Software Resource Architecture and Performance Evaluation of Software Architectures

C. M. Woodside,
Dept. of Systems and Computer Engineering, Carleton University,
Ottawa, Canada K1S 5B6
email: cmw@sce.carleton.ca

Abstract

Performance is determined by a system's resources and its workload. Some of these resources are software resources which are embedded in the software architecture; some of them are even created by the software architecture. This paper considers software resources and resource architecture, as an aspect of software architecture, It considers how resource architecture emerges, the relationship of software and hardware resources, some classes of resource architecture, and what they can tell us about system performance.

1.0 Introduction

Software is just a set of instructions that govern the use of resources such as processors, memory, buses, and peripheral devices. The software also introduces artificial or logical resources into the system to coordinate the use of physical resources such as memory or peripheral devices, (for example, a semaphore), to protect the integrity of data (for example, a lock), and for many other purposes.

Together the software and hardware resources govern the performance of a software system, in the sense of its response delays or its capacity to handle traffic. They can prevent a program from proceeding, until some resource can be allocated to the program. This aspect of *authority to proceed* is common to both software resources and hardware resources; we wish to emphasize their similarities and common properties, in an overall resource architecture for a system.

Resources are seldom treated systematically and carefully in software design, except perhaps for processor time. This may be intentional, as in a design which is intended to be useable over a wide variety of physical platforms (so it deliberately avoids consideration of physical resources). More often resource usage is an emergent property of the design, and resource interactions may introduce undesirable delays. In some systems the patterns of resource use have enough structure that we can identify a resource architecture, while in other cases the patterns are chaotic and the architecture is not obvious.

Software architecture is essentially a system description in terms of components and their interactions; more discussion of the definition is given by Shaw and Garlan [19], Bass et al [1] , or Hofmeister et al [6]. Components may be clients and servers, databases, filters, layers, modules or subsystems. Interactions include messages, calls, communications protocols, database access protocols, and multicasts. The value of architecture is that it creates a coherent overall structure which guides, contains and maintains the functional details.

Coherent overall patterns of resource use are similarly important, particularly for efficient operation. Some software components such as storage (buffers, files) are essentially just resources, while others, such as (e.g. operating systems processes, blackboards, databases) have resource attributes along with other roles. Poor resource use results in resource holding times that are longer than necessary, in fragmented operations which acquire and release resources many times, and in logical bottlenecks, which are discussed below.

Resource aspects of software are mentioned with regard to concurrency decisions, for instance in [19], and in architecture evaluation, for instance in [1],[2]. Klein, Kazman and Clements specifically considered performance as an example of attribute-based architecture evaluation, in [9]. Shaw has recently addressed the impact of changing resource capabilities in "open resource coalitions" which may assemble resources dynamically as they run [20]. However there has been little systematic consideration of resource attributes of software architecture. What we find are architecture evaluations with statements along the lines that an alternative is good for performance, because

it permits concurrency (see e.g. [2]). If we understand resource architecture better, we may be able to:

- develop resource-dominated software architectures from scratch, for applications in which quality of service is important,
- develop a performance model for a given software architecture,
- plan deployment of a given software architecture in different versions with different resources, for example over varying scales.

Performance approaches such as layered queueing models have been developed to study these issues, particularly the later two. A key notion in layered queueing is a resource entity that can take the role of a server, in processing a request, and then turn around and act as a client in requesting service from some lower level resource (e.g. [23], [27], [16]). This is natural behaviour for a software server, and leads to a model with abstract entities that correspond to software resources. Layered queueing has been applied to web servers [3], transaction processing [7], data routers [13] and distributed databases [21].

This paper has the goal of forging a connection between the software architecture and the resource issues, in order to promote all three of the potential uses listed above, and others which may follow from a clearer understanding. In this paper the idea and the essential features of a resource architecture will be described, including several architectural styles (which are also styles of software architecture). We will see that the resource architecture may be strongly structured, or not, and that when it emerges from a software architecture, the resources may or may not have a simple relationship to each other.

2.0 Resources in Software Systems

Software operations use resources as machinery, and this is entirely straightforward as long as one resource is used at a time (for instance a processor, a disk, a network interface). As a reference point for further discussion, consider the following sequence of software activities:

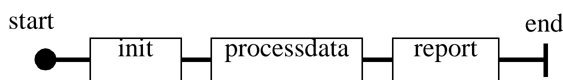

FIGURE 1. Scenario 1: a sequence of activities

The program executing Scenario 1 uses three physical resources: the user I/O interface, the processor, and the disk I/O resources, one at a time. These resources may force it to wait for the user to enter a command, for the operating system to dispatch it to the processor once it is ready, or for the disk I/O subsystem. It also requires a logical resource, which is an operating system thread which must be created or used to run the program. In some systems the total number of these threads may be limited, and a program may have to wait for a free thread before it can start. Figure 1 can be redrawn (see Figure 2) to show the resources for each activity, which will be called its *resource context.* The resource context of activity "in1",

