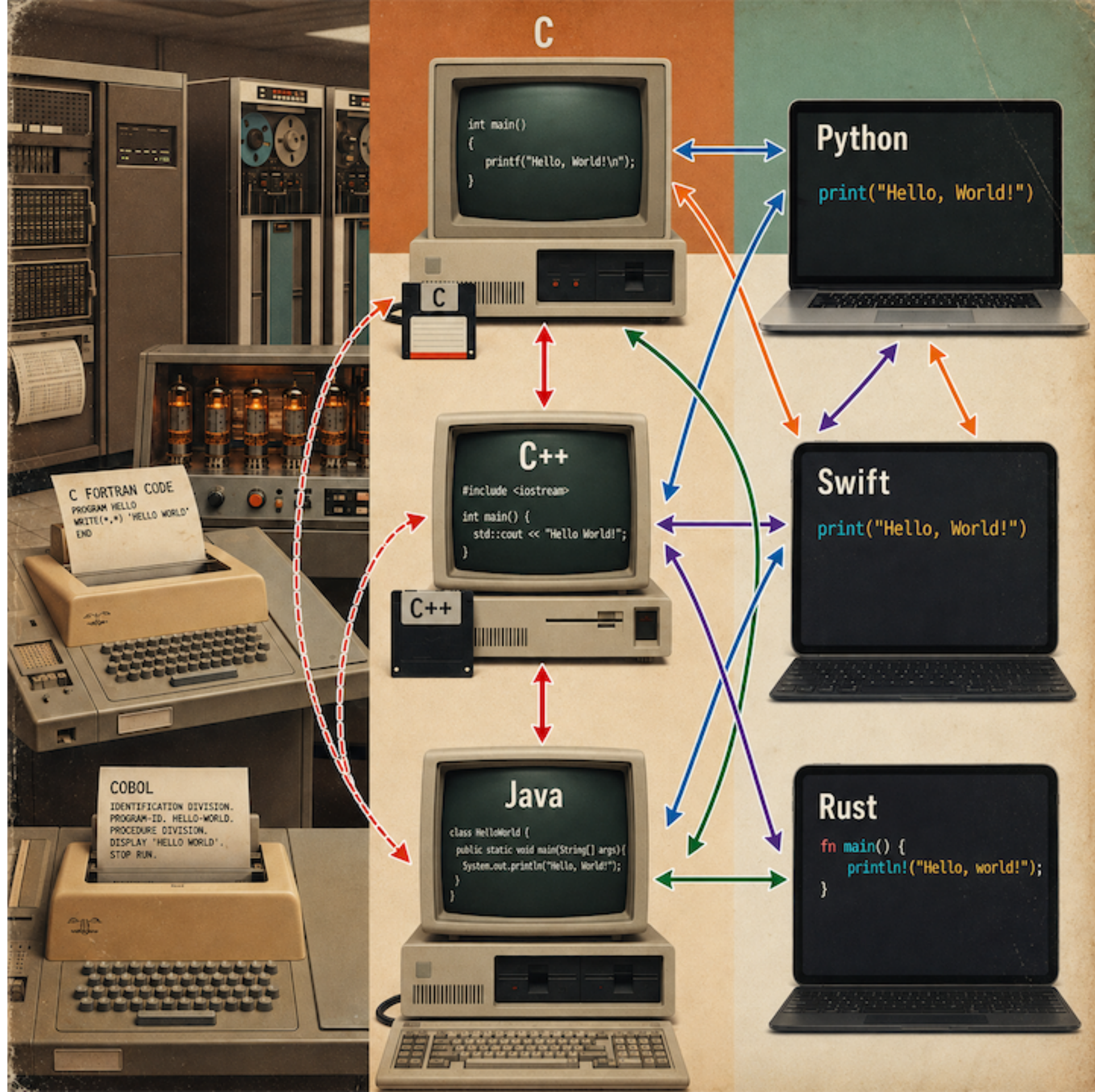


# Achieving Near-Native Performance in XSLT, XQuery, and XPath

Using Host-Language User-Defined Extension Functions

O'Neil Delpratt, [oneil@saxonica.com](mailto:oneil@saxonica.com)





# Motivation

- Java Compiler, JVM, JRE
  - JRE - Many libraries written in C
- Python and Cython
- Python with optional C type annotations
  - Compiles to C for improved performance
  - Computationally intensive workloads
- Integration requirements



