Adapting applications to exploit virtualization management knowledge

Vitalian A. Danciu
Munich Network Management Team
Ludwig-Maximilians-Universität München
http://mnm-team.org./~danciu

Alexander Knapp
Institute for Software & Systems Engineering
Augsburg University
http://www.isse.uni-augsburg.de./staff/knapp

Abstract— **Today's applications do not react to the ad-hoc, dynamic changes in locality, performance and environment that are characteristic of virtualized infrastructure. We illustrate exemplary effects experienced by distributed programs in reaction to change in the infrastructure and explore call interception, library replacement and aspect-oriented programming as alternatives for remedy. We demonstrate the remedial effect of adaptive code introduced without change to the original application code, or its bindings. We sketch a software architecture to make available management knowledge as a base for adaptation.**

I. Introduction

Virtualized infrastructure is becoming pervasive, and it can reasonably be expected to form the basis for computing in general in the near future. While physical hardware is being replaced with virtual hardware as an execution platform, the application software is executed unmodified: neither does it require adaptation in order to function, nor does it leverage the properties inherent to virtual components. We endeavour to make software aware of the properties of virtualized infrastructure and to explore mechanisms that allow applications to adapt dynamically to run-time changes in their environment.

The chief driver property of virtualization is strong encapsulation of computing, storage, network and I/O resources that, at the same time, provides a resource abstraction. It also confers novel properties to the environment for the execution of applications. One of the novel properties of distributed applications executed on virtualized infrastructure is the mobility of their components at run-time. Virtual Machines (VMs) can be migrated manually (in response to management action) or automatically (due to load balancing policy) during execution, along with any software components being hosted on them. Thus, while an application component retains some of its environment (the state provided by the VM), its environment changes with respect to any aspect outside the VM. Its position in the network topology changes with respect to other components, inducing changes with respect to locality network quantities; the target machine may be different with respect to CPU capacity, I/O throughput and load. Network virtualization introduces changes in paths, forwarding policy, delays to other application components, and so on.

Presently, software does not adapt to these quick changes in environment at all, whether to exploit opportunities or to offset a negative impact of the changes. What is more, the application component lacks access to the management knowledge about these changes, which is a pre-requisite for adaptation.

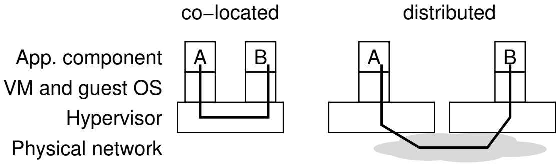

Fig. 1. Experimental setup: Different network connection between VMs, invisible to application software

Proof-of-problem: Consider the two setups shown in Fig. 1: two virtual machines in a VM cluster communicate either over the virtual network within the hosting physical machine (left hand side), or their communication traverses a physical network (right hand side). Within a short time, a VM can be migrated from one host to another and thus changes the path used by the network connection. However, neither the guest operating system of the VM, nor the application are aware of the change in location, which may imply a change in communication performance.

To demonstrate this point, we have implemented this scenario by deploying a simple distributed application on two VMs running on Xen hosts. We measured the throughput between the application's components for a large number of consecutive write operations of different (chunk) sizes to a connectionless (UDP) transport service. We employed no management (flow control, rate limitation, . . .) of the sender.

We repeated the experiments with the VMs

1) co-located on the same host,
2) distributed onto two hosts connected by a standard 1G-Ethernet switch and
3) distributed onto two hosts each connected via 1G-Ethernet ports but via a link with lower (100 MBit/s) transmission speed, i.e. a "choked" channel.

We noted the time necessary to transmit 10 MBytes of application payload and the percentage of messages lost between the VMs, for each of the three cases, as shown in Fig. 2.

Unsurprisingly, the results show differences in time for some chunk sizes, though the differences are rather small. In con-

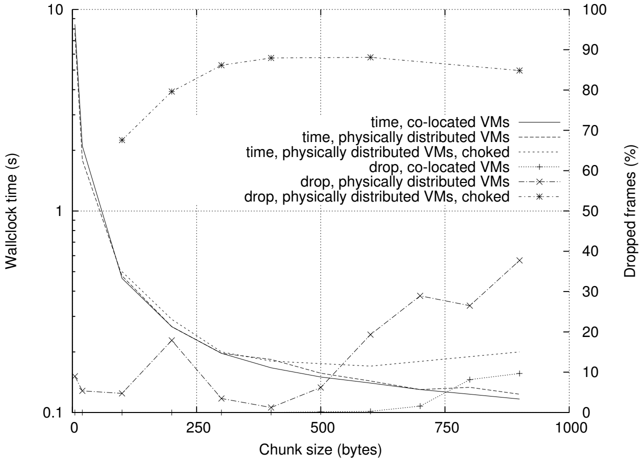

Fig. 2. Duration and drop rate of 10^7 bytes transmission using varying chunk sizes for different setups

trast, the number of dropped chunks varies greatly between the three setups: while the co-located VMs tend to lose only few segments at most sizes, the loss over the standard physical link is significant, and the loss over the choked connection is unacceptable.

