Efficient floating-point number handling for dynamically typed scripting languages

Shiro Kawai
Scheme Arts, L.L.C.
shiro@schemearts.com

ABSTRACT

Typical implementations of dynamically typed languages treat floating-point numbers, or *flonums*, in a "boxed" form, since those numbers don't fit in a natural machine word if a few bits in the word are reserved for type tags. The naïve implementations allocate every instance of flonums in the heap, thus incur large overhead on numerically intensive computations. Compile-time type inference could eliminate boxing of some flonums, but it would be costly for highly dynamic scripting languages, in which a compiler runs every time a script is executed.

We suggest two modified stack machine architectures that avoid heap allocations for most intermediate flonums, and can be relatively easily retrofitted to existing stack-based VMs. The basic idea is to have an arena for intermediate flonums that works as a part of extended stack or as a nursery. Like typical VMs, flonums are tagged pointers that point to native floating-point numbers, but when a new flonum is pushed onto the VM's stack, it actually points to a native floating-point number placed in the arena. Heap allocation only occurs when the flonum pointer needs to be moved to the heap. The two architectures differ in the strategies to manage the arena.

We implemented and evaluated those strategies in a Scheme implementation "Gauche." Both strategies showed 30%-140% speed up in numerical computation intensive benchmarks, eliminating 99.8% of heap-allocation of intermediate flonums, with little penalty in non-numerical benchmarks. Profiling showed the speed improvement came from the elimination of flonum allocation and garbage collection.

1. INTRODUCTION

It is common to use a tagged word to represent an object in the runtime of dynamically typed languages; most objects are allocated in the heap and a tagged word holds a pointer to it, while some frequently used objects, such as small integers, are directly embedded in a word. The width of the word is typically chosen to match the 'natural' width of the underlying CPU for the efficiency, i.e. 32 or 64 bits.

Embedding small integers in a tagged word greatly reduces generation of garbage and improves performance a lot, since such numbers are mostly for intermediate results and need to exist only for short amount of time. This technique, however, can't be applied directly to the floating point numbers, or *flonums*. IEEE-754 single or double precision floating point number format is preferred for the efficiency, but it leaves no room for tag bits.

A naïve approach is to allocate a floating-point number in the heap and uses a tagged pointer to point to it, just like other heap-allocated objects. It incurs a large overhead in computation-intensive programs by allocating and collecting lots of short-living intermediate flonums.

This problem has been well recognized for long. Dynamic language implementations that use separate compilation passes have tried to reduce the overhead by identifying variables that only hold floating point numbers, through optional type declarations[3], storage analysis[11], or type inferences[8, 14].

Nevertheless, quite a few popular implementations of "scripting languages" are still using the naïve approach[1]. We suspect it is partly because scripting languages started with emphasis on rapid development of throw-away programs and gluing existing components, and performance wasn't a priority at first. However, as they have grown to full-fledged languages and large body of libraries and applications have written in them, performance has become more important and most modern scripting language implementations employ a virtual machine (VM) approach[2]—program code is in a source form, and when it is invoked it's compiled to VM instructions, then executed on a VM.

This approach yields much better performance than pure interpreters, but the compilation takes place every time when the program starts up. It limits the amount of time the compiler can spend, preventing the implementation from adopting sophisticated optimization techniques. Spending whether 100ms or 1s for compilation would make huge difference in cgi scripts, for example. It is much desired to develop techniques that avoid heap allocations of floating-

[1]E.g. Python 2.5.2, Ruby 1.8.7, Perl 5, PHP 5.2.6, and Tcl 8.5.2

[2]Even if a language begins with a direct interpreter in favor of the simplicity, it seems inevitable that its core is reimplemented by VM when the language lives more than several years and gets certain user population. Tcl originally started as an interpreter in 1988, and incorporated a bytecode VM in 1996[7]. Ruby was initially developed in 1993 as a tree interpreter and adopted a VM in 2006[10].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

point numbers without taxing compilation time.

We faced this issue in the Scheme scripting engine *Gauche*[3], and addressed it by modifying its VM to avoid flonum allocations as much as possible. The benchmark shows our technique is quite effective in numerically-intensive programs, and otherwise has little impact on other programs. Required modification are confined to the VM itself, with a slight modification of a glue-code generator that allows C functions to be called from Scheme. Some core numerical calculation routines (implemented in C) are modified for further optimization. We believe this technique can be applied in general to the stack-based dynamically-typed VMs.

In the following sections, we first define our goals and discuss existing techniques, then describe our approach in detail, followed by benchmark results and discussions.

2. DESIGN GOALS AND EXISTING TECHNIQUES

First, we make our goals clear. The principal goal is to reduce heap allocation of intermediate flonums. We also have to consider the following constraints.

Target language: We aim at dynamically typed scripting languages. We tested our idea in the Scheme implementation Gauche, although we believe the same technique can be applied to other scripting language families.

No penalties: The technique should not slow down the existing programs that don't use flonums much. Since scripts are compiled every time they are invoked, our goal implies we cannot add sophisticated optimizers to the compiler, such as type inferences[14], [8] or storage use analysis[11]. The technique should also avoid adding run-time overhead in the paths that are unrelated to the flonum operations.