# Declarative Languages: XSLT, XQuery and XPath

```
<xsl:template name="xsl:initial-template">
  <xsl:param name="rpt" select="100000" as="xs:integer"/>
  <xsl:param name="input" select="100000" as="xs:integer"/>
  <results>
    <xsl:for-each select="1 to $rpt">
      <run number="{.}">
        <xsl:sequence select="f:calculate($input)"/>
      </run>
    </xsl:for-each>
  </results>
</xsl:template>
```

...

```
<!-- Expensive function -->
<xsl:function name="xf:calculate" as="xs:double">
  <xsl:param name="iterations" as="xs:integer"/>
  <xsl:param name="context" as="xs:integer"/>

  <xsl:sequence select="
    sum(
      for $i in 1 to $iterations
      return math:sqrt($i) + $context
    )
  "/>
</xsl:function>
```

This math:sqrt example written in XSLT  
executes in 9 seconds using SaxonJ.

Can we speed it up?

- **Combining declarative XML processing with native code?**

# What is available to XML Processors?

## User defined Extension Function

Feature available in several XSLT and XQuery processors:

- Saxon
- BaseX
- SupXML (Rust, XSLT 1.0)
- Are there other implementations with this feature?

# Why User Extension Functions?

- Developers want to reuse existing libraries
- Avoid rewriting subroutines in XSLT
- Integration requirement

# Further benefits

## CPU-Intensive Tasks

- Mathematical algorithms
- Recursive operations
- String processing

# Implementation Details

- SaxonJ
- SaxonC

# SaxonJ

## Java Implementation

To define a user extension function: - extend the `ExtensionFunction` class:

- Specify callback function name
- Argument types
- Return type
- The `call` method

# User Extension Function in SaxonJ

java implementation.

## 1. We extend the *ExtensionFunction* class

```
import net.sf.saxon.lib.ExtensionFunction;
import net.sf.saxon.om.StructuredQName;
import net.sf.saxon.s9api.*;
import net.sf.saxon.value.*;

public class SqrtFunction extends ExtensionFunction {
    private static final StructuredQName FUNCTION_NAME =
        new StructuredQName("www.example.com", "sqrt");

    // Function name: www.example.com:sqrt
    @Override
    public StructuredQName getFunctionQName() {
        return FUNCTION_NAME;
    }

    // Return type: xs:double
    @Override
    public SequenceType getResultType(SequenceType[] suppliedArgumentTypes) {
        return SequenceType.makeSequenceType(ItemType.DOUBLE, OccurrenceIndicator.ONE);
    }

    // Parameter types: (xs:integer, xs:integer)
    @Override
    public SequenceType[] getArgumentTypes() {
        return new SequenceType[] {
            SequenceType.makeSequenceType(ItemType.LONG, OccurrenceIndicator.ONE),
            SequenceType.makeSequenceType(ItemType.LONG, OccurrenceIndicator.ONE)
        };
    }
}
```

## 2. The call method

```
@Override
public XdmValue call(XdmValue[] arguments) throws SaxonApiException {
    long iterations = ((XdmAtomicValue) arguments[0]).itemAt(0).getLongValue();
    long context = ((XdmAtomicValue) arguments[1]).itemAt(0).getLongValue();
    double total = 0.0;

    for (int i = 1; i <= iterations; i++) {
        total += Math.sqrt(i) + context;
    }

    return new XdmAtomicValue(total);
}
```

### Explanation

- arguments[0] → number of iterations (xs:integer)
- arguments[1] → context value (xs:integer)
- Returns the total as xs:double

## 3. Registration of User Extension Function in Java

```
// Create SaxonJ Processor (EE or PE)
Processor proc = new Processor(true); // true = Enterprise Edition

// Create and register the extension function
SqrtFunction example = new SqrtFunction();
proc.registerExtensionFunction(example);
```

## 4. Programmatic Invocation (Java)

```
XsltCompiler compiler = proc.newXsltCompiler();
XsltExecutable exec = compiler.compile(
    new StreamSource("sqrt_calculate.xsl");
XsltTransformer transformer = exec.load();

XdmValue result =
    transformer.callTemplateReturningValue("main", null);
```



The user extension function can now be called from XSLT, XQuery or XPath using:

**example:sqrt(100000, 5)**

# SaxonC

- Built using GraalVM Native Image technology (AOT compiler)
  - SaxonJ compiled to a native shared library
- C/C++ API
- Python API
- PHP API

# SaxonC

## C++/Python/PHP Implementation

To define your own user-defined Extension Function:

- Write the function in the host language (C++, Python, or PHP)
- Arguments passed as an array
- Function registration
  - Method signature specified as JSON string
  - Function pointer passed to the processor

# XSLT Example with User defined function written in C++ - Math sqrt

An XSLT stylesheet calls a user-defined function implemented in C++ that calculates the sum of square roots.

