The CrossTwine Project

Technical White Paper

Ever been bitten by the suboptimal performance of a dynamic language? Tired of having to “drop down” to C to implement your business logic? Interested in providing your customers with either the Python®[1] or Ruby language for scripting, but wary of the consequences? Wondering how to bring your own proprietary language up to modern standards without hiring a team of virtual machine experts? This paper's objective is to present you with the current state of the CrossTwine technology, and let you assess how well it addresses your concerns.

About the author: after many years spent playing a “wizard” in the software industry, Damien Diederen decided to do something about the performance of dynamic languages. He can be reached at

dd@crosstwine.com

or by calling

+49 174 34 89 428.

Introduction

Since the dawn of computing, software companies have tried to move to higher- and higher-level languages as soon as reasonable, to bring projects to completion faster and gain a competitive advantage. A rapid evolution of hardware and implementation techniques allowed the world to move from assembly to C, C++, and later to Java and C#. Interesting outliers such as Lisp and Smalltalk managed to fill niche markets, putting their users in a strange situation: stuck with “dead horses,” yet grateful for their powerful and unique capabilities.

Even though Sun and Microsoft provide very attractive platforms with reasonably versatile languages, a few have started turning their heads to a range of alternatives that offer some of the benefits of Lisp and Smalltalk, and have graduated from “scripting” to “dynamic” status by virtue of being put to serious use. More than a few, actually—enough for both giants to start incorporating support for some of them into their platforms.

The usual caveat is that the available implementations of these productivity enhancers are quite slow. We will explore a number of the reasons in this paper, but suffice to say that Sun and Microsoft plan to leverage their existing technology to provide new implementations—and use them as bait to attract users to their respective environments.

The CrossTwine project takes a different stance: our goal is to augment **existing implementations** with advanced technology, retaining all of their flavor and goodies, and without imposing a “platform tax.” While doing so, we develop a toolkit of building blocks and a portfolio of recipes that permit achieving “economies of scale” across languages. The results of our efforts are commercially-supported, performance-enhanced, “professionalized” drop-in replacements that allow and follow evolutions of the language and remain fully compatible with all but the most exotic extension and embedding scenarios.

1 “Python” and the Python logos are trademarks or registered trademarks of the Python Software Foundation, used with permission.

Copyright © 2009 Damien Diederen, crosstwine.com

Business Environment and Challenges

The videogame industry has been a forerunner: their focus on delivering excellent A.I. and gameplay, both of which require a lot of experimentation and a fast turnaround despite the pressure of extremely short deadlines, has resulted in an increasing adoption of the Open Source language Lua as a component of modern games.[2] Openness, simplicity, high-level of abstraction, and easy extensibility/embedding are often touted as key factors for that commercial success.

More recently, Adobe released Lightroom, an end-user photo manipulation product which also embeds Lua. This time, however, it did not happen under the hood: the language was used to write the complete user interface of the application—40% of the total code base. That audacious choice allowed the interaction designers to come up with a novel user interface that has since proven quite popular.

The Python programming language is often embedded within a larger software system to provide scripting functionality.

The Python programming language is probably the most prominent such language; the same tenets of openness, simplicity and extensibility are usually mentioned as reasons for its popularity. Companies such as Google, USA (NASA's main Shuttle support contractor), Industrial Light & Magic and a lot of others have been stress-testing the default implementation for years—and returned to it after each reevaluation.

Ruby, yet another contender in the same space, stepped out of its native Japan in dramatic fashion when David Heinemeier Hansson released "Ruby on Rails,[3]" a web development framework which created and instantly filled a new category of its own. "Don't repeat yourself" is the motto, and the framework leverages the reflective capabilities of the language in an unprecedented way to achieve its objective of being "optimized for programmer happiness and sustainable productivity."

