High Performance Development for High End Computing with Python Language Wrapper (PLW)*

Piotr Luszczek† Jack Dongarra‡

May 1, 2006

Abstract

This paper presents a design and implementation of a system that leverages interactive scripting environment to the needs of scientific computing. The system allows seamless translation of high level script codes to highly optimized native language executables that can be ported to parallel systems with high performance hardware and potential lack of the scripting language interpreter. Performance results are given to show various usage scenarios that vary in terms of invested programmer's effort and resulting performance gains.

1 Introduction

The essential idea of PLW is to bring Rapid Application Development (RAD) style to scientific computing by leveraging the agility of the Python language and a number of compilation techniques to accommodate various High End Computing (HEC) environments. The solution-oriented nature of Python invites fast prototyping and interactive creation of applications with surprisingly short time-to-solution. However, the default mode of Python's execution is interpretation of bytecode and thus is not suitable for traditional HEC workloads. Using standard compilation techniques is a very hard problem due to so called "duck typing": Python is very permissive when it comes to mixing objects together and using objects in various contexts. This style of programming is elusive for the standard methods used in main stream language compilers – they rely heavily on static type information provided by the programmer. However, by restricting a bit the syntax of Python's programs to a subset that is interesting to the HEC community, it is still possible to generate efficient code for scientific computing and leverage many of Python's RAD features. Hence the name: Python Language Wrapper (PLW). PLW wraps a Python-like language (by allowing only a subset of Python syntax) in the Python code in order to deliver both performance and portability

*This work was supported in part by the DARPA, NSF, and DOE though the DAPRA HPCS program under grant FA8750-04-1-0219.

†University of Tennessee Knoxville

‡University of Tennessee Knoxville and Oak Ridge National Laboratory

of the resulting code. As a result, PLW-enabled codes are still valid Python codes but not the other way around.

This paper is organized as follows: section 2 describes some of the projects similar to PLW, section 3 motivates the choice of Python and various translation methods, section 4 gives a more detailed overview of various aspects of PLW , section 5 shows an example of a parallel code together with optimization process and resulting performance gains, and finally section 6 concludes the paper and hints at extensions and future work.

2 Related Work

There have been efforts to make High Performance Computing more friendly to a casual user. A rather complete approach was taken by the pMatlab project [1] that not only includes bindings to Message Passing Interface (MPI) [2] but also provides support for parallel objects. Another Matlab-based approach was taken by Matlab$*$P [3] which uses client-server architecture to interface the high performance resources. Finally, MathWorks is planning parallel extension to Matlab despite the initial resistance to do so [4]. Titanium [5] is a language whose syntax closely resembles Java (in fact it has become a superset of Java language). But with added constructs such as multidimensional arrays and global address space, Titanium is well suited for parallel applications such as its driving application: a Poisson solver with mesh refinement. Mixing high level language such as Java and a low level one such as C was done by the Janet project [6].

The Python community stepped up to the challenge of making scientific computing easier by creating modules that support multidimensional arrays – an effort collectively called `Numeric` [7]. Over the years, the modules have evolved considerably[1]. Also, a number of extensions provide bindings to MPI: the difference between them is whether they require an extended interpreter [8] or not [9]. The `scipy.weave`[2] project is probably the closest to our approach as it compiles SciPy (Scientific Python) expressions to C++ for faster execution. A different perspective is offered by `Boost.Python`[3] which allows easy access to Python from C++ code – an example of broader category of *hybrid programming*[4].

Finally, efforts to compile Python code into a native executable started with the `py2c` project. It is no longer available but most of its code legacy continues as part of the standard `compiler` package. A different effort called PyPy (see `http://pypy.org/`) has undertaken the task of implementing Python in Python and thus making it much easier to perform various modifications of the language such as optimization through type inference.

[1] See `http://www.stsci.edu/resources/software_hardware/numarray` for more detailed description of `numarray` and `http://www.scipy.org/NumPy` for details on `numpy`.

[2] `scipy.weave` is part of the SciPy package and can be found at `http://old.scipy.org/documentation/weave`.

[3] See `http://www.boost.org/libs/python/doc/` for details on `Boost.Python`.

[4] See `http://www.boost-consulting.com/writing/bpl.html` for a detailed article on the use of `Boost.Python`.

3 Motivation

Python is a programming language that enables data abstraction – by now an old concept implemented first by CLU [10]. But it is hardly a distinguishing feature in today's computer languages. Python's unique assets include very large standard library, strong support for multidimensional arrays through third-party modules, and true multithreading with POSIX threads (rather than just continuation-based multitasking). In addition, Python's Standard Library includes a `compiler` module: a full featured Python language parsers that produces Abstract Syntax Trees (AST). Continuous perusal of this module throughout the translation process is the key to creating systems like PLW where all of the seemingly unrelated components (such as the source code, the source code's directives, and external files with static type information) use the familiar Python syntax (but substantially differ in semantics) and allow for gradual performance tuning of the code as permitted by the available programmer's time and as necessitated by profiling information.

