Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006
© 2006 The authors. All rights reserved.

Rain VM: Portable Concurrency through Managing Code

Neil BROWN

Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NF, England.
`neil@twistedsquare.com`

Abstract. A long-running recent trend in computer programming is the growth in popularity of virtual machines. However, few have included good support for concurrency — a natural mechanism in the Rain programming language. This paper details the design and implementation of a secure virtual machine with support for concurrency, which enables portability of concurrent programs.

Possible implementation ideas of many-to-many threading models for the virtual machine kernel are discussed, and initial benchmarks are presented. The results show that while the virtual machine is slow for standard computation, it is much quicker at running communication-heavy concurrent code — within an order of magnitude of the same native code.

Keywords. Process-oriented programming, Concurrency, Virtual Machine, VM, Rain

Introduction

The over-arching trend of computer programming in the last fifteen years has been the growth of interpreted/bytecode-based languages. It had become clear that computers would always be heterogeneous, and that compiling different versions (‘ports’) of programs in languages such as C usually required a tangled mess of compiler directives and wrappers for various native libraries (such as threading, networking, graphics). Using an interpreter or bytecode (grouped here into the term “intermediate language”) meant that the burden of portability could be centralised and placed on the virtual machine (VM) that runs the intermediate language, rather than the developers of the original programs.

Java, the .NET family, Perl, Python, Ruby — to name but a few — are languages that have become very widely-used and use virtual machines. The developers of .NET coined the term managed code [1] to describe the role that the virtual machine takes in managing the resources required by the intermediate language program. This captures nicely the advantage of intermediate languages; the virtual machine manages everything for you, removing most of the burden of portability.

Support for concurrency is a good example of the heterogeneity of computers. Traditional single-CPU (Central Processing Unit) machines, hyper-threading processors, multi-core processors, multi-CPU machines and new novel designs such as the Cell processor [2] can all provide concurrency, usually in a multitude of forms (such as interleaving on the same CPU or actual parallelism using multiple CPUs). Targeting each of these forms in a natively-compiled language (such as C) requires different code. It often cannot simply be wrapped in a library because the differences between things such as true multi-CPU parallelism and interleaving, or shared memory and non-shared memory are too great.

I believe that the best way to achieve portable concurrency is through the use of an intermediate language. Programs based on a process-oriented programming (no shared data, synchronous channel communications, etc) could be stored in a bytecode format. This bytecode could then be interpreted by a virtual machine that is built to take advantage of the concurrency mechanisms in the machine that it is running on. For example, on a single-core single-CPU machine it could use very fast cooperative multitasking (like C++CSP [3]), whereas on a multi-CPU machine it could use threads. More discussion on such choices is provided later in the paper in section 8.

Existing virtual machines (such as the Java and .NET VMs) tend to rely on Operating System (OS) threads for concurrency (usually provided to the managed program in a threading model with simplistic communication mechanisms such as semaphores). This allows the VM to take advantage of parallelism on multi-core and multi-CPU machines, but is fairly heavyweight. 32-bit Windows can only handle 2,000 threads at the default stack size (which would use 2 Gigabytes of memory), and still only 13,000 if the stack size is cut to 4kB (which would use the same amount of memory due to page sizes) [4], which is infeasibly small for many inherently-concurrent programs where concurrency is a natural mechanism to use.

The only current VM-like system suitable for scalable process-based languages is the Transterpreter, an interpreter for the Transputer bytecode [5]. The Transterpreter is admirably small and concise, and provides portability for the KRoC compiler. There are a number of features that I wanted in a virtual machine, such as C++ integration, security, poisoning, exception handling (all expanded on in this paper) that would have had to be grafted on rather than being part of the design from the outset (never a good idea), therefore the Transterpreter did not suit my needs.

The remainder of this paper details the design and implementation of a new concurrency-focused virtual machine, named Rain VM after the programming language Rain described in [6]. Many of Rain VM's features are motivated by the design of Rain.

1. Virtual Machine Design

Virtual machines can be implemented in a number of ways. The Java virtual machine is entirely stack-based, whereas Parrot (the VM for Perl 6) is register-based. The opposing ideas are discussed in [7] by the designer of the Parrot VM. While the two most-used virtual machines, Java and .NET, use a stack-based architecture, studies have shown that register-based virtual machines can be faster [8]. Based on this research I chose to make Rain VM primarily register-based.

Some modern VMs include Just-In-Time (JIT) compiling — the process of translating virtual machine code into native machine code when the former is loaded. Due to the (man-power) resources needed for JIT compiling, this is not considered a likely possibility for this project. Therefore, the focus in this paper is solely on interpreting virtual machine code.

Rain includes both process-oriented and functional-like programming, thus the virtual machine must support easy context-switching between processes as well as support for function calls. These two aspects are discussed below.

1.1. Security

Security in the context of this virtual machine consists of two main aims: protecting the virtual machine from unintentional bugs in code and protecting the virtual machine from malicious code running on it (often referred to as sandboxing the code). The latter is a particularly ambitious aim. The idea of mobility [9] in process networks yields the possibility of untrusted processes being run on a machine. In this case it is essential to limit their behaviour to prevent the untrusted code attacking the virtual machine.

