On the Performance of Reflective Systems Software

Geoff Coulson, Gordon Blair,
Paul Grace
Computing Dept.,
Lancaster University
Lancaster LA1 4YR, UK
+44 1524 593054
geoff@comp.lancs.ac.uk

ABSTRACT

Reflection is widely acknowledged as a useful mechanism for facilitating the run-time adaptation/ reconfiguration of software. Often, however, it is also thought to impose intolerably high overheads in performance-critical systems software environments like middleware platforms, operating systems, or programmable routers. In this paper we argue that there are many varieties of reflection (i.e. many types of 'meta-models') and that some of these in fact impose zero or negligible overhead. We further argue that in many cases, particularly cases involving highly dynamic software environments, reflection can actually enhance performance by laying open choices of alternative mechanisms that are best suited to current environmental conditions. Finally, we discuss the case of interception meta-models which, in many implementations, do lead to significant performance problems, and present our experience to date in attempting to minimise these problems.

Keywords

Middleware, components, reflection, performance.

1. INTRODUCTION

Recent years have see significant efforts in applying the *reflection* [Kiczales,91] paradigm in systems software. This approach, which builds on earlier foundational work on reflective programming languages, has been particularly directed at middleware systems [RM,00; RM,03], but there has also been significant work in reflective operating systems (e.g. [Murata,95], [Fassino,02]) and programmable networking systems (e.g. [Villazón,00], [Ueyama,03]). Fundamentally, the reflective approach to system-building aims to enhance the flexibility of systems in terms of both deploy-time configurability and (especially) run-time adaptability.

The essence of reflection is to establish and manipulate *causally-connected meta-models* of aspects of an underlying system [Maes,87]. Its broad aim is to maintain a clean architectural separation of concerns between system *building* (often called *base-level* programming) and system *configuration/ adaptation* (which involves the use of *meta-interfaces* provided by the meta-models, and is often referred to as *meta-programming*).

In more detail, systems can be reflectively configured/ adapted in three basic modes: i) *inspection* allows the meta-programmer to gain knowledge of the run-time structure or behaviour of an underlying system as a basis for possible subsequent action—e.g., in a reflective middleware context, to monitor the rate at which incoming method-call requests are arriving, or to check whether a particular protocol is available; ii) *adaptation* then involves changing the structure or behaviour of the system—e.g. switching to a thread-per-connection request-handling strategy as loading increases, and iii) *extension* involves adding new capabilities to the system—e.g. adding a new protocol or inserting a compression module to reduce the throughput requirements of a connection.

As an example of a reflective meta-model, consider the widely-explored notion of *operation interception.* Here, one provides a meta-model of the basic system-level operation invocation mechanism: a meta-interface is provided that allows the meta-programmer to interpose arbitrary code elements (called "interceptors") at object interfaces such that an interceptor is executed whenever one of the interface's operations is invoked (the interceptor may be called either before, or after, or both before and after, the operation invocation). *Inspection* in such a context might involve adding an interceptor that audits the pattern of invocations and their arguments for debugging purposes; *adaptation* might involve switching the invocation to an alternative object instance; and *extension* might involve inserting a security or concurrency control check on an invocation.

The flexibility engendered by reflection is considered very valuable, but it does often come at the expense of *performance*. This is especially easy to see in the above-described interception meta-model: all bindings to interfaces must presumably involve a level of indirection so that interceptors can be inserted dynamically when and where required. Nevertheless, this paper takes the view that reflection *need not* adversely impact performance. There are two facets to our argument: first, reflection can often be implemented in such a way as to incur zero or negligible overhead (at least for normal 'in-band' modes of system operation); second, the flexibility inherent in reflection can be exploited to actually *improve* overall performance in many instances. The paper takes an primarily architectural approach to the performance issue: we discuss principles and patterns rather than present detailed performance analyses. However, our arguments are founded on a solid quantitative basis: we have already demonstrated in previous research that our OpenORB reflective middleware framework (which is based on the OpenCOM component model discussed in this paper) performs excellently in comparison to other highly-optimised CORBA ORBs that do not have anything like the flexibility inherent in OpenORB [Clarke,01].