The essential ingredient for performance is static typing: all major languages used in HEC are statically typed. There were many efforts to introduce static typing in Python and all of them have failed to gain wide spread use [11]. While it is interesting to consider reasons for this, a more imminent consequence is that attempting to add typing to Python should not be the goal but rather the means. Also, adding static typing should not render the code inaccessible to the standard Python interpreter – one of the main drivers behind Python's application development agility. The following modes of static type inference are considered in PLW :

1. Manual,
2. Semi-automatic, and
3. Fully automated.

In the manual mode, the programmer decides what are the types of objects. This is preferable for performance-critical portions of the code and when the other modes fail. In the semi-automatic mode, the programmer guides the type inference engine by narrowing down the potential set of resulting types. For example, unit tests associated with a piece of code could be used for type information and limit the number of usage scenarios. Finally, the fully automated mode would attempt to infer the types of objects without programmers intervention – a rather ambitious task as described later on.

As mentioned earlier, sometimes in language design the programmer's convenience stands in the way of performance of the resulting executable. Thus, by selecting the features included in PLW, performance aspects will most often trump other considerations as long as they do not severely harm the functionality.

4 Design

Figure 1 shows the design of PLW. As the input to the PLW translator, a regular Python code is used possibly accompanied with directives, native code snippets and static type information.

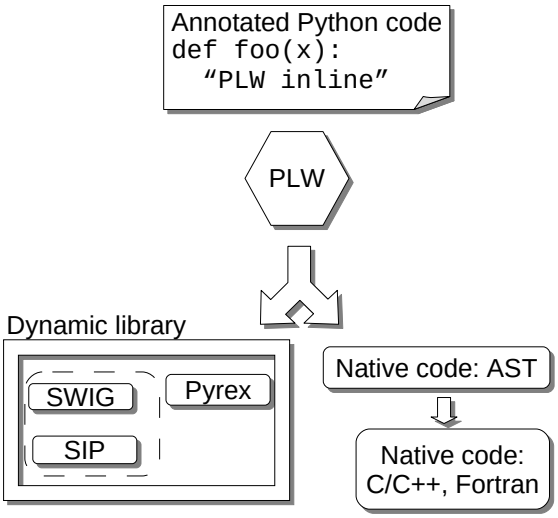

Figure 1: By design, PLW translates annotated Python code into native language by using Python's native-language modules in form of dynamic libraries or generating a stand alone executable that does not require the Python interpreter.

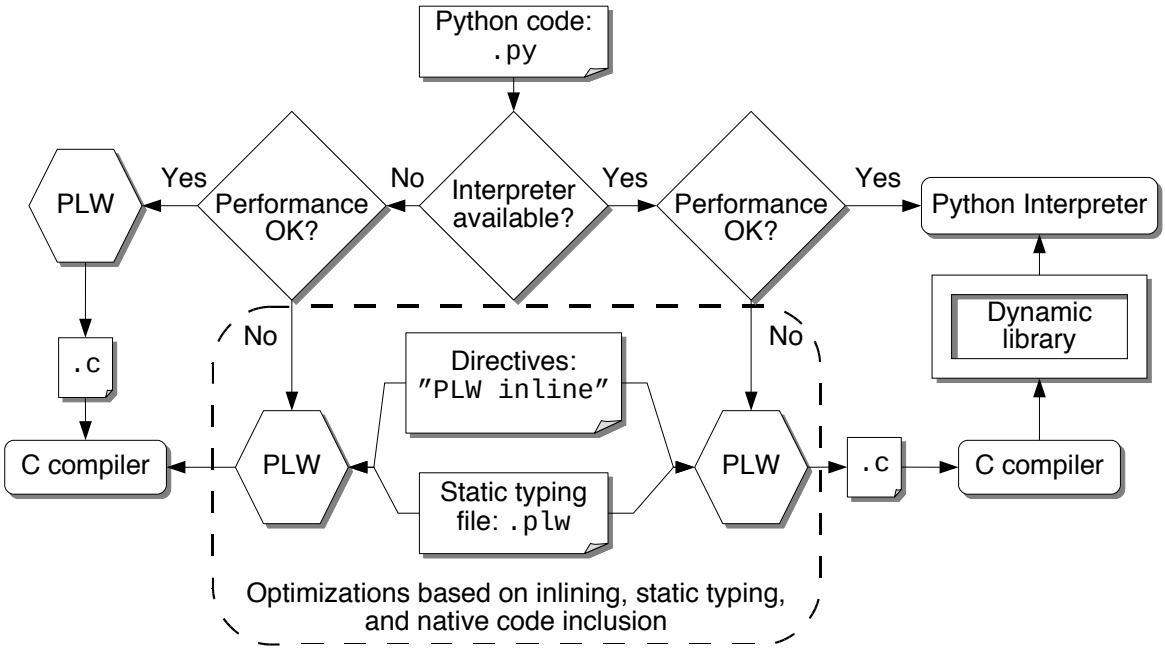

Figure 2: Overview of possible use-cases of PLW. The left portion of the diagram represents the standalone executable scenario while the right hand side corresponds to the interpreted execution.

