Optimization Using Tracing JITs

Or why computers, not humans, should optimize code

Wim Lavrijsen

Future Computing in Particle Physics Workshop
June 15-17, e-Science Institute, Edinburgh

Outline

- Overview of Tracing Just-In-Time Compilation
- Benefits of tracing JITs
  - Within the context of ATLAS software
- The case for Python and PyPy
- Current work on Cppyy
  - C++ dictionaries in PyPy
- Ideas and plans for future work
  - HEP-ification of the JIT

"Classic" v.s. Tracing JIT

- "Classic" just-in-time compilation (JIT):
  - Run-time equivalent of the well-known static process
    - Profile run to find often executed methods ("hot spots")
    - Compile hot methods to native code
  - Typical application for interpreted codes
- Tracing just-in-time compilation:
  - Run-time procedure on actual execution
    - Locate often executed loops ("hot paths")
    - Collect linear trace of one path ("loop iteration")
    - Optimize that linear trace
    - Compile to native if applicable
  - Used both for binary and interpreted codes

Tracing JIT

Program code:

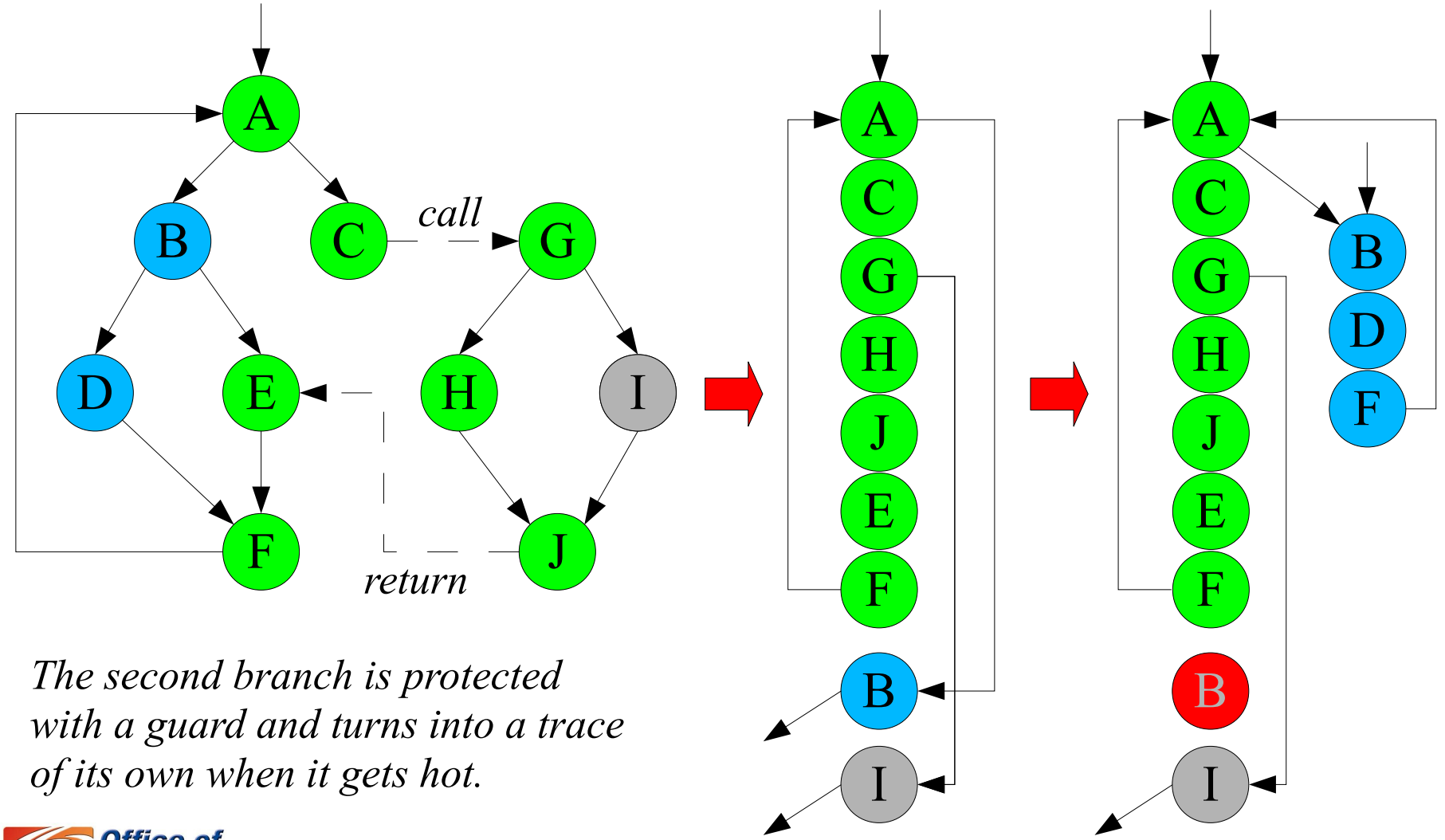
```
     A:
L:   cmp
     inst_a1
     inst_a2
     jne aa          →      call C:
     Call            →      B:
                            inst_b1
                     ←      return
     inst_aN
     goto A
```

