Architecture-Oriented Visualization

Mohlalefi Sefika

Department of Mathematics and Computer Science
National University of Lesotho
P. O. Roma 180, Lesotho
Southern Africa
sefika@macs.nul.ls

Aamod Sane and Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
(sane,roy)@cs.uiuc.edu
http://choices.cs.uiuc.edu

Abstract

Tracking the changing dynamics of object-oriented frameworks[5], design patterns[7], architectural styles[8], and subsystems during the development and reuse cycle can aid producing complex systems. Unfortunately, current object-oriented programming tools are relatively oblivious to the rich architectural abstractions in a system.

This paper shows that architecture-oriented visualization, the graphical presentation of system statics and dynamics in terms of its architectural abstractions, is highly beneficial in designing complex systems. In addition, the paper presents architecture-aware instrumentation, a new technique for building efficient on-line instrumentation to support architectural queries. We demonstrate the effectiveness and performance of the scheme with case studies in the design of the *Choices* object-oriented operating system.

1 Introduction

Designers conceive complex systems as architectures with design patterns[7], frameworks[5], architectural styles[8], and subsystems. As the system evolves, they customize these large-grained components, tune performance, and make reuse decisions. Such complex tasks require understanding component behavior, tracking resource usage, and detecting integration problems. Program instrumentation and visualization tools that support analysis directly in terms of higher-level, architectural abstractions would markedly simplify these activities.

Most conventional object-oriented software analysis tools, however, provide dynamic inspection primarily in terms of lower-level units like classes, instances, and methods[19, 20, 14, 15, 17, 12]. Many, rich architectural abstractions with a granularity that is larger than an object or class are opaque to such tools. Recent research[24, 5, 7] demonstrates the value of gross architectural structures in the design, evaluation, and reuse of complex systems. In our own studies of the customization of the Choices[4] object-oriented operating system, we often needed answers to questions like:

- How many times does this application process visit the file system?
- Is this lock currently in use by any subframework of the virtual memory system?
- Which subframework of the virtual memory system communicates most frequently with the file system?

It is difficult to answer such architecture-level questions using traditional programming tools for several reasons. First, flat "method-level" instrumentation generates too much data and excessively perturbs the system. Second, the more abstract features of the software architecture of the system are opaque to instrumentation, making it hard to identify and control instruments associated with specific architectural entities. Third, architectural investigations often involve system analysis at multiple levels of abstraction and different aspects of design. It is arduous to track and correlate such information with current instrumentation systems as they seldom support multiple perspectives or aid in hierarchical system navigation.

This research demonstrates that *architecture-oriented visualization*, the graphical presentation of system statics and dynamics in terms of its architectural abstractions, makes it considerably easier to understand and customize complex systems. Architecture-oriented visualization permits the same logical components that structure the overall system design to also serve as the fundamental units of animation, offering scalable hierarchical inspection and multiple code perspectives. We demonstrate the utility of our visualization technique with real-world case studies from the *Choices* operating system.

To address the deficiencies of traditional instrumentation, we introduce *architecture-aware instrumentation*. A distinctive feature of architecture-aware instrumentation is that it explicitly represents architectural structures and substructures in a running system, and exploits this knowledge to optimize instrumentation. We present performance data that shows that an architecture-aware instrumentation dramatically reduces trace data size and introduces far less overhead than traditional, unstructured instrumentation.

2 Overview

Architecture-oriented visualization enables the analysis of an object-oriented system in terms of its conceptual organization. Object-oriented systems are aggregates comprised of subsystems, frameworks, patterns, classes, and objects. For example, the *Choices* operating system is an assembly of subsystems like *Process* or *Virtual Memory*, where each subsystem is further subdivided into frameworks like the *Physical Memory* or *Logical Memory* framework. In turn, each framework is organized as a collection of design patterns, class hierarchies, and classes. With architecture-oriented visualization, we can inspect system dynamics structured in terms of these architectural components. We can observe architecture-level properties like "visits to the virtual memory subsystem by a process", where we wish to observe interactions between the architectural units 'process' and 'virtual memory subsystem'.

We begin the paper with case studies that show that architecture-based analysis arises naturally when working with large, complex systems.

Our first case study (Section 3) demonstrates how hierarchical and recursive visualization of the *Choices* system can identify the cause of a system-wide performance bottleneck that reduces CPU and disk utilization. To investigate the problem, we begin by querying statistics at the system level. We identify the virtual memory subsystem as a candidate for further examination. We then trace the internal details of this subsystem by zooming into possible problematic frameworks, class hierarchies, classes, and object instances.

The second study (Section 3.1) concerns a problem in subsystem cohesion and coupling that showed up during a routine check that we apply after every major change in system design[23, 21]. The check tracks subsystem interactions at the subsystem-, framework-, class-, and method-levels and reveals significant architectural changes that could indicate flaws in the modifications. The check is essentially a conformance test and is feasible because architecture-oriented visualization simplifies the process. Before we developed our visualization tools, we would often be unaware of any architectural consistency problems until they affected correctness. It is difficult and time consuming to identify such problems by hand.