## XSLT Stylesheet (calls xf:calculate)

```
1 <xsl:template name="xsl:initial-template">
2   <results>
3     <!-- Repeat the expensive function 100 times -->
4     <xsl:for-each select="1 to $rpt">
5       <run number="{.}">
6         <xsl:value-of select="xf:calculate($input, .)"/>
7       </run>
8     </xsl:for-each>
9   </results>
10 </xsl:template>
```

Calls  
xf:calculate(\$input, .)

## C++ User Function (xf:calculate)

```
1 XdmValue * calculate(int argc, XdmValue ** argv) {
2   long iterations =
3     ((XdmAtomicValue *)argv[0])->getHead()->getLongValue();
4
5   double context =
6     ((XdmAtomicValue *)argv[1])->getHead()->getDoubleValue();
7
8   double result = 0.0;
9
10  for (long i = 1; i <= iterations; i++) {
11    result += std::sqrt(static_cast<double>(i)) + context;
12  }
13
14  return processor->makeDoubleValue(result);
15 }
```

## How it works



XSLT Source  
(Stylesheet)

Compiled &  
processed by  
Saxon Processor



XSLT calls  
xf:calculate(\$input, .)

User Function  
Interface  
(Saxon C API)

Invokes  
C++ function

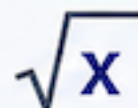


C++ Function  
calculate()  
(Sum of sqrt(i) + context)

Returns  
xs:double result

Result to XSLT  
(xs:double)  
21082060.7469...

Result used in stylesheet output



**What the function calculates:** 
$$\text{result} = \sum_{i=1}^{\text{iterations}} (\sqrt{i} + \text{context})$$
  
The C++ function performs the calculation natively for high performance and returns the result to XSLT as an XDM value (xs:double).

# Register user Extension function in SaxonC C++ API

Before an XSLT stylesheet can call a user-defined function, it must be registered with the SaxonC processor.

## 1. Registration API – Function Signature

```
void registerUserFunction (const char *ns, XdmValue *(*func)(int, XdmValue **),  
                           const char *signatureJSON, bool useNativeTypes=false)
```

**ns**

The namespace URI  
for the functions

**func**

C++ function pointer  
that implements the logic

**signatureJSON**

JSON string that describes the  
function name, return type,  
parameter types and arity

**useNativeTypes (optional)**

If true, uses native types for  
parameters and return values  
(default is false)

## 2. Example Registration Code

```
// Create SaxonC processor  
SaxonProcessor * proc = new SaxonProcessor(true);  
  
// Register the user function  
proc->registerUserFunction ("www.host-lang-func.com",  
    "{ \"calculate\": {  
      \"as\": \"xs:double\",  
      \"param\": [  
        {\"xs:integer\"},  
        {\"xs:integer\"}  
      ],  
      \"arity\": [2]  
    }  
  } }");
```

**&calculate**

**&calculate**

C++ function pointer  
that implements  
the user-defined  
function.

**signatureJSON**

JSON string that describes  
the function (matches the  
const char \*signatureJSON  
argument)

## 3. signatureJSON – What it describes

```
{  
  "calculate": {  
    "as": "xs:double",  
    "param": [  
      "xs:integer",  
      "xs:integer"  
    ],  
    "arity": [2]  
  }  
}
```

Function name  
(used in XSLT as xf:calculate)

Return type  
(XDM type)

Parameter types  
(two parameters,  
both xs:integer)

Function arity  
(number of parameters)

## 4. How it fits together



After registration, the function can be called in XSLT using its namespace and name:  
Prefix **xf** bound to "www.host-lang-func.com" → use **xf:calculate(...)** in the stylesheet.