The remainder of this paper is structured as follows. First, in section 2, we present essential background on OpenCOM, our reflective system-building technology. Then, in section 3, we discuss reflection from a performance perspective. Subsequently, in section 4, we look at related work. Finally, in section 5 we draw conclusions.

2. BACKGROUND ON OPENCOM

2.1 The Component Model

The key element of our approach to reflective systems-building is the use of a fine-grained, low-overhead, language-independent, *component model* called OpenCOM [Clarke,01], [Coulson,02]. This is illustrated in Figure 1 which depicts the key OpenCOM concepts of components (the green rectangles), capsules (the dotted box), interfaces (the small circles), receptacles (the small cups), and bindings (the implied association between the adjacent interface and receptacle).

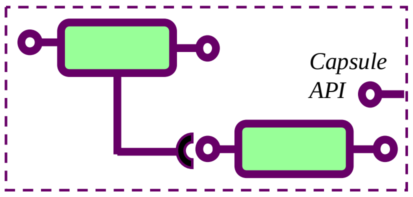

Figure 1: The OpenCOM component model

Let us now explain these concepts. *Components* are encapsulated units of functionality and deployment that interact with their environment through interfaces and receptacles (see below). *Capsules* are loci of component deployment (e.g. address spaces, or collections of related address spaces) which offer a simple capsule services API (see section 2.4) that supports intra-capsule component lifecycle management and binding services. *Interfaces* are units of service provision offered by components. For language independence, these are expressed in OMG IDL [OMG,03]. Components can support multiple interfaces; this is useful to support separation of concerns. *Receptacles* are 'anti-interfaces' used to make explicit the dependencies of components on other components: whereas an interface represents an element of service *provision*, a receptacle represents a unit of service *requirement*. Receptacles are key to supporting a 'third-party' style of component deployment and composition—when third-party-deploying a component into a capsule, one knows by looking at the component's receptacles exactly what other components must be present to satisfy its dependencies. Finally, *bindings* are associations between a single interface and a single receptacle in the same capsule. Like component deployment, the creation of bindings is inherently third-party in nature—it can be performed by any party within a capsule (i.e. not only by the 'first-party' components that themselves own the to-be-bound interface and receptacle).

The OpenCOM per-capsule runtime has only a very small memory footprint (around 27K on Microsoft Windows, and 18K on Windows CE [Grace,03]), and its underlying system-level support requirements are very simple (they basically amount to code loading, threads, and memory allocation). Although these requirements can, of course, be easily satisfied by any operating system (OS), it is important to us that the per-capsule runtime should be able to execute in a bare machine environment with no OS—indeed it must be possible to use OpenCOM itself as an environment in which to implement OS functionality [Clarke,98].

2.2 Component Frameworks

On top of OpenCOM, we advocate the notion of *component frameworks* (CFs) as a coarse-grained system structuring concept (OpenCOM components themselves being the basic, fine-grained, unit of structure). More specifically, in our approach, whole systems (e.g. middleware platforms, operating systems etc.) are built by developing a suitable set of more or less generic CFs, and then selecting and combining these appropriately.

'Component frameworks' were originally defined by Szyperski [Szyperski,98] as "collections of rules and interfaces that govern the interaction of a set of components 'plugged into' them". In our interpretation of the CF concept, each CF addresses the run-time needs of a particular system 'domain' (examples of 'domains' are pluggable protocols, pluggable request demultiplexing strategies, pluggable media filters, and thread management with pluggable schedulers [Coulson,02]). In essence, CFs provide us with domain-specific run-time structure for configurations of pluggable components, and encapsulate domain-specific semantics, rules, etc. with which the application (or other, higher-level, CFs) can control and manage system configuration, deployment, reconfiguration, and longer-term evolution in the domain. CFs accept plug-in components and, furthermore, are themselves built in terms of OpenCOM components: the whole structure is uniformly component-based.

