

# Self-Adaptive Compliant Persistent Architectures

Francis Vaughan and Dave Munro

*Department of Computer Science,  
University of Adelaide,  
South Australia 5005, Australia*

*Email: francis@cs.adelaide.edu.au, dave@cs.adelaide.edu.au*

## Abstract

Systems that can dynamically adapt their operating parameters to changing conditions are increasingly needed, especially for long-lived or large applications. Current self-adaptive algorithms tend to be directed at particular optimisations and rarely operate in concert with other optimisations. The static nature of conventional systems design is not conducive to adaptation. Further, the poor structure and uncoordinated manner in which individual optimisations are incorporated leads to unconstrained complexity of operation. Such lack of structure can lead to poorer than expected or pathological performance, especially where heavy demands are made upon the system. The Compliant Systems Architecture is a design regime which defines a strict methodology of system composition. In this methodology each application layer or operational abstraction is separated into the policy and mechanism parts, and the interfaces include features allowing policy information to be passed. This methodology combined with the simplification of system operation of orthogonally persistent systems allows the construction of adaptive control structures in which it is tractable to reason about interactions of policies and provide a global control of operation. The aim is to provide experience and insights in self tuning and adaptive systems that will also be of value in more general purpose operating systems and application designs.

## 1 Introduction

The key to the run-time efficiency of any software architecture is to ensure that the correct components are available and ready to use at the time they are required. For example, within a storage hierarchy, it is clearly beneficial to run-time efficiency to ensure that the data and programs are in the correct place in the storage hierarchy when they are required for use. Thus register allocation mechanisms, caching techniques and page replacement algorithms should be co-ordinated and optimised for such an effect.

To achieve optimum application efficiency, which requires application-specific knowledge, a software architect must address two major questions. They are:

- How does the system discover what the application is doing?
- How should the architecture be structured to utilise the above knowledge?

Conventional software architectures, which we would term *non-compliant*, provide static abstract layers and operational abstractions to meet the average predicted needs of the majority of cases. The application knowledge discovery is typically performed statically by simulating and benchmarking the applications. The architecture is then structured to perform well under the benchmark conditions. Optimisations are often based on strategies such as overall throughput and may use average or common application execution profiles. Indeed an extensive study of memory allocation [WJN+95] severely questions the value of this methodology.

In contrast a *compliant* architecture is defined to be one which is able to mould itself to the needs of a particular application. It is thus compliant to these needs. The key difference in the Compliant Systems Architecture (CSA) approach is that policy/mechanism separation is provided not just at particular interfaces, but coherently throughout the architecture. This then allows static and dynamic policy decisions at any level to be mapped all the way through to the raw machine.

The advantage of the static interface approach is that it provides a high degree of code reuse and therefore savings, in terms of code re-writing and portability. Many

applications may reuse the interface without regard to its implementation. However, they do so at the possible expense of individual performance since the optimisations necessary for individual cases cannot be accommodated in the general case. Thus, in such an architecture individual applications only occasionally run optimally, and even then it is by accident.

Previous work on self-tuning has typically only been provided for individual mechanisms. For example, rate of garbage collection: Cook et al [CKW+96], reclustering: McIver and King [MK94], buffer allocation: Brown et al [BCL96], caching policies for object and paging systems [GKK+99], [GG97], [JS94],[OOW93].

Our aim is to build a self-tuning and adaptive persistent software architecture as an exemplar of the compliant systems architecture. The realisation of this system is expected to yield insights into integrated self-tuning and adaptive methodologies of value in the broad arena of system architectures. This has not been systematically attempted previously, partly because of the industrial need to meet many other commercial constraints and partly because of the academic need for a wide variety of skills and experience.

## 2 Compliant Systems Architecture

Complex systems are typically a mixture of individual layered sub-systems and lateral compositions of sub-systems. In conventional systems it is seldom that a distinction between the mechanism by which a sub-system is implemented and the policies by which it acts is made. Such a distinction is even less evident when considering the nature of interactions between sub-systems.