# User extension function in Python

the same math sqrt user function, but implemented in Python.

## 1. Registration API – Function Signature

```
register_user_function(ns, func, signatureJSON, use_python_types=False)
```

**func**

Python callable that  
implements the logic

**use\_python\_types** (optional)

If True, uses native Python types  
for parameters and return values  
(default is False)

## 2. Example Registration Code (Python)

```
# Create SaxonC processor
proc = PySaxonProcessor(license=True)

# Python user function implementation
def calculate(iterations, context, **kwargs):
    result = sum((math.sqrt(i) + context) for i in range(1, iterations + 1))
    return result

# Register the Python user function
proc.register_user_function(
    "www.host-lang-func.com",          # ns
    calculate,                         # func
    signature_text="{ \"calculate\": {
        = \"as\": \"xs:double\",
        = \"param\": [
        =   \"xs:integer\",
        =   \"xs:integer\"
        = ],
        = \"arity\": [2]
    }"
    }"                                # signatureJSON
    use_python_types=False            # use_python_types (optional)
)
```

## 3. Compile and Run XSLT

```
xslt30 = proc.new_xslt30_processor()
executable = xslt30.compile_stylesheet(stylesheet_file=stylesheet_file)
result = executable.call_template_returning_value(initial_template)
```

# Benchmark Setup

- Apple M3 Max with 64 GB memory
- SaxonJ & SaxonC both EE edition
  - Java
  - C++
  - Python

# Benchmark

## Pure XSLT vs User-defined extension function

1. Square root calculations on a set of numbers
2. Abs() function on a set of numbers
3. Tree walk with string-length computation
4. Tree walk with predicate filter on 10 MB XMark XML document.  
10,000 repeated

# Benchmark Results

| Benchmark                                |  SaxonJ<br>XSLT |  SaxonJ<br>Func |  SaxonC-<br>Python<br>XSLT |  SaxonC-<br>Python<br>Func |  SaxonC-<br>C++-XSLT |  SaxonC-<br>C++-Func |
|------------------------------------------|--------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| <b>math:sqrt()</b><br>sum<br>calculation | 9 s                                                                                              | <b>0.63 s</b>                                                                                      | 71 s                                                                                                          | 55 s                                                                                                          | 71 s                                                                                                    | <b>0.76 s</b>                                                                                           |
| <b>abs()</b>                             | <b>0.052 s</b>                                                                                   | 0.066 s                                                                                            | 0.08 s                                                                                                        | 0.23 s                                                                                                        | 0.086 s                                                                                                 | 0.21 s                                                                                                  |
| <b>Tree walk</b><br>string length        | <b>7.2 s</b>                                                                                     | 7.6 s                                                                                              | 28.1 s                                                                                                        | 77.8 s                                                                                                        | 23 s                                                                                                    | 202 s                                                                                                   |
| <b>Tree walk</b><br>predicate            | 0.022 s                                                                                          | 0.042 s                                                                                            | <b>0.017 s</b>                                                                                                | 0.342 s                                                                                                       | 2.59 s                                                                                                  | 2.64 s                                                                                                  |

# Where is the Time Spent?

- XSLT/XPath/XQuery evaluation
- Host-language callback overhead
- Type conversion
- Python runtime overhead

# Observations

- User-defined functions provide **large gains** for CPU-intensive mathematical workloads.
  - `math:sqrt()` computation benefits from host-language implementation
  - `abs()` shows little benefit from a host-language implementation
- Tree-walk operations are generally **better expressed directly in XSLT**.
- C++ user functions achieve performance close to native Java.

# Key Takeaways

## User-Defined Extension Functions: A Powerful Optimisation Tool

-  **Use host-language user extension functions** for computationally intensive algorithms and recursive operations.
-  **Large performance gains** are possible when logic is difficult or inefficient to express in XSLT or XQuery.
-  **Keep XML processing in XSLT/XQuery** for tree navigation, predicates and document-oriented transformations.
-  **Use selectively**—choose the technology that best matches the task.

# Future Work

- More experiments for additional languages:
  - PHP, C# and JavaScript
- Use the results to help improve XML processors
- Include additional XSLT processor implementations

# Questions?

## Thank you



**Source code, benchmarks, experiments, and presentation slides**

<https://github.com/Saxonica/balisage2026-user-function-paper>