2.3 Reflection

In our earlier work, we implemented reflective meta-models by building them directly into the OpenCOM runtime [Clarke,01]. Our current view, however, is that it is better to implement reflective meta-models 'on top of' OpenCOM as unprivileged CFs (there are, however, important exceptions; see section 2.4). Implementing meta-models as unprivileged CFs means that new meta-models can be added as required to meet emerging system environments or application needs. It also allows us to reduce overhead by leaving out meta-models that are not required in a particular deployment.

Nevertheless, all the non-trivial component-based systems that we have so far developed using OpenCOM have relied on a core set of three (CF-implemented) meta-models which we have found to be widely applicable and mutually orthogonal. This core set is as follows:

1. the *architecture* meta-model: this represents compositions of components within a capsule as a nested graph-like software architecture which can be inspected to learn about the underlying component topology, and adapted to effect corresponding changes to this topology (e.g. adding a node to the graph results in the deployment of a new component; breaking an arc results in the breaking of a binding, etc.);

2. the *interception* meta-model: this supports the insertion of arbitrary code into bindings (as discussed in the introduction); our implementation of interception is very efficient as the interceptors are linked in using direct

knowledge of the machine-specific calling convention [Brown,99];

3. the *interface* meta-model: this provides Java core reflection-like facilities—i.e. the ability to enumerate at run-time the interfaces/ receptacles supported by a target component, to identify operation signatures, arguments etc., and to support the dynamic invocation of interfaces discovered at run-time; the essential difference between the interface meta-model and Java core reflection is that the former is based on IDL and is thus language independent;.

We have also found widespread use for a so-called *resources* meta-model [Blair,99] which enables fine-grained control over the resourcing of dynamically-delineable units of work called 'tasks' (e.g. message reception, codec decompression, packet forwarding, etc.). These tasks can span multiple components which cooperate to perform some logically-separable unit of work (such as the above) to which is it useful to dedicate ringfenced resources. Using the resources meta-model one can assign resources (e.g. threads, memory, buffers, network connections) to tasks, and also assign resource *factories* to tasks so that newly created resources come from potentially-ringfenced pools that are owned by particular tasks. There is an explicit (base-level) task-switch operation. When this is called, a new set of resource factories (those owned by the new task) comes into scope. In addition, a thread switch is performed to a thread that is owned by the new task.

2.4 The Capsule API

OpenCOM's Capsule API offers simple support for loading components into capsules, and for binding the receptacles and interfaces of components loaded into the capsule. Importantly, the API presents such loading and binding functionality as reflective CFs[1] that take plug-in *loaders* and *binders*. It is therefore possible to extend an OpenCOM capsule with many and various implementations of loading and binding.

The (base-level) API for loading and binding is as follows:

```
load(component);
load(loader, component);
unload(component);

bind(receptacle, interface);
bind(binder, receptacle, interface);
unbind(receptacle, interface);
```

The following meta-interfaces expose a meta-model of the loader and binder CFs:

```
install_loader(loader, is_default);
intercept_loading(interceptor);
remove_loader(loader);

install_binder(binder, is_default);
intercept_binding(interceptor);
remove_binder(binder);
```

The meta-interfaces are used to install/ uninstall loaders and binders to extend/ reduce the set available to callers of *load()*, *bind()* etc. For example, we might provide a 'recursive' loader that recursively satisfies all the dependencies of a to-be-loaded component; or we may provide a binder that incorporates an interception meta-model (this, in fact, is how our interception model is implemented; see section 3.2.2). The boolean argument *is_default* determines whether the loader (binder) being installed should be treated as the 'default'. This works as follows: the default loader (binder) is selected implicitly when the base-level programmer calls the *load()* (*bind()*) signature that does not explictly specify a loader (binder). In such a case, the expectation is that the default loader (binder) will not itself directly perform loading (binding) but will rather act as a 'dispatcher' that will select and call a 'real' loader (binder) on the basis of attributes and predicates attached to components, interfaces, receptacles, loaders and binders. Alternatively, the base-level programmer can explicitly select a particular loader (binder) by employing the alternative *load()* (*bind()*) signature.