While the absolute values are of little concern (their meaning depends heavily on hardware capabilities and on the method of measurement), the relative values show significant differences in behaviour, as a consequence of the change in environment.

Challenge: As changes to the environment occur dynamically, we propose to equip operating systems and application software to be able to react. Note that "change" may denote both improvement and deterioration of environmental parameters for an application, and it may be one the following:

- gain/loss of topological and geographical locality
- change in channel (I/O, network, ...) capacity
- change of CPU capacity; hence, reaction time
- the security and administrative context of the target physical machine, i.e. the trust potential of the host

This list cannot be comprehensive: it reflects potential changes that have been recognised by the authors, and it might be extended at any time. As each class of change may require its own remedy, we cannot devise a specific method, but rather require a generic method for rendering applications adaptive.

However, adaptivity itself is dependent on knowledge of the environment. Hence, only when provided adequate information from outside the application, can any benefits be expected.

Synopsis: We present different avenues of approach in the following Sect. II before reviewing related approaches from other domains in Sect. III. We proceed by describing a prototype based on aspect-oriented programming in Sect. IV and showing, that it achieves a change in application behaviour. We discuss the requirements, obligations and limitations of the approach in Sect. V. It becomes obvious, that software needs information from outside the VM in order to take correct decisions regarding whether and how to adapt its behaviour. To this end, we sketch an architecture in Sect. VI, that addresses this concern. We conclude by proposing interesting topics for future extension of this work in Sect. VII.

II. Approach

To modify the behaviour of a (distributed) application is to modify the code that it comprises itself or the code that it requests to be executed (library and system calls). The code stack involved in the execution of an application includes typically the application logic itself, application-specific libraries, system libraries and the OS kernel. The desired modifications can be inserted at any of these points, depending on the scope and the degree of invasive modification and the required effort.

A. Alternatives to approach

The following techniques lend themselves to our goal to modify application behaviour:

1) Adaptation of system and application libraries: we could modify library code (e.g. in the C standard library or in libraries linked to the application) to exhibit the desired change in behaviour.
2) System/library call interception (e.g. symbol substitution): we could intercept selected library calls and, without modifying the library, execute alternative code.
3) Programmatic modification: we could alter the application code systematically to change its behaviour at selected points.
4) Manual modification: we could re-write the application code to change its behaviour.

To select the suitable method for modifying the application, we take into account the properties sketched in Fig. 3:

scope: applications differ in natural behaviour (communication volume and patterns), thus they require specific modifications. For example, solutions introduced to optimise throughput in one application (such as the buffering scheme exemplified in Sect. IV) may harm the performance delay-sensitive ones, as well.

required authorisation: the effective options are constrained by the deployment platform for the software, e.g. a Platform-as-a-Service cloud offering may lack the option of replacing system libraries to avoid platform/system malfunction.

required knowledge and effort: the alternatives differ in the required knowledge about the inner workings of the application and its requirements on the execution platform. Obviously, the most knowledge-intensive option is the manual modification of the application code; it also requires the largest effort, and carries the highest probability to introduce errors.

Taking into account that different applications will require different modifications, we are forced to exclude the alternatives that are very broad in scope: modification to the libraries affect all applications, and even application-specific

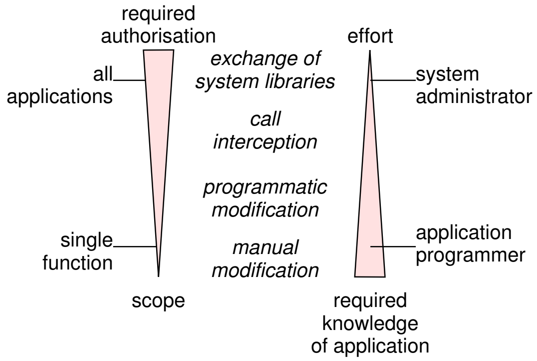

Fig. 3. Alternatives to modifying application behaviour

call interception applies the modification to all instances of a call. Manual modification must also be excluded in the general case, simply taking into account the enormous amount of code potentially being executed on virtualized infrastructure, and the application-specific expertise required compounded by the effort of manually modifying and testing the application. The remaining option, a programmatic modification of the application code, does not require modifications to the system, is application-specific and can ideally be applied automatically.