Naturally, security checks will impose a performance (and in some cases, memory) overhead on the virtual machine. While security will be vital for some uses of the virtual machine, there will be users who will consider such security features to be unnecessary. To this end, it is intended to produce a compiled version of the virtual machine without these checks (where that is possible).

1.2. Concurrency

The difference between this virtual machine and most others is that it will be designed for concurrency — this means that there will be many contexts, one for each thread of execution. In the case of a solely register-based machine, this would mean one register block per thread. However, consider the following pseudo-code:

```
int: x,y;
x = 3;
y = 4;
par
{
  x = 2;
  y = 5;
}
y = 2 * x;
```

Assume for the sake of discussion that the above code is compiled un-optimised. Using the register-based virtual machine design, each of the two parts of the `par` would have their own register block. Putting the initial value of `x` and `y` into the register blocks could be done as they were created (before they were put on the run queue), but getting the values back into the parent's register block after the `par` is not as easy. The underlying mechanism of the (un-optimised) `par` is that the parent process will fork off the two parallel parts and then wait on a barrier for them to complete. The proper mechanism for getting values back would be channel communication — which seems somewhat over-the-top for such straight-forward code.

Instead a stack could be used; the values of `x` and `y` could be on the stack, which could be accessed by the sub-processes. For security, sub-processes could be given a list of valid stack addresses in the parent process that they are permitted to access, to avoid concurrent writing. Further justification for having a stack for special uses alongside the general-use registers is given in the next section.

1.3. Functions

The Rain programming language contains functions. Functions that are guaranteed to be non-recursive can be inlined during compilation — however, Rain supports recursive and mutually-recursive functions, which means that this virtual machine must too. Allowing arbitrary-depth recursion (the depth of which cannot be predicted at compile-time) with only registers would require many contortions that can be avoided by using a stack for arguments to be passed on.

Function calls also require the return address for a function to be stored. This is usually done by storing it on the stack. An incredibly common technique in remotely breaking into machines is that of buffer overflow, which often works by trying to overflow a stack buffer and over-write the return address on the stack. Theoretically, our type system (described later in section 3) and other measures should prevent such abuse. Security works best in layers, however.

Function call return addresses will therefore be maintained on their own stack. Having two stacks in a native program would be considered wasteful and cumbersome. One of the advantages of a virtual machine is that there is greater design flexibility to do such things. The overheads of having an extra stack are minimal — the equivalent of two registers.

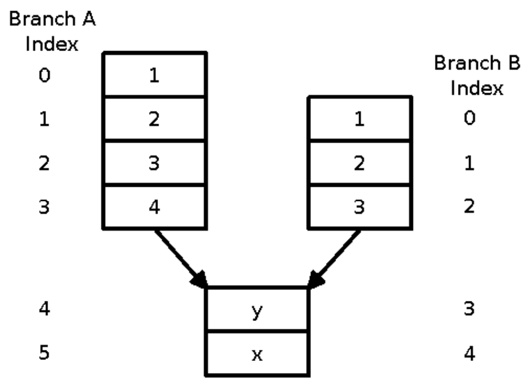
1.4. Forked Stacks

Consider the following pseudo-code that combines the preceding ideas:

```
int: x,y;
x = 4;
y = 3;
par
{
  x = factorial(x); #Branch A
  y = factorial(y); #Branch B
}
```

It has already been explained that the two parts of the `par` have access to the stack of their parent's process. It has also been decided that function calls use the stack for parameter passing. If the two factorial calls tried to use the same stack (inherited from the parent process) for their parameter passing then there would be a race hazard — they would both be writing to the same stack locations. Therefore the two sub-processes must have their own distinct stacks, yet still be able to access their parent's stack. Hence the idea of forked stacks.

This concept is shown in a diagram below, at the point where both stacks will be at the deepest level of the factorial function (`factorial(1)`). The 1, 2, 3, 4 progression are the arguments to the factorial function. Return addresses are stored on a different stack as described in the previous section. Return-address stacks are not forked, as there is no need to access the parent process's stack.



Stacks are random access, with zero being the topmost item on the stack. Note that the storage locations for `x` and `y` have different indexes in each branch. Accessing index 5 in branch A is no different to accessing index 2 in terms of the access method; the forking mechanism is transparent for the purposes of stack item access.

1.5. Exceptions

The paper on the design of the Rain language [6] explores whether or not to include exceptions. It concludes that exceptions should be featured in the language (in a small way), and hence as the target for the language, Rain VM must also include them. Like functions, exceptions are a mechanism best suited to using a stack. For security reasons, the exception stack will also be separate to the other stacks.

Unhandled exceptions cause a process to be terminated. Deciding what to do beyond that is unclear — for now, pessimism prevails, and the entire virtual machine is terminated. Processes cannot catch their (parallel-composed) children's exceptions. In future this could be changed according to the latest research conclusions.