In each study, we use information from higher levels to select lower-level components for further study. The system offers many graphical presentations, each suitable for viewing behavior at some abstraction level. When we want aggregate, relative statistics, we use diagrams like bar charts and *ternary diagrams*[9]. When the information must be correlated to the system structure, we use *space-filling diagrams*[2] that make it easy to relate the run-time statistics to the relevant system parts. When we have to display pair-wise component interaction, we use *affinity diagrams*[23] and *object interaction diagrams*[3]. At each point, the user may choose from a palette of possible diagrams. Our system also supports multiple simultaneous diagrams and combined static and dynamic perspectives. Moreover, the diagrams are hyperlinked to one another so that information displayed on the screen can be easily used to guide system navigation.

The queries used in the case studies motivate the design of architecture-aware instrumentation (Section 4). Architecture-aware instrumentation is cognizant of the software architecture of the system. The instruments gather data in a task-specific manner so that data processing and manipulation is minimized. Moreover, the instrumentation run-time exploits knowledge about the architectural units of the system to condense data right where it originates. Our performance studies (Section 5) exhibit the dramatic difference between instrumentation that optimizes data collection on a per-architectural model basis versus one that does not.

In the paper, we describe parts of the instrumentation architecture in terms of the design patterns used in building the system. We also present a simple query language (Section 4.5) for the architectural queries supported by the instrumentation.

3 Case Study 1: Performance debugging

This case study shows how architecture-oriented visualization helps in identifying subtle performance bottlenecks starting from problem symptoms that are manifest at the system level.

In one version of the *Choices* system, we discovered that CPU utilization was reduced by 8% and disk utilization by 14%. Many parts of the system had been changed between

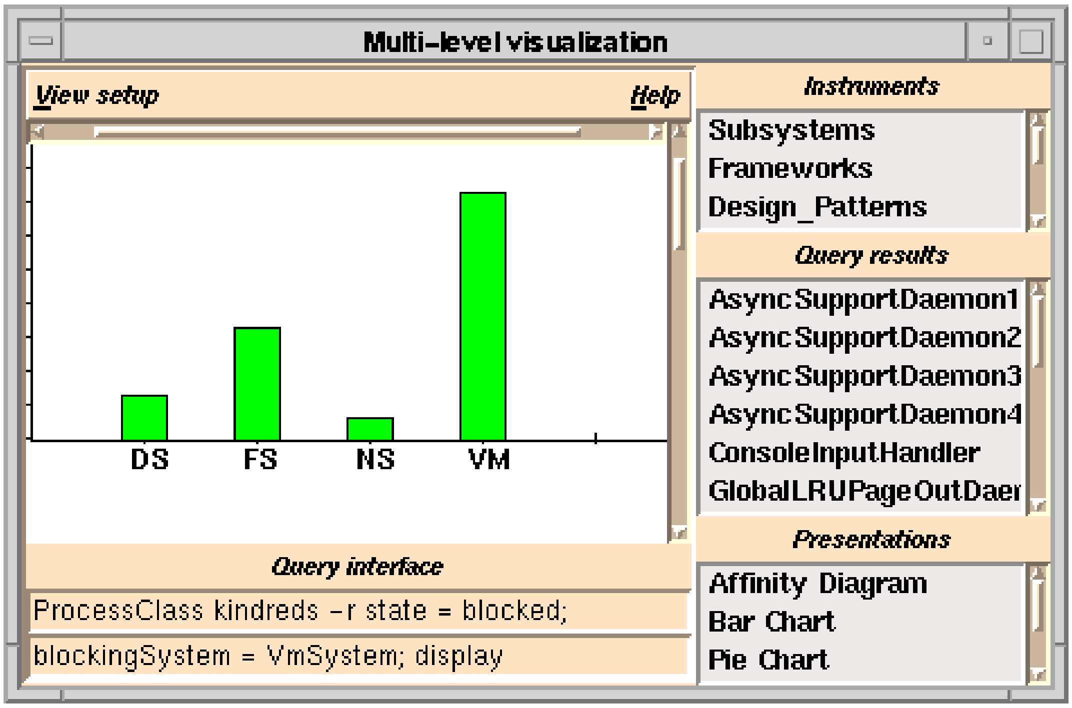

Figure 1: Animated process locking activity in four subsystems. The length of a bar changes to reflect the number of processes currently blocked in the corresponding subsystem.

versions, so the cause of the degradation was unknown. In examining process statistics, we discovered that the average time for which both system and application processes were blocked had also increased. Consequently, one possibility was a synchronization problem somewhere in the system. We report the steps in our investigation below.

- *Subsystem*: We first investigated relative process blocking across four *Choices* subsystems: the naming system (NS), file system (FS), devices system (DS), and virtual memory system (VM). These subsystems had been altered in producing the latest version.

  Figure 1 depicts processes that are blocked in each subsystem in a bar chart. The length of each bar changes to reflect the number of processes currently blocked in the corresponding subsystem. The diagram can also display a running numerical average. From the figure, it appears that on the average, the virtual memory system blocks the largest number of processes.