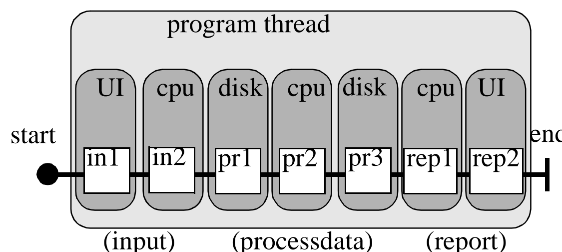

FIGURE 2. Scenario 1 with resources and resource contexts indicated on finer-grained activities

for example, is the user interface UI and the program thread. The resource context of a program changes over time.

Many performance models (notably all basic queueing network models) assume the program uses just one resource at a time. They ignore the thread resource in Figure 2 by only considering the set of active threads. Extended queueing models may be used to consider logical resources, but they easily become complicated, and their construction is difficult.

Now consider a slightly more complex program in Figure 3, in which the program obtains a lock before reading and modifying a file, and in which the file is on a network file server. The lock is an additional resource, held for only a part of the time, and the file server has its own thread or threads and CPU.

Clearly, from Figure 3, resource contexts are complex and they may change rapidly. For instance the program may switch between the file server cpu (fsc) and the disk (dsk) many times. A diagram like Figure 3 may have to be drawn in an abstract way to hide some of the complexity.

The concept of resource in a software system implies some concrete entity used by the program, which is either a device or some kind of logical token (which may be related to a device), or a member of a pool of devices or tokens. The concept includes the notion of authority to proceed, when the token is allocated by its manager (which may be a part of the operating system). Some of the diverse possibilities are:

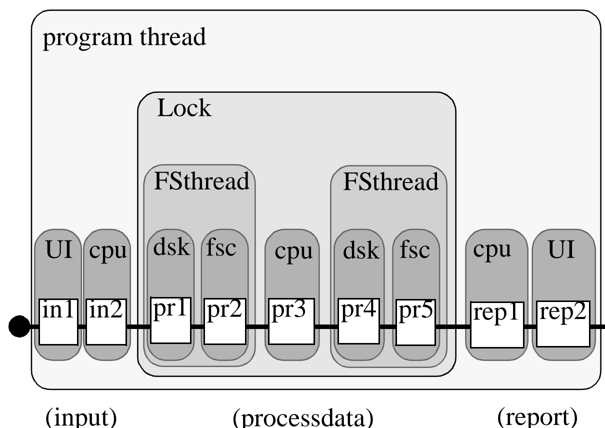

FIGURE 3. Scenario 2, with lock and file server resources, showing the resource context for each activity.

1. physical devices that execute operations, including processors, interface devices, buses, and disk devices and controllers,
2. memory or storage, including memory blocks, buffers, disk sectors
3. access to data (semaphores and locks). A counting semaphore is a resource pool
4. active threads, execution contexts, access control permits, or permission to carry out an operation or to transmit data. A flow control window for data communications can be regarded as a pool of resource tokens; so can a pool of execution threads.

A process with unlimited threads, such as a server which creates a thread per request, is an infinite set of resource tokens which actually cannot withhold permission to proceed. In this sense an unlimited resource pool is missing some of the attributes of a resource.

The concept of resource considered above is different from that of a URL or web page, or a resource in the RDF (Resource Description Framework) [11], which essentially refers to resources which are documents. The question of access limitations and permission is central to our notion of resource, but not to RDF (although RDF is designed to describe "all kinds" of resources and could perhaps be stretched to include the above examples).

3.0 Resource interactions

Resources are related to each other through the pattern of acquisition and release, and through being held simultaneously. The overall execution has a series of resource contexts with transitions between them, due to acquisition and release of individual resources. From the viewpoint of the resources, we can collect any sequence of operations within one resource context together, and call it a *context-operation*. The same context may also be used again for other context-operations. One way to relate the resources to the software is to identify the context-operations in the program. This may or may not reveal a structure we could call "architectural".

In some important cases resource use is highly structured:

- *Layered:* A common class of systems provides resources and services on demand. A request for a resource causes the program to add that resource to its context.

If resources are released in the reverse order to which they are obtained, as in Figure 3, this gives a kind of procedure-call or client-server resource style. The role of server in the interaction is taken by the resource, and the role of client is taken by the process requesting the resource. This gives a set of nested contexts, as shown in Figure 3.

This may be described as a "layered" resource style. In this style, outer contexts are at a higher layer and inner ones at a lower layer. Lower level resources are held for shorter time spans nested within the holding times of higher level resources. The "bottom" or innermost resource in any context is normally the device that is actually executing the current operation.

