High-level yet policy free?

Configurable protocols and fixed policies in the Movitz platform

Frode Vatvedt Fjeld
Department of Computer Science
University of Tromsø, Norway
frodef@cs.uit.no

ABSTRACT

We identify three ways a programming platform can address programming concerns: having application-configurable protocols by means of which applications can provide their own policies, providing mechanisms with fixed policies, or by doing nothing except ensuring that none of the platform's fixed policies unduly restricts application systems' design space. We believe awareness of these categories is important when designing programming platforms, because improper and unchangeable policies in a platform can severely impact the quality of the final system. A platform with support for higher-level programming constructs is more likely to introduce more policy decisions in order to address more programming concerns. Movitz is a platform for (x86) hardware-near programming based on the high-level Common Lisp language, and whose design is heavily influenced by the above observations. We describe some examples of how Movitz tries to resolve the tension between supporting high-level programming while remaining policy-free by identifying some programming concerns and motivating how these are addressed in terms of the aforementioned categories.

1. INTRODUCTION

For the last few decades, the C programming language has been a very successful technology for hardware-near programming, ranging from tiny embedded systems to massive OS kernels. In trying to understand the reason behind this success, we attribute a substantial portion of it to the policy-free nature of C. By this we mean that employing the C programming language (or its close relatives, such as C++) carries with it very few design-space restrictions relative to the base-level design-space offered by assembly language.[1] On the other hand, C also does not *provide* a lot in terms of aiding and empowering the developer, again compared to the base level as defined by assembly language programming.

[1]By assembly language we mean any language with an "obvious" mapping to machine code.

Libre Software Meeting July 5-9, 2005, Dijon, France.

Except perhaps for the smallest embedded systems and the simplest components of larger systems, programs written in C require minute attention to the irrelevant. This makes it a low-level language in the words of Alan Perlis [5], so we find C's nickname "portable assembly" to be of some merit.

What is the causality relationship between the two observations that C is low-level and policy free? It is not unreasonable to think that the former is a necessary requirement for the latter. This would imply that C would not have enjoyed such success if it was not so low-level (and this is perhaps even true, given the historical context of computer architecture development). However, our hypothesis is that there is no such causality. That is, while it may be easier to achieve a policy-free platform if it is low-level, it should be possible to retain the essential policy-free properties with careful design of a higher-level platform. Verifying this hypothesis is one of the goals of the Movitz project.

The above discussion suffers under the vague nature of the terms "policy-free" and "low-level". Arguably, it is impossible by definition to provide a high-level and policy-free platform, because removing irrelevant concerns from the view of the programmer implies that some policy decision must have been made. For example, C's modulo integer arithmetics can be considered a prime example of a mechanism that requires attention to the irrelevant, but any platform support for e.g. bignums must inevitably imply several policy decisions about data-structure layouts, memory allocation, etc. Furthermore, in some situations it is clearly *not* irrelevant to be aware that, say, the CPU can perform 32-bit modulo arithmetics a hundred times faster than the corresponding arbitrary-length operations. Consequently, we see our task as twofold: Firstly to identify which policy decisions must be configurable, and which are mostly irrelevant to programmers and can appropriately be embedded in the platform. Secondly, to provide mechanisms to configure or circumvent such policies, insofar possible.

Our approach is to establish a program execution model that on the one hand is the basic infrastructure needed to support many high-level programming mechanisms, and on the other hand establishes configurable protocols for program execution. This protocol can be understood as a "switchboard" between the lower-level execution model of the CPU and our higher-level execution model, where some of the wires can be rearranged while others are fixed. In implementational terms, Movitz is centered around the "run-time-

context" data-structure. This data-structure holds program execution state, including the configuration parameters that codify the particular policy decisions for that execution context. The design of this data-structure and the associated protocols is therefore the manifestation of our effort to identify and arrange programming concerns roughly into these categories:

- Application-relevant concerns, for which the platform provides infrastructure and policy configuration protocols (with reasonable defaults).
- Application-irrelevant concerns, for which the platform provides mechanisms and fixed policies.
- Concerns that are irrelevant to the platform, for which neither mechanisms nor policies are provided.

The first two categories partition the set of concerns that must be addressed by the platform in order to be "high-level". The first and third categories comprise the concerns for which Movitz remains "policy-free".