No type declaration: It may be an option to allow an optional type declaration (as offered in Common Lisp), so that the compiler can use it to optimize flonum allocations[3]. Although it is a nice option to have, here we avoid it, since asking type declarations to speed up things tends to end up cluttering source code with declarations[4].

There have been several techniques suggested to achieve similar goals.

One of such techniques is *tagged unboxed floating point numbers*, which uses shortened representations of floating point numbers to put a few tag bits in it. Back in 1987, *Self* used 30bit floating-point number with two bits tag to fit a flonum in a 32bit machine register[2]. More recently, Jason Evans described a method to "steal" three tag bits from the exponent field of IEEE double-precision floating point numbers[5]. It can preserve 53-bit precision, but limits the exponent range between 2^{-127} and 2^{127}, assuming most "usual" instances of flonums that occur in typical applications fall in this range and thus fit in a single 64-bit word. Koichi Sasada also implemented a similar but more sophisticated technique in the Ruby runtime; he took only one bit from IEEE double, and with clever bit-operations he kept the overhead of extra type checking small and gained 39%–54% speedup in benchmarks[9].

The advantage of this method is that it won't add any overhead to non-flonum operations. On the other hand, since typical run-time tagging methods don't allow tag bits in the middle of a word, extra bit twiddling is required to extract native floating-point values from the truncated representation of flonums.

Another technique is to use invalid floating-point number bit-patterns (such as NaNs in the IEEE floating point numbers)[13]. In IEEE double precision floating point numbers, NaN is defined as all 1 exponent bits with non-zero mantissa, and denormalized numbers is as all 0 exponent bits with non-zero mantissa. It means there are $2^{52} - 1$ bit patterns of both. Even if we set two patterns aside for a quiet NaN and a signaling NaN, we can use the rest to encode other objects. It may be possible to take advantage of the hardware to signal NaNs as a lightweight type dispatching mechanism. This method requires the hardware's FPUs and other libraries to guarantee that they never produce NaN patterns that can be misinterpreted as other objects.

Both techniques run best on 64bit architectures. They are applicable to 32bit architectures with IEEE single-precision floating point numbers, but the loss of exponent range or the limited number of available bit patterns will be a severe restriction for general-purpose languages. (Using such restricted floating point numbers might be plausible for the embedded, special-purpose scripting, however.)

If we give up the idea of squeezing flonums in a machine word, there are still ways to avoid flonum allocations. Since we can assume intermediate results are short-lived, we can set-aside special "floating-point registers" in the VM to keep those intermediate numbers, avoiding calling heap allocators and putting a burden on GC. In fact, Jeffrey Mathews reported that adding one floating point register in the ocaml byte-code VM increased floating-point computation performance by 55-72%[6]. However, doing so requires the compiler's help to emit flonum-specific VM instructions, which is incompatible with our goal.

Yet dynamically typed languages can still use the idea, by effectively widening the VM data word sufficiently to hold floating-point numbers, like the register-based VM of Lua 5 does[4]. It adds overhead when the VM's data value is copied around, but in general it results in good benchmark score in computation-intensive tasks[7].

Our technique can be regarded as an extension of the floating-point register approach, but we apply it to *stack machines* instead of register architectures. Furthermore, we keep the VM data word the same as the underlying CPU word (either 32bit or 64bit), so it won't affect the performance of non-flonum operations.

3. OUR APPROACH

3.1 Gauche VM

Before going into the details, we explain the Gauche VM architecture we use as our testbed.

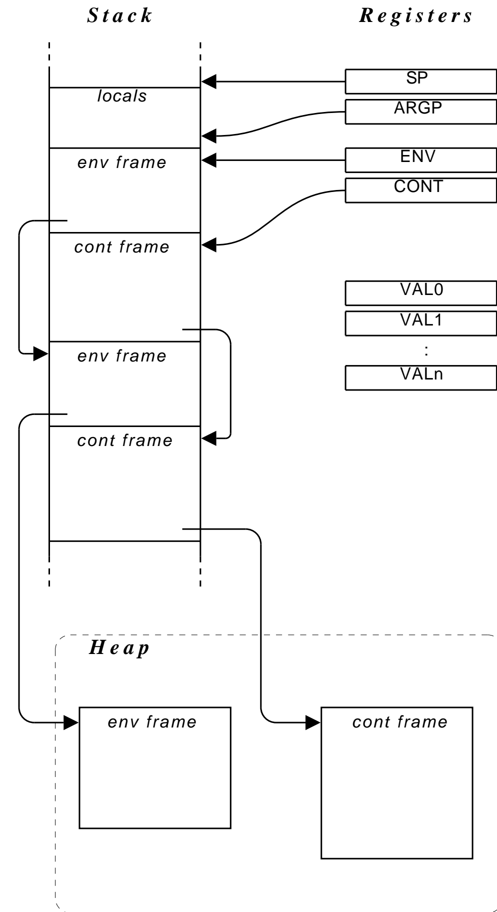
[3] `http://practical-scheme.net/gauche/`

[4] We don't deny the effectiveness of optional type declarations; they can be adopted orthogonally with the method we propose here.