Depending on the invocation, the translator generates Python modules or a stand alone executable, both of which come from the generated native code that may speed up the application execution.

Figure 2 illustrates various PLW's usage scenarios. And so from the Python interpreter perspective, a PLW code is a regular Python source code and can be executed directly without changes. If the performance of the code is acceptable and Python interpreter is available on the target platform then no further action needs to be taken by the programmer: the use of PLW should not impede Python's standard development cycle. However, if either of the two is not true then the PLW's translator needs to be involved. If the translation is done because of portability problems, then (most often) the code does not need to be changed and the porting is done by simply running the PLW translator in the "dynamic typing" mode. If performance is an issue, then PLW may be used to produce better performing native code from the original Python code. The native code can be made available to a regular Python interpreter using Pyrex [12] or similar technologies. If using the interpreter is not an option then the native code is generated for execution. The generation of native code is done in two phases: first the AST of the native code is generated and then the actual text is produced. This of course gives opportunity to perform additional optimization step on the AST of the native code but currently is not done.

As was mentioned before, a subset of Python is available while working with PLW translator. Currently the following data types, modules and libraries are available to PLW programs:

- Most of Python's syntax is supported, except for highly dynamic features of Python that modify various namespaces (modules, classes, functions) at runtime and the newest additions to the language, e.g. generators, that have not gained wider acceptance from programmers' community yet.
- Standard Python user data types: booleans, strings, integers, floats, complex values, lists, tuples, slices, files, etc..
- Essential Python built-in functions.
- Essential Python modules: `array`, `os`, `socket`, `string`, `sys`, `time`.
- Multidimensional array module (a subset of `numarray` module).
- Numerical linear algebra libraries: BLAS [13, 14, 15, 16], LAPACK [17], ScaLAPACK [18], PBLAS [19]
- Communication libraries: MPI [20, 21, 22] and BLACS[23].

This has been sufficient to generate code for many useful applications but extending the above list is certainly planned in the future.

4.1 Target Languages and Platforms

As mentioned above, one of the possibilities while developing an application is to translate it to native code (for either performance or portability). PLW does not use assembly language as the

target but rather the C language. The generated code uses only a subset of C for maximum portability and lack of extra linker dependences (as opposed to languages like C++ or Fortran which require extra libraries linked in to produce a binary). The disadvantage of using C as the target is that many optimization techniques have to be implemented – techniques that, for example, C++ programmers take for granted such as inlining, template instantiation and template expressions. Consequently, C++ is considered as a possible target but is only in experimental phase as of this writing. In the future, the use of C++ might lead to a more compact output code by, for example, the use of the Resource Allocation Is Initialization (RAII) technique and smart pointers for reference counting (the current automatic garbage-collection method). But the biggest advantage of generating C++ is ability to directly interface to existing C++ libraries. By the similar argument, using Fortran as translation target is also considered. Fortran 2003 has extensions to previous versions of the standard specification that make it an easy target for code generation and allow for seamless interfacing with existing C libraries. Targeting multiple languages would require special syntax for code inlining depending on the targeted language: such syntax is already present in the current PLW implementation (see below).

As mentioned already, PLW targets architectures that are of interest for HEC tasks. It would be hard (if not impossible) to port the Python interpreter to some of these architectures. As an example, consider dynamic libraries – a feature heavily relied upon by the interpreter. Dynamic linking is a problematic issue on IBM's BlueGene/L. In general, on systems with light-weight OS kernels (for example Cray XT3) many standard OS features are missing which might be a major obstacle for porting the Python interpreter. This is when PLW's translation to C becomes very useful – the resulting C code has minimal requirements from the C compiler. Another feature targeted for such non-standard environments is ability to remove features and modules from PLW's runtime to accommodate porting of the translated code. This is done by splitting the runtime into separate libraries each of which may be built in a dummy mode – without any functionality and no dependences. For example, if the target architecture does not support sockets, then the library that implements the `socket` module is built in the dummy mode – applications can be ported to such an architecture as long as they do not communicate through the socket interface. In the standard Python environment, the `socket` module resides in a dynamic library and cannot be disabled during the building and installation process.

Finally, to make the exposition more specific we list here the architectures on which the PLW runtime was deployed. The operating systems that we used are AIX 4.3, AIX 5.2, Apple Darwin PowerPC G5, IRIX64 6.5, Linux 2.4.21 ia64, Linux 2.4.18 x86, Linux 2.6.{8,14} x86 Linux 2.6.{5,13} x86-64, Linux 2.6.14 ppc64, Solaris 5.9, Solaris 5.10, UNICOS/mp 3.1.05, Cray XT3 Catamount OS (no socket networking support). The tested compilers were from GNU, The Portland Group, Cray, IBM, Sun, SGI, Intel (version 8.0 on x86 and ia64 architectures), and PathScale. Tested processors were from Intel, AMD (x86, x86-64 and Itanium 2), IBM Power3,4,5, PowerPC 440/G5, UltraSPARC II/T1(Niagra), MIPS R12000, and Cray X1E. In the future we plan investigate possibility of porting PLW to non-UNIX platforms such as Microsoft Windows and in particular the Compute Cluster Edition as it sees acceptance for HPC workloads.