B. Aspect-based technique

Several *generative programming* techniques [3] lend themselves to programmatic modification of existing and emerging applications. The simplest form of modification, still encountered in many programming projects, is to use pre-compiler directives (e.g., `#ifdef`/`#endif` in C) to control insertion or suppression of code fragments. Widely used, this technique tends to render the source code difficult to maintain [5]. In contrast, top-down code generation, as in Model-Driven Architecture (MDA) style [2] development enables changes to the generic or platform-independent models to affect the desired modification in the resulting, generated code. While promising the best integration with the development process of the software, this approach is limited to software developed by means of MDA or similar model-driven frameworks.

To maintain the integrity of the pre-existing code and avoid limiting our approach to one software development paradigm or other, we use *aspect-oriented programming* (AoP) techniques to achieve the necessary modifications. They allow reuse of existing ("legacy") code as-is while avoiding additional internal complexity.

III. Background and related work

A. Context-aware adaptation

Making applications in a virtualized environment aware of and reactive to environment changes can be classified as turning these applications into *context-aware systems*. Context-aware systems, in general, reflect knowledge of their environmental status and show the ability to adapt their behaviour according to this environmental information without explicit user intervention in order to increase their usability or efficiency [1]. The contextual information used can be of rather different kinds, like physical location, user identity, or as in our case, the state of the virtualization environment.

Context awareness is a cross-cutting, pervasive concern of a system that affects many of its components. Thus it has been suggested before to use aspects for the design and implementation of context-aware systems [11] which has been taken up in various application domains, like services [7]. The introduction of context-awareness into existing software by using aspects has recently been considered in the field of high-performance computing for the case of multi-core applications [4], though also involving manual code changes.

B. Adaptivity in software

Some software is adaptive by design, or even self-adaptive, meaning it adapts itself without the benefit of external influence. Autonomic computing concepts fall within this category, as does self-management in the distributed system—though they concern management, not application software.

a) Example: Network protocols as self-adaptive software: Some reliable network protocols, such as the Transmission Control Protocol (TCP) [9], are excellent examples of self-adaptive software: they inspect or observe their environment and change their behaviour according to the knowledge gathered, in order to achieve their goals. The information they consume originates both locally, accessible via the operating system (e.g., the maximum transfer unit of a network interface, that determines the TCP segment size) or externally. TCP, for instance, acquires information unavailable locally by probing the maximum transmission rate, that is possible on an end-to-end path, as part of a congestion control strategy.

TCP is self-adaptive: it acquires the knowledge for effective adaptation itself, reasons about a necessary adaptation of behaviour and executes the adaptation.

b) Avoiding self-adaptivity: We do not endeavor to make applications self-adaptive, for the following reasons:

overhead: each application makes an effort to create and use knowledge; this might mitigate any benefits achieved by the modification.

invasiveness and size of modification: the code necessary to introduce self-adaptivity is located within the application, as opposed to outside (in the OS, in the management system, ...), resulting in increased complexity of the resulting code and sensitivity to errors.

manageability of the application eco-system: an environment of self-adaptive software may be more difficult to manage centrally, since its decision processes are local.

Instead, we wish to render applications *reactively adaptive*, so that they change behaviour in response to external stimuli.

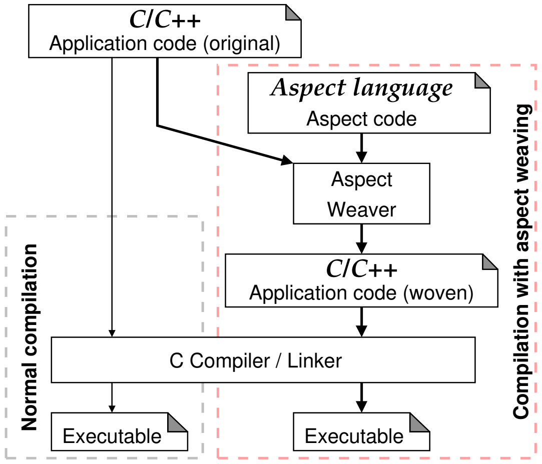

Fig. 4. Interposition of an aspect weaver

C. AoP primer

“Aspects” describe cross-cutting concerns of a program, i.e., properties, that are not localised to a specific part (class, module, source file, . . .) of the code, but that concern the application as a whole.