- *Separated:* In real-time systems a program is often divided into distinct "tasks" (like the activities in Scenario 1), which are scheduled separately. Deadlines are enforced by the scheduler, backed up with calculations of schedulability, and mechanisms like priority inheritance (see, e.g. [12]). The scheduler allocates the necessary resources to a task before it is launched, and they are released when it ends.

We may call this a "separate" or "operation-centred" resource style; a sequence of separate non-overlapping resource contexts are created. Figure 4 gives an illustration in which each task has its own operating system thread, and a physical resource (RC, RE, or RF in Figure 4), and each task also has its own collection of logical resources (RA, RB, RD, RX). Even though RA is used again in Task2, it is released after Task1 and then reacquired; this is in contrast with the case in Figure 3.

- *Pipeline:* Pipelines are widely recognized and used, in hardware and software, and in their classic form they are a "one-at-a-time" resource style. However they may also appear in a generalized form as a sliding series of overlapping contexts, as shown in Figure 5, in which resources are acquired in some order and

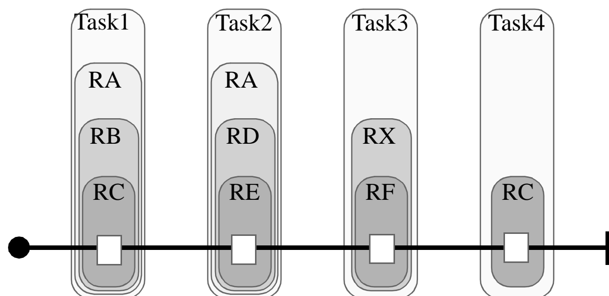

FIGURE 4. Separate resource context for each activity

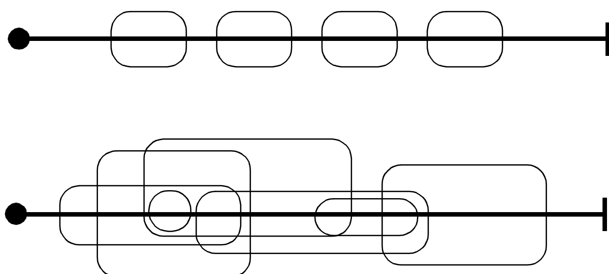

FIGURE 5. Resource pipeline (above), and sliding overlapping resource contexts over time

released later in the same order, while operations are carried on. As an example, a file may be opened in one stage and closed in some later stage, with ownership passed along the pipe.

3.1 Use of System Views

The discussion so far has been based on a path-oriented view of the system, as described in the three-view model for performance engineering in [28] (with views of Maps or modules, Paths or scenarios, and Resources). The following discussion will develop the corresponding Resource view. The Path view is similar to the execution view of Hofmeister, Nord and Soni [6] (and the Map view is similar to their module view). In [28] it is argued that the Resource view is essentially different from the others.

3.2 Resource-operations

In the special configurations just considered, the context-operations can be refactored or clustered into operations associated with each resource, initiated when the resource is obtained and covering the span of time the resource is held. These will be called *resource-operations*.

A resource-operation collects together all the software operations, that are executed in the context of a given resource. The resource-operation can be characterized by its workload, which includes its immediate execution (if it is a device that can execute programmed instructions, such as a CPU), and demands for other resource-operations. Thus the resource R1 in Figure 6 has the operation R1-op, which makes demands on CPU and on R2 (and R2-op). The CPU operation is to execute the code of other operations such as R1-op, and it is usually clear (and less cluttered) to not show the CPU-operations separately in a diagram. A request for a resource is shown by an arrow; the execution path goes to the resource or the resource operation (like a procedure call). When the resource is released the execution path follows the arrow backwards to the calling (or requesting) context.

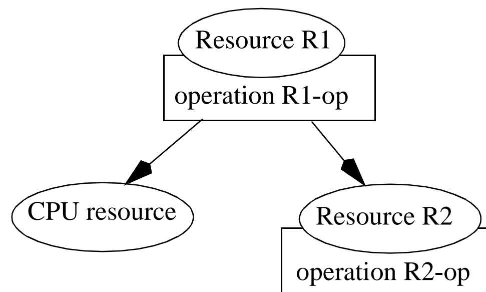

FIGURE 6. A Resource-operation

This view of resource-operations matches very well with the layered style of resource architecture and also with several other styles. Figure 7 uses it to show the layered resource architecture involved in Scenario 2, and the nested contexts of Figure 3. In the case of the file server there are two resource-operations, Read and Write, and the request arrow for the resource goes to the particular operation being requested. In the case of the lock, there is only one shown but others might exist, so the request is also shown going to the operation. In the case of the CPU, file-server processor FS-CPU, and the disk there are multiple operations with the details suppressed, and the request is shown going to the resource itself.