The *intercept_*()* calls are used to provide a basic interception capability for Capsule API calls which is independent of whether or not a more generic interception meta-model is currently available in the capsule. The use of these calls is illustrated in section 3.2.

3. REFLECTION AND PERFORMANCE

3.1 Overview

Given the above background on our approach to reflective systems-building, we are now in a position to discuss in detail the performance implications of our approach in particular and reflection in general. As outlined in the introduction, we structure this discussion under two headings: First, we argue that, in many instances, the overhead of reflection need only be incurred 'occasionally' in such a way that critical 'in-band' performance is not impacted. We use the term 'in-band' to refer to segments of essential code that are repeatedly executed in the normal course of events and are therefore particularly performance critical. Conversely, 'out-of-band' code is executed only occasionally and its impact on overall performance is negligible. Second, we argue that using reflection to select alternative underlying mechanisms (e.g. alternative loaders, binders, protocols, thread schedulers, etc.) can actually *enhance* performance by helping to ensure that the underlying system is always configured optimally for the current application mix and environmental circumstances.

3.2 Avoiding In-band Overhead

Happily, it turns out that two of the three 'core' meta-models discussed in section 2.3 (i.e. the architecture and interface meta-models) hardly incur any 'in-band' overhead. This is also true for the resources meta-model. In this section we analyse each of these cases together with the less encouraging case of the interception meta-model

3.2.1 The Architecture Meta-Model

The architecture meta-model is updated only when components are loaded and bound, at which times the presence of the meta-model adds slightly to the overhead of these operations; but loading and binding operations are already heavyweight and are furthermore assumed to occur relatively rarely (e.g. when first deploying/ configuring a capsule or when performing some

[1] Referring back to the comments made at the start of section 2.3, these are the only built-in, privileged, meta-models supported by OpenCOM.

adaptation or extension operation). Apart from this updating, the architecture meta-model incurs overhead only when it is being inspected, adapted or extended. Again, these operations occur relatively infrequently in an out-of-band manner.

To demonstrate the overhead of performing architecture meta-model operations in a realistic scenario we carried out an experiment[2] using our OpenCOM-based ReMMoC middleware [Grace,03]. This exercises ReMMoC's capability of dynamically changing its remote binding protocol so as to selectively exploit alternative middleware communication standards depending on their current availability; in particular, we repeatedly reconfigured between SOAP and IIOP (a single such reconfiguration involves architecture meta-model operations to switch between a six component personality for SOAP and a five component personality for IIOP).

The graph in Figure 2 shows that the cost of the initial reconfiguration (that of SOAP to IIOP) is quite expensive (about 2.3 seconds) but that subsequent reconfigurations are cheaper (this is because subsequent reconfigurations do not require DLLs to be loaded again as they are already present in the address space). The key point, however, is that, as discussed above, these reconfigurations do not at all impact typical in-band performance. In the given scenario they only occur in the relatively rare event that a new server speaking a different protocol becomes available.