Examples include logging and security: all parts of the source code might at some point contain output for information or debug purposes, and they may be concerned with implementing common security practises (e.g., checking buffer sizes or pointers in C programs). Instead of cluttering the application logic with these functions, they can be formulated as aspects: “output a log message every time a file is opened” or “check pointers after memory allocation”. The definition of an aspect is separate from the application logic and is applied by an *aspect compiler*, or *weaver*, in a process called *code weaving*. For our experiments, we used the AspectC++ weaver [10]. Fig. 4 juxtaposes the common compilation process with one where the weaver is interposed to allow the introduction of aspect code into the original application. To formulate aspect code, several concepts are typically supported:

pointcut – specification of locations in the original code: the cross-*cutting* concern is inserted at certain *points* in the original code, that are specified in a *pointcut* expression, e.g. by string matching.

advice – effect of the aspect code: Having identified the points in the original code, that are to be altered, we specify the manner, in which it is to be modified in an *advice* expression.

The environment of the original code: The code to be added within an aspect often needs to refer to the values of variables, formal parameters or functions in the original code.

IV. Proof-of-concept

```
1  void write_file(long count) {
     int sbuf = strlen(chunk);
     int file = open("foo.txt",
                  O_CREAT | O_TRUNC | O_WRONLY,
5                 S_IRUSR | S_IWUSR);
     int c = 0;
     while (c++ < count)
       write(file, (void*)chunk, (size_t)sbuf);
   }
10
   int write_udp(const char* targetip,
                 unsigned int port,
                 long count) {
     int transmit_socket = socket(AF_INET,
15                                SOCK_DGRAM,
                                  IPPROTO_UDP);
     connect(transmit_socket,
             (struct sockaddr*)&si_other,
             sizeof(si_other));
20   int c = 0;
     while (c++ < count)
       write(transmit_socket,
             (void*)chunk, (size_t)sbuf);
   }
```

Listing 1. Abridged application code

Listing 1 shows the salient lines of code of the application mentioned in Sect. I, with comments and sanity checks removed. It is a simple C language program which writes to a file or transmits UDP segments in reaction to a number of POSIX `write()` calls. We shall call this program “the application” in the following: it represents the original, unaltered, non-virtualization-aware source code. To demonstrate the programmatic modification of the code, we introduce buffering of the write operations in order to improve the throughput of the application when it transmits or writes long sequences of small data chunks.

Please note, that we chose this modification as an example, to prove a point. We are aware, that indiscriminate use of blocking and buffering may introduce latency issues; nevertheless, we will ignore these for the time being, for the benefit of the demonstration. The balancing of multiple aspects against each other goes beyond the scope of this paper.

A. Analysis of the code

Standard POSIX functions are employed to open the file and to create the socket, as well as for the file write and transmit operations involved. The functions called, being part of the same standard library, can be said to be homogeneous: both `open()` (Lst. 1, l. 3) and `connect()` (l. 15) yield an integer file handle, later used by the `write()` (l. 8, 22) function.

In addition, both `write()` calls are candidates for the introduction of additional buffering: while writing to a file and transmitting over the network may be different operations, they both profit by blocking of sequences of small data chunks.

B. Introducing a buffering aspect

To supplement buffering we need to control the whole lifecycle of a file stream or socket, from its creation, including the I/O operations performed on it, to its destruction. In addition, if every file and socket is to have its own buffer, we need

```
1   int buffill[MAX_FILE_HANDLES] = {0,0,0,0,0};
    char* writebuf[MAX_FILE_HANDLES];
    int fhmap[MAX_FILE_HANDLES] = {-1,-1,-1,-1,-1};

5   aspect Buffering {
      /*match POSIX open(2) and socket(2) calls*/
      pointcut openpc() = "% ...:: open(...)";
      pointcut socketpc() = "% ...:: socket(...)";
      advice call(openpc()||socketpc()) : around() {
10      tjp->proceed();
        int fh = *((int*)tjp->result());
        int myfh = _getnextfh();
        if (myfh == -1)
          return; //filehandles exhausted
15      writebuf[myfh] = (char*)malloc(bufsize);
        fhmap[myfh] = fh;
      }

      pointcut writepc() = "% ...:: write(...)";
20    advice call(writepc()) : around() {
        int fd = *((int*)tjp->arg(0));
        const void* buf =
                  *((const void**)tjp->arg(1));
        unsigned int count =
25                 *((unsigned int*)tjp->arg(2));
        int myfh = _getmyfh(fd);
        if (myfh != -1) { //managed by us?
          if ((buffill[myfh] + count) < bufsize) {
            memcpy(writebuf[myfh] + buffill[myfh],
30                 buf, count);
            buffill[myfh] += count;
          }
          else {
            write(fd, writebuf[myfh], buffill[myfh]);
35          buffill[myfh] = 0;
            tjp->proceed();
          }
          *((int*)tjp->result()) = count;
        }
40      else
          tjp->proceed();
      }
    }
```