The shaded resources and path in Figure 7 show the resource context of one kind of disk operation, as a path down through the model from the root resource (the program thread) to the active execution resource (the disk).

The layered graph notation has the advantage of compactly representing quite complex patterns of resource use, provided they are nested. It will be shown that it can also, with a few additional conventions, represent other structured patterns.

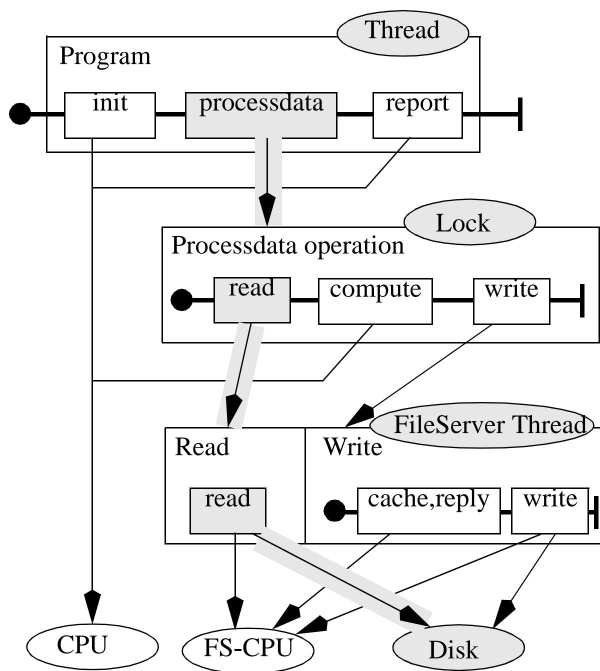

FIGURE 7. Resource architecture, with highlighting of the resource-operations in Scenario 2

3.3 Layered resource architecture notation

There is not space here for a full definition of a notation for resource architecture, but Figure 7 shows the key features. Each resource has one or more resource-operations, within each of which there is one activity or a sequence of activities, which may in turn make requests for other resource-operations. Resource-operations may be hidden (as for the Disk) if there is only one, or to hide detail. This notation provides a basis for describing structured resource architectures. It looks like a call graph, and like a call graph it implies that the requesting resource-operation is blocked until the resource it has requested is released. A full notation needs additional features, such as non-blocking operations and request-frequency parameters.

Figure 7 has a superficial resemblance to the software architecture, a resemblance which may have substance, if resource-operations are embodied exactly in the software components. Then there is a one-to-one correspondence of resources to the software components. On the other hand the boundaries of software components could be quite different, and have no relationship to the resource-operations. For example the Processdata operation associated with the Lock is actually performed in the Program software module. The software architecture corresponding to Figure 7 might look like Figure 8, which is much simpler:

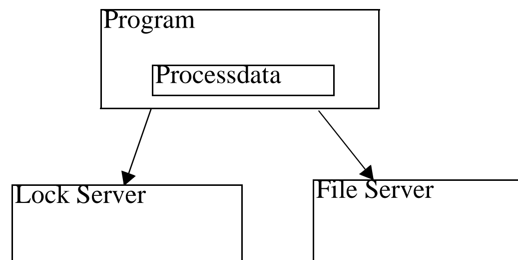

FIGURE 8. Software architecture that might correspond to Figure 7, as a box-connector diagram

A notation for performance modeling, called layered queueing networks (LQNs), has been developed based on the ideas in Figure 7. The models and tools have been developed by several authors and are described in [26][27][16][4][5][15]. An LQN determines the delay in waiting for all resources, at every level in the layered hierarchy. It accounts for how the holding time of a resource includes waiting and holding for lower level resources. The acyclic request graph of a layered architecture has the advantage of imposing a condition which avoids resource-based deadlock. That is, because two concurrent programs within the same architecture request their resources in the same order, they cannot deadlock in a situation where each is waiting for a resource already held by the other (a consequence which is well known in the use of an acyclic graph to establish or maintain a locking order).

Applications of LQNs include client-server systems, web servers [3], transaction processing [7], distributed databases [21], telecom connection management [8] and intelligent network (IN) servers [23].

A single-layer LQN, such as would arise from Scenario 1 of Figure 2, is just an ordinary queueing network model. The Program resource in Figure 2 would be the customer or customers, and the other resources would be the servers.

Our resource architecture model can benefit from some generalizations that have been made in LQNs, beyond what was described in the previous section. These include:

- non-blocking requests, in which the execution path goes on to a new context when it releases a resource, rather than back to a previous one,
- early release of the requester, while the requested resource continues to be used,
- forwarded requests, which are asynchronous at a low level while holding some set of higher level resources.
- multiple copies of a resource, all identical and managed as a resource pool.

3.4 Resource pipelines and non-blocking interactions

In a classic pipeline a package of data is passed from one resource to the next, as a buffer or message or file. As it arrives at the next stage it releases the one before. The resource-operations are exactly mapped to the pipeline resources.