Excellent. And Good Enough

Most testimonials praise the following qualities, which apply to each of the languages under consideration:

- **Openness** cannot be underrated. People are—not unreasonably—wary of depending on single-vendor proprietary technologies. Becoming the milking cow is to be avoided at all cost, be it by using "inferior" solutions;
- **Simplicity** is probably is a close second. These languages have been gathering feedback from large and vibrant communities of users for years, feedback which has been refined and integrated by a "benevolent dictator" who keeps an eye on the general design;
- **Extensibility** is key for two reasons: first, it is very unlikely that the base language would provide a special effects company with the exact tools they need, yet custom add-ons have to fit in homogeneously lest simplicity be lost; second, being able to drop down to C/C++ to implement the bits exhibiting inadequate performance is often required;

2 http://lua-users.org/wiki/LuaUses
3 http://rubyonrails.org/

- **Embedding** proves to be important at some point in most scenarios, either because it becomes desirable to integrate some of the knowledge and assets developed on sideband projects into the mainline, or because the prototyping abilities and productivity improvements brought to the table by dynamic languages find a growing place in traditional product development.

The Problem

Faster implementations would enable the use of dynamic languages in a lot more situations.

Quite a few of the testimonials also mention how the languages are “fast enough.” Which, while true, is not exactly a praise: no matter how thankful both developers and managers are for the high-level goodness, the not-uncommon need to drop to low-level plumbing and integration certainly results in some cursing. Not to mention the swearing that happens when an otherwise perfectly adequate language has to be rejected for a task—just because of the performance characteristics of its implementation.

Moreover, companies have started turning away from the “throw hardware at it” mantra: they realize that tremendous economies can be achieved by consolidating some computations “in the cloud,[4]” where IT support costs are inexistent, but billing is directly proportional to the amount of processor time used. Performance matters again, while productivity doesn't cease to.

With their supporters having to choose between existing, flexible but slow implementations, and the promise of (potentially) faster ones tied to heavyweight—if not proprietary—platforms such a Java and .net, we believe that most popular dynamic languages are not achieving their potential level of adoption.

The fierce competition in the web browser space pushed Google, Apple, Mozilla and Opera to recently start competing on the performance of their JavaScript implementations. As JavaScript, the “web page language,” shares a lot of characteristics with Python, Ruby and other siblings, this proves to the world that dynamic languages don't *have* to be slow.

Not everybody is going to jump ship, however: not only must existing investments be preserved, but JavaScript lacks a useful standard library, a package system, and mature extension/embedding mechanisms. Instead, people expect the same improvements to be brought to the tools they use for their business-critical operations; or, as Joe Gregorio puts it:[5]

> *“What we are privileged to witness, something that wasn't happening a year ago, and will probably be complete in another year or two, is the professionalization of scripting languages. There was a time when you could whip out a parser in lex and yacc, stitch together a naive VM and throw it over the wall and you'd have a new scripting language. Those days are coming to a close…”*

The CrossTwine project aims to provide a de-facto solution to that problem, across languages, that solves it in the most efficient possible manner: by introducing an agile technology—and an associated methodology—which overcomes the deficiencies of the original implementations while retaining all their desirable qualities.

---

4 http://en.wikipedia.org/wiki/Cloud_computing
5 http://bitworking.org/news/321/The-Professionalization-of-Scripting-Languages

How CrossTwine Helps

There is certainly a lot of variety in the landscape we have been looking at, but the common lack of performance can be chiefly attributed to two properties:

1. Extreme dynamism, and
2. Simplistic implementations.

Let's assume that dynamism is a desirable feature, or it would have disappeared as the language evolved. An important objective of ours is to preserve the dynamism of these languages by applying **auto-adaptive solutions**—as opposed to "tuning knobs."

We believe the fear of vendor lock-in to be a major deterrent to the adoption of Smalltalk.

What might be less obvious is that the availability of a simplistic, but "clean" implementation is also desirable: it allows power users to assist the inventor in evolving the language, fostering a lively community and healthy progress curve. We certainly do not intend to disrupt that ecosystem: "forking" languages is not an objective; our solution intends to accelerate, but provide otherwise **indistinguishable behavior** from reference implementations.