Because of the rigorously defined separation of mechanism and policy in CSA we can meaningfully understand, and hence control the interaction of adaptive policies. In a layered structure the interfaces between the levels of the architecture define a set of up-calls and down-calls. The down-calls constitute the mechanisms that the higher layers may call upon. These include calls for passing policy information. Up-calls constitute entry points to the higher layers that may be used to request policy information. At any particular layer, the mechanism provided from below, together with the policies implemented at that level, constitute mechanism for the higher layers. Thus for layering:

$$\mathbf{policy}_n \oplus \mathbf{mechanism}_{n-1} = \mathbf{mechanism}_n \quad (\mathbf{rule\ 1})$$

In a similar manner, laterally composed operational abstractions communicate policy and mechanism information using an agreed protocol effectively via an IPC. Again the interfaces constitute entry points that can be used to request policy information but such policy/mechanism passing can be bi-directional. So between any two such lateral abstractions, P and Q, then P's policy regarding Q (written  $P \rightarrow Q$ ) together with the mechanism used to communicate from Q to P constitute the mechanism by which P communicates with Q. Hence we have:

$$\mathbf{policy}_{P \rightarrow Q} \oplus \mathbf{mechanism}_{Q \rightarrow P} = \mathbf{mechanism}_{P \rightarrow Q} \quad (\mathbf{rule\ 2})$$

and symmetrically

$$\mathbf{policy}_{Q \rightarrow P} \oplus \mathbf{mechanism}_{P \rightarrow Q} = \mathbf{mechanism}_{Q \rightarrow P}$$

A system can thus be defined through recursive use of rule 2 composed with rule 1.

In [MBG+99a] we suggest that the following decisions have to be made to instantiate such an architecture:

- the number of operational abstractions in the architecture and layering with in these
- the system functions that the architecture allows applications to control (e.g. recovery, scheduling, clock ticks etc)
- the method used for specifying policy information
- the method used for passing system information between architecture components and system functions (up-calls, down-call and lateral IPC calls)

Any architecture implementing the above can be made to be compliant without the need of any common or regular interface mechanisms.

## **Adaptive and Self-Tuning Control**

Adaptive systems may be defined as systems which automatically use the results of current and past performance and operating conditions to alter their behaviour. A system which alters its behaviour so as to best serve the needs of a client application is compliant to that application.

Adaptive control of computer systems can be characterised by purpose into three categories:

- External performance goal: where the system seeks to maintain an externally measurable performance specification, such as a defined rate of transactions
- Internal performance goal: where some internal performance specification is generally expected to help system operation if it is maintained, such as maintaining garbage below a given fraction of heap space.
- Ah Hoc: where no individual metric is measured to define action, but some self directed action is taken based upon a general expectation that system operation will be enhanced, such as pre-fetching of disk blocks based upon recent access patterns.

Clearly each of these regimes can be modified in behaviour dynamically. For instance, by varying the target transaction rate; changing the fraction of heap; or modifying the heuristic for block pre-fetch. Further, these modifications may themselves be the result of some additional adaptive system's action. Thus a hierarchy of adaptive control is created. This hierarchy is one defined through the policy interface of each layer.

The functioning of a self-tuning system may be broken into three phases, measurement, prediction and reaction[GKK+99]. Following this taxonomy we discuss self-tuning and adaptive paradigms.

### **Measurement**

Measurement in control systems is typically a matter of measuring the output of a process, e.g. output rate of a product. In adaptive computer systems measurement takes the form of maintaining a history from which more complex results are derived. Broadly this history is used for two purposes:

- To directly drive some heuristic algorithm, e.g. predicting which pages to prefetch
- To maintain a dynamic prediction function, that is used in a feedback system.

### **Prediction**

Typically the most difficult part of an adaptive system is in the prediction phase. Conventional process control systems are usually blessed with a reasonably well understood process, one that is often amenable to mathematical analysis, and with typically well behaved transfer functions. This both aids analysis and helps guarantee stability of the system. Computer systems are sadly not nearly so well behaved.

The majority of adaptive control systems described explicitly assume that the prediction function is well behaved. Indeed the function must be monotonic. Functions that depart from this shape, or worse functions that exhibit steps, or even dual values almost certainly will result in unstable behaviour.