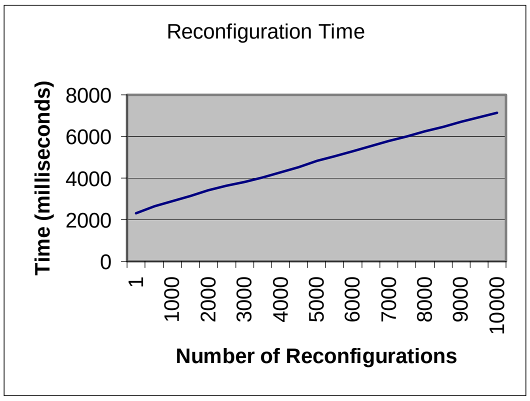

Figure 2: Cost of architectural reconfiguration

3.2.2 The Interception Meta-Model

The case of the interception meta-model looks, on its face, to be more problematic. This is because, as mentioned above, a level of indirection is necessary to enable interceptors to be inserted in interface-receptacle bindings [Brown,99]. This is problematic in three respects: *i*) the indirection adds a significant overhead, *ii*) this overhead must apparently be present in *every* binding so the programmer has freedom to insert an interceptor in any binding s/he likes, and *iii*) bindings are a fundamental system mechanism and as such are likely to be frequently used in in-band code segments.

[2] The experiment was performed on an Evesham workstation equipped with 128Mb RAM, and an Intel Pentium III processor rated at 500Mhz. The operating system used was Microsoft Windows 2000.

To get a sense of the overhead of interception, consider Figure 3 which presents the raw performance of intercepted methods in OpenCOM in terms of calls/sec throughput on a method with a null body (the same setup was used as for the figures in 3.2.1). Four measures interception-related cases are compared against non-intercepted baseline OpenCOM invocations. The "delegated" category represents the inherent overhead of an interception-capable binding but with no interceptor present. The "hooked" category is where the original method is replaced with a new (null) method. The "Pre & Post" category adds null interceptors both before and after the method call, while the "Pre" category adds only a 'before' interceptor. The Figures demonstrate that interception adds significant overhead.

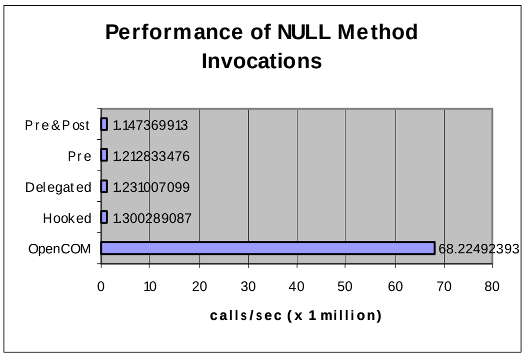

Figure 3: Performance of interception (null methods)

However, the *absolute* overhead of interception is still small and the effect on a real system is not necessarily as bad as Figure 3 might suggest. To illustrate this, Figure 4 shows the effect of replacing the null method with a short non-null method (in particular, one that implements a 1000-iteration null loop). It can be seen that even this modestly-sized method dwarfs the distinction between intercepted and non-intercepted methods.

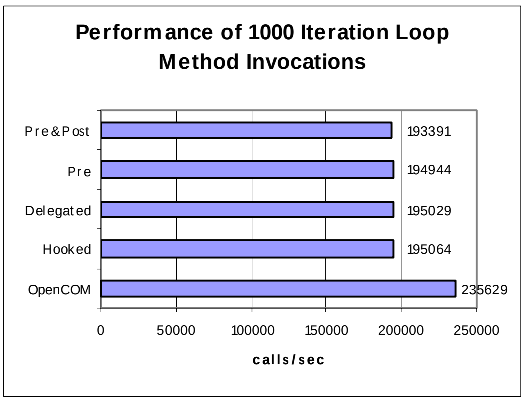

Figure 4: Performance of interception (non-null methods)

We primarily address the interception problem by *selectively scoping* the availability of interception, thus enabling application control over the trade-off between flexibility and performance. More specifically, we associate interception with specific plug-in binders and offer some binders that support it and others that don't. In this way, the programmer can choose an interception-capable binder (using the Capsule API) where/ when s/he knows that interception is likely to be required, and another, unencumbered, binder when this is not likely to be the case. In addition, it is always possible, thanks to the third-party nature of binding establishment, to dynamically replace one type of binding with the other if the initial choice has proven inappropriate[3].