4.2 Adding Directives to Python Code

Code directives in scientific programs have a long history, the well known examples of standardized code directives include High Performance Fortran (HPF) special comments and OpenMP pragmas and comments. This concept can easily be used in Python. Python's `compiler` module removes comments during the parsing process. As a consequence, directives in the Python code cannot be done with comments as it is the case with Fortran[5]. Rather, strings could be used for that purpose. The following example shows how adding directives is done in PLW: a function has directives inside the so called doc-string (documentation string) and a `for` loop has a directive included in a preceding string that has no effect on execution:

```
def square(x):
  """PLW inline"""

  "PLW parallel"
  for i in range(len(x)):
    x[i] *= x[i]
```

Again, both of these directives do not affect the execution process (and in fact are removed if the Python interpreter is started with optimization enabled) but are included in the resulting AST (generated by the `compiler` module) for PLW's back-end perusal. It is beneficial for the directives not to use a new syntax but rather keep them as Python expressions and/or statements so that the Python's `compiler` module can be used to parse the directives. In addition, adhering to Python syntax makes the resulting code look consistent.

The previous example showed a directive that looked like an OpenMP pragma – such functionality is currently experimented with. The advantage of using OpenMP at Python level allows for implementing transparent fallback to, for example, POSIX threads where there is no OpenMP compiler available. A similar procedure for C-level pragmas would require interfacing with the C compiler front-end or parsing the C code – neither of which is comparable to the ease of the PLW approach. The same technique may be applied to, for example, request loop unrolling of certain depth or inform the PLW translator of data dependences (or lack thereof) between objects to allow more efficient code generation.

A more evolved example with static typing directives is shown on the left hand side of Figure 3. The "`PLW{`" and "`PLW}`" mark the beginning and the end, respectively, of the section with type declarations. The section itself is a valid Python code and can be easily parsed with the standard `compiler` module. The meaning of the directive is deduced by traversing the resulting AST: in this instance, the types of `e` and `x` are prescribed to be `float` values. There is an alternative form of providing static typing information: an external file (with extension `plw`) that has Python syntax but semantics is valid only within PLW's translator – an example is shown in Figure 3 on the right. In fact, the code on the left in the figure may be considered a template while the declaration on the right may be considered a template instantiation – a novel approach that makes generic programming syntax shorter than, for example, C++ templates.

[5]It is possible to recover comments by using line information contained in the AST. However, it is complicated and simpler methods are readily available.

```
def power(e, x):
  """
  PLW{
  e, x = float
  PLW}
  """
  return e ** x
```

```
def power(e, x):
  e, x = float
  return float
```

Figure 3: Two ways of providing static type information: in function doc-strings (left) and in external file with Python syntax (right).

Another use for directives could be programming by contract and aspect oriented programming. The latter is particularly interesting in the context of parallel computing. However, both of these features are beyond the scope of this article.

Figure 4 shows a sequential code for the STREAM test – a part of the HPC Challenge benchmark suite [24]. There is no explicit `for` loop: the iteration is made implicit using overloaded operators `+` and `*` from a third-party module for multidimensional arrays called `numarray`. However, operator overloading as it is implemented now allocates additional arrays to store temporary results. While in general, this does not create a substantial overhead, it might be necessary in some situations (for example in constrained memory environments) to remove necessity for temporaries. PLW offers another directive for such situations: the programmer may insert the native code. In the figure, the native code consists of a few initialization statements and an explicit `for` loop. Upon translation to C, the overloaded operator expression is replaced by the C code from the directive. In this way, the programmer can include raw native code to be used whenever PLW cannot provide a faster alternative – this could be used especially for performance-critical portions of the code as is the case with the main `for` loop of the `RandomAccess` function. The directive includes (in the square brackets) the language of the native code. Thus, if the example was translated to Fortran, the code between "`PLW[C]{`" and "`PLW[C]}`" would not be used and an alternative form in Fortran would have to be provided.

Finally, decorators could be used for Python directives. However they have a number of drawbacks. To begin with, only functions can have decorators so using directive for loops is out of question. That, combined with still fresh deployment status (decorators were introduced in core Python in version 2.4) of decorators makes them an unlikely tool for general purpose code directives.