After describing the Movitz platform from a birds-eye perspective in Section 2, we describe in Section 3 some programming concerns of the categories we have just identified, and how they are addressed by the Movitz platform.

2. TECHNICAL OVERVIEW OF MOVITZ

Movitz is based on the Common Lisp language [1], and embraces the associated paradigms such as dynamic typing and incremental, interactive and exploratory development.

As we have already mentioned, Movitz is centered around a data-structure called the "run-time-context" (RTC for short). This data-structure can be understood as a software extension of the basic CPU state. For example, just as the CPU state (i.e. registers) must be replicated for each thread or process in the system, so too must the RTC. It contains the entry-points to a number of low-level functions that address application-relevant concerns, such as dynamic memory allocation and dynamic variable binding. The RTC also holds some state for these functions (because they are not lisp-level functions, they cannot hold their own state as closures), as well as other kinds of thread-local state such as space for holding multiple-values.

A major challenge in designing configurable protocols for some of the basic concerns of a programming platform, is efficiency. For example, the concern of dynamic memory allocation could be addressed by defining a single "malloc" function signature (using the standard lisp-level infrastructure). Such a protocol would be configurable in that the implementation of malloc could be changed. However, this would be inefficient in several dimensions, and some of the reasons for this are rooted in the fact that Movitz is a high-level platform that needs to see the result of such a malloc call as more that just a pointer to a block of memory. From this perspective, the RTC is an optimized, special-purpose data-structure and protocols that address these efficiency requirements. To this end, the Movitz run-time maintains the currently active RTC in a designated CPU register, allowing it to be accessed efficiently.

The RTC can take on the role as a special kind of CLOS object. That is, the RTC can be sub-classed and specialized using the normal CLOS infrastructure. (In CLOS/MOP terminology, there is a "run-time-context-class" meta-class, and every RTC object is an instance of an instance of this meta-class. [4]) In principle, every program execution protocol could be implemented in terms of the CLOS infrastructure. That is, the low-level functions mentioned in the previous paragraph, with generic functions and methods specializing on the (currently active) RTC. However, for many concerns this would be prohibitively expensive, as we discussed in the previous paragraph.

The remainder of the Movitz run-time is a more-or-less standard Common Lisp implementation, much of which can be considered to fall into the second category of concerns, namely the application-irrelevant concerns for which mechanisms with fixed policies are provided. For example, basic data-structures such as cons-cells, strings, bignums, and function-objects, basic protocols such as the function-call and pseudo-atomic sequences, are the basic building blocks of our high-level platform. Additionally, Movitz implements an increasing portion of the standard CL operators, system functionality such as an interactive read-eval-print-loop, a debugger, and library functionality and infrastructure for the OS kernel application domain.

Movitz currently supports only a cross-compiling development model, where code is edited and (incrementally) compiled on a standard Common Lisp system, from which bootable kernel binaries can be dumped. Interpreted functions can be dynamically added from inside Movitz, and compiled functions can also be serialized and loaded into running kernels (the compiler is not yet self-hosting).

3. EXAMPLES

In this section we describe four platform-level programming concerns, and how these are addressed by Movitz.

3.1 Synchronization and atomic sequences

The issue of platform support for synchronization is a difficult design problem. For example, is it justifiable to make access to basic data-types be thread-safe by introducing the necessary synchronization, incurring synchronization overhead also for single-threaded applications and other cases when it is completely redundant? This has for example been identified as a source of performance overhead with Java [2]. Inter-CPU synchronization is considerably more expensive than intra-CPU synchronization, and which variant is required, or if platform-level synchronization is required at all, is also not obvious. Under our previously introduced terminology, our task is to identify the relevant kinds of synchronization and place them in one of the three categories application-relevant, application-irrelevant, or platform-irrelevant.

With notable exceptions, Movitz assumes synchronization to be platform-irrelevant. That is, while Movitz provides basic synchronization building blocks such as mutexes, operations are not in general guaranteed to be thread-safe, motivated by the principle of remaining policy-free. The following paragraphs describe the exceptions to this design choice.

