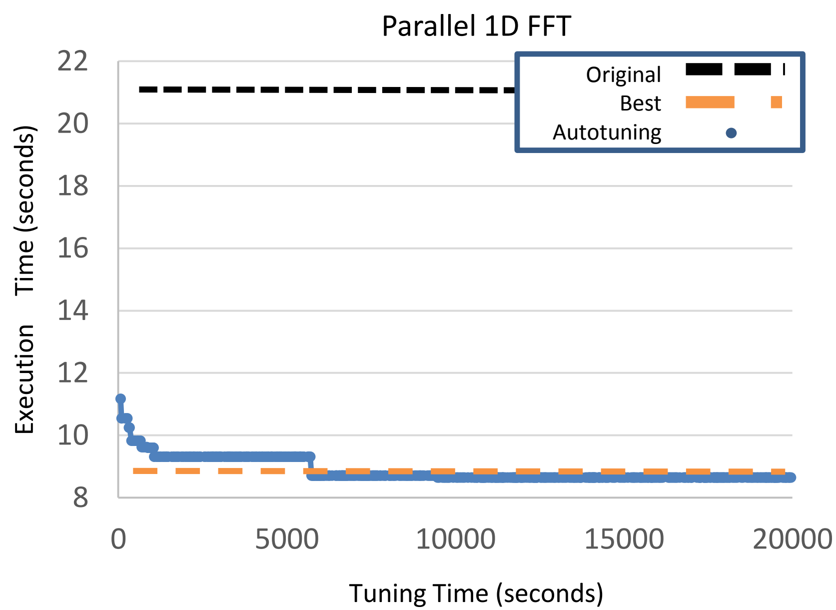


While full search takes **71,944** sec., OpenTuner can achieve almost the same performance in about **3,000** sec. (4.2%).

Achieved Performance

TOHOKU UNIVERSITY





1.6x higher performance for Himeno benchmark.

✓ The autotuned version outperforms the Himeno code compiled with –O2 and –O3 options.

2.3x higher performance for parallel 1-D FFT.

Auto-tunable Himeno Kernel

- Auto-tunable code is likely to be **messy**
  - Even a simple loop nest becomes very complicated if various optimizations are taken into account.

Original

```
do loop=1、nn
  gosa= 0.0
  do k=2、kmax-1
     do j=2、jmax-1
        do i=2、imax-1
           s0=a(I、J、K、1)*p(I+1、J、K) &
                +a(I、J、K、2)*p(I、J+1、K) &
                +a(I、J、K、3)*p(I、J、K+1) &
                +b(I、J、K、1)*(p(I+1、J+1、K)-p(I+1、J-1、K) &
                            -p(I-1、J+1、K)+p(I-1、J-1、K)) &
                +b(I、J、K、2)*(p(I、J+1、K+1)-p(I、J-1、K+1) &
                            -p(I、J+1、K-1)+p(I、J-1、K-1)) &
                +b(I、J、K、3)*(p(I+1、J、K+1)-p(I-1、J、K+1) &
                            -p(I+1、J、K-1)+p(I-1、J、K-1)) &
                +c(I、J、K、1)*p(I-1、J、K) &
                +c(I、J、K、2)*p(I、J-1、K) &
                +c(I、J、K、3)*p(I、J、K-1)+wrk1(I、J、K)
           ss=(s0*a(I、J、K、4)-p(I、J、K))*bnd(I、J、K)
           GOSA=GOSA+SS*SS
           wrk2(I、J、K)=p(I、J、K)+OMEGA *SS
        enddo
     enddo
  enddo
```

Auto-tunable

Discussions

- **Productivity**
  - Code transformation rules : **102** lines in total
    - One rule file of 51 lines for loop transformation
    - Another rule file of 51 lines for data layout optimization
  - Auto-tunable Himeno code : **185** lines in total
    - The kernel becomes **6.5**x longer than the original one.
- **Benefits from the combination**
  - We can use AT while keeping the original code unchanged
  - Even for a small benchmark, the total number of transformed code lines is larger than that of lines for transformation rules
    - Generally, a practical application has more kernel loops.

Both maintainability and autotunability are achieved.

Summary

- Happy Marriage of Autotuning and Code Transformation
  - Autotuning can adapt one code to individual systems.
    - The number of code transformation rules can be reduced because similar systems can share some rules.
  - Code transformation can avoid degrading the code maintainability.
    - Application developers need to care about only the original code.

Future Work

- An interface is needed for effective collaboration of autotuning and code transformation.
  - They have been so far developed independently.
    - Every parameter needs to be described in two different configuration files. → **redundant** and **error-prone**

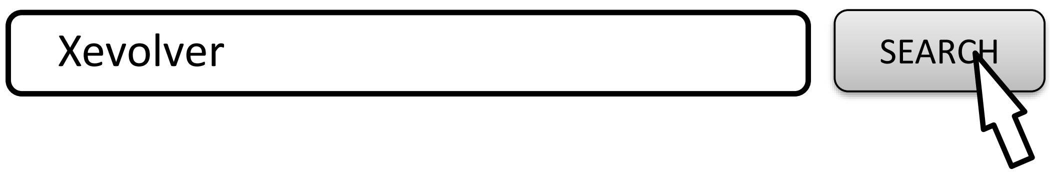
A part of autotuning scenario file for auto-tunable Himeno code

```
class UserDefinedTuner(MeasurementInterface):
  def manipulator(self):
    manipulator = ConfigurationManipulator()
    manipulator.add_parameter(
      PowerOfTwoParameter(‘BLOCK_SIZE’, 1, 128))
    manipulator.add_parameter(
      IntegerParameter('VARIANT', 0, 5))
  return manipulator
```

```
def run(self, desired_result, input, limit):
    cfg = desired_result.configuration.data
    gcc_cmd = 'gfortran himenoBMT.f90 -o ./tmp.bin'
    gcc_cmd += '-Dvariant[0]'.format(cfg['VARIANT'])
    gcc_cmd += '-DBLOCK_SIZE=[0]'.format(cfg['BLOCK_SIZE'])
    compile_result = self.call_program(gcc_cmd)
    assert compile_result['returncode'] == 0
    run_cmd = './tmp.bin'
    run_result = self.call_program(run_cmd)
    assert run_result['returncode'] == 0
    return Result(time=run_result['time'])
```

Danke!

- **Acknowledgements**
  - This work was supported by JST Post-Peta CREST.



Xevolver with some sample translation rules is online available at http://xev.arch.is.tohoku.ac.jp.

Your feedbacks (and bug reports) are welcome!