Linear trace:

inst_a1, inst_a2, G(aa), inst_b1, inst_aN

- In interpreted mode:
  - Process user code
  - Identify backwards jumps
  - Collect trip counts
- If threshold crossed:
  - Collect linear trace
  - Inject guards for all decision points
  - Optimize trace
  - Compile trace
  - Cache & execute
- In compiled mode:
  - Process user code
  - Collect trip counts on guards
- If threshold crossed for guards:
  - Create secondary trace
  - Rinse & repeat

Traces, Guards, Branches



The second branch is protected with a guard and turns into a trace of its own when it gets hot.

Tracing JIT Optimizations

- To the linear trace itself, e.g. for guards:
  - Removed if implied, strenghten for larger role
- Loop unrolling and function inlining
- Constant folding and variable promotion
  - Much more effective at run-time than statically
- Life-time and escape analysis:
  - Move invariant code out of the loop
  - Place heap-objects on the stack
- Load/store optimizations after address analysis
  - Collapse reads, delay writes, remove if overwritten
- Parallel dynamic compilation

Examples of Tracing JITs

- Dynamo for PA-RISC binary
- PyPy's meta-JIT for Python
- MS's SPUR for Common Intermediate Language
- Mozilla's TraceMonkey for JavaScript
- Adobe's Tamarin for Flash
- Dalvik JIT for Android
- HotpathVM for Java
- LuaJIT for Lua

First International Workshop on Trace Compilation (IWTC), being organized for August 24-26 in Denmark

Dynamo (PA-RISC)

- Of interest because it's a tracing JIT on binary
  - User-mode and on existing binaries and hardware
    - No recompilation or instrumentation of binaries
  - Run-time optimization of *native* instruction stream
- Gained over static compilation because:
  - Conservative choice of production target platforms
    - Incl. legacy binaries existing on end-user systems
  - Constraints of shared library boundaries
    - Actual component to run only known at dynamic link time
    - Calls across DLLs are expensive
  - Linear traces simpler to optimize than call graphs

Benefits of Tracing JIT (1)

- Profile on current and actual input data on hand
  - ATLAS: huge variety in shape of physics events
- Compile to actual machine features
  - ATLAS: restricted by oldest machines on the GRID
- Inline function calls based on size and actual use
  - ATLAS: many small functions w/ large call overhead
- Co-locate (copies of) functions in memory
  - ATLAS: huge spread across many shared libraries
- Remove cross-shared library trampolines
  - ATLAS: all symbols exported always across all DLLs

Benefits of Tracing JIT (2)

- Remove unnecessary new/delete pairs
  - ATLAS: tracking code copies for physics results safety
- Judicious caching of computation results
  - ATLAS: predefined by type, e.g. Carthesian v.s. Polar
- Memory v.s. CPU trade-off based on usage
  - ATLAS: predefined by type (ptr & malloc overhead)
- Smaller footprint comp. to highly optimized code
  - ATLAS: maybe relevant, probably not
- Low-latency for execution of downloaded code
  - ATLAS: not particularly relevant

Also to Consider ...