Technical highlights

This section will focus on various common qualities of the target languages, the pain points caused by their simplistic implementations, and how our *CrossTwine Linker* kernel technically addresses them.

Language Hallmarks

Various trade-offs made during the design of a language evolve into "hallmarks" that are well defended by its supporters: the—sometimes arbitrary—initial choices morph into very useful specificities users come to rely on, and become an integral part of the experience.

Such hallmarks, however, actively prevent the cross-pollenization of implementations, and that lack of horizontal component renders the latter extremely vulnerable to the law of diminishing returns; after a while, slightly improving any particular aspect requires herculean efforts. Worse: when they happen, these efforts often end up being rejected because they tend to touch and "uglify" many parts of the system.

The idea of countering diminishing returns by leveraging cross-language economies of scale is not new: there have been quite a few attempts at overarching architectures, but they have proven to be an extremely difficult task. The recent "dynamic language" initiatives tied to the Java and .net platforms try to reach the same goal by extending and repurposing existing technology.[6]

Our unique approach is based on a "toolkit" of versatile components developed in heavily-templatized C++, which allows us to "inject" the platform into the language, as opposed to the more common solution of reimplementing the language on top of an existing, possibly tailored

6 These, however, are likely to succeed, if only because their sponsors spare no expense in trying to draw potential customers to the platform.

platform. Compile-time specialization permits code reuse without incurring runtime cost, whereas the low-level nature of C++ provides the required level of control.

Other benefits of the toolkit approach include:

- **Easy prototyping**: components are initially developed within a particular host to achieve an immediate objective, the useful commonality being extracted later, when the then-proven solution is generalized to accommodate other languages;
- **Incremental improvements**: improving upon an existing implementation allows us to keep the language—and its test suite—working at all stages of development, in TDD-style.[7] In addition to enabling extremely short iteration cycles, this permits the early discovery of the inevitable surprising corner cases.

No Mandatory Compilation Step

One of the distinguishing characteristic of most dynamic languages is that they do not require an explicit compilation step. Moreover, they usually provide an "eval" primitive that allows the injection of new code into the environment at runtime. Implementations widely differ in how they massage source code text into an exploitable data structure, e.g.:

- **Ruby 1.8** parses the source into an abstract syntax tree (AST); no further processing is done as the evaluator takes AST fragments as input;
- **Ruby 1.9** parses the source into an AST which is then compiled to direct threaded code to be executed by an efficient interpreter;
- **Python** parses the source into an AST which is then compiled to bytecode to be dispatched by a "big switch" interpreter.[8]

Interpretation can be faster than compilation for code that is rarely executed.

None of these translate the program into native code to be executed directly by the CPU. Instead, the implementations provide interpreters, which have the advantage of relative ease of maintenance and portability over native code compilers, but result in an additional level of overhead during execution.

Our kernel does not disable these mechanisms, but further analyzes their products, and prepends executable fragments with a tiny stub that hands control back when about to be interpreted. What happens on subsequent hits depends on "flags" that are set and updated by various mechanisms and heuristics inside the virtual machine:

- The stub is **inactive**: control is relinquished and execution continues using the default platform mechanisms;
- The fragment is **uncompiled**: an initial compilation is performed, during which an abstract bytecode closely modeled after the host language's own primitives is converted to generic native code.[9] Upon successful "just-in-time" (JIT) compilation, the stub becomes

7 http://en.wikipedia.org/wiki/Test-driven_development
8 The Python interpreter recently incorporated a patch to activate an alternative "computed goto" *dispatcher*, but the bytecode is left untouched.
9 The use of per-language, tailored instructions is a key factor in our approach.

compiled, and execution proceeds as per the next point;

- The fragment is **compiled**: the default platform mechanisms are cut short, execution is diverted to native code for the whole fragment.[10] Compiled is the normal state for frequently-executed code paths.
- The fragment is **pending recompilation**: as in the uncompiled case, the abstract bytecode is JIT-compiled to native code, but this time with profile-guided optimizations (PGOs[11]) enabled. Pending recompilations are caused by various active subsystems detecting opportunities for optimistic compilation, or repeatedly violated invariants; these are discussed in the following subsections.