[5] `http://www.canonware.com/~ttt/2007/07/tagged-unboxed-floating-point-numbers.html`.

[6] `http://dem.inim.us/ocamlfp/README-FP.html`.

[7] `http://shootout.alioth.debian.org/`.



The register ENV and CONT point to the head of the chain of environment frames (*env frames*) and continuation frames (*cont frames*), respectively. Env and cont frames may be on the stack or in the heap. Argument register ARGP keeps the bottom of intermediate local values on the stack. VAL0 through VALn holds the results of the most recent operation (VAL1... are used for multiple return values).

Figure 1: Gauche VM architecture

Gauche VM is a simple stack machine: It has a VM stack where environment and continuation frames are pushed. It also has value registers VAL0 ... where return values of a procedure are placed (See figure 1).

For a non-tail call, a continuation frame is pushed to the stack first, then arguments are evaluated and their results are pushed one by one. Finally the operator is evaluated, and result is invoked as a procedure, with the arguments on the stack, which become the topmost environment frame. When the procedure exits, the continuation frame is popped, discarding the environment frames above it altogether.

For a tail call, we start pushing arguments without pushing a continuation frame; after evaluating the operator, we shift the accumulated arguments between ARGP and SP down to the place just above the previous continuation frame, overwriting the environment of the current executing procedure, and jump to the procedure entry. This realizes tail call optimization[6].

The basic instructions operate between VAL0 and the stack; for example, the 'POP' instruction puts the content of the location pointed by the stack pointer (SP) into the VAL0 register and decrements SP, and 'PUSH' puts the value of VAL0 on the location pointed to by SP and increments SP. VAL0 effectively works as the cache of the stack top. It also has quite a few instructions that transfer data inside stack (e.g. "take the value of local variable #3 and push it onto the stack top.")[8]

Frequently used operations (e.g. 'cons' and '+') have dedicated VM instructions. If the core procedures are not redefined by the time the expression is compiled, they are inlined into those instructions.

When a closure is created, the environment frame chain is moved to the heap. When a continuation is captured, the continuation frame chain and the environment frames pointed from them are moved to the heap. The frames are also moved to the heap when stack overflow is detected. Once environment frames and continuation frames are moved to the heap, they sit there until being GC-ed. Thus, each frame may be moved at most once.

There are some more registers such as dynamic handler chains, but we omit them as they don't affect our discussion here.

The important invariance is that, except for a handful known registers, no data in the stack is pointed to from outside of the stack. For wider portability and easier integration with C, Gauche uses a conservative GC. So, in general, we can't move around data during GC. However, we can distinguish real pointers if they are in the VM's stack and registers, since we have full control over their contents. The invariance allows us to adjust all pointers correctly when data in the stack is moved around.

Since closure creation may trigger copying environment frames, closure elimination by lambda-lifting improves performance a lot. Gauche's compiler performs simple-minded closure optimization in a limited time, and typically converts (possibly mutual) recursion of inner closures into simple loops.

Gauche VM itself is written in C. Gauche's compiler is written in Scheme, and pre-compiled by Gauche itself to an array of VM instructions.

3.2 Fast Flonum Extension

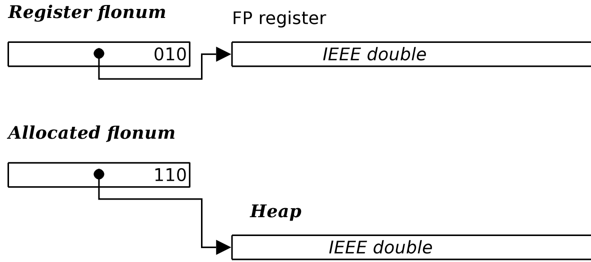
A flonum is represented as a tagged pointer that points to an IEEE double-precision floating number (we call it simply a *double* from now on). In the original implementation, all doubles are allocated in the heap. We call such flonums *allocated flonums*.

With our suggested method "fast flonum extension", an array of doubles are pre-allocated in the VM and serves as floating-point (fp) registers. When a new flonum is created, a double is stored in an fp register whenever possible, and the flonum pointer points to it. We call such flonums *register flonums*.

We use one bit in the tag to distinguish two kinds of flonums, as shown in figure 2. When we extract a double, we can just mask the tag bits in the flonum pointer without caring whether it is in the heap or in an fp register[9].

[8] Allowing operations to work with not only the stack top but deep in the stack, we can consider the VM as a sort of register machine with moving a register window. The distinction between stack and register VMs is rather vague.

[9] This doesn't affect the language semantics, since flonums



If Scheme object's lower bits are `010`, the pointer points to the location of an fp register. If they are `110`, the pointer points to a double allocated in the heap. Normal type checking is done by looking at the lower two bits, and value extraction is done by masking lower three bits, regardless of which one the flonum representation is.