An extension to the idea of manually switching from one binding type to the other when interception may be required/ not required, would be to automate this process and make it transparent. Thus all binders would support the interception meta-interfaces even if they were not themselves interception-capable. In case an attempt is made to add an interceptor to a currently non-interception capable binding, the associated binder would transparently remove the current binding and instantiate another by delegating to an interception capable binder. Similarly, when an interceptor was removed, the current (interception-capable) binder could delegate to a more efficient, non-interception-capable, binder.

3.2.3 The Interface Meta-Model

The interface meta model involves two modes of operation: *i*) responding to queries for information about the interfaces/ receptacles etc. owned by a component (inspection), and *ii*) allowing dynamically-formulated invocations to be made on interface types that are discovered at run-time and are not explicitly bound using the Capsule API's binding services (extension). Again, both of these modes of operation are expected to be applied in an essentially out-of-band manner. It is possible that badly-designed systems may make excessive use of the dynamic invocation facility (which will admittedly perform far less well than a typical Capsule API binding). However, our experience is that this facility is relatively rarely used as the Capsule API's binding services are sufficiently flexible for most purposes.

3.2.4 The Resources Meta-Model

The resources meta-model incurs a level of indirection on calls to resource factories (as explained above, these are associated with the current 'task' context). But once a resource has been allocated, it is used directly without further overhead (so, again, the in-band case is hardly affected). The main in-band overhead incurred by the resources meta model occurs on a 'task switch'. As mentioned, this is an explicit (base-level) operation. Therefore, there is no 'hidden' reflection-related overhead: the application can freely choose the overhead it is willing to trade for the additional functionality offered (e.g. it can choose the default case of a singe task-per-capsule and thus never invoke the task switch operation).

3.2.5 The Memory Overhead of Meta-Models

A further aspect to the avoidance of in-band overhead in our approach is that the core meta-models themselves need only be loaded into a capsule when they are actually needed. They are loadable on demand and can be unloaded (or garbage collected) if/ when no longer required. Even if this does not directly save CPU cycles, it certainly saves memory and increases cache-hit probability. As mentioned above, we also provide a 'recursive' loader that knows how to recursively load all the dependencies of a to-be-loaded component, so that if such a component has receptacles for some particular meta-interfaces, the corresponding meta-model will be transparently loaded if not already present. This automation further reduces the likelihood that memory will be wasted by currently unnecessary components.

Overall, we argue that reflective systems can be structured on the principle that "you only pay for what you get" and that unused functionality need not hinder general-case, in-band, computation or unduly consume resources.

3.3 Choosing Optimal Mechanisms

As explained, a prime benefit of reflection is that one can alter the operation of the underlying system infrastructure using adaptation and extension. Although it is not usually advertised as such, this capability can be explicitly leveraged to actually *enhance* performance.

For example, one can optimise the use of network bandwidth by using the architecture meta-model (as in section 3.2.1) to swap between alternative compression schemes when one is switching between one type of network connectivity and another (e.g. WaveLAN to GPRS, or SOAP to IIOP) [Coulson,99]. Or, one can optimise request handling in an ORB by using the resources meta model to switch to a new threading strategy for request handling as load increases [Gokhale,98]; or one can alter the capsule's thread scheduling policy as the application mix in a capsule evolves [Coulson,02b]. Further examples are given in [Coulson,02a]

4. RELATED WORK

In the middleware area, several researchers have investigated reflective component-based middleware architectures. Prime examples are the University of Illinois' DynamicTAO [Kon,00] and LegORB [Roman,00]. These are flexible ORBs that employ a dependency management architecture that relies on a set of 'configurators' that maintain dependencies among components and provide hooks at which components can be attached or detached dynamically. This is related to our architecture meta-model. Another example is work at Syddansk University on building real-time control middleware in terms of JavaBeans [Joergensen,00]. This system emphasises interception and is thus, as discussed, likely to incur significant in-band overhead. Performance of reflective mechanisms has not, however, been a focus for much of this work and there is little explicit discussion on this topic.