- *Framework*: Now we want to see which internal subparts of the virtual memory subsystem block the processes. We select the VM bar from Figure 1 to get the space-filling diagram[2] of Figure 2.

  The diagram depicts the virtual memory subsystem frameworks and their class hierarchy components, with filled rectangles indicating the relative number of processes blocked. The class hierarchies are represented by rectangles whose size is proportional to the number of classes in the corresponding hierarchy. A rectangle label shows the most abstract class in the hierarchy. We observe five frameworks: address space management (F1), logical memory (F2), address translation (F3), physical memory management (F4), and caching support (F5). From the figure, it is immediately obvious that the *MemoryObject* class hierarchy of the logical memory framework is currently blocking most processes.

- *Class Hierarchy*: Clicking on the *MemoryObject* region displays Figure 3. This space filling diagram shows some classes in the *MemoryObject* hierarchy separated by thick black lines. The inner squares depict per-class instances. Filled squares indicate the specific instances that are blocking the processes. The figure reveals that instances of *MoView* (*MemoryObjectView*) are blocking most processes.

A *MemoryObjectView* denotes a contiguous set of memory locations or region within a memory object and allows the data in the physical pages of that region to be pinned in place so that they cannot be paged out. The I/O system uses the *MemoryObjectView* to move data between a disk and a memory object. The framework supporting caching had been changed to use disk I/O in units of 64 KB. However, the *MemoryObjectViews* still supported I/O

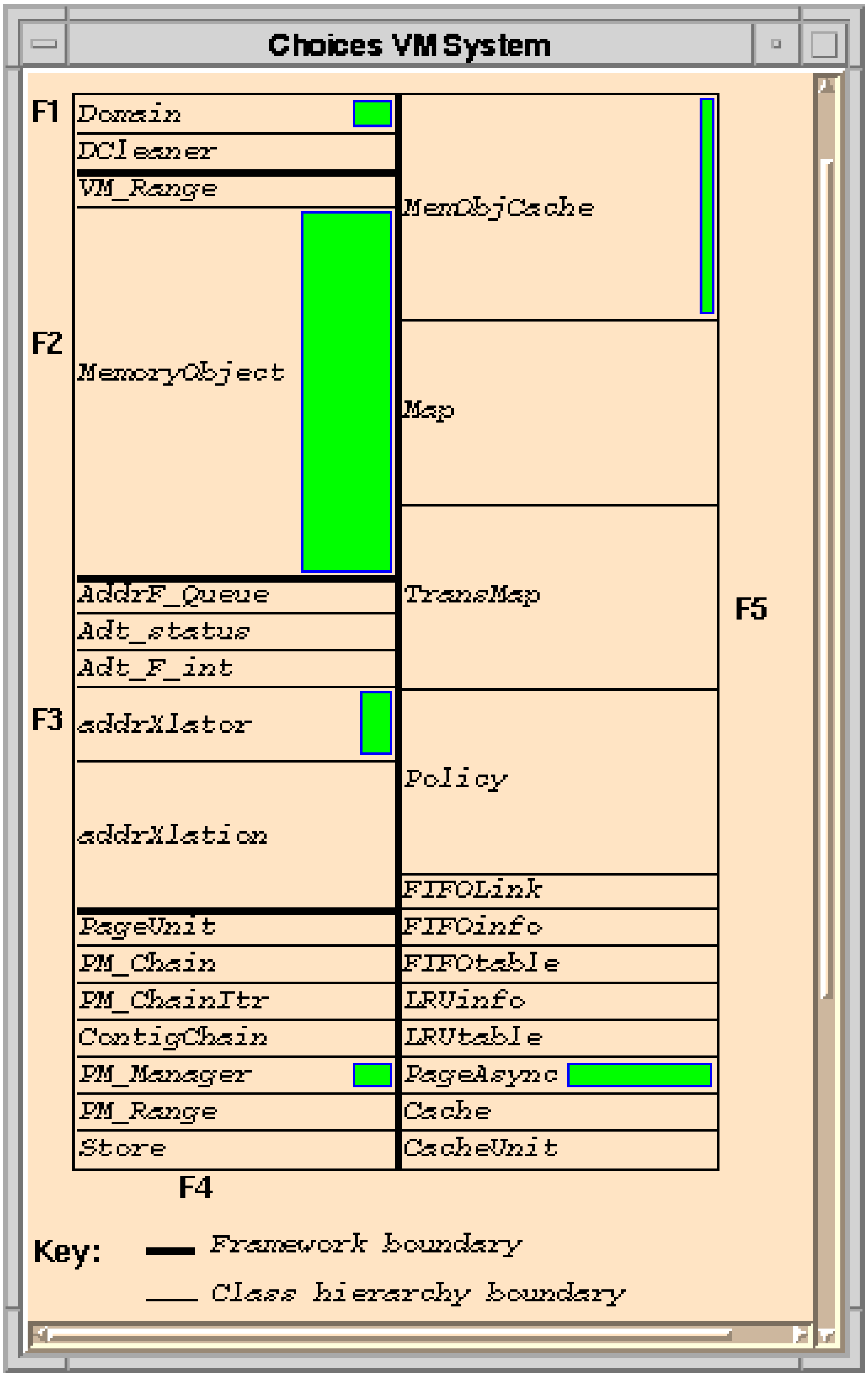

Figure 2: A zoomed view of the virtual memory system. The diagram reports process blocking statistics in terms of subframeworks, inheritance structures, and classes, making it straightforward to relate the statistics to the relevant system components.