Some systems attempt to create a dynamic predictive model based upon recent system performance. E.g. Brown, Mehta, Carey and Livny [BCL96] base the correlation of transaction performance to buffer allocation on a moving window of recent transactions. This function is used to provide the buffer size for new transactions, so that the transaction will have a known performance.

The crucial flaw in such predictive systems is that they only work well if the underlying system is itself well behaved and free from phase changes. It is assumed that the underlying database never incorrectly allocates buffer pages. Thus they assume it never suffers from thrashing behaviour.

The prediction function must clearly cost less to evaluate and maintain than the fraction of system operation saved through the optimisation. Furthermore the operation of the optimisation itself may change the operating parameters of the system (for instance through cache pollution) and thus careful measurements are always needed to validate operation.

In self adaptive systems described thus far, optimisations have been limited in operation because they are only able to build a predictive function based upon a limited understanding of system action (e.g. through a pattern of page accesses). In contrast the CSA approach explicitly provides for policy information to be transmitted to any layer, and used to enhance the value of the prediction functions.

### **Reaction**

The reaction phase embodies the optimisation technique. As such it can take almost arbitrary form, however a number of general paradigms are evident.

- Change of operational parameter, whereby a simple system tuning parameter is changed with the intention of affecting system operation.
- Direct action, whereby the system takes a particular action, such as initiating pre-fetching of data, or starting a garbage collector.
- Changing the selection of a subsystem's operation, possibly by the selection of a new policy mechanism.

### **Abstract Layer Flattening**

A significant optimisation of system operation can be achieved when a multi-layered implementation is flattened, removing inefficiencies due to unnecessary data copying, and control transfer. Moreover, such flattening may be done so as to favour a particular activity or usage pattern. The ability to create such flattening dynamically, and later, to replace them dynamically is a powerful paradigm for an adaptive system. Flattening of a layered implementation typically requires significant programmer effort, and deep understanding of the operation of the target system. In the future, it might be expected that utilities that can automatically generate flattened implementations might be created for specialised tasks, based upon the insights gained from the hand generation of earlier flattenings. Examples of optimisation incorporated into such a flattening might include choices of particular buffer, hash table and cache management strategies, and use of protocols that are suited to particular needs (i.e. latency vs bandwidth vs reliability tradeoffs.) These tradeoffs are, in principle, of the same nature as those choices made in policy for the individual mechanisms in the layered implementation. The ability to take those same optimisations of policy and apply them in consort with the inherent performance advantages of a flattened implementation provide the basis for even greater system performance.

### **System Stability**

The integration of multiple subsystems, each with adaptive control regimes is a goal that is fraught. The reliability and stability of a higher-level control regime depends upon on all the subsystems behaving predictably and, most importantly, smoothly.

For instance, optimisations for memory allocation are typically designed to control and maintain locality of data. Data locality will clearly affect other layers of the storage hierarchy, and indeed the optimisations are usually designed with some knowledge of the behaviour of the hierarchy (e.g. clustering related objects on pages). Such techniques often work extremely well in many circumstances, and fail spectacularly in others, sometimes each behaviour evidencing itself at different times in the same application.

One of the most important aspects of adaptive control involves the detection and management of phase changes [ABJ+92]. Phase changes are those points where sudden changes in performance occur. Well known examples include the onset of page thrashing in a virtual memory system; and, in general, sudden drops in performance as a data set size exceeds the cache used to hold it. Garbage collection in the face of memory exhaustion similarly exhibits pathological phase changes.

The literature is replete with examples of targeted and special case optimisations and control algorithms to improve system performance, usually performance as measured by processing speed. The difficulty is that each individual algorithm is inherently selfish, and will only act to optimise its own local goals. This can (and often does) lead to conflicts, which can further lead to pathological or oscillatory behaviour. Building a monolithic optimisation and control system in which the entire system's

behaviour is monitored and controlled is both extremely difficult and fixes the interfaces, yielding an system that is very difficult to evolve with future demands.