Figure 2: Two variations of flonums

To keep the invariance that "only known pointers can point to a movable data", register flonums are allowed only in `VAL` registers or in the VM stack. If a register flonum needs to be moved to the heap, an IEEE double is allocated in the heap and the flonum is converted to an allocated flonum.

The key is *when* and *how* we move the doubles from an fp register to the heap. The registers will eventually gets full and we need to make room for new flonums. If we scan all pointers to find which fp register is in use, it defeats the very purpose of fp registers since it is just another GC. On the other hand, if we give up fp registers too early (e.g. only if value registers uses fp registers) the benefit of fp registers diminishes. We also don't want to use complicated algorithm to decide which fp register to be flushed.

Based on these considerations, we tested two strategies to manage fp registers.

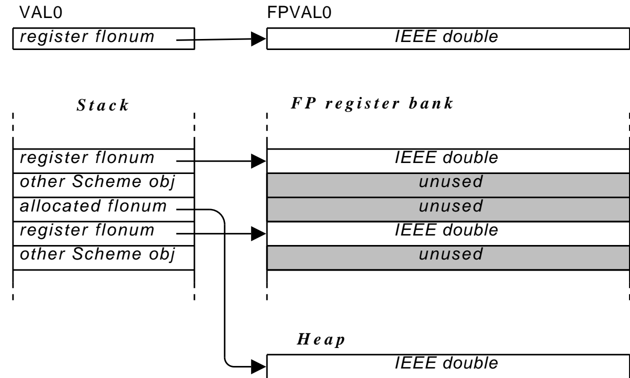
3.2.1 Wide-stack

The first strategy is to pair up every stack word with every fp register, and to make the stack allocated flonums use the paired fp register. We also add an fp register `FPVAL0`, to pair up with `VAL0`, etc. (See figure 3). Whenever a stack word is moved to the heap, the content of fp registers pointed to from the word is also moved to the heap, and the word is adjusted to point to the heap-allocated double. This effectively widens the stack and registers, so we call it the wide-stack strategy.

Note that it is slightly different from register machines with wide values like Lua. If we fetch a flonum from a heap-allocated object, the flonum refers to the heap allocated double; we don't need to copy the double value into a fp register.

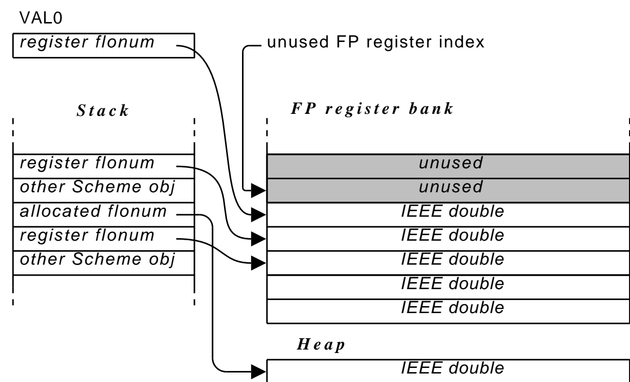
The advantage of this strategy is that we can straightforwardly determine which fp registers are in use, and by whom—as far as the stack is not full, there's always an accompanying fp register is available. There's an overhead, though. Whenever we move a value between stack and `VAL0`, shift the stack frame, or when a variable on the stack is `set!`, we have to check if the value is a register flonum, and if so, we have to copy the fp register and adjust the word to point to the new fp register.

of the same value doesn't need to be `eq?` in Scheme.



Value registers and every stack word have corresponding fp register. If a register flonum is placed in a value register or the stack, it always points to the corresponding fp register. It is possible that an allocated flonum is put in a value register or the stack, in which case it points to a heap-allocated double.

Figure 3: Wide-stack strategy



A new register flonum uses the fp register pointed by the unused fp register index. If the fp register bank gets full, the stack and value registers are scanned and all the doubles are moved to the heap.

Figure 4: Nursery strategy

3.2.2 Nursery

Another strategy is to use the fp registers as a nursery of doubles, as shown in figure 4. The VM keeps a pointer to a next fp register to be used. When a new flonum is created, the fp register pointed to is used and the pointer is incremented. When the pointer reaches the end of the fp register bank we flush fp registers by scanning the stack and move the live doubles into the heap. Like the wide-stack strategy, once a double is moved to the heap it will never moved back to VM registers. In this strategy, the fp register bank effectively works as a special nursery for doubles[10].

The advantage is that we can move the words around the stack and VM registers freely; only when we move the word

[10]This can be considered as a specialization of the "Cheney on the M.T.A." approach by Henry Baker[1], which uses a machine stack as a nursery of *all* objects.

to the heap do we need to check if it is a register flonum. The overhead depends on how frequent the flushing of fp registers occurs.

For the sake of benchmark comparisons, we allocate the same number of fp registers as with the wide-stack strategy, which is the same as the VM stack size (10000 entries).

We also tried the third strategy that used separate stacks, one for VM words and another for doubles. However, the overhead of managing two stack pointers, and of dealing with the case that one double was pointed to by more than one flonums, made the strategy much less attractive than the other two and we dropped it from the options.