Passing data like this will be termed a *non-blocking interaction* between the resource operations. Non-blocking interactions are not limited to pipelines; they also occur when a series of resources is triggered one at a time to work on a job, in any order.

Also, a chain of non-blocking interactions can occur in the midst of a layered system, with some blocked resources also being held throughout the sequence. In this case we may interpret that the request to the first resource in the chain is being forwarded through the sequence, and these are called *forwarding interactions*. They are remarkably common, for example where an input thread dispatches requests to a set of worker threads.

3.5 Early release of the requester

We say a requester is released early if the resource-operation making a request can resume while the requested resource is still busy. The two resources can then be active in separate concurrent resource contexts, which increases the system concurrency levels.

In one kind of early release interaction, when the requester releases a resource R the execution path splits and an independent resource context is started up, based on R, in parallel with the continuation in the context of the requester. This pattern of resource behaviour compactly captures a fairly common behaviour, for instance:

- a server returns a result to an RPC client, and then does any work which is not in the critical path of the reply, such as buffer clean-up, or logging,
- a pipeline stage accepts its input from a blocked upstream stage (perhaps using a shared buffer), and then releases the upstream stage and continues on by itself,
- a task is handed off to a server to be performed independently.

In layered queueing terminology the part of the resource-operation after the early release is called a “second phase”.

3.6 “Real-time” separable architectures

In schedulable real-time systems the resource contexts of activities are separate, and a scheduler allocates all the resources it needs to a task when it is time for it to run. Resources are acquired atomically, all or none, which means that order is not significant. This requires a single all-knowing scheduler (although some aspects of its operation may be distributed), which limits the applicability of this architecture. This architecture is used in real-time control and real-time databases and networks (e.g. [12]).

3.7 Resources with delayed release

A common pattern of resource holding, which breaks the layered structure, makes an interesting study. Consider a layered system with one exception, a resource which is obtained in a deeply layered context, and retained when other resources “above” it are released. For instance, a system might obtain rights to a buffer on a remote system, through the use of remote resources which are then released, and then later pass the buffer rights to an agent on the remote system, to use. The agent might then use the buffer and then pass it back, release it or pass it on. The resource contexts for such a system are shown in Figure 9, and we can see that they are not nested. This thoroughly breaks the layered resource pattern.

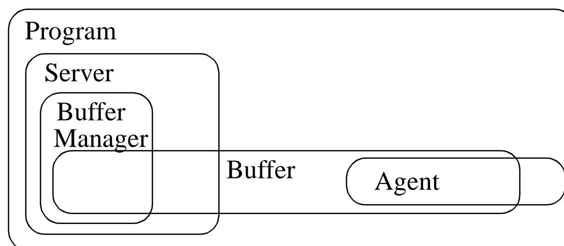

FIGURE 9. Delayed release of a Buffer resource, with non-nested resource contexts.

If resources are layered except for a subset, the subset can be shown as exceptions. For one isolated delayed-release resource, the request and entry into the resource-operation are just as before, but the resource-operation completes without releasing the resource; this must be indicated as a special class of resource. Then the release is a separate interaction, which could be indicated graphically by a special class of arc. Figure 10 shows a Buffer in such a special class of resources, labelled (DR) for delayed release, and the release interaction indicated by an arc labelled Release. There could be multiple alternative release points. A weakness of this notation is that the scope of the resource contexts covered by the Buffer is not clearly identified.

3.8 Multiple copies (multiple threads)

A resource may be provided in multiple copies. For instance, the buffer described just above would normally be one of a pool all managed by the Buffer Manager. A server process may be multi-threaded; a processor may be

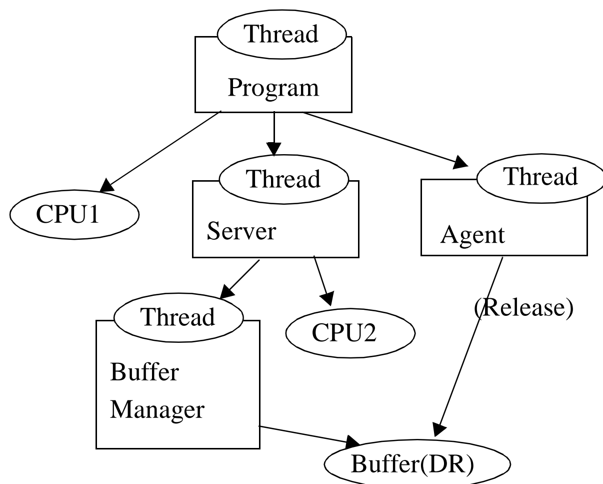

FIGURE 10. Delayed-release resource shown as an exception within a layered architecture

a multiprocessor. Multiplicity should be a parameter of a resource. Sometimes a resource is multiple without any limit, for example a server process which creates a thread per request.