While native compilation brings a non-negligible amount of performance in itself, it also lets us instrument the code to gain insight about its dynamic execution profile, to be used later in PGO: lightweight bookkeeping is done on execution counts, the concrete types seen at method call sites are remembered, etc.; basically, a record is kept for every operation that involves dynamic behavior, to allow the framework to later deduce the static equivalent.

Note that even though JIT-generated native code is not a hard requirement for monitoring the early runs, it has been chosen over instrumenting the interpreters for the following reasons:

- Instrumenting an interpreter requires invasive changes, and we would rather leave the default mechanisms untouched. Moreover, such changes would have to be statically compiled in, which makes it impossible to totally disable them at runtime.
- The various bookkeeping utilities would have be written in C, and could not rely on our cross-language infrastructure. As we have complete control over native code, we can arrange for it to call reusable utilities which have been customized using template specialization.

CrossTwine Linker does not currently make use of the LLVM (llvm.org) framework. Some of our techniques require full control of the generated code.

Our code generators currently only support the ubiquitous AMD/Intel x86-64 instruction set. Other backends could be implemented given enough demand, an obvious candidate being the ARM architecture, which is very popular in the embedded market. We are still waiting to see how well it fares in the presence of Intel's new “Atom” offerings—some of which implement the x86-64 ISA.

Dynamic Dispatch

Dynamic dispatch means that the mapping between an abstract “message” sent to an object and its concrete implementation is delayed until the latest possible moment. The dispatch mechanisms offered by our target languages offer a lot of flexibility, supporting declaration-free coding as well as techniques such as “duck-typing” and “monkey-patching.”

The fact that methods can be redefined at any time, however, combined with operators such as + offering the full flexibility of a method call, puts

---

10 The native code might, however, decide to drop back to the interpreter under exceptional circumstances. Conversely, multiple fragments can be chained without involving the interpreter.

11 http://en.wikipedia.org/wiki/Profile-guided_optimization

Copyright © 2009 Damien Diederen, crosstwine.com

these languages at a serious performance disadvantage compared to their mostly-static equivalents.

Indeed, a considerable amount of work has to be shifted at runtime, even though most call sites up end up repeatedly invoking the same primitives. The default Python and Ruby implementations use a variety of caching schemes to speed up the common case, but a lot of things remain to be determined every time a method call happens.[12]

Our solution implements stateful call sites, which combine inline caching[13] (IC) with optimistic, profile-guided compilation. In their base state, call sites perform minor bookkeeping around dynamic dispatch via the default mechanisms, and notify their associated “lifecycle manager” of interesting events such as the repeated invocation of a same implementation.

“Hot” call sites matching one of the patterns found in an (extensible) library of optimizable cases are quickly transitioned to a corresponding advanced state. A variety of such states exist per host and operation, and can have effects ranging from thunk patching (a limited form of self-modifying code) to complete optimistic recompilation. Advanced states must:

1. Minimize the runtime impact of the call site;
2. Provide a fallback “slow path,” which performs additional checks and bookkeeping before either fixing-up or dropping down to less optimized code;
3. Guard the optimized fast path, and divert to the fallback in the event its invariants are not met.

Point 1 is the raison d'être of stateful call sites. As an extreme example, consider the use of the + operator in the expression `3+x`, where `x` is known to hold a floating-point value most of the time: during optimistic recompilation, the full method call—which includes call frame setup, type coercion of the first argument, and boxing of the result—gets replaced by a single `addsd` instruction.

Points 2 and 3 are pure overhead. To minimize the impact of point 3, which stands in the critical path, optimistic guards are usually used; they consist in a very short instruction sequences which can lead to false negatives, in which case a simple fix-up is done by the provided fallback before reverting to normal execution (*minor* vs. *major* misses).