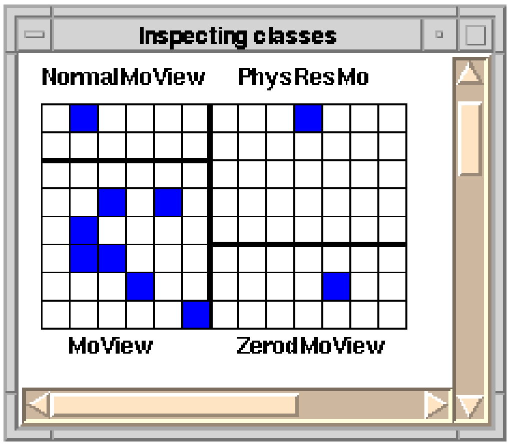

Figure 3: A zoomed view of the MemoryObject inheritance structure. The diagram highlights the particular classes and instances where processes are blocked.

in terms of 4KB pages. This mismatch led to a fragmentation of I/O across numerous small *MemoryObjectViews*, so processes waited for a long time to complete their I/O. Modifying *MemoryObjectView* code to use the larger sizes reduced process blocking and improved CPU utilization by 5% and disk utilization by 10%.

Discussion The above case study shows how architecture-oriented inspection helps navigate a complex system to uncover subtle performance bottlenecks. We used both system-level views and lower-level views to discover *indirect* side-effects in the logical memory framework caused by customizing the neighboring caching support framework. Architecture-aware instrumentation enabled us to track the dynamics of both coarse grain and fine grain components until we identified one major cause of system-wide performance degradation.

3.1 Case Study 2: Evaluating design properties

Architecture-oriented visualization makes it easier to check complex interactions of components and subsystems. We exploit this facility to routinely check the design properties of *Choices* after every version[23, 21]. This case study considers an instance where we recognized and repaired undesirable subsystem couplings during our system evaluation.

- *Subsystem*: The ternary diagram[9] in Figure 4 portrays relative communication between four device driver frameworks and three subsystems: the naming system, the remote procedure call (RPC) system, and the device driver system. The diagram helps quickly compare the degree of interaction among each framework and the three subsystems. A Framework is animated as a moving circle. During the animation, the

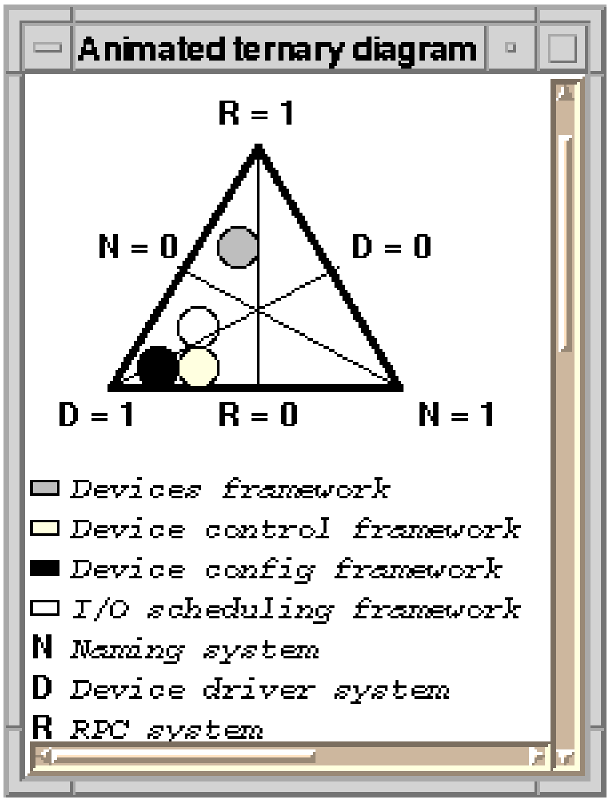

Figure 4: A ternary diagram animating the degree of communication between device driver subframeworks and three subsystems.

framework drifts towards the subsystem(s) it is currently communicating heavily with. From the snapshot, we observe that the *Device config* framework communicates intensively with the devices system, with communication ratio nearly 1. This gives a high degree of confidence that the *Device config* framework interacts most with its own subsystem components. But the *Devices* framework communicates more with the remote procedure call (RPC) subsystem than with its own *Device Driver* subsystem. This coupling is probably undesirable.

- *Framework Details*: To examine the details of the suspicious communication, we click on the *Devices* framework circle and the *RPC* system axis to generate an affinity diagram[23] that shows how the framework and the subsystem components interact. In Figure 5, classes attract each other if they communicate frequently, otherwise they repel. The higher the communication between two classes, the thicker their connecting lines. The line arrows indicate the direction of calls. For the most part, the framework and the subsystem appear to be cohesive: intra-component interaction is much greater than inter-component interaction. However, Disk and RpcBuffer exhibit anomalous, heavy communication.