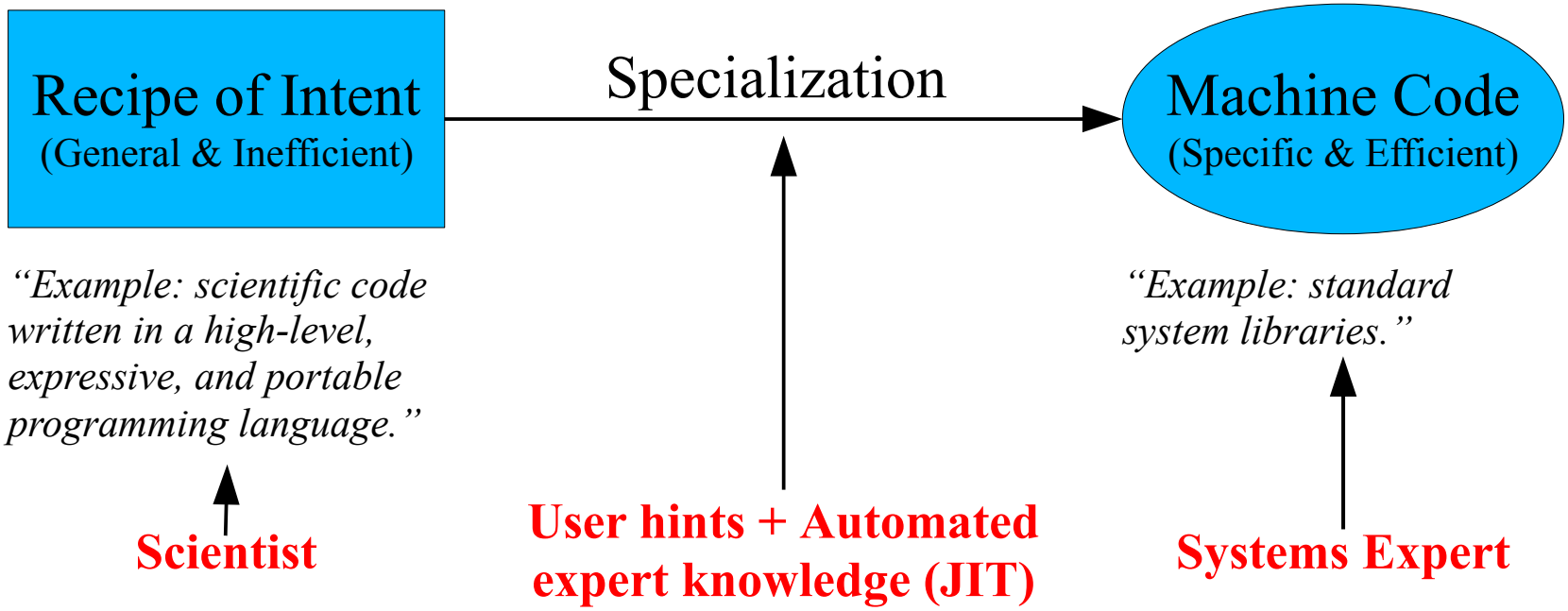
- Cost of effort amortizes well for reconstruction ...
  - But for analysis? Smaller (non-LHC) experiments?
- Machines scale better than humans
  - Re-optimizing the same codes over and over is *boring*
    - New platforms, new functionality, etc.
  - Our number of experts is frightfully small
- Bit-rot: optimized code gets stale really quickly
  - Added complexity hampers maintenance
  - Re-optimizing code harder than first time around
  - Underlying hardware rules are changing quickly
    - Even from machine to machine on the grid

Use of Python

- Relevant project: PyPy (see later)
- Why lose C++ speeds first just to regain w/ JIT?!!
  - Pure mathematics code is easy to unbox & optimize
    - Will need to add a few basic classes, e.g. 3/4-vectors
  - Higher level code allows for broader optimizations
    - Control memory model, code locality, and prefetching
    - Automatic loop parallelization similar to OpenMP
    - HEP-specific examples:
      - Encode the event-loop with a low threshold
      - Branch activation for TTrees
      - Algorithm parallelization based on event data model
      - Shared memory for e.g. pile-up studies
  - CPU cost of call may be higher CPU cost of code
    - General problem due to OO nature of our codes

Benefit of Abstractions

Performance of a program is improved by specializing it to the target environment and the input that it takes.