4.3 Automatic Type Inference

It is possible to perform basic type-inference [25] in systems with recursive types and subtyping [26]. However, open questions still remain thus preventing broader use of such techniques in common languages such as C [27], C++ [28] and Java [29] – at least to the extent it is done in ML [30]. The inference mechanism becomes yet harder for the dynamically typed languages such as Smalltalk [31], Self [32] or Python. Consequently, in order to use existing tools for automatic

```
def STREAM_Triad(a, b, x, c):
  """
  PLW[C]{
  double *A, *B, X, *C;
  int n;
  PLW[C]}
  """

  """
  PLW[C]{
  X = plw_float_to_double(x);
  A = (double*)plw_numarray_obj(a)->data;
  B = (double*)plw_numarray_obj(b)->data;
  C = (double*)plw_numarray_obj(c)->data;
  n = plw_numarray_obj(a)->dim[1];
  for (i = 0; i < n; ++i) {
     A[i] = B[i] + X * C[i];
  }
  PLW[C]}
  """
  a[:]  = b + x * c
```

Figure 4: Reference implementation of STREAM-Triad test from the HPC Challenge suite.

```
def pingpong(nmax):
  buf = netlib.zeros(nmax, netlib.Int32)

  np_me = mpi.COMM_WORLD.rank()
  npall = mpi.COMM_WORLD.size()
  if npall < 2 or np_me > 1:  return

  n = 1
  while n <= nmax:
    t = -mpi.Wtime()
    if np_me:
      mpi.COMM_WORLD.Recv(buf[:n], 1-np_me, 0)
      mpi.COMM_WORLD.Send(buf[:n], 1-np_me, 0)
    else:
      mpi.COMM_WORLD.Send(buf[:n], 1-np_me, 0)
      mpi.COMM_WORLD.Recv(buf[:n], 1-np_me, 0)
    t += mpi.Wtime()

    size = n * 4
    if not np_me:
      print size, t, size/t

    n += n / 10 + 1
```

Figure 5: Python code for a simple MPI ping-pong test.

type analysis we are considering further restriction of Python syntax similarly to the notion of RPython (Restricted Python) used by the PyPy project. The work on this aspect of PLW is still preliminary.

5 Performance Results

The driving applications for PLW so far were the computational server of the LFC project [33] and the HPC Challenge benchmark. Both of them are too large to be discussed here in sufficient detail. Instead, simplified examples were chosen: an MPI ping-pong code and a reduced version of `RandomAccess` function from the HPC Challenge suite.

Figure 5 shows the MPI ping-pong program written in Python – the code should be self-explanatory. The code could be used to test an MPI installation and is often used as a benchmark for bandwidth and latency. The latter usage will be emphasized here to show how PLW allows continuous refinement of Python source code to achieve performance on par with low-level compiled language.

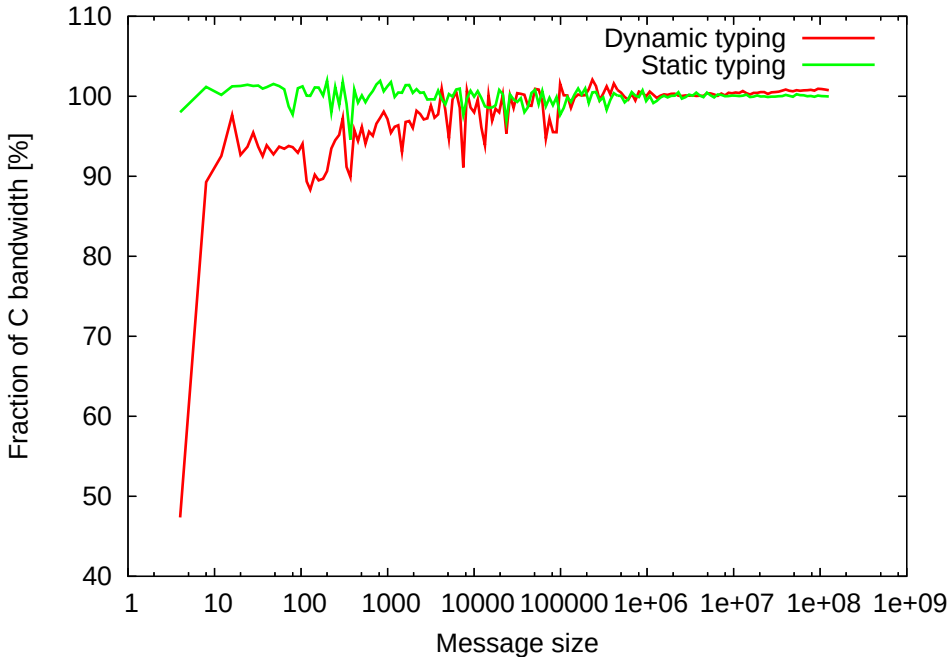

Figure 6: Performance results on a GigE switch for the MPI ping-pong test.