Listing 2. Aspect code for introducing buffering

to allocate buffer memory and associate it with the file or socket. We hold a (limited) array of file handles (l. 3), that are associated with buffers (l. 2) and the amount of valid data (the buffer's fill level) present in each of the buffers (l. 1) by index. In addition, we specify helper functions (unlisted, for brevity) `_getnextfh()` to retrieve a common array index that is unused, and `_getmyfh(`*file handle*`)` to match a given file handle to its array index.

Relying on these data structures, we proceed to identify the positions in the original source code, where the opening occurs, in order to modify them to incorporate buffering.

c) Identifying the targets of modification: We formulate *pointcut* expressions, shown in Lst. 2, that match the relevant function calls `open(...)` and `socket(...)` (l. 7 and 8, respectively) and `write(...)` calls (l. 19). We also specified pointcuts for `close(...)` and `flush(...)` calls (not shown in Lst. 2), to guard the border cases when a file/socket is closed while data remains in the buffer we introduced.

Having selected the candidates for modification, we now formulate advice blocks to specify the modification to be applied to each of them.

d) Specifying the manner of modification: Whenever a `open(...)` or `socket(...)` call is encountered, the advice code (beginning at l. 9) changes the behaviour of the resulting code, if the original call returns a valid file handle. This is effected by the following operations:

1) retrieve an available index for the set of arrays (l. 12)
2) allocate buffer memory for the socket or file (l. 15), and
3) register the file handle at that array index (l. 16).

Whenever a `write(...)` call is encountered, the associated advice code (beginning at l. 20) first analyses the parameters of the call:

1) retrieve the file handle used, to check whether we have introduced buffering for it (l. 21, 26, 27)
2) retrieve the payload and payload size given in the call (l. 22–25)

Then, it weaves our buffering policy into the original code:

3) if room (l. 28), copy the payload to our buffer associated with the file handle (l. 29, 30), but do not execute a `write()` call
4) if the buffer is full, write out and reset the buffer (l. 34, 35), then allow the current `write()` call to proceed (l. 36)
5) change the return value conveyed to the application to reflect the size of the payload in the current call (l. 8)

e) Resulting change in behaviour: The changes applied to opening and writing to a socket can be summarised as follows:

1) When a socket is created in the original code, a buffer is allocated and associated with the socket.
2) Each time a socket is written to in the original code, the payload is stored in the buffer, instead.
3) When the buffer is full, it is written to the socket stream.
4) When the socket is closed in the original code, any remaining data in the buffer is written to the socket before closing it.

C. Observing the change in behaviour

We applied the aspect code to the original application and we produced executables from both the woven code and the original code, precisely as shown in Fig. 4. We proceeded to compare the throughput achieved by the two variants of the application and varied both the size of the chunks passed to `write()` calls and, for the modified application, the size of the buffer employed. We transmitted 10 MBytes between two VMs distributed by a (non-choked) physical network, and used the application variants as the sending end-point.

The results are shown in Fig. 5, where the upper surface shows the measurements made with the unmodified application, and the lower surface those with the woven application variant.

We observe a significant dependency of the throughput on the size of the chunks sent by the unmodified application (note

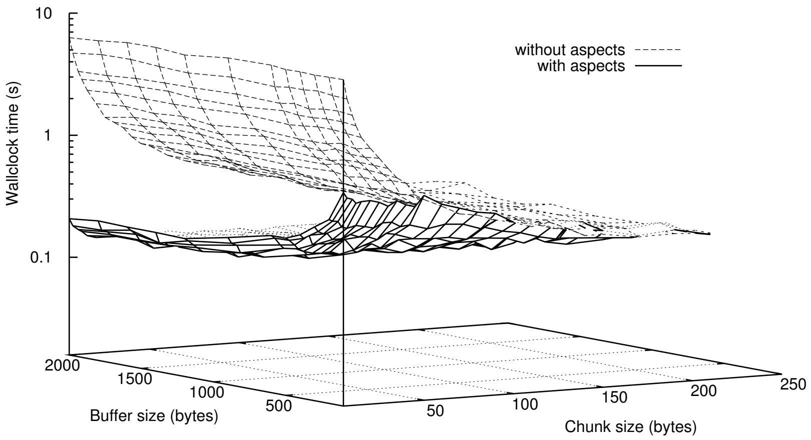

Fig. 5. Duration of 10^7 byte throughput using varying chunk sizes between VMs on different physical hosts

the logarithmic time scale). We observe the same effect with the woven application, for small buffer sizes, i.e., when the buffering is less aggressive, in particular for small chunk sizes. This effect is indicated by the slope of the lower surface, that is higher at the lower values juncture between the chunk size and buffer size scales.