4.0 Consequences of resource architecture

The resource architecture, with the parameters of the resource-operations, governs the performance of the system. Models based on the architecture can be used for evaluation. This has limitations deriving from the degree of abstraction in the architecture; if the architecture is abstract and misses some fine-grained resources, a prediction based on it will miss the effect of those resources. This should not be seen simply as an *error*; it is an aspect of *abstraction*. The predictions have to be understood as abstract; it is not intended as a joke to say that abstractions in predictions just *look* like errors. Financial budgets, for instance, contain similar abstractions. They are correct at their given level of abstraction.

4.1 Resource overhead

Acquiring and releasing resources incurs overhead which is part of every resource-operation. Fine-grained resource manipulation can cause explosive overhead costs. A nice example is data access under the Simple Network Management Protocol (SNMP), where the protocol states that every remote value is retrieved separately from a Management Information Base (MIB). This can cause heavy data traffic to be visible in the parameters of the resource architecture [22].

4.2 Software bottlenecks

One consequence of a software architecture may be a software resource bottleneck. Layered queuing has been used to investigate this phenomenon, as it relates to process threads, in [14]. A process thread remains “busy” when it is blocked, waiting for an event or message or reply from some other process or device. A typical source of blocking is waiting for disk I/O to complete. While one thread is blocked, another one could be using the processor, if there is one ready to run. An insufficient thread pool is an example of a software bottleneck, which can relieved by changes which are entirely in the software. Another cause of thread starvation is that all threads are blocked, no matter how many are provided; in this situation the bottleneck is in the lower subsystem (e.g. in the file server or the disk). In [14] a measure of “bottleneck strength” was described, to identify where the cause of the bottleneck is located.

Other similar software bottlenecks due to logical resources may occur at flow control windows (provide a larger window) or at a lock (provide finer-grained locking). Similar bottleneck patterns occur in layered hardware resources, for instance a bus bottleneck when the bus is used to access memory and various interfaces [13].

Patterns in resource relationships that lead to bottlenecks, and strategies for relieving them, have common forms in very different kinds of system. The resource architecture can be the same, with a role taken by a process in one system, filled by a critical section or a bus in another.

5.0 Relationship to software architecture

In a given system, how do the resources relate to the software architecture? We can look at this question in three ways: resources as an emergent property, software deliberately designed around its resources, or resources deliberately kept orthogonal to the software structure. There is also the real-time separated-resource-context case, where there is effectively no resource architecture as such.

5.1 Emergent resource architecture

The software architecture is determined using a variety of relevant criteria, including performance and resources. A wide-ranging discussion of methods for creating and evaluating architectures is given by Bass, Clements and Kazman [1], and real-time system software architecture evaluation is addressed directly by Kazman et al. in [9]. Some resource issues are dealt with explicitly but other resources for controlling data access, for storing temporary data, for concurrent threads, and so forth may just accumulate from the totality of considerations. In this way a resource architecture emerges, and it may or may not

have a recognizable structure. How can it be recognized and extracted?

Kazman and Carriere have considered a similar question for software architecture, and found the relationships from static analysis [10]. For resources, it is clear that scenarios or traces must be analyzed to identify resource demands and holding times of higher level resources. This is the basis of classic software performance engineering recommendations by Smith [24], for instance. However Smith concentrated on demands from hardware devices, and gave only limited assistance for dealing with logical resources and concurrent execution. In [25] Smith and Williams considered software architecture and performance, but the architecture was at a very fine-grained level (data objects) and mainly affected the hardware demands. Hrischuk et al. [7] have described an approach for finding resource contexts and layered performance models from traces, even when objects have been dynamically created and linked.

Emergent resource architecture partly follows software structure. Concurrent processes involve resources, for instances. Semaphores can be identified, but the requests may be buried in low level modules even though they govern higher-level operations by other modules. Thus the scope of a resource-operation may be difficult to determine. Buffer resources pose similar questions.

5.2 “Resources first” development

Systems with critical performance requirements, that are not amenable to separate-context design may be designed around the critical resources. Internet routers come to mind as an example, with the routing table as one critical resource.

In designing reactive software systems (systems where the function is mainly to respond in a timely way, and in the correct order, to external events) Selic et al. have insisted that architecture should not only identify components and interactions, but must also describe behaviour [18]. This is certainly also true where resources are considered first. It is implied by the central position of resource-operations and their interactions, in the architecture. Scratchley and Woodside have considered concurrency-related architecture decisions within an integrated scenario specification in [17], and have evaluated substantial alternatives in software architecture for a group communications system. Some ideas for generating architectural alternatives around performance concerns were described in [28].