Due to the nature of computer systems, there is not the mature theoretical approach that exists in control systems. Thus tactics for ameliorating interactions will be derived from expert understanding of the actual operation of the system and are unlikely to be easily codified as an automatic mechanism. Some general tactics include:

**Dampening of action:** Many pathological behaviours are evidenced by a catastrophic change in system function. A well know example is page thrashing which is unambiguously characterised by a system where CPU usage is mostly idle, about one fifth system, and a small fraction user. Using such metrics, an overlooking system control mechanism can detect failures in optimisation policy, and act to change the operating parameters of the individual policy mechanisms. In the face of many failures moving to a generic operating paradigm (or indeed the random selection paradigm described below) will move a system back to a workable (if sub-optimal) operating condition.

**Random choice:** For the purposes of analysis, and as a valuable policy in its own right, a baseline policy is always possible, one where a random choice of the possible actions is taken. For activities such as page discard or scheduling, such a policy can have real benefits, in particular it can decorrelate any relationship that two policies may have that might lead to pathological behaviour. Further, using a random choice provides a basis to which all other policies can be compared. Indeed a system could be constructed in which every policy decision was made randomly. Such a system might be expected to perform tolerably well, and be immune to oscillatory behaviour, and interaction between policy layers. An ability to dynamically switch optimising policy to one of random choice allows the system to act to remove interactions that may be creating oscillatory or other pathological behaviour.

## Conclusions and Future Work

We believe that a system based upon:

- the strictly defined and controlled layering of policy and mechanism of our compliant systems design,
- the complete control of all layers afforded by our persistent systems implementations
- the major semantic simplifications of an orthogonally persistent environment

allows an integrated adaptively optimising system to be synthesised. A system based upon these regimes should yield insights into the nature and utility of self-tuning and adaptive designs for a wide range of system architectures. Further work is required to better understand the nature of interactions, and to codify and generate more rigorous methods of dealing with these interactions.

## Bibliography

- [ABJ+92] Atkinson, M.P., Birnie, A., Jackson, N. & Philbrow, P.C. "Measuring Persistent Object Systems". In **Persistent Object Systems**, Albano, A. & Morrison, R. (ed), Springer-Verlag, In Series: Workshops in Computing, van Rijsbergen, C.J. (series ed) (1992) pp 63-85.
- [BCL96] Brown, K.P., Carey, M.J. & Livny, M. "Goal-Oriented Buffer Management Revisited". In Proc. ACM SIGMOD International Conference on Management of Data, Montreal, Canada (1996) pp 353-364.
- [CKW+96] Cook, J.E., Klauser, A., Wolf, A.L. & Zorn, B.G. "Semi-automatic, Self-adaptive Control of Garbage Collection Rates in Object Databases". In Proc. ACM SIGMOD International Conference on Management of Data, Montreal, Canada (1996) pp 377-388.

- [GG97] Gray, J. & Graefe, G. "The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb". SIGMOD Record 26, 4 (1997) pp 63-68.
- [GKK+99] Weikum, G., Konig, A.C., Kraiss, A. & Sinnwell, M. "Towards Self-Tuning Memory Management for Data Servers". Data Engineering Journal 22, 2 (1999) pp 3-11.
- [JS94] Johnson, T. & Shasha, D. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm.". In Proc. VLDB, Santiago de Chile, Chile (1994) pp 439-450.
- [MBG+99a] Morrison, R., Balasubramaniam, D., Greenwood, M., Kirby, G.N.C., Mayes, K., Munro, D.S. & Warboys, B.C. "A Compliant Persistent Architecture". To Appear: (1999).
- [MK94] McIver, W.J. & King, R. "Self-Adaptive, On-Line Reclustering of Complex Object Data". In Proc. ACM SIGMOD International Conference on Management of Data, Minneapolis, MN (1994) pp 407-418.
- [OOW93] O'Neill, E.J., O'Neil, P.E. & Weikum, G. "The LRU-K Page Replacement Algorithm For Database Disk Buffering.". In Proc. ACM SIGMOD International Conference on Management of Data, Washington, D.C (1993) pp 297-306.
- [WJN+95] Wilson, P.R., Johnstone, M.S., Neely, M. & Boles, D. "Dynamic Storage Allocation: A Survey and Critical Review". In **Lecture Notes in Computer Science 986**, Baker, H.G. (ed), Springer-Verlag (1995) pp 1-116.