2. Implementation Notes

The virtual machine has been developed using a rigorous unit-testing approach. Mainly this is done test-first. The virtual machine is written in C++; advantages of this are expanded on in section 10.

Registers are addressed by one byte, therefore 256 registers can be addressed. The virtual machine will guarantee that any of the 256 registers can be used. As an optimisation however, compilers targeting the virtual machine are encouraged to allocate the lowest-indexed registers first. The virtual machine could initially allocate, say, eight or 16 registers and then increase the size of the register block when higher-indexed registers were accessed. This would allow the memory footprint of short processes (that do not use many registers) to be very small.

A process context is a combination of: register block, instruction pointer, data stack, return-address stack, and exception stack. Context-switching in one virtual machine thread (i.e. where no locks are needed) is as simple as changing the current context pointer (and possibly doing a small amount of processing to the run queue). This should mean that context-switching is as fast as executing any other virtual machine instruction, although this is a combination of the speed of the former as well as the slowness of the latter. This expectation is tested in section 11.

3. Type System

This virtual machine is strongly statically typed. This is primarily to mirror the design of Rain. As described in [6], Rain currently contains the following data types:

- Boolean values.
- 8-,16-,32- and 64-bit integers, signed and unsigned.
- 32- and 64-bit floating point numbers (with the possibility of 128-bit or larger in future).
- Tuple types of sizes 1-255, containing any mixture of the types in this list[1].
- List types (implemented as either array or linked list) of any single type in this list[1] .
- Map types from any single type in this list[1] with a total ordering (see below) to any other single type in this list[1] .
- Channels for communicating any single type in this list[1] .
- Reading/writing ends of the above channels.
- Barriers and buckets.
- Functions and processes.

The types that have a total ordering are: all numbers, tuples where all types have a total ordering and lists containing a totally ordered type. Maps, functions, processes and communication primitives are not ordered. All types support testing for equality.

These types are compositional, to an unlimited depth. Booleans, integers and floating point numbers (which are contained by value in the 64-bit registers) are referred to here as primitive types, and all other types (that store a reference in the 64-bit register) as complex types. Complex types must all support four operations (referred to as type functions): creation, destruction, copying and comparison.

Naturally, because the types are compositional to an unlimited depth, the type functions are as well. So the list copying function allocates a new list of the same size as the source list, and then calls the copy function of its inner type for each of the elements in the list. These functions could either have been written into the virtual machine itself, or could have been

[1]With the exception that channels, channel ends, buckets and barriers cannot be contained in any other type

written using virtual machine code to be executed by the virtual machine. It was decided, for the sake of speed, to write them into the virtual machine (in C++).

When some bytecode is initially loaded, the virtual machine stores the types in a lookup table (which prohibits duplicate keys for the same type), allocating the type its unique key. Type functions for the type are created once, and stored by the same key. Whenever a type function is enacted on a class afterwards, it uses the key for speed — because the types can be of unlimited depth, the exact description is of unlimited length, which could take a long time to compare.

3.1. Type Safety

Each register in the virtual machine could have been simply a 64-bit integer. Any other values (32-bit integers, references to lists or maps) would have been converted into integer form and stored. This would have allowed an instruction to treat the same integer as any of those types without restriction. This, however, is bad from a security stand-point; an integer could be treated as a pointer, and thus arbitrary memory locations could be accessed/written to. Therefore type safety was added.

Each register in the virtual machine is an instance of data-storage. Every data-storage is a 64-bit value with an additional type identifier. Currently this type identifier is a 32-bit integer that references the lookup table described in the previous section. This means the data-storage is 12 bytes in size. A possible optimisation is to increase this to 16 bytes for better alignment on 64-bit machines (at the cost of memory usage). Every operation on this data-storage is type-checked.

Deciding what to do when type-safety is broken is a difficult problem that has not yet been fully examined. Unlike invalid class casts in Java that can be caught by the language, or type errors in languages such as Perl where a conversion is usually attempted, type exceptions in this virtual machine are considered fatal, just as an invalid instruction code is. They are not visible to the virtual machine code — that is, they cannot be detected or caught by the virtual machine code. They are currently treated as unhandled exceptions, and therefore (as mentioned in section 1.5) terminate the virtual machine.

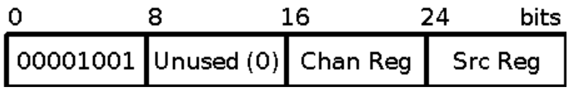
3.2. Decoupling

Rain VM is the target compilation platform for Rain. Since Rain will already enforce correct typing at the language level, adding these extra checks may seem superfluous. The checks are born of a desire to decouple the virtual machine from the language; while I do not envision Rain targeting anything but the virtual machine, I hope that the virtual machine may be targeted by other languages. Rain VM has a complete assembly language that the Rain compiler uses; there is no reason why other languages could not target this assembly language. These checks are therefore for other languages and also for untrusted code. A virtual machine would also lose all its safety advantages over native code without these checks.