- *Class Interactions*: To understand the unusual interaction, we click on the thick arrow, generating the object interaction diagram[3] of Figure 6. This diagram portrays a trace of method calls between instances of Disk and RpcBuffer. The diagram statistics indicate that the method getState() is called with high frequency on the Disk object.

Code inspection revealed that getState() was initially used for debugging RPC data buffering to and from disk (for example, for long-lived client/server connections exchanging very large data). Therefore, some RPC related state was kept in Disk. Unfortunately, this debugging related implementation decision perpetuated in the final RPC system implementation, creating undesirable system-level dependencies. We removed the erroneous coupling by factoring the state properly to the RpcBuffer. Figure 7 shows the ternary diagram animation that results after correcting our design flaw. Now all the device subsystem frameworks communicate mostly with device subsystem components, giving a high degree of confidence that the subsystem maintains high cohesion and low coupling.

4 The design of architecture-aware instrumentation

We need architecture-aware instrumentation to support various dynamic analysis tasks as exemplified in our case studies. The requirements for the instrumentation system are driven by the information demanded by both low-level and architecture-level queries. We will derive these requirements by referring to the needs of the visualization experiments in the last section.

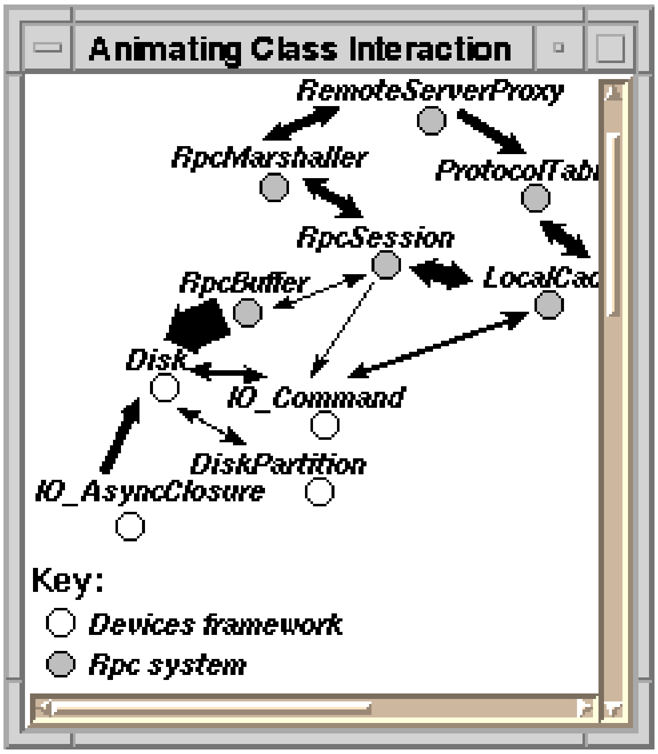

Figure 5: An affinity diagram animating the interaction between the Device subframework and key classes of the RPC system.

Consider Figure 4 where we examine the dynamic interaction between selected frameworks and subsystems. To capture such interactions efficiently, the instrumentation system must activate only those specific *Choices* instruments dedicated to the objects of the target frameworks and subsystems. Thus, out of a very large number of instruments in the operating system, the instrumentation system must be able to name and identify those particular instruments belonging to the few subsystems and frameworks of interest. This leads to our first requirement:

- **Requirement 1:** *The instrumentation system must know how instruments are associated with architectural components.*

Next, observe the affinity diagram of Figure 5, where we narrow our focus of attention to the *Devices* framework and the *RPC* system interactions. At this point, the instruments belonging to the naming system and the other frameworks must be deactivated to reduce unnecessary system perturbation. Thus we have:

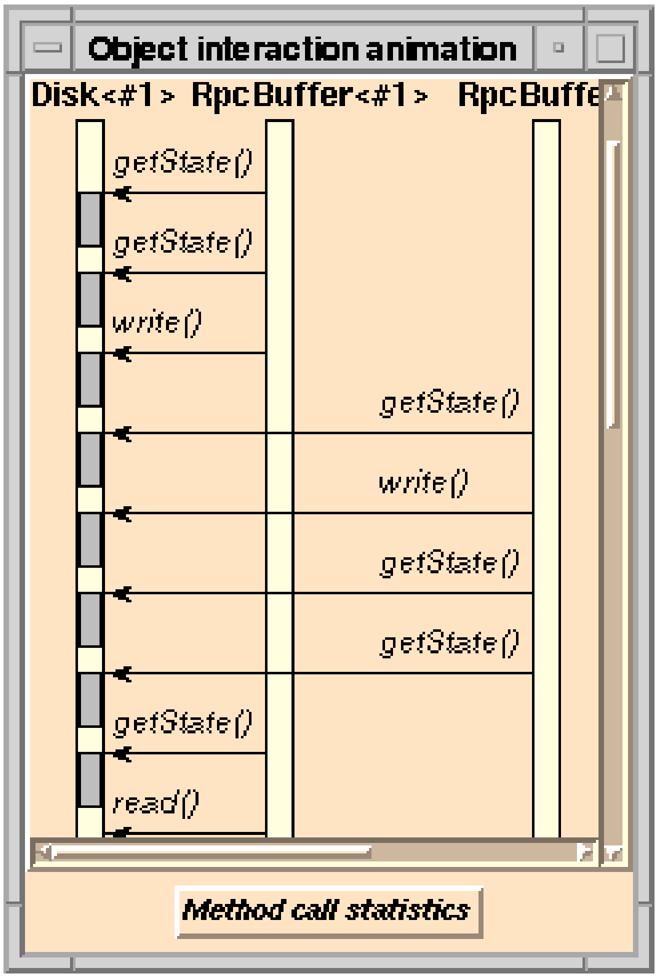

Figure 6: Animated object interactions among Disk and RpcBuffer objects.

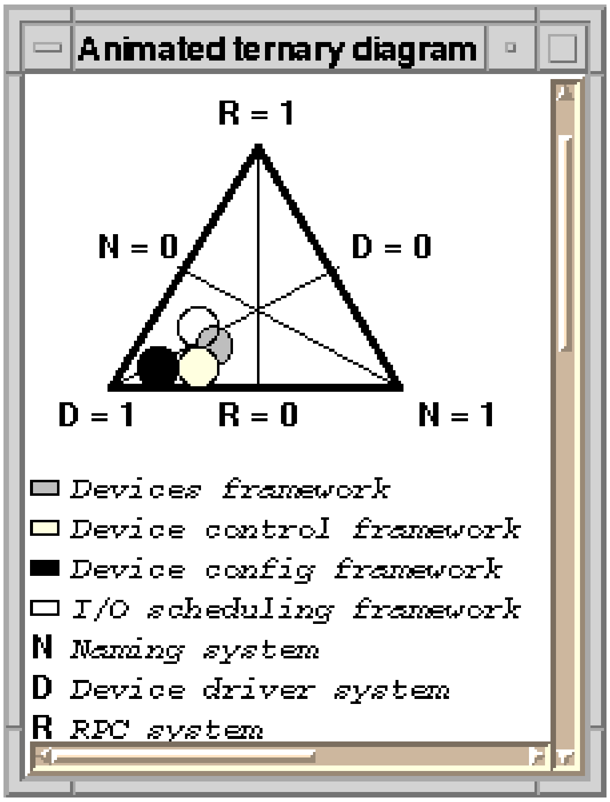

Figure 7: A typical ternary diagram animation after debugging the unwanted couplings.

- **Requirement 2:** *The system must support dynamic control of groups of instruments associated with architectural units.*

Going back to the ternary diagram in Figure 4, to place the framework circles, we have to count the number of calls between specific frameworks and subsystems. We certainly cannot afford to send data about every call to the visualizer. Instead, we want to accumulate the number of calls in *one* running counter per pair of framework and subsystem. In general, we require that:

- **Requirement 3:** *The system must accumulate aggregate statistics based on the architectural models.*

In the course of our experiments, we switch between abstraction levels, selectively collecting data about classes, frameworks, or subsystems. But the events are actually detected by per-method event sensors, so depending on our level of abstraction, we may want the same sensor to update a class, framework, or subsystem instrument. Therefore,

- **Requirement 4:** *Event sensors must be dynamically bound to instruments.*

In addition, we have the generic instrumentation requirement

- **Requirement 5:** *Minimize system time and space perturbation.*

Traditional instrumentation does not have enough runtime information about the system structure to meet our requirements. For our first two requirements and their example uses, it will be probably impossible to efficiently name, select, and animate a specific group of instruments belonging to a large architectural unit like an *RPC* system. The reason is because conventional instrumentation does not have knowledge about the decomposition of the system into its architectural components. Consequently, component-driven identification or regulation of instruments would be unduly difficult in real time.

Similarly, the third requirement would be extremely difficult to meet because there are no means to identify the particular objects belonging to a given architectural unit. The fourth requirement is unnecessary unless multiple levels of abstraction are supported.

An alternative approach simply gathers all the data and then filters it to choose data packets of interest. This approach, however, entails considerable I/O traffic to send the data to the visualizer and adds more time to filter the data. It would be better suited to post-mortem processing.

4.1 The instrumentation architecture

Figure 8 shows the gross design of the architecture-aware instrumentation system. We explain the design by showing how event sensors detect interesting events and notify instruments, and how the system interprets interactive user queries to collect data or control instruments. The requirements met by each component are indicated in parentheses.

Consider a query that generates the affinity diagram of Figure 5. The user specifies the framework and subsystem to be observed, and the visualizer formulates a *query* to count inter-class method calls. The QueryInterpreter maps the framework and subsystem to the instruments (Requirements 1 and 2) and triggers the InstrumentManager (Requirement 2) to activate all the relevant instruments. The manager tells the instruments[1] to count every inter-class call (Requirement 3).

Another data path through the architecture begins with EventSensors. Event sensors are selectively activated by the InstrumentManager according to the query. They are triggered upon method entry and exit and generate *event notifications*, rather like "printf debugging" statements. The sensors report to the EventAnnouncer which directs the event notification to the instrument defined by the query (Requirement 4). The instrument will take action according to its mode, either count the call, keep time, or maintain history.