3.3 Library interface

Flonum results produced in the VM's arithmetic operations are always register flonums. However, that's not enough in order to take full advantage of the fp registers: We also need to consider how this change affects the way the VM calls C-defined functions (foreign functions).

We can't pass register flonums to arbitrary foreign functions, since such functions may store flonums in the heap where the VM is not aware of, and thus impossible to adjust when the flonum is converted to an allocated one. Another issue is how those foreign functions return flonums: If a foreign function knows the returned value is received by the VM, it can use an fp register. However, there's no general way for a foreign function to know if its return values are directly used by the VM, or received by another foreign function and may be stored in the heap. We don't want to impose large changes in already-existing foreign functions, yet we want to minimize the use of allocated flonums.

Gauche's standard foreign-function interface (FFI) relies on stub code generated from foreign function descriptions, which describe input and output types of foreign functions. A stub generator program creates a small C function for every foreign function that converts the Scheme arguments to C objects, calls the foreign function, and converts back the resulting C objects to Scheme objects.

If a foreign function takes C double arguments, or if it does not take numeric arguments at all, we can safely omit the process of moving doubles to the heap, since we know the passed flonums will never be stored in the heap—in the former case the doubles are immediately extracted, and in the latter case the flonums are rejected by argument type checking. Similarly, if a foreign function returns a C double, the stub generator can create code that places the resulting double in an fp register. This modification of the stub generator eliminates checking of flonums pointing fp registers in majority of the built-in C-defined procedures; it is especially effective to avoid overhead for calling non-numeric functions.

Numeric foreign functions ask a bit more care; the core numeric functions need to deal with all types of Scheme numbers—exact integers, rational numbers, real numbers and complex numbers—so they aren't defined to take and return C doubles. From the stub definitions the stub generator can't know whether it can safely pass register flonums or not.

For this, we introduced a special flag in the foreign function definition to indicate when the VM could safely pass the register flonums. Most built-in numeric functions ended up having the flag. It might have been better to make register flonums default, and only indicate the unsafe situation by a special flag. We didn't do so because it would've broken backward compatibility and affected existing large numbers of existing third-party foreign function bindings for Gauche[11].

Gauche's runtime is intended to be used also as a general list-processing library from C, so some core numeric functions are provided as a public C API. They don't know if they are returning the value to the VM or C-written application code, so they can't use fp registers to return flonums. We just took a naïve approach here; we prepared two distinct APIs, one was to be called from the VM, and the other was from general C code.

To summarize, the changes required to the code to implement our strategy are almost exclusively in the VM implementation, the collection of Scheme core numeric functions, and the stub generator.

4. BENCHMARKS

4.1 Benchmark Programs

We prepared five programs that used floating-point computation intensively (flonum-oriented), and four programs that did not use floating-point numbers (non-flonum-oriented). The latter group is to see the impact of the fast flonum extension to the programs that don't involve flonum calculation.

A list of flonum-oriented benchmarks is as follows:

`simple` Just calculates `(+ 1.0 2.0 ... 10.0e8)` by iteration. It is simple enough that the inner loop runs purely on the VM (i.e. no calls to the library functions) and do no allocations when the fast flonum extension is turned on. That is, its result is basically as good as our method gets, and can serve a reference point.

`gauss` Matrix inversion by Gauss-Jordan elimination. A matrix uses `f64vector`, a homogeneous numeric vector as a backing storage. It keeps array of doubles in unboxed form. Homogeneous numeric vectors are defined in SRFI-4[12].

`ray` Simple ray tracer program,[13] originally written for a Scheme compiler Stalin, adapted to Gauche.

`nbody` Physical simulation of planet movement considering gravity[14].

`plot` Draws graphs of complex transcendental functions. This is a straightforward port from the code shown in *Common Lisp the Language 2nd Ed.*[12], pp. 339–349. It involves lots of complex number arithmetics. Internally, a complex number is represented by two doubles. The Scheme code extracts real and imaginary parts very often.

And this is a list of non-flonum-oriented benchmarks:

`ack` Ackerman function.

`anc2` Puzzle solving.

[11] A new implementation might find it worth to consider the idea to make the arguments explicitly marked if it needs longer extent.

[12] `http://srfi.schemerts.org/srfi-4/`

[13] `http://www.ffconsultancy.com/languages/ray_tracer/`

[14] `http://shootout.alioth.debian.org/`

`tak` Takeuchi function.

`ssax` XML parsing. I/O intensive.

We used two platforms to run these benchmarks:

- 32bit Linux/x86 (Pentium 4, 2.0GHz, 2GB RAM, Fedora 8). Gauche VM is compiled by gcc 4.1.2.
- 64bit Linux/x86_64 (Athlon64 X2 3800+, 2GB RAM, Ubuntu 7.10). Gauche VM is compiled by gcc 4.1.3.

4.2 Result

Table 1 shows the average execution time (arithmetic means) and speedup compared to the original VM. Figure 5 plots the relative speed of the two strategies on two architectures.