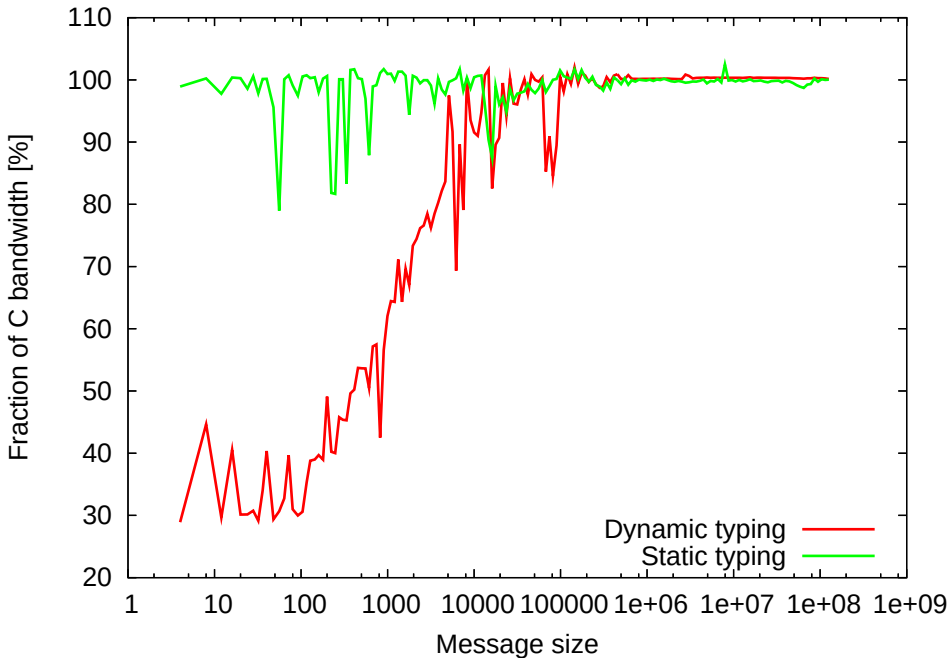

Figure 7: Performance results using memory-memory copy for the MPI ping-pong test.

```
def pingpong(nmax):
  nmax = int
  buf = netlib.array(int, 1)
  np_me, npall = int
  n = int
  t = double
  size = int
  return void
```

Figure 8: Static typing directives for the ping-pong code.

Figures 6 and 7 show results from running the program from Figure 5 on Intel Xeon 2.4 GHz cluster with dual processor (single core) nodes and LAM 6.5.8 as an MPI implementation. Both figures show performance relative to that of a reference C implementation. For each data size, a few measurements were done and the maximum value is reported on the figures. The lines are still not smooth enough due to the fact that the tested system was not completely dedicated during runs and the ping-pong test code was simplified from the one that is used in the HPC Challenge. Figure 6 shows performance numbers when data is sent through a GigE switch while the results from Figure 7 were obtained on a single box with two processors: the MPI processes were communicating through shared memory and memory copying. Both of these test environments show different performance characteristic with respect to PLW's translation methods.

Two types of Python-to-C translation scenarios were tested. The "dynamic typing" scenario did not use any of the static typing information and treated all objects (even the primitive ones like `int`'s) as generic Python objects – the PLW runtime figured out the interactions between the objects and thus added an interpretation overhead. The "static typing" scenario involved an external file whose content is shown in Figure 8. This file informed the PLW translator about the types of the objects. The information was then passed from PLW to the C compiler by generating an appropriate source code with more specific type information. It was up to the C compiler then to deduce at compile time the right interactions between objects.

The least effort approach is of course the dynamic typing scenario when the user only writes Python code. As seen from Figures 6 and 7 this approach delivers the bandwidth of the native C code at message sizes of about 100 KiB. However, if this is not satisfactory from the user perspective, just adding static type information from Figure 8 fully recovers the performance of C (note that the static typing file uses Python syntax and can be parsed with Python's `compiler` module). Except for possibly message sizes below 10: if this is still not satisfactory, the programmer may use techniques from section 4.2 to further improve the results.

Figure 9 shows a sequential code for the RandomAccess test – a part of the HPC Challenge benchmark suite. The main `for` loop is preceded by a directive that includes C source code. When the `RandomAccess` function is translated to C, the loop is replaced by the C code from the directive rather than the translated Python code. In this instance, this is the only way to achieve optimal performance as no Python modules can provide functionality required by `RandomAccess` at the speed of the native code. Figure 10 shows the performance (measured in Giga Updates Per

```
def RandomAccess(table, n):
  """
  PLW[C]{
  int i, N, Ran=1, *Table;
  PLW[C]}
  """

  ran = numarray.array(1, type=numarray.Int32)
  """
  PLW[C]{
  N = plw_int_to_long(n);
  Table = (int *)plw_numarray_obj(table)->data;
  for (i = N; i; --i) {
    Ran = (Ran << 1) ^ ((Ran < 0) ?  7 :  0);
    Table[Ran & (N-1)] ^= Ran;
  }
  PLW[C]}
  """
  for i in range(n):
    ran = (ran << 1) ^ (numarray.any(ran < 0) and 7 or 0)
    table[ran & (n-1)] ^= ran
```

Figure 9: Reference implementation of the RandomAccess test using directives with embedded C code.