To complete query processing, data collected by the instruments must be displayed. The QueryInterpreter orders a DataCollector to collect method calls statistics for every pair of objects at regular time intervals. The DataCollector sends *data packets* back to the Visualizer.

In this architecture, the primary design decisions are:

- How should the query interpreter map architectural units to their instruments.
- How should Instrument managers control the instruments.
- How might data collectors visit instruments to collect data.
- How should events be directed to the interested instruments.

Two primary forces govern the system design. First, the system must incur low space and time overhead. Second, the system must be flexible, in that the user should be able to orthogonally change instrument organization/management and data collection strategies. We present our design components as triples of *Problem*, *Solution*, and *Consequences*. Some components are instances of well-known design patterns, while others are specific to our system.

[1] Instruments either count, keep time, or maintain history.

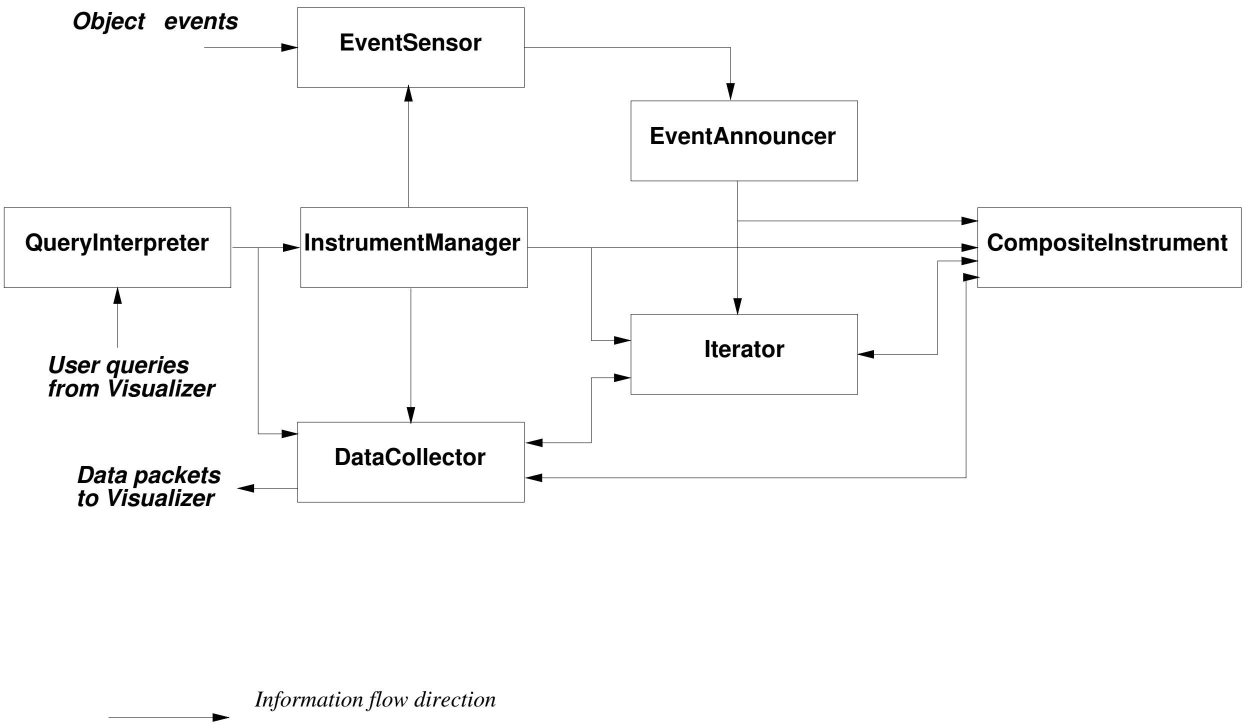

Figure 8: The components of architecture-aware instrumentation and the way they interact.