4. Bytecode Design

The virtual machine instructions are entirely in 32-bit increments — usually one 32-bit word per instruction. This makes fetching the instruction straightforward. The majority of the instructions are in the format: 8-bit instruction opcode, 8-bit instruction sub-opcode, 8-bit destination register, 8-bit source register. This is not fixed however, and is merely the convention that is usually followed. A possible optimisation for the future is to fetch 64-bits at a time (given that most machines in the future will be 64-bit, this would be more efficient) and then decode it into instructions.

The below diagram depicts the `output` bytecode instruction. The bits on the left are the most significant. The first (most significant) byte is the (provisional) output instruction opcode. The next byte is unused, so it must be set to zero. The third byte is the index of the register that holds the channel writing-end. The fourth (least significant) byte holds the data to be sent down the channel.



5. Assembly Language

The virtual machine has its own assembly language and assembler. This should make it easy for other programming languages to target the virtual machine. The syntax is of the form `move r0,r2` — that instruction moves the contents of register 2 into register 0. A full documentation of assembly syntax, bytecode instruction and semantics for all instructions is currently being worked on, and should be made available when the virtual machine is.

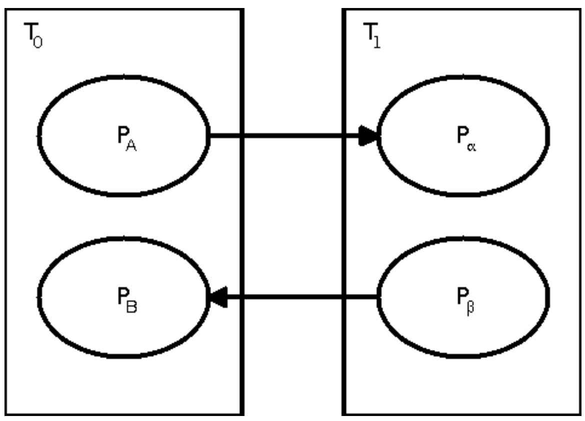
6. Terminology

Terms involved in concurrent programming often have ambiguous meaning. In this paper they are used as follows. Parallelism refers to two physical devices acting in parallel (be it multiple CPUs or multiple cores on the same CPU), whereas concurrency encompasses parallelism and other techniques such as interleaving that convey the effect of two processes acting at the same time. A thread is used here to refer here to concurrency at the Operating System (OS) level — this may be termed threads or (confusingly) ‘processes’ (used in quotes to differentiate this use from mine) by the OS. To achieve parallelism, multiple threads must be used. A process refers to a block of code and an accompanying program state. The virtual machine contains a set of active processes to be run concurrently. A process in Rain VM is more fine-grained than a process in Rain; a `par` block in Rain is a process in Rain VM.

7. Communication

This section details a few issues which are common to all of the approaches to implementing concurrency given in section 8.

Consider the OS threads T_0 and T_1. T_0 has two processes it is currently inter-leaving between: P_A and P_B. T_1 is inter-leaving between P_α and P_β. P_A and P_α are connected by a channel, as are P_B and P_β.



P_A makes a call to write on its channel. Its thread T_0 cannot block on this write, as T_0 must instead switch to P_B while the communication is pending. That is, the communication must take place asynchronously.

There are two methods of making asynchronous calls: polling and interrupts/callbacks. Inter-thread communications via interrupts or callbacks is a tricky area that is not explored here — as the thread would be interrupted asynchronously, it could be in an unknown/partially-invalid state when it was interrupted, such as updating its run queue.

Polling would require T_0 to check back periodically for completion of the communication by T_1. This could be done by having shared memory between the two threads protected by a mutex (or similar), but if T_0 holds the mutex when T_1 goes to check the memory then T_1 must either block (which this is intended to prevent) or must come back again later. The latter could lead to an indefinite wait if T_0 always gets the mutex just before T_1 arrives.

Therefore polling would have to be done through some sort of message-passing system. Each thread could have its own inbox of messages (from which it could receive messages from any virtual machine OS thread) for communication notifications. To implement this, a Message-Passing Interface (MPI) [10] package could have been used. Using buffering, MPI systems can send messages asynchronously (without blocking), which would solve the problem. However, MPI cannot guarantee this behaviour. Alternative mechanisms will be explored in future; TCP sockets are one possibility, but individual operating systems may offer their own solution.

7.1. Mobility

There are complications with channel communications between concurrent processes in the virtual machine, related to the implicit mobility of channel ends.

Consider a process P_A that allocates a one-to-one channel C. P_A spawns P_B and P_C in parallel, passing them the two ends of the channel. P_B spawns P_D and P_E, giving the latter its end of C. When P_E and P_C come to use the channel, at least one of them will need to know where the other end is. This is not a matter of the address in memory (that will be constant), but whether the two processes are in the same thread (and can have a very quick communication) or in different threads (that will require a different communication mechanism). This is compounded in some of the concurrency suggestions below, where processes can move from being in the same thread to being in separate threads during their execution.