In the component-based operating system area the situation is similar. MMLite [Helander,98] is a component-based operating system built using MS COM components. It offers (limited) support for dynamic reconfiguration through a 'mutation' mechanism (again, related to our architecture meta-model) which enables the replacement of a component implementation at run-time. Think [Fassino,02] is another lightweight component model that is targeted at the construction of system software. It is close to OpenCOM in its goals but has less sophisticated reflective

[3] However, it must be ensured that third-party replacement of a binding is not attempted when some thread is performing an invocation over the binding! We currently provide no explicit support to prevent this, rather leaving it to 'the application'.

capabilities. Knit [Reid,00] is a component model developed for the OSKit component framework which enables the flexible configuration of operating systems but does not emphasise run-time reflection. Aperios [Murata,95] is an older, non-component-based, reflective operating system. The performance of this was shown to be inferior to traditional OSs; but the reflective facilities (which were based on "meta-spaces" into which different OS functions could be migrated to achieve different behaviour/ semantics) were *i*) radically different from our design, and *ii*) not explicitly singled out for differential performance analysis.

5. CONCLUSIONS AND FUTURE WORK

We have discussed the impact of reflective meta-models on the performance of systems software. The discussion has made it clear that interception, which is synonymous with reflection in many people's minds, certainly does carry a significant overhead, particular where components are small and fine-grained. However, we have also identified several other meta-models apart from interception which, besides being very useful in the configuration and dynamic reconfiguration of systems software, do not have anything like the performance overhead of interception. As argued, this is primarily because of their 'out-of-band' nature.

We have further argued that system reconfiguration, as enabled by reflective-meta-models can, in many instances, actually improve performance in dynamically changing environments because it can help ensure that the system is able to best exploit its current conditions by choosing the most performant of an range of alternative mechanisms.

Finally, we have argued that even the detrimental effects of interception meta-models can be alleviated by avoiding an interception-related overhead in *all* bindings. We proposed that this be achieved by offering alternative binding implementations, some of which are not interception-capable (and therefore are 'fast'), and others of which are (and are therefore 'slow'). Furthermore, we have discussed how it is possible to break and remake bindings so as to switch between interception-capable and incapable versions as requirements evolve.

In our future work we expect to pursue this latter issue and look at dynamically configuring-in or configuring-out the capability for interception in a transparent, on-demand, manner. The notion of partial evaluation [Jones,96] is one approach that we are currently exploring in this regard. In addition, there are interesting issues to be explored in determining when it is 'safe' to carry out these transparent optimisations so that applications are not disrupted.

6. REFERENCES

[Blair,99] Blair, G.S., Costa, F., Coulson, G., Duran, H., Parlavantzas, N., Delpiano, F., Dumant, B., Horn, F., and Stefani, J.B., "The Design of a Resource-Aware Reflective Middleware Architecture", 2nd Intl. Conference on Meta-Level Architectures and Reflection (Reflection'99), St-Malo, France, Springer-Verlag LNCS, Vol 1616, pp115-134, 1999.

[Brown,99] Brown, K., "Building a Lightweight COM Interception Framework Part 1: The Universal Delegator", Microsoft Systems Journal, January 1999.

[Clarke,01] Clarke, M., Blair, G.S., Coulson, G., "An Efficient Component Model for the Construction of Adaptive Middleware", IFIP/ACM Middleware 2001, Heidelberg, Nov 2001.

[Clarke,98] Clarke, M., Coulson, G., "An Architecture for Dynamically Extensible Operating Systems". Proc. 4th Intl. Conference on Configurable Distributed Systems (ICCDS'98), Annapolis MD, USA, May 1998.

[Coulson,02a] Coulson, G., Blair, G.S., Clark, M., Parlavantzas, N., "The Design of a Highly Configurable and Reconfigurable Middleware Platform", ACM Distributed Computing Journal, Vol 15, No 2, pp109-126, Apr 2002.

[Coulson,02b] Coulson, G., Moonian, O., "A Quality of Service Configurable Concurrency Framework for Object Based Middleware", Concurrency and Computation: Practice and Experience, 2002.