A limited, but growing amount of inter-site collaboration helps in partly eliminating redundant checks and unnecessary boxing so that e.g. `*` in the larger `(3+x)*2` expression suppresses a test for “floatness,” and receives the intermediate result via a floating-point register. Similarly, the literal 2 is coerced at compile-time, and loaded directly into another floating-point register. The general mechanism is abstracted as a concept of “transfers.”

Note how no accounting is done in the fast path; an orthogonal mechanism provides an estimate of the number of executions for each extended basic block; these counters are used by heuristics to decide whether to

---

12 The Python and Ruby 1.8 interpreters use static caches, whereas Ruby 1.9 opted for a more elaborate inline caching mechanism, with one slot per call site. While possibly more efficient, the interpreted nature of the VM still prevents the latter from caching more than just method lookup.

13 http://en.wikipedia.org/wiki/Inline_caching

deoptimize the fast path in the case of repeated major misses.

At all times, each particular instance of generated code is under the control of a “lifecycle manager” which communicates with the stateful call sites, schedules recompilations, and recycles unused code buffers. Multiple versions of generated code can be alive for each stubbed fragment at any given time, be it to be used as a fallback or because they are active in the call chain of one thread of execution.

It was previously mentioned that we are not using LLVM, but that framework might reenter the picture later as a powerful way to optimize combinations of very hot fragments that can be optimistically reduced to long chains of primitive operations.

Property Access

As far as property access is concerned, objects in the aforementioned languages consist of little more than dictionaries, *per-instance* mappings of names to values, which has the following implications:

- Variables can be added/removed to instances at any time;
- A full “dictionary lookup” is required for each access;
- Variables do not exist at a fixed offset across a set of objects sharing a type.

Again, these have a major effect on the performance that can be expected compared to member access in more static languages.

Our kernel currently speeds up repeated dictionary lookups with a constant key by keeping a finger on the last entry; this strategy is particularly effective in loops which execute numerous accesses to the same property. Given that in the Python programming language, for example, method calls are split into distinct lookup and invocation phases, and that the former goes through generic property resolution, this condition is not as uncommon as it may seem.

An experimental implementation of Self-style maps, which, combined with a dedicated IC, should provide property access times comparable to what static languages enjoy, is currently disabled because of a net slowdown in a number of synthetic benchmarks—for reasons that still are to be investigated; this slowdown is going to be one of our points of focus in the near future.

Current Status

After an initial research phase, our focus was moved to 100% compatibility.

So far, our technology has been implanted in the “standard” development versions of CPython (3.1alpha+), and of the next-generation Ruby interpreter (1.9.1+). The augmented virtual machines pass the full test suite of their respective language with the same results as the default interpreter, including:

- **Garbage Collection** tests: a naive implementation of ICs and other caching mechanisms can result in object retention, and prevent proper garbage collection; our components use a variety of weak reference and guard “tricks” to avoid withholding more objects than

the basic environment;

- **Introspection** tests: even when running natively compiled code at full speed, our implementation reports correct source line numbers and fully supports various other introspective facilities lots of modules and utilities rely on;
- **Debugging** tests: the Python platform supports a number of advanced debugging features such as line-by-line tracing and jumping, which are exploited by the debugger to provide its services. Our implementation supports activation of the tracing functionality at arbitrary points, including when native frames are present in the call chain.

Note that we also have a—very preliminary—augmented version of the former generation Ruby interpreter (1.8+): that version is very likely to remain in use for a long period of time as it is currently used in production on most “Rails” sites, and the two generations of the Ruby language are not quite compatible.

The numbers in this section reflect the state of CrossTwine Linker as of February 23, 2009.

Our focus being on performance, we have been tracking our progress using a number of more-or-less synthetic benchmarks. While benchmarks do not necessarily reflect real-world scenarios, they are useful tools for isolating inefficiencies, and subsequently measure the effectiveness of targeted solutions. We present some of the observed results in the following sections.