One of the design goals of the Movitz run-time is that it should be able to handle interrupts reasonably. That is, no policy for interrupt handling—such as automatically deferring them to some convenient time in the future—should be embedded in Movitz: an interrupt triggers triggers immediately a new stack-frame for the handler function.[2] Being a high-level platform, this design choice has certain implications for Movitz. Some of the application-relevant protocols, most notably those concerning dynamic memory allocation, require nstruction sequences that would leave the RTC or memory objects in inconsistent states if interrupted. Movitz addresses this problem by defining a protocol for "optimistic rollback atomic sequences"[3]. This protocol allows a program to specify that if an interrupt occurs, it should be resumed at a certain rollback point, which perhaps performs some cleanup operation before restarting the atomic sequence. We refer to this protocol as pseudo-atomic sequences, because the atomic property is only guaranteed within an RTC context.

The protocol for pseudo-atomic sequences is not configurable, and is so placed in the application-irrelevant category. The primary reason for this is that its current incarnation incurs so little performance overhead that making it configurable (i.e. by introducing an indirect low-level function call) would increase its overhead dramatically. Since the primary motivation we see for making this protocol configurable would be the ability to specify a no-op implementation for single-threaded systems. Consequently, a configurable protocol would be ultimately useless.

Pseudo-atomic sequences are implemented in Movitz by means of a data slot in the RTC. Whenever the value of this slot is non-zero an atomic sequence is in effect, and that value is a designator for the rollback point to which an interrupt handler must transfer control rather than the point at which the interrupt occurred. The cost of both entering and leaving a pseudo-atomic sequence is therefore one memory write. The cost incurred if a pseudo-atomic sequence is interrupted is bigger, but not prohibitive.

Besides pseudo-atomic sequences, Movitz might support (optional) automatic synchronization for some data-types, such as hash-tables, or define configurable protocols for same. This aspect of Movitz is work-in-progress.

3.2 Dynamic memory allocation

Movitz defines a configurable protocol for dynamic memory allocation, in order to accommodate a wide range of system requirements with appropriate policies for storage allocation and reclamation.

The design of a protocol for dynamic memory allocation carries with it implications for how garbage collection can work. For this reason, Movitz defines three categories of data-types: those objects that *cannot* hold pointers[3] (such as strings, which require no scanning for live pointers during GC), those objects that *only* hold pointers (such as cons-cells, a memory block of which consists of only live pointers, regardless of object boundaries), and finally those objects that hold both pointers and non-pointers (such as function objects, which require per-object parsing in order to find the live pointers). Movitz provides a separate allocation protocol path for each of the three kinds of objects, which allows an implementation to place the different kinds of objects in separate memory blocks, thereby aiding the GC implementation.

Each of the three protocol paths is arranged in two steps, which must be executed inside a pseudo-atomic sequence. First a low-level function[4] returns a pointer to an object (which is not yet legal) of a given size, the contents of which may now be initialized so as to form a legal object of the desired type and contents. Then, a new low-level function is called to commit the object allocation (with a size equal to or smaller than the size first requested), before the pseudo-atomic sequence is exited. The essential property of this protocol is that it is $O(1)$ relative to the object's size. In contrast, if the first call was to return a legal object, the memory would have had to be initialized to GC-safe values, resulting in $O(N)$ behavior. Another benefit of the two-step scheme, is that the object's contents can be initialized before its exact size is known. This is exploited e.g. in the implementation of bignum multiplication, where the object's initialization phase is the actual multiplication operation, and where the result's exact size isn't (easily) known beforehand. This way, we avoid having to either allocate too much space for the bignum result, or the overhead of using a separate temporary buffer for the result which is copied into a fresh bignum only after the computation is completed.

For certain frequently-used small objects, the constant factor hidden by the $O(1)$ notation can be uncomfortably large. Therefore, Movitz defines specialized allocation protocols (i.e. low-level functions) for creating 32-bit bignums, and for allocating cons-cells. The latter protocol can also be used to allocate objects that are isomorphic to cons-cells, such as standard CLOS instances (which consist of pointers to the class and slot-vector, just as the cons-cell consists of the car and cdr pointers).