KRoC.net solves a similar problem [11] by coordinating the channel ends via an administration node for the channel. That is, when a channel end is moved, it contacts the administration node to register its new location and to find the current position of the other end of the channel. The implication of this and similar solutions is that moving a process to a new thread involves a larger overhead than simply manipulating the run queues. This cost should be borne in mind when considering the options presented in the next section.

8. Concurrency in the Virtual Machine

8.1. Overview

In section , I posited that virtual machines would be the best way to take advantage of the variety of mechanisms for parallelism available on heterogeneous systems. Implementing this virtual machine alongside a full programming language (detailed in [6]) has been a large under-taking, so currently there is only a simple form of concurrency present in the virtual machine; a single-threaded interleaving through all the available processes. This section discusses the various implementation ideas for a multi-threaded virtual machine and attendant problems for the future.

8.2. Context-Switching

There are a variety of methods for choosing when to context-switch between processes. Concurrent systems will invariably context-switch when one process makes a blocking call. OS threads usually switch asynchronously when a timer interrupt occurs. Programs are usually able to suggest or force a context switch through some form of `yield()` call. occam-π [12] can compile in a possible context-switch at the end of loops. In a virtual machine, very fine-grained control is possible — for example, a context-switch could be done after every 25 executed instructions.

In channel-based process-oriented systems communications are usually frequent and thus context-switches occur quite often (because the channel reads and writes will block roughly one attempt in two[2]). However, if a process is doing some computation it may not be desirable to let it monopolise the processing resources. Hence, being able to interrupt the virtual machine after an arbitrary number of instructions to perform a context-switch is a useful option.

8.3. Interleaving

Interleaving is an incredibly simple mechanism for implementing concurrency in the virtual machine. One thread executes virtual machine instructions for the current context until a switch is made. The current context pointer is changed to point at the new context, and those instructions are then executed until a further context switch. The disadvantage is that there is no parallelism involved, but it can be combined with other concurrent mechanisms to form a many-to-many approach (multiple parallel threads interleaving various concurrent processes).

8.4. Thread-per-process

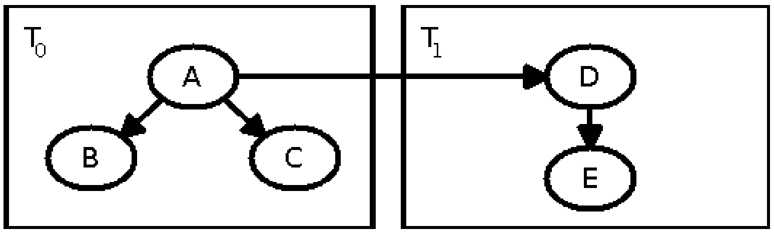
Operating systems almost always offer parallelism through the mechanism of threading. This may or may not be distinct from running multiple OS ‘processes’ — Linux has historically favoured the use of `fork()` to create new ‘processes’, whereas Windows favours threads inside a single ‘process’. This is therefore an easy semi-portable way for our virtual machine to utilise true parallelism on multi-processor and multi-core systems.

One way to use threads would be to allocate one thread for each process. This would make all inter-process communications identical (each thread could simply block when waiting for a channel communication), which would make the implementation relatively simple. It would be wasteful however in terms of overhead, and would mean that the virtual machine had gained no performance benefits over the clunkiness of threads and the restrictive limits on their scalability described in section .

8.5. Strictly Hierarchical Many-to-many Approach

One way to combine the OS thread and interleaving approaches would be a hierarchy. Each OS thread would be in charge of interleaving a set of processes. Sub-processes spawned would remain in the same OS thread, unless the set of processes grew beyond an arbitrary limit, at which point another OS thread would be created to handle that process and its sub-processes. The diagram below shows this approach — the arrows indicate a parental relationship (e.g. A is the parent of D):

[2]With two parties involved in a communication, the first one to attempt to synchronise will block, but the second will not as the first is already waiting.



This approach mirrors the hierarchical nature of process-oriented programs. Most communications between processes will occur between sibling processes spawned by the same parent — these communications would be quick ones between interleaved processes in the same OS thread. The main drawback is its inflexibility. If the program has a worker/manager pattern, then all the workers may end up in the same OS thread — clearly undesirable. The virtual machine could be amended to accept hints from the programmer as to when to split child processes into separate threads, but ideally allocation would be done without such hints.

8.6. Pooling

In threading, pools of threads are a relatively common technique. To avoid the cost of the creation and destruction of threads on demand, a pool of threads is created; if no work is available then they sit idle. In this case, the work would be virtual machine processes to execute. The pool in what I term 'peered pooling' would be held in a shared memory area, guarded by a mutex. By contrast, the idea of 'dictated pooling' involves an active manager thread handling the pool of processes. These ideas are explored below.

8.6.1. Peered Pooling

The threads could work in one of two ways: either a thread could take a single process at a time and return it to the pool when the process blocks, or each thread could hold a number of processes to interleave, and only pick up more when all its current processes are blocked.