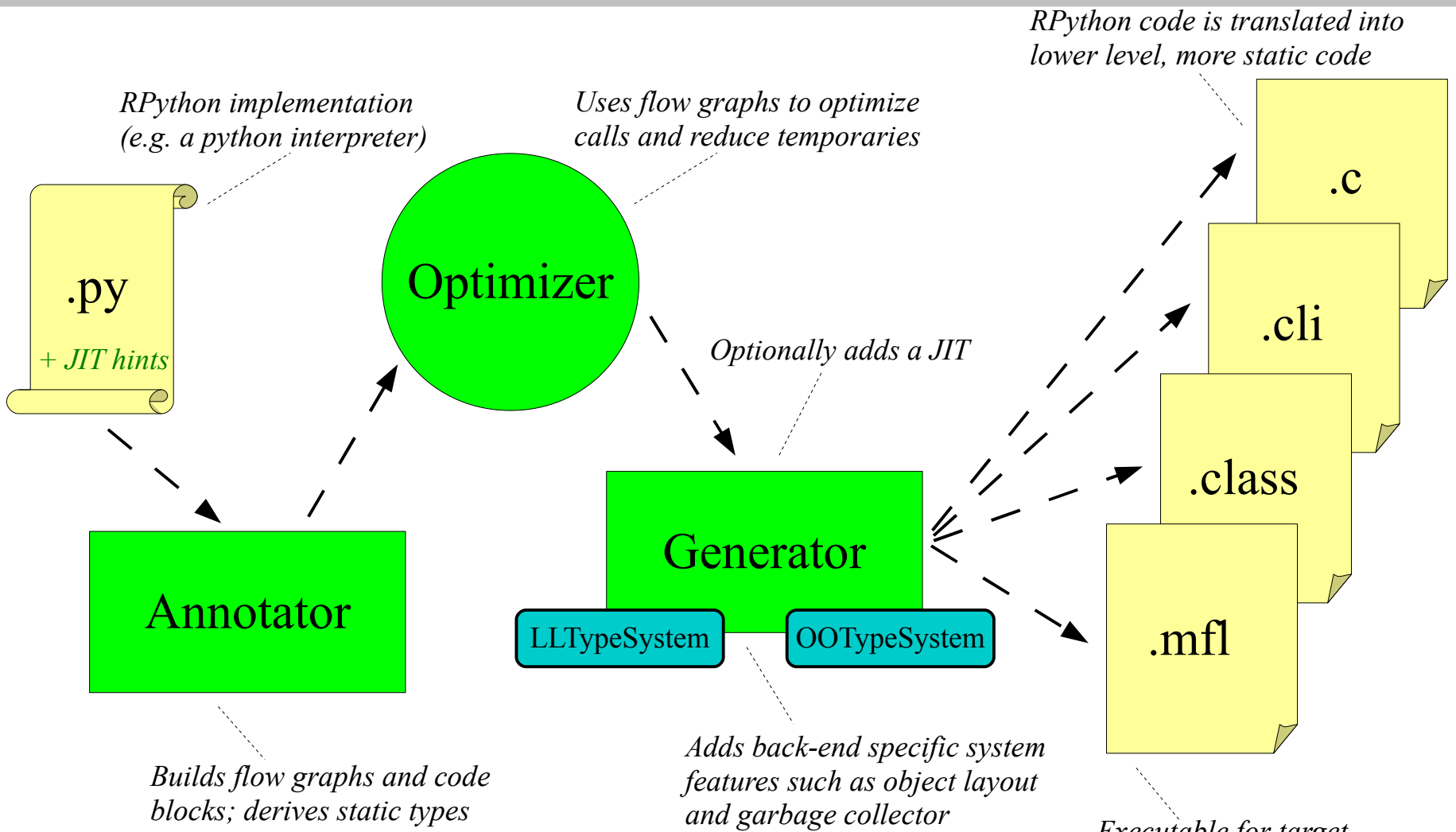


In HEP today, we force the Scientist to be the Systems Expert, with expert knowledge only available on some twiki page ...