For flonum-oriented programs, fast flonum extension yielded 30–140% boost in the performance. The two strategies showed similar performance overall, except the `simple` benchmark which showed rather large disparity between 32bit and 64bit architectures.

Turning to the non-flonum-oriented programs, the wide-stack strategy tends to incur more overhead than the nursery strategy. It is understandable since the wide-stack strategy requires checking register flonums for every operations when the value is moved in the stack or between the stack and the value registers. The wide-stack strategy performs exceptionally bad in the `tak` benchmark.

Table 2 compares the cumulative number of allocated flonums in the flonum-oriented benchmarks. It clearly shows the fast flonum extension eliminates almost all flonum allocations (more than 99.8%) except `plot`, in which 96% of allocations are avoided. The `plot` program constructs a list of numbers during calculation, which forces flonums to be moved to the heap.

Figure 6 shows the time spent in GC and other calculation, normalized by the execution time on the original VM. It is based on per-function profile taken by `qprof`[15] on 32bit machine. For the flonum-oriented benchmarks, it shows that the speed up shown in the figure 5 did come from elimination of memory allocation and GC. (Note: The large allocation/GC time of `ack` is caused by the stack overflow handler, which moves live frames to heap).

4.3 Discussion

An interesting observation is that, even in scripting languages, we can avoid allocating flonums 99.8% of the time with local modifications in the implementation and no change in the compiler.

Except for the obvious cases of flonums stored in polymorphic containers (such as lists and vectors), almost all flonum allocations occur only when the Scheme values are moved from the stack to the heap. This means that generic storage optimizations (such as better closure optimization to reduce data spilling from the stack to the heap) directly reduce flonum allocations altogether; no type-aware dataflow analysis is needed for flonum allocation optimization. This is a good news for scripting languages seeking to avoid longer compilation time.

There is some impact of the fast flonum extension to the non-flonum-oriented code, although it tends to show in the micro benchmarks like `tak` and `ack`. Particularly, we've known from our experience that `tak` is extremely sensitive in how the VM inner loop is compiled to the CPU instructions, since in the `tak` benchmark almost all execution time is consumed within the VM loop. (The `ack` benchmark has similar property, but it tends to recurse deeper and to trigger stack overflow handler that moves the stack contents to the heap. It dilutes the impact of the overhead.)

In general, a few percent difference in micro benchmarks becomes invisible in the real applications so we don't worry much about the overhead. However, the rather large performance degradation of the `tak` benchmark with the wide-stack strategy on the 32bit architecture is worth to investigate further.

We turned off the code of the fast flonum extension in various parts of the VM selectively to see which operations caused the overhead. It turned out that the two largest causes of the overhead were the following two cases, both of which involving an extra mask, test, and jump CPU instruction sequence.

- When the value in `VAL0` is pushed onto the stack, the VM needs to check if it is a register flonum pointing to `FPVAL0`.
- When the VM retrieves an object from a environment frame, it needs to check if it is a register flonum.

In the early stage of experiment, we found the wide-strategy yields much better performance in simple numeric calculations on 32bit architectures (as shown in the `simple` benchmark). However, the fact that it tends to incur more overhead in the non-numeric programs makes it much less attractive for adoption.

There is one hazard we learned to watch out. During the tuning process of the fast flonum extension, we sometimes observed that small change in the VM code could cause rather big overhead on 32bit architectures. We disassembled the VM code for each change and identified the cause: In the original VM (and the final versions of the fast flonum extension used for benchmarks), the pointer to the structure keeping the VM state is always kept in `%ebp`, so that VM data access is just a single indirection. When the fast flonum extension is turned on and extra code is inserted, the pointer to the VM structure can spill out from the CPU register, making any reference to the VM state a double indirection. This spilling couldn't be prevented by the GCC extension to specify a machine register to a local variable explicitly.

It only shows up on the IA32 processors since it has very few registers. To test whether a value is a register flonum, the processor needs to extract the lower 3 bits into a CPU register; if the original value is also needed immediately after the test, gcc tends to keep both on the CPU registers. When we have multiple flonums to work on, it's easy to use up all available registers. After we've noticed this, we try to not cluster the tests of register flonums so that gcc can keep the pointer to the VM structure on a register.

5. CONCLUSION

We proposed a fast flonum extension that greatly reduced heap allocation of intermediate flonums on the stack-based VM for dynamic languages. It doesn't involve a sophisticated optimizing compiler, so it is suitable for scripting languages which cannot afford long compilation time. It also