The first approach would make channel communications difficult; if a process is moving between threads then the message-passing system described in section 7 would lack a permanent or even semi-permanent destination for a process. The communication would have to change to being stored in a common shared memory area (protected by a mutex). This would make the shared memory area very heavily used by all the threads, which in turn would slow all the threads down as only one could access the area at once.

The second approach would involve a thread taking on a new process each time that all of its currently managed processes ones blocked. This would have to be cleverly limited, to stop the thread taking on too many processes because its current ones happen to all block simultaneously.

With either approach, new processes would have to be placed into the shared area once created, awaiting a thread able to execute them. Ideally, if no thread was near-immediately available, a new one would be created. This would require some form of central co-ordination amongst the threads, which leads to the dictated pooling idea.

8.6.2. Dictated Pooling

Dictated pooling extends the peered pooling idea by introducing a central coordinator thread. Rather than having a shared memory area for the shared information, the coordinator thread keeps all such information. Information is fed in via its message passing system, and sent back out in the same way. Processes that block could be sent back to the coordinator. Threads with no more processes to run could request more from the coordinator.

The peered pooling and dictated pooling are similar ideas. Peered pooling appears simpler (although very inefficient), but dictated pooling more closely fits the channel-based communication that is used by process-oriented systems. The best way to judge performance trade-offs would be a trial implementation rather than extensive on-paper design considerations.

8.7. Linux 2.6

During the development of the Linux kernel 2.6, the old O(n) 2.4 scheduler was replaced with a new O(1) scheduler [13]. The Linux kernel scheduling problem is very similar to our VM kernel scheduling problem, so the Linux kernel methodology is useful here for reference.

The Linux kernel is broadly similar to the 'Strictly Hierarchical Many-to-many Approach' described above in section 8.5. Each CPU is in charge of maintaining a run-queue for itself that is protected by a lock. Forked processes are kept on the same CPU initially. In order to load-balance, CPUs with a light load find a busy CPU and lock its run-queue in order to move some of the busy CPU's processes onto the lightly-loaded CPU.

8.8. Threads and CPUs

All of the above ideas are predicated around arranging the VM processes into OS threads. The number of threads to be used should be determined by the number of CPUs/cores. Running only two threads on a four-CPU machine would not fully utilise the parallelism available. Running ten threads on a single-core single-CPU machine would be very inefficient; channel communications would be done between threads rather than using the much quicker mechanisms available within the same thread.

The ideal equilibrium would appear to be a thread per core/CPU. However, the OS will not necessarily schedule each thread on a different CPU, so the parallelism may still be underutilised. The optimum number of threads is likely to be slightly higher than the number of cores/CPUs available, except in the case of a single-CPU (single-core) machine, where one thread would be optimal. Experimentation (see the next section) will help to determine the optimal amount.

8.9. Outcome

Various options have been proposed above for the concurrency mechanism for the virtual machine. Rather than making an early decision, I intend to implement the most promising options and compare their performance. It may be the case that multiple systems are retained in the virtual machine — the flexibility allowing good performance across multiple different systems.

8.10. Implementation Transparency

Whichever mechanism is chosen on a particular OS, it will be transparent to the programmer. All implementations will have the same semantics (e.g. synchronised communication). While the exact scheduling choices may differ, the advantage of process-oriented programming is that these choices should not affect the overall behaviour of the program.

9. Debugging

While programming process-oriented systems, there inevitably comes a time when a program runs and at some point exits with the message "DEADLOCK". The instinctive reaction is to want to know why — that is, what every process was doing at the time, and which was the last process to block (and hence cause the deadlock). An advantage of using a virtual machine is that this information should be easy to provide.

It is not just post-mortem inspection that a virtual machine can facilitate; stepping though a program with a debugger should be easier in a virtual machine than in native code. As with many features, there has not yet been time to implement a debugger, but it should be possible. The challenge will be to make the debugger work with some of the threading that is intended. There will inevitably be a performance hit for such a feature, but it would only be used by a programmer during program development, and not when a program is actually deployed.

10. C++ Interface

Inevitably there will be a reason for programmers of interpreted languages to interface with C/C++. This can be for a variety of reasons, such as needing to access a C/C++ API for which the particular language has no libraries, or for writing high-performance code. Accordingly, most languages offer a C interface. Java has the Java Native Interface [14]. Many other languages are served by SWIG [15].

The other advantage of such an interface is that it gives programmers a measure of comfort when trying a new language. As Advanced Perl Programming [16] notes: “The ability of languages such as Perl, Visual Basic, Python, and Tcl to integrate well with C accords them the status of a serious development language, in contrast to awk and early versions of BASIC, which were seldom used for production applications.” As described in [6], these practical measures will encourage the transition from C++ to Rain.

The virtual machine is written in C++. Concepts such as data-storage (described in section 3.1), lists, channels and maps are all C++ objects. This makes exposing a C++ interface to them very easy. Its channel and process concepts can be integrated with C++CSP. This tight integration should be be very useful to C++ programmers wishing to use Rain (or any other languages that target Rain VM) with existing C++ code.

