Minimising virtual machine support for concurrency

Simon Dobson, Alan Dearle and Barry Porter

School of Computer Science, University of St Andrews UK
simon.dobson@st-andrews.ac.uk

Abstract

Co-operative and pre-emptive scheduling are usually considered to be complementary models of threading. In the case of virtual machines, we show that they can be unified using a single concept, the *bounded execution* of a thread of control. Furthermore this technique can be used to surface the thread scheduler of a language into the language itself, allowing programs to provide their own schedulers without any additional support in the virtual machine, and allowing the same virtual machine to support different thread models simultaneously and without re-compilation.

1 Introduction

Multiple threads of control can make programs easier to write, by allowing logically concurrent activities to be coded independently, regardless of the availability of true concurrency. These benefits accrue equally to all levels of the software stack: applications, operating systems and virtual machines. In multicore environments there can also be performance benefits, but improved expressiveness alone makes multiple threads attractive even for single-cored systems.

Virtual machines (VMs) can further improve expressiveness, by providing a base level of abstraction targeted to the needs of the higher-level languages being written and hiding the underlying complexities of different machine architectures. A typical virtual machine embodies a particular model of threading and hard-codes a particular model of thread scheduling, prioritisation and so on. This choice will almost always be sub-optimal for some class of applications, especially in the case of resource-constrained embedded systems and sensor networks where it may be desirable for applications to exert close control over all aspects of the system's operation. Equally, we want to keep the VM well-defined and not shuffle important features into platform-dependent libraries.

In this paper we observe that it is possible to construct a virtual machine that makes no *a priori* choices about thread scheduling and concurrency control, yet without relegating these vital functions to external libraries. Put another way, we allow the *same* VM to support *different* concurrency models. We do this by simplifying the VM's support for threading to a single function offering *bounded execution* of the virtual instruction stream. This approach allows us to surface all other aspects of concurrency out of the VM and into the language.

2 Virtual machines and concurrency

Several modern programming language implementations adopt a virtual machine approach. The most notable are Java (the Java Virtual Machine or JVM); C♯ and F♯ (the Common Language Infrastructure [3]); GNU and Squeak Smalltalk; Lua; and Python. These VMs all adopt the *bytecode* style, in which the VM defines the instruction set of a processor that is "ideal" in some sense for the language being defined. This focus allows the language and VM designers to collaborate to define an instruction set that exactly matches the needs of the language, and so minimise space, time and compilation overheads, and the trade-offs between them. Some

VMs are general enough to be targeted by several languages: by design in the case of CLI, or through ingenuity in the case of the JVM. It also allows different implementations of the same instruction set, targeted at different classes of machine (for example for Java standard [5], bare-metal [2] and just-in-time [4] VMs, and highly portable Smalltalk [1]).

The core of a virtual machine is an *inner interpreter* that identifies and interprets virtual instructions. For a bytecode VM the inner interpreter uses a virtual instruction pointer (IP) to read an instruction and select an appropriate behaviour to execute. The bytecode may be "packed" to include (for example) a small integer literal or a jump offset to allow common instructons to take up less space, and so may require some decoding to extract the *opcode* specifying the instruction's implementation (a *primitive*) before execution (figure 1). In a multithreaded architecture, each thread must give up the processor after some time (referred to as its *time quantum*) to allow another thread to execute. In a *co-operative* (or coroutine) scheduler the programmer embeds explicit instructions that yield control at programmer-selected execution points. This slightly complicates programming and means that poorly-written code may not yield often enough (or at all) to avoid *starving* other threads of processor time, but has the advantage of requiring little or no concurrency control on data structures, since a thread may manipulate shared data without fear of interference from another thread. (In single-core systems, at least: multi-core requires slightly more care, for example separate control structures for each core.) By contrast, a *pre-emptive* scheduler forcibly interrupts the executing thread, suspending it and allowing another thread to be selected. This gives more power to the scheduler and prevents starvation, but requires concurrency control over all shared data.

```
void bytecode_inner_interpreter() {
  while(TRUE) {
    bytecode = *ip++;
    opcode = unpack_opcode(bytecode);
    switch(opcode) {
      case OP_NOOP:
        break;
      ...
      case OP_JUMP:
        ip += unpack_offset(bytecode);
        break;
      ...
    }
  }
}
```

Figure 1: Interpreting bytecode

Typically each thread maintains its own stack space and instruction pointer cache. Changing threads (a *context switch*) saves the VM's stack and instruction pointers and replaces them with those of the newly-scheduled thread. Implementing co-operative scheduling on a VM is conceptually straightforward, with a `yield()` primitive invoking a context switch. Pre-emptive scheduling often leads VM designers to use OS-level threads, swapping between several different VM-level instruction streams. Context switches also need to be triggered when a thread blocks on some event, such as a semaphore or a file read. This reduces the control the VM can offer to the language level, and often makes its behaviour platform-specific. (Java, for example, has primitives to block on object semaphores, but none for thread creation or scheduling [5].)