[15] `http://www.hpl.hp.com/research/linux/qprof/`

Table 1: Benchmark results, time and speed up

Program	*32bit*					*64bit*				
	original	*wide-stack*		*nursery*		*original*	*wide-stack*		*nursery*	
`simple`	52.93s	21.77s	(2.43)	24.59s	(2.15)	29.97s	16.56s	(1.81)	15.26s	(1.96)
`gauss`	23.71s	16.05s	(1.48)	16.25s	(1.46)	16.66s	10.75s	(1.55)	10.91s	(1.53)
`ray`	72.59s	44.73s	(1.62)	46.00s	(1.58)	45.48s	28.91s	(1.57)	30.11s	(1.51)
`nbody`	50.82s	36.12s	(1.41)	34.89s	(1.46)	32.37s	23.20s	(1.40)	23.49s	(1.38)
`plot`	11.28s	9.13s	(1.24)	8.62s	(1.31)	7.59s	6.09s	(1.25)	6.23s	(1.22)
`ack`	18.61s	19.73s	(0.94)	19.11s	(0.97)	13.09s	13.24s	(0.99)	13.24s	(0.99)
`anc2`	10.45s	10.95s	(0.95)	10.53s	(0.99)	7.95s	8.37s	(0.95)	8.00s	(0.99)
`tak`	26.87s	30.26s	(0.89)	26.81s	(1.00)	23.99s	25.85s	(0.93)	24.66s	(0.97)
`ssax`	10.65s	11.49s	(0.93)	10.63s	(1.00)	6.83s	6.80s	(1.01)	6.95s	(0.98)

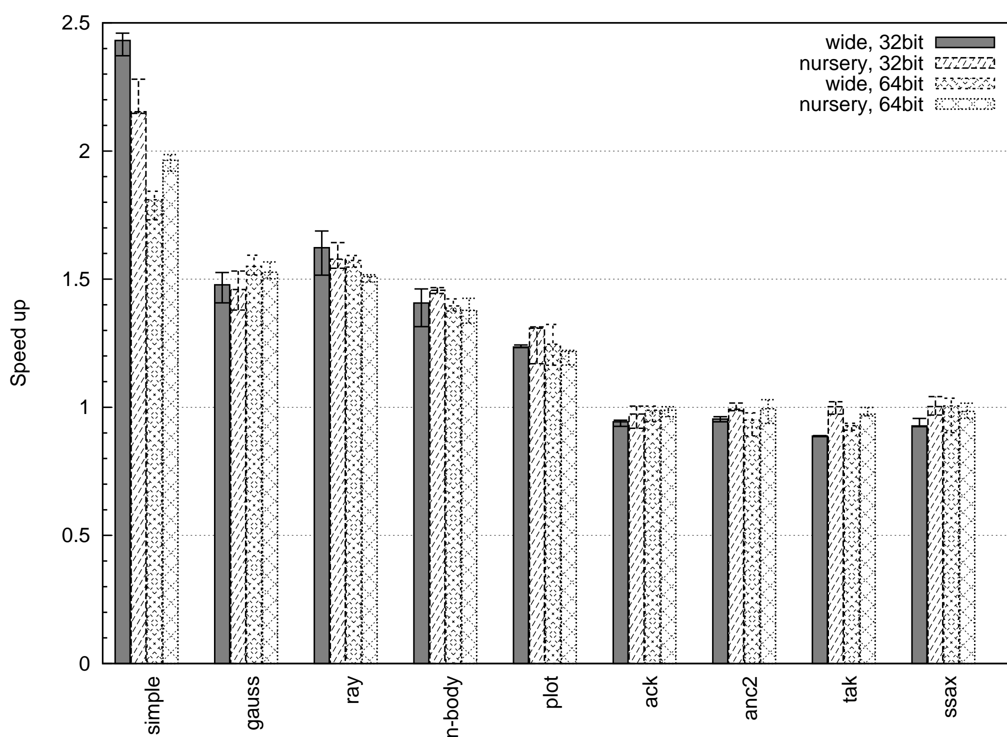


Figure 5: Speed-up comparison

Table 2: Cumulative number of allocated flonums

Program	Original	Wide-stack	Nursery
`simple`	200,000,006	10	40,008
`gauss`	65,448,341	20,051	28,245
`ray`	201,458,039	162,346	226,902
`nbody`	106,501,178	162	5,362
`plot`	19,923,144	782,252	787,891