[Coulson,99] Coulson, G., Blair, G.S., Davies, N., Robin, P., Fitzpatrick, T., "Supporting Mobile Multimedia Applications through Adaptive Middleware", IEEE Journal on Selected Areas in Communications (JSAC), Vol 17, No 9, pp 1651-1659, IEEE Press, September 1999.

[Fassino,02] Fassino, J.-P., Stefani, J.-B., Lawall, J., Muller, G., "THINK: A Software Framework for Component-based Operating System Kernels", Usenix Annual Technical Conference, Monterey (USA), June 10th-15th, 2002.

[Gokhale,98] Gokhale, A. and Schmidt, D.C., "Principles for Optimising CORBA Internet Inter-ORB Protocol Performance", Proc. HICSS '98, Hawaii, Jan 9th 1998, http://www.cs.wustl.edu/~schmidt/HICSS-97.ps.gz.

[Grace,03] Grace, P., Blair, G.S., Samuel, S., "ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability", Proc. Intl. Symposium on Distributed Objects and Applications (DOA 2003), Catania, Sicily, Italy, November 2003.

[Helander,98] Helander, J., Forin, A., "MMLite: A Highly Componentized System Architecture", 8th ACM SIGOPS European Workshop, pp96-103, Sintra, Portugal, Sept 1998.

[Joergensen,00] Joergensen, B.N., Truyen, E., Matthijs, F., and Joosen, W., "Customization of Object Request Brokers by Application Specific Policies", IFIP Middleware 2000, New York, April 3-7, 2000.

[Jones,96] Jones, N.D., "An Introduction to Partial Evaluation", ACM Computing Surveys, 28(3), pp480-504, Sept 1996.

[Kiczales,91] Kiczales, G., J. des Rivières, D.G. Bobrow, "The Art of the Metaobject Protocol", MIT Press, 1991.

[Kon,00] Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L.C., and Campbell, R.H., "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB". IFIP Middleware 2000, New York, April 3-7, 2000.

[Maes,87] Maes, P., "Concepts and Experiments in Computational Reflection", Proc. OOPSLA'87, Vol. 22 of ACM SIGPLAN Notices, pp147-155, ACM Press, 1987.

[Murata,95] Murata, K., Horspool, R.N., Manning, E.G., Yokote, Y., Tokoro, M., "Unification of Compile-time and Run-time Metaobject Protocols", Proc. ECOOP Workshop in Advances in Metaobject Protocols and Reflection (Meta'95), August, 1995.

[OMG,03] Object Management Group, Interface Definition Language (IDL), http://www.omg.org/cgi-bin/doc?formal/02-06-07.

[Reid,00] Reid, A., Flatt, M., Stoller, L., Lepreau, J., Eide, E., "Knit: Component Composition for Systems Software", Proc. OSDI 2000, pp 347-360, Oct 2000.

[RM,00; RM,03] Workshops on Reflective Middleware at IFIP/ACM Middleware Conferences, New York, 2000; Rio, 2003; http://www.comp.lancs.ac.uk/computing/RM2000/.

[Roman,00] Roman, M., Mickunas, D., Kon, F., and Campbell, R.H., "LegORB", Proc. IFIP/ACM Middleware 2000 Workshop on Reflective Middleware, IBM Palisades Executive Conference Center, NY, April 2000.

[Szyperski,98] Szyperski, C., "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, 1998.

[Ueyama,03] Ueyama, J., Schmid, S., Coulson, G., Blair, G.S., Gomes, A.T., Joolia, A., Lee, K, "A Globally-Applied Component Model for Programmable Networking", Proc. Intl. Workshop on Active Networks (IWAN 2003), Kyoto Japan, 10-12 Dec 2003.

[Villazón,00] Villazón, A., "A Reflective Active Network Node", Proc. 2nd Intl. Working Conf. on Active Networks (IWAN 2000), Tokyo, Japan, Oct 2000.