A general approach is to develop the resource architecture first, from an analysis of scenarios, then to estimate budgets for operation times, validate the performance measures on the basis of the budgets, and finally go on to develop all the other aspects of the software within the budgets. This process mimics the way projects are managed to fit within financial budgets, and allows for iterations and adjustments as problems are revealed. This is a subject of current research.

5.3 Resources orthogonal to software

This name is applied here to systems in which the software design avoids resource commitments, so it can be deployed in many different situations, with different resources. This approach appears to be implicit in many theoretical ideas of distributed systems. Ideally, resources can be completely ignored in the software, and included in a configuration step which specializes the system to a particular deployment option.

In practice, there are still *resource roles* in the design, or implied by it. They may be employed only in some versions of the system. For instance semaphores to protect data shared by multiple threads are not needed in a small-scale single-threaded deployment. The resource architecture is a property of the deployment, and is effectively isolated from the software architecture. However the semaphore programming has to be provided in the system, even if its use is made optional.

In this situation there may be many resource architectures that could be used with a single software architecture, and they may be quite different. For instance a series of operations in a subsystem might be configured as a resource pipeline in one deployment and a hierarchical master-slave style in another. The possibility exists of optimizing the deployments within a general plan, and some of these issues were explored in a recent study of scalability of software architectures [8].

The programming to support multiple resource configurations is likely to be complex, which is a negative aspect of this concept.

5.4 Separate resource contexts

The approach of separating the resource contexts of operations or “tasks” is distinctive, and has been described earlier as characteristic of real-time and embedded systems. Design of these systems is a large subject, and the way resources are treated is a side effect of the design philosophy. In one approach, resources and operations together are put (as much as possible) under a single system-wide layer of control, instead of having the control dispersed to separate controllers for each resource. The resource behaviour is analyzed by schedulability analysis and enforced by the controlling layer. Resource relationships once discovered are embodied in the analysis and the control.

For some resources (especially in hardware), dispersed request-based control may remain, and may cause difficulties in the analysis and control. Priority inheritance can sometimes be used to mitigate these problems.

Software architecture may still be influenced directly by the raw hardware demands. For instance in [9], performance issues which arose in evaluating a real-time embedded software architecture centred around a hardware limitation (a channel bandwidth).

As a general solution, system wide resource control and resource context separation has the attraction that it establishes some control over timing of execution of operations. However systems built this way are circumscribed in their applications and may be sensitive to changes. For instance the design may depend critically on a clock rate, or on how many functions have to be executed in an execution cycle. Once any aspect of these tightly controlled solutions breaks down the system must be reconsidered from the beginning. The composition of systems into larger systems has to be done with great care and may not be possible. So this approach does not appeal to designers of systems in which it is not essential, for instance in telecommunications.

5.5 Styles

It is not the intention of this paper to go into architectural styles in depth, but it has already been pointed out that a resource architecture may have a pipelined style, or a heirarchical (call-return) style, etc. There is usually some similarity between the style of software architecture and the resource architecture, but it is not necessary. This point needs further consideration.

6.0 Conclusions

A concept of resource architecture has been described, which applies to software resources as well as hardware. There are resource-operations attached to each resource, and resource interactions between the operations. This kind of structure can be found in many software systems. The form of the architecture model description is backed up by a performance modeling methodology called layered queueing.

Resource architecture may be imposed on a system from the beginning, or may be identified as an emergent property of a mature system. Identifiable styles of resource architecture include pipelines, hierarchies and layers, and operation-centred styles. A chaotic architecture, with resource relationships that have no particular structure, may be subject to inefficient resource use and to resource deadlock.

For understanding performance issues, and adapting designs to solve performance problems, software resources will usually be considered together with hardware resources, in a *system resource architecture*, which could in principle be described by the approach of this paper. The integrated study of software and hardware resources is essential for understanding the operation of the system. However a given software system is likely to be deployed in many different environments, so its software resource architecture should be developed first and used as a component in many system architectures.

Acknowledgements

The comments of the anonymous referees were very helpful. This research was supported by the Natural Sciences and Engineering Research Council of Canada through its program of Research Grants.

7.0 References

[1] L. Bass, P. Clements, R. Kazman, *Software architecture in practice,* Addison Wesley, 1998

[2] L. Cheung, B.A. Nixon, E. Yu, "Using non-functional requirements to systematically select among alternatives in architectural design", *Proc. of First Int. Workshop on Architectures for Software Systems*, April, 1995, pp 31-43.

[3] J. Dilley, R. Friedrich, T. Jin, J.A. Rolia, "Measurement Tools and Modeling Techniques for Evaluating Web Server Performance", *Proc. 9th Int. Conf. on Modelling Techniques and Tools*, St. Malo, France, June 1997

[4] R.G. Franks, S. Majumdar, J.E. Neilson, D.C. Petriu, J.A. Rolia and C.M. Woodside, "Performance Analysis of Distributed Server Systems", Proc. Sixth International Conference on Software Quality, Ottawa, Canada, October 28-30, 1996, pp. 15-26.