Note that the timings of the augmented versions comprise the complete startup and shutdown of the virtual machine, including the various analysis, bookkeeping, and compilations tasks performed by our embedded kernel.

CPython 3.1alpha+

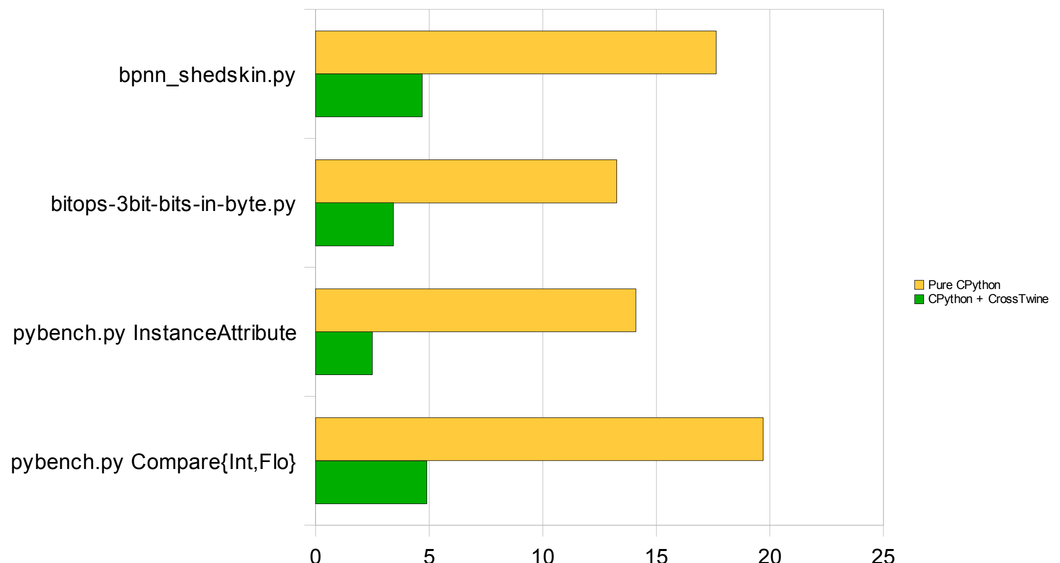

Illustration 1: CPython 3.1alpha+ Benchmark Results

CPython, the default and most widely used Python interpreter, was our first target. Illustration 1 presents the current execution times in seconds for the

following benchmarks (lower is better):

- `bpnn_shedskin`: A back propagation neural network benchmark based on a real-world use case. Compute intensive, it performs numerous floating-point operations on the elements of matrices stored as object properties (stressing floating-point math, property access, and array access). Our engine results in a global speedup of 3.8×.
- `bitops-3bit-bits-in-byte`: A synthetic benchmark lifted from the SunSpider suite, commonly used to compare modern JavaScript implementations. Stresses local variable access, and low-level bit-twiddling operations; global speedup is 3.9×.
- `pybench.py` InstanceAttribute, Compare{Int,Flo}: Synthetic benchmarks bundled with the Python distribution. Stress object access and number comparisons; speedups are 5.6× and 4.0×, respectively.

A lot of other performance tests have been built from various sources, and periodic runs have been setup to monitor the performance impact of our changes, e.g.: Knuth–Morris–Pratt string searching algorithm, 2.9×, binary tree search, 3.5×. In general, we expect our current Python implantation to result in a speedup of 2 to 4× on compute-intensive tasks.