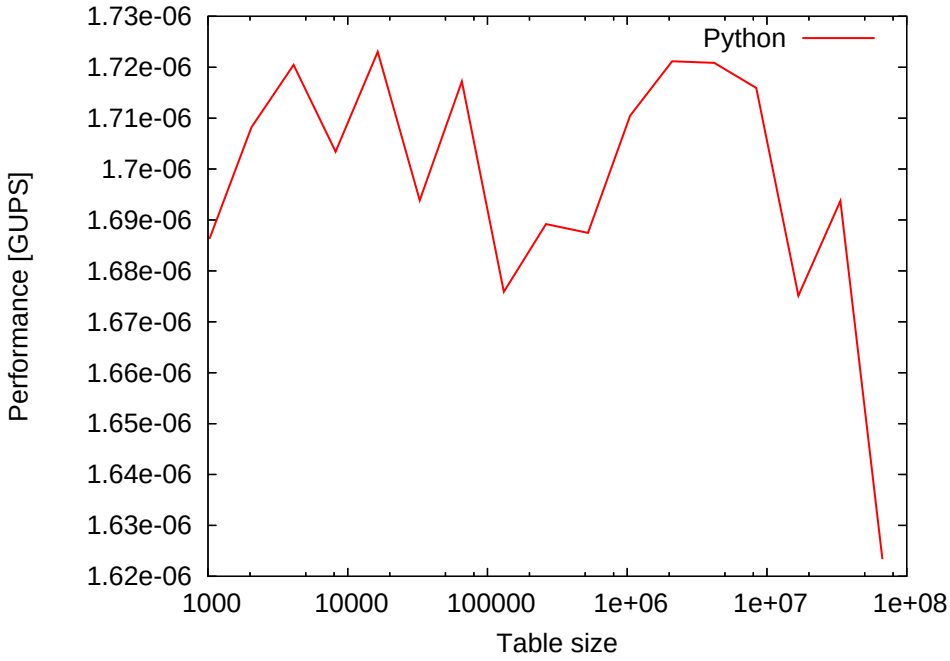

Figure 10: Performance of Python version of `RandomAccess` function.

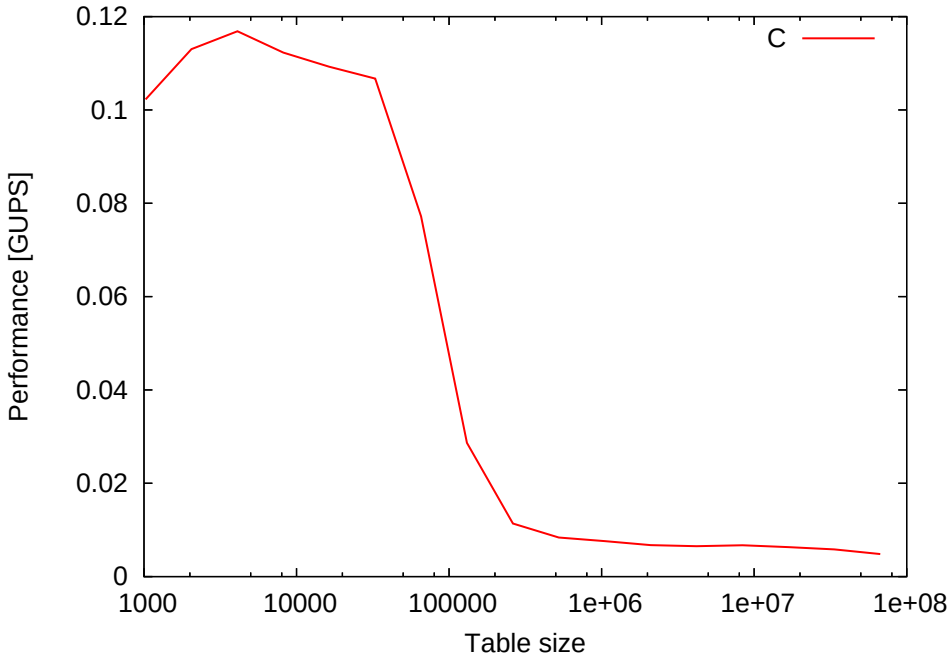

Figure 11: Performance of translated Python version of `RandomAccess` function with inserted native C code.

Second – GUPS) achieved at Python level while Figure 11 shows results of the code translated using the directive with native code. The difference is three orders of magnitude. Also, the native code version clearly shows how the performance of `RandomAccess` depends on the main table size as it spills various cache levels. Any such effects in the Python version are suppressed by the interpretation overhead.

6 Concluding Remarks

This paper showed a design and implementation of a software system called PLW that improves application development time by leveraging agility of the Python language. At the same time, PLW allows many options for gradual and selective improvement of performance of the resulting code so that it can achieve the speed of the native code. This allows very focused tuning effort of the performance-critical portion of the code while keeping the rest of the code unchanged and thus easy to maintain.