PyPy

- A dynamic language development framework
  - Framework itself is implemented in Python
  - One language/interpreter thus developed is Python
    - Is a CPython alternative (currently at 2.7.1)
    - Makes it “Python written in Python”
- A translation tool-chain with several back-ends
  - E.g. Python => C to get *pypy-c* (compiled)
- A JIT generator as part of the toolchain
  - Operates on the interpreter level
    - Makes it a “meta-JIT”

PyPy Toolchain



PyPy's Generated JIT

- JIT applied on the interpreter level
  - Optimizes the generated interpreter for a given input
    - Where input is the user source code
  - Combines light-weight profiling and tracing JIT
    - Especially effective for loopy code
- Can add core features at interpreter level
  - Interpreter developer can provide hints to the JIT
    - Through JIT API in RPython
    - Optimization choices, libffi type information, etc.
  - JIT developer deals with platform details
  - All is completely transparent for end-user

Current Work: Cppyy

- Teach PyPy JIT about our C++ software
  - Uses Reflex dictionary information
  - Cppyy itself written in RPython
    - Transparent to the JIT, so that it can peel off layers
- Reflex-support branch with module cppyy
  - Note the limitation of Reflex for some HEP codes
  - Best results with minor patch to Reflex
- (Oldish) prototype available on AFS
  - /afs/cern.ch/sw/lcg/external/pypy
  - https://twiki.cern.ch/twiki/bin/view/AtlasProtected/PyPyCppyy **(protected ...)**

JIT Closer To Our Codes: C++ Bindings

With CPython:

+ modules
+ PyROOT.so

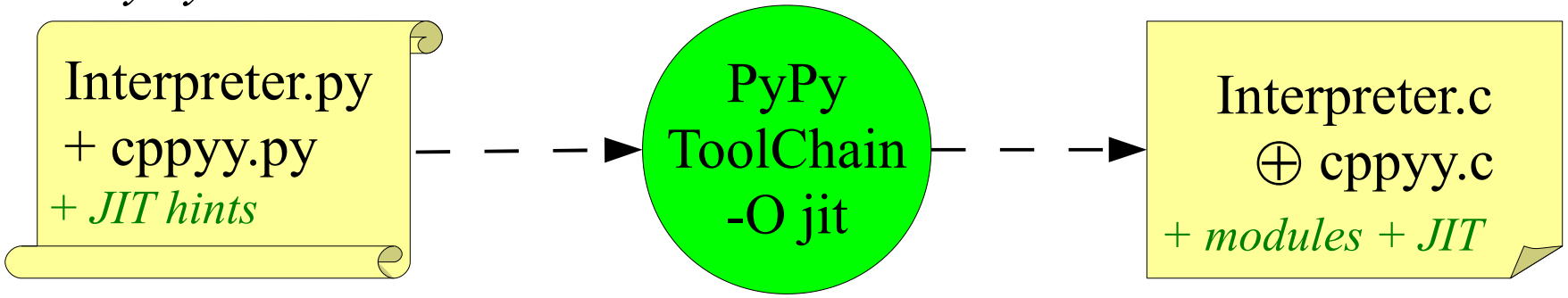
→ CPython is blind to the bindings

→ *No optimizations for GIL, types, direct calls, etc.*

→ PyPy+JIT know the bindings natively

→ *Full optimizations possible*

With PyPy:



+ modules

Cppyy: Current Features

- Bulk of language mapping is implemented:
  - Builtin types, pointer types
  - Namespaces, functions
  - Classes, static/instance data members and functions
  - Simple inheritance, some virtual inheritance support
  - Template types, basic STL support
  - Basic operator mapping
- Short-list of missing features:
  - Automatic memory mgmt (heuristics)
  - LLVM Precompiled Headers back-end
  - Various corner cases to be fleshed out

Install From Mercurial

(Note: prototype dates from SVN days)

- Mercurial repository on Bitbucket server:
  - https://bitbucket.org/pypy/pypy/overview
- Steps to check-out and install:

```
$ hg clone https://bitbucket.org/pypy/pypy
$ cd pypy; hg update reflex-support
$ cd pypy/translator/goal
$ python translate.py -O jit targetpypystandalone.py –withmod-cppyy
$ <setup or install ROOT; or an ATLAS release>
$ [opt: patch pypy-reflex/pypy/module/cppyy/genreflex-methptrgetter.patch]
```

- Result is `pypy-c` in 'goal' directory

Note PyPy is self-hosting, so for 2nd build can use it (or could have downloaded a pypy-c binary from bitbucket's downloads page):

```
$ pypy-c translate.py -O jit targetpypystandalone.py –withmod-cppyy
```

Prototype Usage

- /afs/cern.ch/sw/lcg/external/pypy/<plat>/setup.sh
  - For ROOT, genreflex, lcg gcc compiler, etc.
- Start pypy-c executable, use like CPython

```
>>>> print “Hello World”
Hello World
>>>> import cppyy
>>>> cppyy.load_lib( “libMyClassDict.so” ) # (*)
>>>> inst = cppyy.gbl.MyClass()
>>>> inst.MyFunc()
>>>>
```

(*) libMyClassDict.so being a Reflex dictionary.