As chunk sizes increase, the benefits of buffering become less significant, indicated by the surfaces converging. In one instance (for high chunk and low buffer size), the throughput of the original application becomes higher than that of the woven variant.

V. Discussion

When adapting an application with aspects to make it virtualization aware, the following questions arise:

1) What to adapt, i.e., what behaviour to modify, under which circumstances?
2) In what manner should the behaviour be modified?
3) What information is required to determine the target circumstances?
4) How to locate the source of the behaviour in the application's code for defining the pointcuts?
5) What to substitute to achieve the desired effect, i.e., what to put in the advice blocks?

A. Determining target behaviour

Changes in the application's environment will affect rather different functional and non-functional performance indicators of the application. In order to enable an application to react appropriately to such changes in the virtualization environment, different types of information need to be available.

As demonstrated before, a channel's throughput, latency, or drop rate will be influenced by the effective distance between two VMs. Such effects can be countered by introducing buffering, the usage of compression, or, conversely, by limiting the transmission rate. The decision, how to react, depends on the physical location of the involved VMs as well as their physical connection. In the same vein, certain computational tasks may suffer from a lack of computational resources, like the effectively available CPU or memory; here, decreasing the precision of the computation or the limitation of thread pools can mitigate the environmental changes. On the other hand, when several VMs are co-located on a single host it may become advisable to scrub sensitive information before freeing memory in order to avoid page snooping.

B. Adaptation strategy

A particular behaviour of the application may thus be detrimental, or at least influential, to the application's functionality or performance in a particular situation of the virtualization environment. Let us assume that the effective detection of such a situation is indeed possible using the available information sources (see Sect. VI). In order to react to the occurrence of the environmental situation of discourse, it is still necessary to locate the causes of the particular behaviour in the source code of the application and to cast these causes into pointcuts for the aspects—which, in general, will be a complex and difficult task. On the other hand, it has to be decided how to react to the change in the virtualization environment and how to integrate the reaction into advice code.

Ideally, the overall software architecture of the application with its components and their relations can provide the necessary guiding clues where to look. For example, when suspecting that the throughput on a channel in a distributed application may suffer from re-locations of VMs, the communication structure expressed in the software's architecture is an obvious source of information. On the source code level, however, the manifestation of the (mis-)behaviour's causes consists in various idioms, i.e., groups of program statements, which have to be identified and altered in order to introduce the intended reaction. For the throughput example, the encoding of writing to the possibly affected channel will be relevant and thus be used for a pointcut. But not all occurrences of these idioms in the source code will pertain to the particular behaviour, and

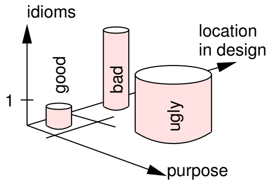

Fig. 6. AoP affinity depending on the variety of idioms, purposes and location in an application's design.

the non-relevant ones have to be excluded using contextual information. To wit, for a particular channel the idioms for establishing and closing the channel have to be included in the context of the writing idiom leading to additional contextual pointcuts, as the managing of the file handles in Sect. IV.

Finally, the complementing code to implement the reaction to the occurrence of a particular virtualization environment depends quite heavily on which effect should be achieved and what the application itself offers for the adaptation. In simple cases merely some internal parameters need to be tuned, e.g. the buffer size exemplified in Sect. IV. More complex adaptations could range from the de-/activation of some of the application's internal functions, e.g., employing compression of data to be transmitted, to influencing user-facing functions, e.g., when access to a cryptographic key is requested by the user. In any case, the contextual information influencing whether the modification shall be applied has to be managed. Besides the aspects for handling the behavioural modification proper, additional contextual aspects are needed that apply for the contextual pointcuts and record in their advice code all information that is needed to decide whether the behaviour has to be adapted. Furthermore, when several modifying behaviours are available (typically including the application's default behaviour), some cheap selection strategy depending on the virtualization environment and the contextual information should be provided in the modifying aspect.

C. Limitations

The availability of information on the state of the infrastructure is a hard prerequisite of our approach.

Due to the properties of our chosen technique, aspect-oriented programming, our approach is additionally sensitive to the quality of the application's design and implementation.