3 Bounded concurrency control

We assume a single core and therefore no "true" concurrency. Similar techniques can be used in multicore environments.

The core problem in context switching is to take control away from a running language-level thread (either voluntarily or forcibly) and give it to another. We propose to accomplish this by changing the inner interpreter so that – instead of running an *unbounded* loop over a single

virtual instruction stream which must be interrupted to regain control – it runs a *bounded* loop to context-switch into a given thread and execute a certain maximum number of virtual instructions before returning (figure 2). Here `activate()` context-switches the VM's instruction pointer and other registers to those of a given thread. If we assume that all primitives are non-blocking, then the such a bounded inner interpreter will always return to its caller in a finite time. Non-blocking primitives are a strong assumption, but multithreading actually simplifies the creation of blocking structures on a non-blocking substrate by allowing condition checking to be made independent of the main program flow (exactly as an operating system does).

We could simply use this construction to avoid the need for interruption and to re-factor the scheduler into another primitive that uses the bounded inner interpreter. However, the construction facilitates a more interesting approach whereby we remove the *entire* scheduling and concurrency control regime from the VM and surface it to the language level.

```
void bounded( bound, thread ) {
  oldthread = activate(thread);
  n = bound;
  while(n--) {
    bytecode = *ip++;
    opcode = unpack_opcode(bytecode);
    switch(opcode) {
      ...
    }
  }
  activate(oldthread);
}
```

Figure 2: Bounded interpretation

Having bounded the inner interpreter, we may now treat it as a virtual instruction in its own right (since it is non-blocking). This means that we may use it in defining new behaviours, and specifically we may use it in thread scheduling and concurrency control. The significance of this change is two-fold. Firstly, it blurs the distinction between co-operative and pre-emptive thread scheduling. Suppose we provide language-level threads, each of which is represented as a VM-level thread. *At the VM level*, thread scheduling is essentially co-operative: the bounded inner interpreter runs the thread for a time quantum specified in terms of virtual instructions and performs a voluntary context switch. *At the language level*, however, threads are pre-empted arbitrarily (from their perspective), since they have no control over when the underlying VM will switch them out. Secondly, bounded execution means that thread scheduling can itself be provided by a thread, rather than as a primitive. The scheduling thread chooses a worker thread, boundedly executes it for its time quantum, receives control back and selects another (or the same) thread for execution. Thread scheduling therefore need not be considered as a VM function, and may instead happen at language level: one language-level thread can use bounded execution to run another for a given period, without losing overall control of the program's execution.

Thread creation and scheduling. A thread is created by allocating memory for its stacks and essential registers – tasks that can be performed without primitive support – before scheduling the thread by adding it to the scheduler's run queue. Using a simple C-like language running on top of our VM, we might encode the simplest roun-drobin scheduler as shown in figure 3. The point is that this is *program* code and not VM code: it need not be primitive, and so may be redefined independently of the VM.

```
while(TRUE) {
    task = dequeue(runqueue);
    bounded(quantum, task);
    enqueue(task, runqueue);
}
```

Figure 3: Language-level scheduling

Within this style more complex schedulers are clearly possible. A scheduler might maintain multiple run queues of differing priorities and select the next thread from the highest-priority queue having runnable threads. The `quantum` parameter determines the latency of context switches in terms of virtual instructions: one might reduce this number to regain control into the scheduler more frequently, and consult a timer to determine whether to perform a context switch, leading to language-level threads that effectively have time quanta specified in wallclock times (at some granularity) rather than in virtual instructions.

Program-level concurrent objects. A small modification of the bounded inner interpreter allows us to migrate semphores and control of other program-level concurrent objects to the language level alongside the scheduler.

If we ignore the possibility that the thread may be pre-empted, implementing semaphores at language level is straightforward. A semaphore consists of a counter and a thread queue. The wait (P) function decrements the counter and, if it is less than zero, enqueues the thread onto the thread queue and de-schedules it as far as the thread scheduler is concerned. The signal (V) operation increments the counter and, if it remains less than zero, dequeues a thread from the semaphore's thread queue and adds it to the scheduler's run queue. None of these functions require explicit VM support.