[5] Greg Franks, Murray Woodside, "Performance of Multi-level Client-Server Systems with Parallel Service Operations", *Proc. First Int. Workshop on Software and Performance (WOSP98)*, pp. 120-130, Santa Fe, October 1998.

[6] C. Hofmeister, R. Nord, D. Soni, *Applied Software Architecture*, Addison-Wesley, 1999.

[7] C.E. Hrischuk, C.M. Woodside, J.A. Rolia, "Trace-Based Load Characterization for Generating Software Performance Models", IEEE Trans. on Software Engineering, v 25, n 1, pp 122-135, Jan. 1999.

[8] Prasad Jogalekar, Murray Woodside, "Evaluating the Scalability of Distributed Systems", IEEE Trans. on Parallel and Distributed Systems, to appear in 2000.

[9] R. Kazman, M. Klein, P. Clements, “Evaluating Software Architectures for Real-Time Systems”, Annals of Software Engineering, Vol. 7, 1999, 71-93.

[10] R. Kazman, S. J. Carriere, “Playing Detective: Reconstructing Software Architecture from Available Evidence”, Journal of Automated Software Engineering, 6:2, April, 1999, 107-138.

[11] O. Lassila, R. R, Swick (eds), Resource Description Framework (RDF) Model and Syntax Specification, W3C Reccommendation 22, World Wide Web Consortium, Feb. 1999.

[12] S. Levi, A. K. Agarwala, *Real Time System Design,* McGraw-Hill, 1990.

[13] P. Maly, C.M. Woodside, “Layered Modeling of Hardware and Software, with Application to a LAN Extension Router”, Proc. Performance Tools 2000, Chicago, March 2000.

[14] J.E. Neilson, C.M. Woodside, D.C. Petriu and S. Majumdar, “Software Bottlenecking in Client-Server Systems and Rendez-vous Networks”, IEEE Trans. On Software Engineering, Vol. 21, No. 9, pp. 776-782, September 1995.

[15] S. Ramesh, H.G. Perros, “A Multi-Layer Client-Server Queueing Network Model with Synchronous and Asynchronous Messages”, *Proc. of First Int. Workshop on Software and Performance (WOSP98)*, pp. 107-119, Oct. 1998

[16] J.A. Rolia, K.C. Sevcik, “The Method of Layers”, IEEE Trans. on Software Engineering, v 21, no. 8, pp 689-700, August 1995.

[17] C. Scratchley, C. M. Woodside, “Evaluating Concurrency Options in Software Specifications”, Proc 7th Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecomm Systems (MASCOTS99), College Park, Md., pp 330 - 338, October 1999.

[18] B. Selic, G. Gulleckson, P. T. Ward, Real Time Object Oriented Modeling, publisher, 1994.

[19] M. Shaw and D. Garlan, *Software Architecture*, Prentice-Hall, 1996

[20] M. Shaw, Sufficient Correctness and Homeostasis in Open Resource Coalitions”, WORKSHOP at Nth Int Conf on Software Engineering (ICSE2000), Limerick, June 2000.

[21] F. Sheikh and C.M. Woodside, “Layered Analytic Performance Modelling of a Distributed Database System”, *Proc. 1997 International Conf. on Distributed Computing Systems*, May 1997, pp. 482-490.

[22] F. Sheikh, J.A. Rolia, P. Garg, S. Frolund, A. Shepherd, “Performance Evaluation of a Large Scale Distributed Application Design”, *Proc. of World Congress on System Simulation*, Singapore, September, 1997 .

[23] C. Shousha, D.C. Petriu, A. Jalnapurkar, K. Ngo, “Applying Performance Modelling to a Telecommunication System”, Proc. of First International Workshop on Software and Performance (WOSP98), pp. 1-6, October 1998.

[24] C.U. Smith, Performance Engineering of Software Systems, Addison-Wesley, 1990.

[25] C.U. Smith, L.G. Williams, “Performance Evaluation of Software Architectures”, *Proc First Int. Workshop on Software and Performance (WOSP98)*, pp. 164-177, October 1998

[26] C.M. Woodside, “Throughput Calculations for Basic Stochastic Rendezvous Networks”, Performance Evaluation, v 9, no 2, April 1989.

[27] C.M. Woodside, J.E. Neilson, D.C. Petriu and S. Majumdar, “The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software”, IEEE Transactions on Computers, Vol. 44, No. 1, January 1995, pp. 20-34.

[28] C.M. Woodside, “A Three-View Model for Performance Engineering of Concurrent Software”, IEEE Trans. On Software Engineering, Vol. 21, No. 9, pp. 754-767, Sept. 1995.