In particular, pointcuts are specified with respect to syntax. To achieve pointcuts that are sufficiently selective and precise requires certain minimum standards on code quality. Consider for example a program, that uses multiple idioms to open files for writing: it might use the functions specified by the POSIX standard (e.g., `open` or `creat`) or the standard C functions (e.g., `fopen`). Pointcut declarations that target file opening need not only take into account the different function names, but also their different parameter types: if open files need to be tracked (shown in our example, Lst. 2), the tracking would require management of both file handles (integers) and `struct FILE` records. This would require several pointcuts, with their own advice blocks to achieve the same goal, but for different idioms. Hence, consistent use of a single idiom for the statements targeted by the aspect is advantageous, but cannot be assumed in every body of code.

Another metric for code quality is determined by the "low coupling, high cohesion" principle [8] that dictates to minimize dependencies stretching over large areas of the code. In code, that violates this principle, the scope for the application of pointcut expressions becomes of necessity very large; thus, unmanageable or, at least, error prone.

Figure 6 illustrates the variety of idioms employed in the code for a given purpose and the distribution of idioms and purpose within the component parts of an application. Code with a high localisation of purpose, using only a single idiom for expressing that purpose is an ideal candidate for AoP techniques, to achieve adaptation.

Thus, an effective use of AoP in any application domain is dependent on the quality of the application's code and design. Applications, that are problematic in this respect may instead profit from the other techniques mentioned in Sect. II, e.g., replacement of libraries or re-factoring the application's code.

VI. Architectural sketch

In Sect. IV we described non-adaptive modifications, validated by measurements of effects observed on VMs in a fixed environment. When the environment changes while the application is running, the fixed configuration values (e.g. buffer size) may no longer be suitable. For example, if the VMs hosting the sending application and the receiver are migrated to the same physical host, the use of additional buffering may introduce only additional latency, but no benefit in throughput.

Therefore, the aspects woven into existing application code need to acquire and leverage knowledge about the changes in the virtualized infrastructure. Such knowledge is typically available to the management systems governing the setup of the infrastructure. What is more, the deliberate changes introduced are not only known, but planned and executed—i.e., known before they occur—by the management system.

A. Making use of virtualization management knowledge

Given a source of management knowledge available to application code, we could devise aspect code that not only changes the behaviour of the application, but that adapts it to a current (or soon-to-be) state of the virtualized infrastructure. While commonly, such information is not available to applications, it could be made available in the same manner in which they can access certain knowledge about their environment held and made available by the (guest) operating system they are run on (the time and date, the amount of memory available to the machine, the availability of network access, etc).

B. Requirements of the application

Environment information is made available by an operating system by means of standard system calls, e.g., gethostbyname(*DNS name*) to resolve a host name into an IP address, or time() to access the system clock. In the same manner, we could ask: is_co_located(*communication partner*). Depending on the result of the call, the aspect code woven into the application could decide which measures to take, if any, and which parameters to choose. We propose to employ the same kind of interface in order to allow application code to gain knowledge about the virtualized environment.

The relevance of a given change to the infrastructure is highly dependent on the purpose and behaviour of an application. To convey only relevant knowledge, a publish/subscribe pattern seems suitable: each application subscribes only to management events, that are relevant for itself.

C. Obligations of the application

Given such an approach, aspect code should detect the existence of the system event interface and become inert if detection fails. In case detection is successful, it subscribes to particular types of notification, that are of interest for the application instance. This may entail subscriptions to notifications pertaining to certain managed objects currently relevant to the application, e.g. "notify me, if interface with IP address a.b.c.d changes position". On receiving a notification, the aspect code adapts the behaviour of the application to the new situation, e.g., changes the buffer sizes for a communication channel.

To illustrate a possible way of achieving this, we sketch an architecture capable of supplying selected management knowledge to applications in the following.

D. Architectural components

The requirements and obligations of the woven application describe only a need for information and a protocol between the application and the operating system code. In Fig. 7, that illustrates the envisioned architecture, these elements are drawn in black. They include an event service, that implements a publish/subscribe pattern offered by means of an OS interface. The aspect code subscribes to certain events types offered by the event service and is notified, when their instances occur.

The origin of the required information may well lie outside the guest operating system. The remainder of the architectural sketch, drawn in grey, illustrates a possible source of notifications, envisioned for an open, managed infrastructure. In the presence of a management system, information can be drawn from its knowledge base, perhaps filtered according to some management policy. The management system acquires the knowledge by recording its own decisions and management actions and by observing the virtualized infrastructure (hypervisor software and physical machines, as shown in the diagram, are only a subset of the infrastructure). The figure depicts the interaction between management system and event service as adhering to a publish/subscribe pattern, as an example. A polling pattern might be substituted at this point.