First Prototype Results (1)

- Benchmark measuring bindings overhead:
  - PyROOT: 48.6 (1000x)
  - PyCintex: 50.2 (1000x)
  - pypy-c-jit: 5.5 (110x)
  - pypy-c-jit-fp: 0.41 (8x)
  - pypy-c-jit-fp-py: 3.46 (70x)
  - C++: 0.05 (1x)

Notes:
1) "overhead" is the price to pay when calling a C++ function
2) bindings overhead matters less the larger the C++ function body
3) "-fp" is "fast path" and requires Reflex patch
4) "-py" is the pythonified (made python-looking) version, which still needs to be made JIT-friendly

First Prototype Results (2)

- Overhead w/ "realistic" C++ function body:
  - PyROOT: 52.6 (52x)
  - PyCintex: 51.8 (51x)
  - pypy-c-jit: 6.6 (6.5x)
  - pypy-c-jit-fp: 1.18 (1.2x)
  - pypy-c-jit-fp-py: 4.37 (4.3x)
  - C++: 1.02 (1.0x)

Notes:
1) "Realistic" means "a lot" of computation being done in the C++ function body: the atan() function is called
2) "-fp" is "fast path" and requires Reflex patch
3) "-py" is the pythonified (made python-looking) version, which still needs to be made JIT-friendly
4) "C++" is gcc -O2 (other codes are -O2 as well)

JIT Closer To Our Codes: Some HEP-ification Ideas

- Support ROOT I/O at interpreter level
  - Automatic selection of “best practices”
    - PROOF, TTreeCache, buffer sizes, read ordering
  - Instruct JIT to allow direct access (like C++)
    - Get C++ speeds w/o boilerplate code
- Parallelization based on data model
  - Framework written in RPython, thus transparent to JIT
- Parallel processing of I/O
  - Unzipping, T/P separation on different cores
- Memory reuse across parallel event threads
  - Let JIT take care of COW and GC for life-time

Near Future Plans

- Build out functionality
  - Most needed is dealing with destructors
    - PyPy has a GC, so no dtor on scope-exit
- Support for CINT dictionaries
  - Probably easier to work with for end-users
  - But no fast-path options possible
    - Still a factor of 100 improvement expected
- Support for LLVM's Clang Precompiled Headers
  - New direction taken by ROOT/CINT (Summer '12)

Backup Slides

Goals

- Speedup of existing python codes
  - While maintaining C++ integration
- Future-proofing analysis codes
  - Event-parallel processing okay on multi-core, but not so much on many-core

Note how these are related: future-proofing is done by adding a layer for the needed flexibility, speed-up works by removing layers ("unboxing").

=> *The connection is provided by JIT technology*

Why Python?

- Python is a natural fit:
  - Already in use for analysis
  - Clear layering exists (the interpreter itself)
  - JIT technology exists (PyPy)
  - Computer science and industry interest
- And if you prefer C++:
  - Integration is straightforward
  - Common approach of hybrid (C++ for serial code; with parallelizations and messaging in Python) offers a clear separation of concerns

Why Future-Proof?

- No simple solution to using many-core:
  - Train physicists in parallel programming,
  - Pay software engineers to do the work,
  - Or let the extra silicon go to waste

=> *In time or money, there's a significant cost, that negatively impacts the scientific output*

- Need ability to integrate new solutions
  - As these arrive from CS and industry
  - “Like switching to a better compiler”