11. Benchmarks

The main aims for this virtual machine were support for concurrency, portability and security. Time and time again however the barrier to adoption of a new programming tool amongst programmers has been speed. Java faced a barrage of criticisms that it was too slow for years after its initial release. I hope that this ‘speed is all-important’ mentality will one day be left behind but nevertheless in this section I present some initial performance benchmarks.

The benchmark timings are listed as real-time, not system time; each results is the average of 50 runs and thus any interruptions on the machine are expected to average out to allow for a fair comparison. The benchmarks were carried out on an AMD Athlon 64 3000+ (1.8 Ghz raw clock speed) CPU with dual-channel DDR memory running Gentoo GNU/Linux. Both Rain VM and the GCC-compiled tests were using native 64-bit code, but KRoC (the occam-π compiler [17]) was compiled for 32-bit as some of its dependencies do not currently support compilation for 64-bit.

11.1. Instruction Speed Test

The first benchmark is intended to measure the rough speed of the virtual machine through timing a simple counting loop. The C++ version is simply:

```
for (usign64 i = 0; i < 1000000; i++) {}
```

and the occam-π version is the equivalent code. The Rain assembly version consists of two instructions (an add and a conditional jump) to achieve the same effect. The timing for each loop iteration (averaged out over fifty separate runs of a million loops) is as follows:

Language/Compiler	Time per loop iteration (nanoseconds)
occam-π/KRoC 1.4.0	11
C++/GCC 3.4.5	1
Rain VM	404

The results show that the virtual machine is roughly 400 times slower than the native-compiled code. This is a slightly disappointing result.

11.2. Context-switching Test

The next benchmark tests (when compared to the previous benchmark) test the context-switching speed. The C++CSP code ran x copies of this code:

```
for (usign64 i = 0; i < 1000000; i++) {yield();}
```

in parallel. The Rain assembly version had three instructions (a yield, an add and a conditional jump), with x copies of that code running in parallel. The occam-π version was the equivalent of those two. The results for $x = 2$ and $x = 10$ are given below.

Language/Compiler	Time per loop iteration $x = 2$ (nanoseconds)	Time per loop iteration $x = 10$ (nanoseconds)
occam-π/KRoC 1.4.0	26	128
C++CSP/GCC 3.4.5	534	2645
Rain VM	1063	5126

The times given above 'per loop iteration' are for the time for all x iterations. That is, given the time t for 2 copies of the loop running 10^6 times in parallel, the times recorded above are $t/10^6$, not $t/(2 \times 10^6)$. These times are perhaps more useful when compared amongst the same system. Such comparisons are given below. In an ideal (zero-overhead for concurrency) world, the ratios would be $2 : 1$ and $5 : 1$.

Language/Compiler	($x = 2$):Sequential Ratio, 2 d.p.	($x = 10$):($x = 2$) Ratio, 2 d.p.
occam-π/KRoC 1.4.0	$2.36 : 1$	$4.92 : 1$
C++CSP/GCC 3.4.5	$534.00 : 1$	$4.95 : 1$
Rain VM	$2.63 : 1$	$4.82 : 1$

The impact of introducing the context-switches to the C++(CSP) code is immediately apparent. Running two yielding copies in parallel is approximately 250 times worse than the ideal ratio. By comparison, Rain offers a reduction not far from the optimal $2 : 1$, as does KRoC. All the systems scale approximately linearly in the number of copies being run in parallel. The performances being slightly better than linear is thought to be related to the cache.

A naive bit of arithmetic, subtracting the calculation times in section 11.1 from the times with context switching for $x = 2$ in the table above (divided by two), gives a rough idea of the time that context-switching takes:

Language/Compiler	Rough estimate of context-switch time (nanoseconds)
occam-π/KRoC 1.4.0	2
C++CSP/GCC 3.4.5	266
Rain VM	128

This indicates that the virtual machine can switch contexts faster than the native-compiled C++CSP code, which is very encouraging. As ever, KRoC performs incredibly well.

11.3. Commstime Test

The virtual machine is intended for concurrency, so a benchmark with concurrent communicating processes is more apt. The quasi-standard CommsTime [18] benchmark is chosen. This benchmark consists of a process ring containing a prefix process, a successor and a delta process also connected to a recorder. In essence, CommsTime is a concurrent version of the counting loop used in the above benchmark. C++CSP, KRoC and the Rain VM all use interleaving concurrency in the same OS thread, so the comparison is fair (whereas JCSP uses OS thread-level concurrency, so it is not included). The timings are given below.

Language/Compiler	Time per CommsTime iteration (nanoseconds)
occam-π/KRoC 1.4.0	272
C++CSP/GCC 3.4.5	1327
Rain VM	1991

The results show that the virtual machine is only around fifty percent slower than the native-compiled C++CSP code at performing CommsTime. This indicates that the performance difference between native-code and the virtual machine on communication-heavy code (as CSP code is wont to be) will be less than an order of magnitude. KRoC has the best time again, but is less than an order of magnitude different to Rain VM.