Alternative sources: Other sources of knowledge might entail gathering information from the underlying hypervisor(s), techniques for collaborative discovery among the guest operating system instances supporting an application's distributed components, or even configuration files created and updated by the guest OS administrator. In any case, the event service within the guest OS acts as a proxy to information offered by such sources, while at the same time providing OS specific but unified subscription services to all applications.

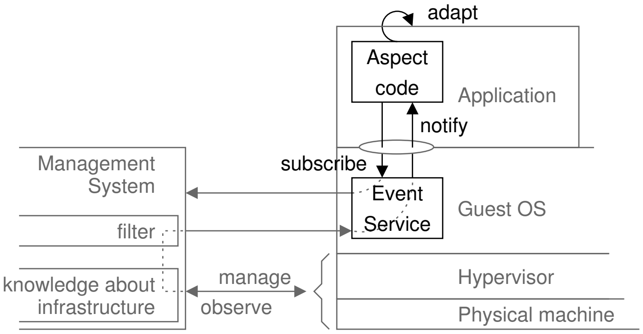

Fig. 7. A simple software architecture, to provide VMs with knowledge about their environment

VII. Conclusions and future work

We observed significant behavioural differences in the same VM-based application when varying its communication channel within the virtualized infrastructure. In response, we introduced buffering as an adaptation of behaviour, using AoP. To address the need for information to guide behavioural change, we postulated an architecture that assumes a permissive management system. From the discussion of limitations and requirements of this method we conclude, that such an approach seems plausible for larger, complex applications. However, a substantial amount of further research work is necessary. Apart from obvious next steps, such as experimentation with event-driven adaptation, we offer in conclusion the following endeavours as being worthy of attention:

- Detect and resolve conflicts between aspects (e.g., between throughput and latency optimisations)
- Identify techniques to be incorporated in the adaptive code (buffering, change in communication patterns, execution of security functions in response to location, etc.)
- Explore the pro-active support for pre-existing design patterns in applications, e.g. supporting a *chain-of-responsibility* pattern [6] by optimising communication paths between the participating software components.
- Classify applications with respect to their specific adaptation needs, to derive adaptation patterns
- Complement the event service with a query API to allow on-demand acquisition of information
- Realise distributed discovery techniques, as knowledge source alternatives

Acknowledgment

The authors wish to thank the members of the Munich Network Management Team (MNM Team) for helpful discussions and valuable comments on previous versions of this paper. The MNM Team directed by Prof. Dr. Dieter Kranzlmüller and Prof. Dr. Heinz-Gerd Hegering is a group of researchers at Ludwig-Maximilians-Universität München, Technische Universität München, the University of the Federal Armed Forces and the Leibniz Supercomputing Centre of the Bavarian Academy of Science.

References

[1] M. Baldauf, S. Dustdar, and F. Rosenberg. A Survey on Context-aware Systems. *Int. J. Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007.

[2] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Laypool Publ., 2012.

[3] K. Czarnecki and U. Eisenecker. *Generative Programming*. Addison-Wesley, 2000.

[4] A. Danylenko and W. Löwe. Adaptation of Legacy Codes to Context-Aware Composition Using Aspect-Oriented Programming. In T. Gschwind, F. D. Paoli, V. Gruhn, and M. Book, editors, *Proc. 11th Int. Conf. Software Composition (SC'12)*, volume 7306 of *Lect. Notes Comp. Sci.*, pages 68–85. Springer, 2012.

[5] J.-M. Favre. Preprocessors from an Abstract Point of View. In *Proc. 1996 Int. Conf. Software Maintenance (ICSM'96)*, pages 329–339. IEEE, 2006.

[6] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[7] H. Hafiddi, H. Baidouri, M. Nassar, and A. Kriouile. An Aspect Based Pattern for Context-Awareness of Services. *Int. J. Computer Science and Network Security*, 12(1):71–78, 2012.

[8] G. J. Myers. *Reliable Software through Composite Design*. Mason and Lipscomb Publ., 1974.

[9] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), Sept. 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[10] O. Spinczyk, D. Lohmann, and M. Urban. AspectC++: An AOP Extension for C++. *Software Developer's J.*, 5:68–76, 2005.

[11] É. Tanter, K. Gybels, M. Denker, and A. Bergel. Context-Aware Aspects. In W. Löwe and M. Südholt, editors, *Proc 5th Int. Conf. Software Composition (SC'06)*, volume 4089 of *Lect. Notes Comp. Sci.*, pages 227–242. Springer, 2006.