There is only implicit support for garbage collection in Movitz. There is no explicit protocol for GC as such. Rather, GC is considered to be a part of the dynamic memory allocation implementation. Typically, this is realized by having the allocator function raising a CPU exception if a request for memory cannot be satisfied. This exception is then handled by running the GC to free up some space for the allocator. The pseudo-atomic mechanism will then ensure that the allocation request is restarted. However, none of this is specified or enforced by the Movitz run-time: the allocators and garbage collector can cooperate in arbitrary ways.

3.3 Dynamic variable binding

The implementation strategy for dynamic variable binding might be intuitively thought to be an intrinsic aspect of the run-time system. Or, in the terminology we have introduced: it could easily be considered an application-irrelevant

[2] On *which* stack the handler frame appears is subject to how the CPU (i.e. the x86 "interrupt descriptor table") is configured, and is another platform-irrelevant concern.

[3] By "pointer" here we mean a "lisp-val", a 32-bit value subject to the dynamic typing regime.

[4] I.e. a function whose calling conventions are not restricted by the standard lisp function-call protocol, and whose entry-point is immediately accessible from the RTC.

concern. Still, for Movitz we have classified dynamic binding strategy as an application-relevant concern, and provide a configurable protocol. This is motivated by the fact that there are many different strategies for dynamic binding with varying properties, and the solution spaces and intrinsic costs are very different for single-threaded, single-CPU, and multi-CPU systems.

A dynamic variable binding is a dynamically scoped mapping from a variable name to a value. The naive "shallow binding" implementation strategy is to store the current value in a global per-variable-name cell, and to push the previous value onto the stack upon entering a new scope for that variable-name, and popping the old value back when leaving the scope. Conversely, there is a naive "deep binding" strategy where a new scope is entered by pushing an explicit association between the name and value on the stack, such that finding the variable's current value entails searching the stack for this association. Consequently, shallow binding exhibits $O(1)$ lookup but doesn't work with multi-threading, while deep binding yields $O(N)$ lookup (with the number of active bindings) but works even with multi-CPU threading. Between these two extremes, typically various forms of (per-thread) caching is introduced to achieve optimal behavior for a particular system.

Movitz' configurable protocol for dynamic variable binding consists of five low-level functions in the RTC. Two of these are called when a new dynamic variable scope is entered and exited, two functions are for loading and storing a variable's current value, and the last function is used while unwinding the stack due to e.g. dynamic control transfers. In terms of this protocol, we have implemented both naive shallow binding and naive deep binding in about 100 lines of code (i.e. adorned inline assembly) for each, the latter of which is Movitz' default dynamic variable binding implementation. Switching between the two strategies can be done dynamically. The configurable protocol's viability in terms of flexibility and performance appears so far to be reasonable, although extensive experience and benchmarks for serious application systems are not yet available.

3.4 Threading

Threading is yet another programming concern that a platform must determine for itself how to address. While the current trend is to place threading more or less in the application-irrelevant category (Java, C#, etc.) and provide fixed mechanisms and policies for threading, Movitz puts threading in the platform-irrelevant category. That is not to say that Movitz is not to support threading, but rather that threading comes in the form of optional library functionality, potentially in numerous incarnations.

On the other hand, there is implicit support for threading in Movitz, as we have also touched upon in the previous sections: Since interrupts can occur at any time, preemptive scheduling can easily be implemented. Because the implementations of dynamic memory allocation is configurable, it can be made thread-safe for most any threading scheme. And ditto for dynamic binding. Currently, rudimentary threading is implemented in a 200-line file of library code for Movitz.

4. CONCLUSION

We believe it is very important in the design of a programming platform to be aware of which concerns are to be addressed and how: with configurable protocols or fixed policies. Furthermore, for every fixed policy it is important to identify how this policy restricts the design space for any application system built on that platform. Movitz has been designed with these principles in mind, and our experiences so far indicate that it is possible this way to bridge the gap between policy-free and high-level platforms in a fruitful way.

5. REFERENCES

[1] ANSI X3.226:1994 programming language Common Lisp, 1994. Also available in hypertext form at www.lispworks.com/reference/HyperSpec/.

[2] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for java. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 258–268, New York, NY, USA, 1998. ACM Press.

[3] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 223–233. ACM Press, 1992.

[4] G. Kiczales, J. des Rivires, and D.G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

[5] Alan J. Perlis. Epigrams in programming. *SIGPLAN Notices*, 17(9), 1982.