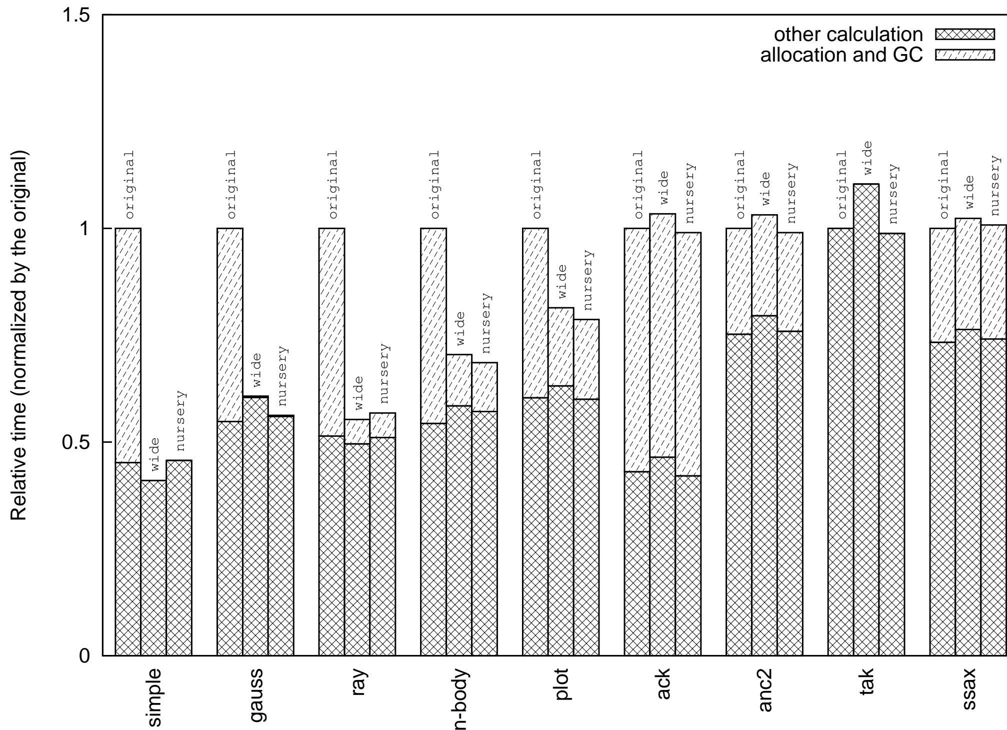


Figure 6: Time spent in allocation/GC and other calculation

only requires local changes in the runtime module—the VM itself and an FFI stub generator, plus small extra annotations in the FFI definitions—so that it can be relatively easily retrofitted to the existing implementations.

We implemented and benchmarked the proposed techniques on the Scheme scripting engine Gauche. The result showed that it virtually eliminated allocations from intermediate flonums, except the case that they were stored in polymorphic containers. The performance improvement depended on how much such allocations consumed in the execution time in the original implementation—in Gauche it turned out the cost was about 25% to 50% in the numerically extensive programs. Eliminating them was a big win.

The overhead of extra checking to manage fp registers is a few percent at most in the the nursery strategy, making it feasible for real applications. The wide-stack strategy shows higher peak performance, but it also shows higher overhead in particular situations and we find it less attractive.

For a future extension, an interesting possibility is to apply this technique to 64bit integers. Gauche VM supports unboxed integers (fixnums) up to two bits less than the width of the machine word. If an integer doesn't fit in it, it becomes a bignum which is allocated in the heap. We can use 64-bit VM fp registers to keep 64-bit integers as well, so that the programs that work on full-width integers can also avoid allocations.

Reducing flonum allocations benefits not only to overall computation performance, but it also makes an implementation easier to be adopted as an embedded scripting engine for interactive and semi-realtime applications, such as videogames. Using a powerful scripting engine in realtime computer graphics applications helps the content production tremendously[5], but the delay or the pause from garbage collection cycles have been a problem. Although we've tried not to allocate in the inner loop, GC has been unavoidable if flonums are heap allocated. Now, it becomes possible to write mostly alloc-free inner loop.

6. REFERENCES

[1] H. G. Baker. Cons should not cons its arguments, part ii: Cheney on the m.t.a. *ACM Sigplan Notices*, 30(9):17–20, September 1995.

[2] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 49–70, New York, NY, USA, 1989. ACM.

[3] R. J. Fateman, K. A. Broughan, D. K. Willcock, and D. Rettig. Fast floating-point processing in common lisp. *ACM Trans. on Mathematical Software*, 21(1):26–62, March 1995.

[4] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The implementation of lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, 2005.

[5] S. Kawai. Gluing things together - scheme in the

real-time CG content production. In *Proceedings of ILC2002, the International Lisp Conference*, pages 342–348, 2002.

[6] R. Kelsey. Tail-recursive stack disciplines for an interpreter. *NU-CCS-93-03*, March 1993.

[7] B. Lewis. An on-the-fly bytecode compiler for tcl. In *Proc. 4th Intl. Tcl/Tk Workshop*, pages 103–114. USENIX, 1996.

[8] T. Lindahl and K. Sagonas. Unboxed compilation of floating point arithmetic in a dynamically typed language environment. In *Implementation of Functional Languages: Proc. of the 14th International Workshop number 2670 in LNCS*, pages 134–149. Springer, September 2002.

[9] K. Sasada. A lightweight representation of floating-point numbers on ruby interpreter. In *Proceedings of the workshop of programming and programming languages (PPL2008)*, Sendai, Japan, March 2008.

[10] K. Sasada, Y. Matsumoto, A. Maeda, and M. Namiki. YARV: Yet another rubyvm. the implementation and evaluation. *IPSJ Transaction on Programming*, 47(SIG 2 (PRO 28)):57–73, February 2006.

[11] M. Serrano and M. Feeley. Storage use analysis and its applications. In *Proc. ICFP '96: the first ACM SIGPLAN international conference on Functional programming*, pages 50–61, 1996.

[12] G. L. Steele, editor. *Common Lisp: the Language, 2nd Edition*. Digital Press, 1990.

[13] K. Umemura. Floating-point number lisp. *Software—Practice and Expecience*, 21(10):1015–1026, October 1991.

[14] W. F. Wong. Optimizing floating point operations in scheme. *Computer Languages*, 25(2):89–102, 1999.