MRI 1.9.1+

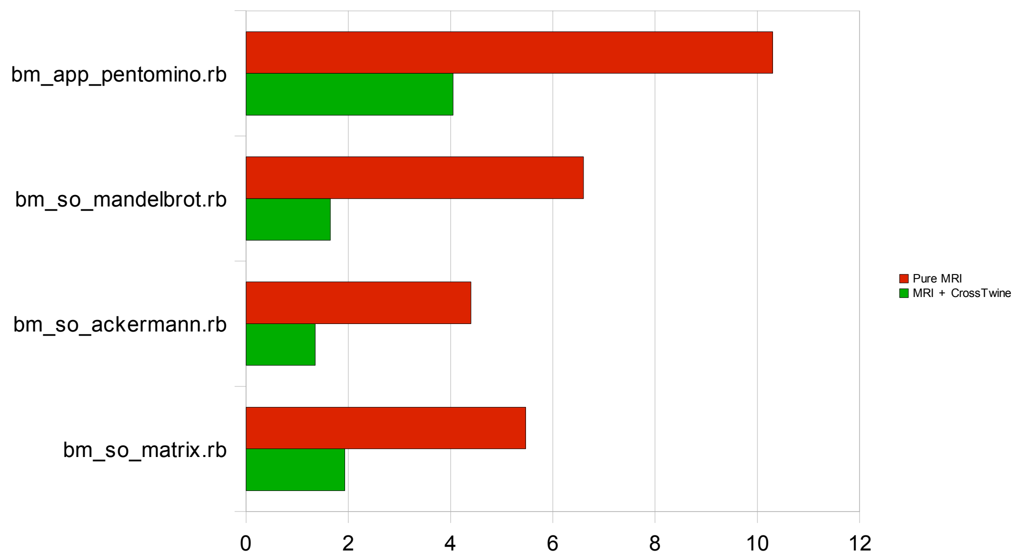

Illustration 2: MRI 1.9.1+ Benchmark Results

MRI (Matz's Ruby Interpreter) 1.9.1 is the basis for the still-to-be-released Ruby 2.0 language. Illustration 2 presents the current execution times in seconds for the following benchmarks, which come from the suite bundled with the implementation (lower is better):

- `bm_app_pentomino`: A memory intensive benchmark which stresses array access and mapping operations (i.e. `array.each { |x| … }`); speedup is 2.5×.

- `bm_so_mandelbrot`: Generates a bitonal representation of the Mandelbrot set; stresses iteration primitives and floating-point operations. Speedup is 4.0×.
- `bm_so_ackermann`: A very synthetic recursive benchmark which stresses control flow; speedup is 3.3×.
- `bm_so_matrix`: performs multiplication of large integer matrices, global speedup is 2.8×.

While the execution engine underlying the default MRI 1.9.1 is YARV, a fairly efficient interpreter which leaves less "low-hanging fruit" to be collected than the Python one, we expect our current Ruby implantation to also yield a speedup of 2 to 4× on compute-intensive code.

The "transfers" mechanism still has to be reintegrated in the Python branch.

This is because this more recent integration uses a newer generation of the technology incorporating a lot of improvements that still have to make their way into the Python line; we expect further speedups in the latter as soon as we start reintegrating these improvements.

MRI 1.8.6+

While our adaptation of MRI 1.8.6 is in its very early stages, initial testing already show speedups of 3× and 4.1× on the `bm_so_mandelbrot` and `bm_app_pentomino` benchmarks—while still passing the complete test suite.

Even though its underlying execution engine is radically different from the one in 1.9.1, and is based on AST interpretation instead of bytecode, we have been able to perform the integration and obtain the results above after only four days of development—a testimony of the flexibility of our framework and methodology.

Conclusion

Despite its youth, the *CrossTwine Linker* toolkit already goes a long way towards making the "dynamic languages are slow" concern a thing of the past.

We do not invent a new language, nor do we reimplement enormous amounts of libraries and modules. Instead, we rely on the vast experience and effort gathered by various communities during many years, and provide a final push to bump existing implementations over some of their barriers of adoption.

These languages already have lots of notable qualities, and will evolve to gain many others; while contributing negatively to performance, simplistic implementations are an important vehicle for that evolution to happen. A totally orthogonal approach allows us to provide commercially-supported, high-performance, yet fully compatible alternatives for the scenarios that require them—relieving their adopters from a serious source of concerns.

Please visit crosstwine.com or email info@crosstwine.com for additional information.

Copyright © 2009 Damien Diederen, crosstwine.com