Out of many possible future directions to consider for PLW is support for Global Address Space (GAS) by using a back-end library such as GASNet [34] with a fallback support using one-sided communication available in MPI-2. This would allow us to experiment with global address space semantics in Python. Also, as mentioned previously, adding new computing platforms and exploring type inference are interesting research directions to pursue.

References

[1] Nadya Travinin, R. Bond, Jeremy Kepner, and H. Kim. pMatlab: High productivity, high performance scientific computing. 2005 SIAM Conference on Computational Science and Engineering, February 12 2005. Orlando, FL. 2

[2] Jeremy Kepner and Stan Ahalt. MatlabMPI. *Journal of Parallel and Distributed Computing*, 64(8):997–1005, Aug 2004. 2

[3] Long Yin Choy and Alan Edelman. MATLAB*P 2.0: A unified parallel MATLAB. Technical report, Massachusetts Institute of Technology, January 2003. URI: `http://libraries.mit.edu/dspace-mit/`. 2

[4] Cleve Moler. Why there isn't parallel Matlab. *Mathworks Newsletter*, 1995. Cleve's corner. 2

[5] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11-13), September-November 1998. 2

[6] Marian Bubak, Dawid Kurzyniec, and Piotr Luszczek. Convenient use of legacy software in Java with Janet package. *Future Generation Computer Systems*, 17(8):987–997, June 2001. Available: `http://janet-project.sourceforge.net/`. 2

[7] Paul F. Dubois, Konrad Hinsen, and J. Hugunin. Numerical Python. *Computers in Physics*, 10(3), May-June 1996. 2

[8] P. Miller. pyMPI – an introduction to parallel Python using MPI, 2002. Available: `http://www.llnl.gov/computing/develop/python/pyMPI.pdf`. 2

[9] Konrad Hinsen. ScientificPython. URI: `http://starship.python.net/~hinsen/ScientificPython/`. 2

[10] Barbara Liskov and Stephen Zilles. Programming with data types. In *ACM SIGPLAN Conference on Very High Level Languages*. ACM, 1974. 3

[11] Michael Salib. Faster than C: Static type inference with Starkiller. In *PyCon Proceedings*, Washington DC, March 24 2004. 3

[12] Paul Prescod. Building python code with Pyrex. In *PyCon Proceedings*, Washington DC, March 24 2004. 5

[13] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and S. Hammarling. Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, March 1990. 5

[14] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:18–28, March 1990. 5

[15] Jack J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, March 1988. 5

[16] Jack J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:18–32, March 1988. 5

[17] E. Anderson, Z. Bai, C. Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and Danny C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999. 5

[18] L. Suzan Blackford, J. Choi, Andy Cleary, Eduardo F. D'Azevedo, James W. Demmel, Inderjit S. Dhillon, Jack J. Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David W. Walker, and R. Clint Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1997. 5

[19] Antoine Petitet. *Algorithmic Redistribution Methods for Block Cyclic Decompositions*. Computer Science Department, University of Tennessee, Knoxville, Tennessee, December 1996. PhD dissertation. 5

[20] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994. 5

[21] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard (version 1.1), 1995. Available at: `http://www.mpi-forum.org/`. 5

[22] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, 18 July 1997. Available at `http://www.mpi-forum.org/docs/mpi-20.ps`. 5

[23] Jack Dongarra and R. Clint Whaley. A user's guide to the BLACS v1.1. Technical Report UT-CS-95-281, University of Tennessee Knoxville, March 1995. LAPACK Working Note 94 updated May 5, 1997 (VERSION 1.1). 5

[24] Jack Dongarra and Piotr Luszczek. Introduction to the HPC Challenge benchmark suite. Technical Report UT-CS-05-544, University of Tennessee, 2005. 8

[25] Jens Palsberg and Patrick M. O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, 1995. Preliminary version in Proc. POPL'95, 22nd Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 367-378, San Francisco, California, January 1995. 8

[26] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, 1993. 8

[27] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978. 8

[28] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990. 8

[29] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. 8

[30] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, 1990. 8

[31] Adele Goldberg and David Robson. *Smalltalk-80 – the Language and its Implementation*. Addison-Wesley, 1983. 8

[32] David Ungar and Randall B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation, Kluwer Academic Publishers*, 4(3), June 1991. First published in *Proc. OOPSLA '87, Object-Oriented Programming systems, Languages and Applications*, pages 227-241, 1987. 8

[33] Zizhong Chen, Jack Dongarra, Piotr Luszczek, and Kenneth Roche. Self-adapting software for numerical linear algebra and LAPACK for Clusters. *Parallel Computing*, 29(11-12):1723–1743, November-December 2003. 10

[34] Dan Bonachea. GASNet specification, v1.1. Technical Report CSD-02-1207, UC Berkeley, October 29, 2002. 15