Dealing with pre-emption requires that we change the bounded inner interpreter in three ways. Firstly, we add a state flag to each thread which by default is `RUNNABLE` indicating that the thread may continue to execute. Secondly, we introduce two other states: `BLOCKED` for a thread that is blocked on a thread queue and so cannot be scheduled; and `PRIORITISED` for a thread that should not be pre-empted. Setting a thread's state to `PRIORITISED` forces the bounded inner interpreter to keep executing virtual instructions in this thread, even if it comes to the end of its allocated quantum. Finally, we return the thread state from the bounded inner interpreter. This new scheme is shown in figure 4a, and is VM-level code: `set_thread_state()` is another primitive that sets the running thread's state. We modify the scheduler (at language level) so that it runs prioritised and, after receiving control back from `bounded()`, it only enqueues the thread back onto the run queue if it is `RUNNABLE` (figure 4b, language-level code).

```
int bounded( bound, thread ) {
  oldthread = activate(thread);
  n = bound;
  set_thread_state(RUNNABLE);
  while((state = thread_state()),
        (state == PRIORITISED) ||
        ((state != BLOCKED) && (n--))) {
      opcode = unpack_opcode(bytecode);
      switch(opcode) {
          ...
      }
  }
  activate(oldthread);
  return state;
}
```

(a) Inner interpreter

```
set_thread_state(PRIORITISED);
while(TRUE) {
    task = dequeue(runqueue);
    state = bounded(quantum, task);
    if(state == RUNNABLE)
        enqueue(task, runqueue);
}
```

(b) Round-robin scheduler

Figure 4: Language-level concurrency control

We can now write a `wait()` primitive (for example) at language level rather than as a VM

primitive. We first set the thread's state to `PRIORITISED`. The thread can then manipulate the semaphore's counter and thread queue safely using the full features of the language, since it will not be pre-empted. At the end of the definition we set the thread's state to `RUNNABLE` if we pass the semaphore or `BLOCKED` if the thread has been enqueued on the semaphore's thread queue. A `RUNNABLE` thread will keep running or, if it has reached its quantum, will be de-scheduled and enqueued on the run queue; a `BLOCKED` thread will not be enqueued. The complementary implmentation of `signal()` will prioritise the thread, dequeue a blocked thread (if any) from the sempahore, enqueue it on the run queue, and then make itself `RUNNABLE` again to restore normal scheduling.

What this shows is that the bounded inner interpreter with simple atomic thread state-setting offers sufficient VM-level support to allow concurrency primitives to be lifted to language level, and so allow a program to take complete control of its own concurrency control regime. These operations then have access to the full scope of the language: they are not restricted to the functions available primitively within the VM.

Clearly there is scope for errors in this scheme if program code is allowed to arbitrarily make itself `PRIORITISED`, which essentially turns the pre-emptive scheme into a co-operative scheme again. However, this is a problem for the language level that may be addressed using permissions, encapsulation or whatever mechanisms (if any) the designer chooses: it is not an issue for the VM, which can support any scheme chosen.

4 Conclusion

We have briefly presented an approach to opening-up the concurrency mechanisms in a virtual machine, allowing the VM to provide minimal support (two primitive operations) and building the rest of the concurrency regime at the language level. We have shown that this supports a number of different approaches to concurrency, including allowing the definition of new thread schedulers within a language so that they can be changed and specialised at run-time. One may also choose between traditional and speculative concurrency, blocking and non-blocking data structures and the like, entirely on top of the VM and therefore completely portably.

This scheme allows us to further enrich the program-level handling of concurrency on top of a minimal virtual machine. It is possible, for example, to unify the treatment of threads and delimited continuations, making these powerful features available on-demand with little or no VM support – and therefore no overhead where they are not required. This is potentially of great significance for sensor networks and other systems with severely limited resources, and we are currently exploring what place such advanced language features have in such environments.

References

[1] Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 1997.

[2] The Squawk Virtual Machine: Java on the bare metal. In *Proceedings ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 2005.

[3] Common Language Infrastructure (CLI). Technical Report ECMA-335, ECMA International, 2012.

[4] Joshua Ellul and Kirk Martinez. Run-time compilation of bytecode in sensor networks. In *Proceedings of the Fourth International Conference on Sensor Technologies and Applications*, July 2010.

[5] Bill Venners. *Inside the Java Virtual Machine*. McGraw Hill Osbourne Media, 2000.