12. Conclusions and Future Work

This paper has detailed the design of a new virtual machine designed specifically for highly-concurrent process-oriented programs. The virtual machine is intended to run programs for the Rain programming language [6], but can be targeted by any language. There is a full assembly language for programming the VM. Much of the virtual machine has been implemented and tested.

The VM contains type safety to remain stable if it encounters defective code and dynamic register allocation will soon be added to allow small processes to have a very small memory footprint. A C++ interface is planned, to allow interaction with existing C++ code via C++CSP. A debugger is also intended for implementation to aid programming in Rain and other languages that target the VM.

A detailed discussion of threading models and associated problems was provided in sections 7 and 8. The process-oriented programming style allows concurrency to be used naturally and easily but in order to provide this, tools such as this VM must be built on top of existing unwieldy and unsafe concurrent mechanisms offered by modern operating systems. Experimentation with implementing the various suggestions has been proposed — the outcome of which will guide the final choice of threading model.

The benchmarks show that the virtual machine is two orders of magnitude slower than native code for executing standard instructions. However, the concurrent CommsTime benchmark was only twice as slow as native C++CSP code, showing that for concurrent communication-oriented code the virtual machine is within an order of magnitude of native C++CSP and KRoC.

The primary focus of future work will naturally be finishing and optimising the virtual machine. This includes both implementing the remainder of the instruction set and also experimenting with and implementing the threading model. Process priorities are an obvious candidate feature for inclusion when implementing the threading model.

Trademarks

Java is a trademark of Sun Microsystems, Inc. Windows is a registered trademark of Microsoft Corporation. Linux is a registered trademark of Linus Torvalds. Python is a trademark of the Python Software Foundation. 'Cell Broadband Engine' is a trademark of Sony Computer Entertainment Inc. occam is a trademark of SGS-Thomson Microelectronics Inc. Gentoo is a trademark of Gentoo Foundation, Inc. 'AMD Athlon' is a trademark of Advanced Micro Devices, Inc.

References

[1] Brad Abrams (Microsoft). What is managed code? `http://blogs.msdn.com/brada/archive/2004/01/09/48925.aspx`, June 2006.

[2] IBM. Cell Broadband Engine Architecture. `http://domino.research.ibm.com/comm/research_projects.nsf/pages/cellcompiler.cell.html`, June 2006.

[3] N.C.C. Brown and P.H. Welch. An Introduction to the Kent C++CSP Library. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156, 2003.

[4] Raymond Chen (Microsoft). Does Windows have a limit of 2000 threads per process? `http://blogs.msdn.com/oldnewthing/archive/2005/07/29/444912.aspx`, June 2006.

[5] Christian Jacobson and Matthew C. Jadud. The Transterpreter: A Transputer Interpreter. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 99–106, 2004.

[6] N.C.C. Brown. Rain: A New Concurrent Process-Oriented Programming Language. In Peter Welch, Jon Kerridge, and Fred Barnes, editors, *Communicating Process Architectures 2006*, pages 237–251, September 2006.

[7] Dan Sugalski. Registers vs stacks for interpreter design. `http://www.sidhe.org/~dan/blog/archives/000189.html`, June 2006.

[8] Andrew Beatty Yunhe Shi, David Gregg and M. Anton Ertl. Virtual Machine Showdown: Stack versus Registers. In *ACM/SIGPLAN Conference of Virtual Execution Environments (VEE 05)*, pages 153–163, June 2005.

[9] F.R.M.Barnes and P.H.Welch. Prioritised Dynamic Communicating and Mobile Processes. *IEE Proceedings-Software*, 150(2):121–136, April 2003.

[10] A. Skjellum W. Gropp, E. Lusk. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[11] Mario Schweigler. Adding Mobility to Networked Channel-Types. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 107–126, 2004.

[12] Fred Barnes. occam-pi: blending the best of CSP and the pi-calculus. `http://www.cs.kent.ac.uk/projects/ofa/kroc/`, June 2006.

[13] Josh Aas (Silicon Graphics Inc). Understanding the Linux 2.6.8.1 CPU Scheduler. `http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf`, June 2006.

[14] Sun Microsystems Inc. Java Native Interface (JNI). `http://java.sun.com/j2se/1.5.0/docs/guide/jni/`, June 2006.

[15] Simplified Wrapper and Interface Generator (SWIG). `http://www.swig.org/`, June 2006.

[16] Sriram Srinivasan. *Advanced Perl Programming*. O'Reilly, 1997.

[17] University of Kent at Canterbury. Kent Retargetable occam Compiler. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/kroc/`.

[18] Roger M.A. Peel. A Reconfigurable Host Interconnection Scheme for Occam-Based Field Programmable Gate Arrays. In Alan G. Chalmers, Henk Muller, and Majid Mirmehdi, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 179–192, IOS Press, Amsterdam, The Netherlands, September 2001. IOS Press.