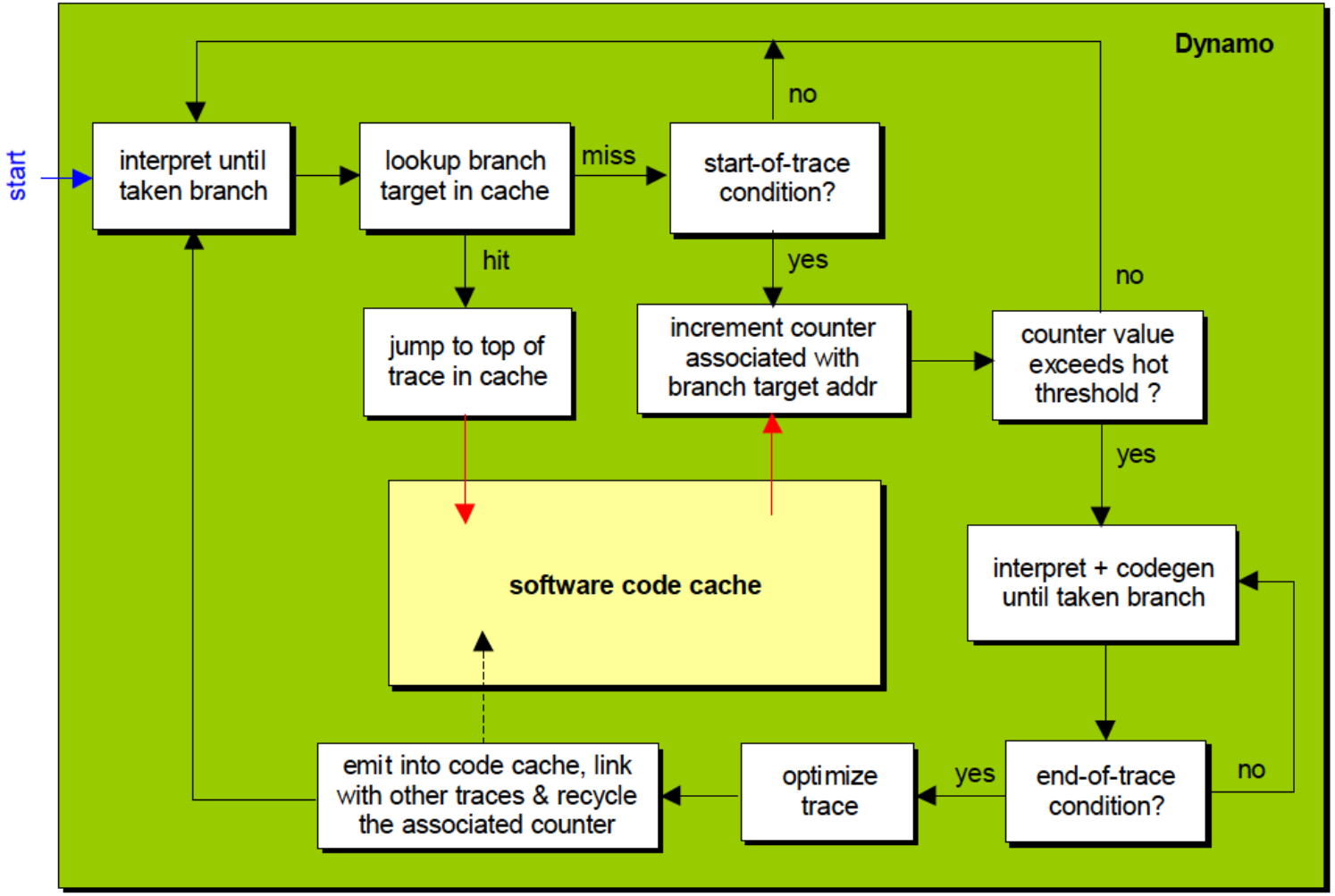
Why Bother?

- After all, no expectation to solve the difficulty of parallel programming, right?
- Yes. Instead, HEP-ify existing approaches:
  - Use JIT technology
    - Keeps a clear path to new CS solutions
  - Instruct JIT about HEP processing
    - Framework invariants
    - Data access patterns
    - Memory models
    - Code locality and prefetching

Solution Idea

- HEP data processing framework in PyPy
  - Abstracts the framework from user runtime
- User sees a normal Python interpreter
  - Framework (base) classes available as if they were written in Python
- Combine HEP and platform specifics in JIT
  - E.g. analyze data flow for intra-event parallelism
  - *The specialization bit in the "Abstraction" but no longer involving the user*

Dynamo Program Flow



Old: Python Optimization

- Optimize algorithmic parts of analysis
  - Pull through PyPy toolchain → Compiled C
  - Several problems:
    - (Too) Restrictive in dynamic typing usage
    - Slow translation/compilation process
    - Difficult to understand error messages
      - Even several bugs in error reporting
- Turns out: no more productive than C++
  - Better approach now exists
  - Completely given up on this idea ...