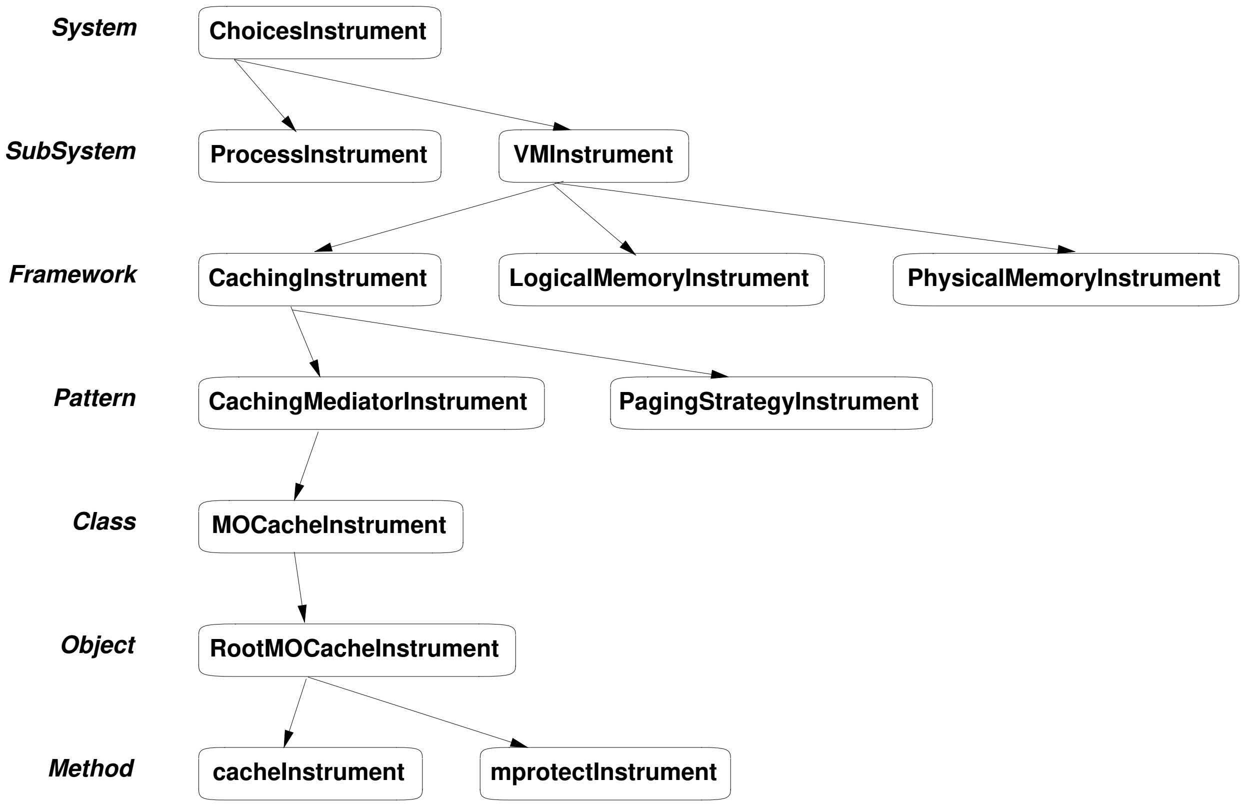

Figure 9: A possible hierarchical instrument object structure in running *Choices*.

4.2 Architecture-directed organization of instruments

• *Problem:* We want to associate system architectural units with instruments. Clients of the instruments (e.g., data collectors and instrument managers) should treat both aggregate and primitive instruments uniformly for simplicity.

• *Solution:* We represent the hierarchical architectural components of the system with a corresponding hierarchy of instruments as in Figures 9 and 10. The hierarchy is built using the COMPOSITE[7] pattern.

• *Consequences:* Using the COMPOSITE-based design, it is transparent to the client code whether it is accessing aggregate instruments like a Subsystem or primitive instruments like a Method. Thus, client programs become independent of the granularity of an instrument. In this way, the design achieves architecture-awareness without complicating the program code that uses various instrumentation levels. We also have a natural data structure to support data accumulation and organization within the instrument objects themselves – the composite structure.

Since the architectural units are directly represented in the instrument hierarchy, and since all instruments are uniformly accessible, mapping the units to their instruments is highly efficient.

A drawback is that we need run-time checks to enforce the containment constraints Subsystem > Framework > Pattern > Class > Object > Method. The respective instruments must honor this ordering.

4.3 Navigating and controlling instruments

• *Problem:* The instrument managers and data collectors have to traverse the collection of instruments to initiate, control, or terminate data collection. For fast response, traversal policies should be tailored to queries. Moreover, the traversal strategy and instrument organization must be orthogonal so that either might be changed independently. For instance, a user might want to use the same traversal strategy over multiple abstraction levels, or to rely on customized navigation policies for individual levels.

• *Solution:* We separate the responsibility for traversing the instruments in an ITERATOR[7] object. Each iterator has a specialized algorithm for traversing its collection of instruments.

• *Consequences:* Iterators know the details of the collection, but the instrument managers and data collectors are isolated from those details.

But the collection of instruments may grow and shrink as objects are created and destroyed, so we must ensure that iterators know about these changes and adjust themselves. Luckily, the instrumentation system considers object deletion and creation events to be significant, so the events are intercepted and used to keep the instrument Composite and Iterators consistent.

4.4 Dynamic binding of eventSensors to instruments

• *Problem:* When switching abstraction levels, we want to bind event sensors to target instruments dynamically without undue effects on efficiency.

• *Solution:* We exploit the fact that classes (and hence objects) that contain the sensors themselves do not change their architectural units dynamically. For every object (and sensor) we can statically determine a list of instruments, one for every level of abstraction. Thus, we only have to indicate the level of abstraction to a sensor, and it can dispatch the data to the right instrument.

We introduce an intermediate EventAnnouncer object that knows about the abstraction level for data collection. The sensors report the event and their list of instruments to the announcer, and the announcer uses the abstraction level to select the right instrument.

• *Consequences* The binding of sensors to instruments can be switched by visiting the announcer and changing a single field that indicates the instrument at the desired level of abstraction. A single indirection suffices to achieve this binding without significant impact on efficiency.

More details of the instrumentation system organization can be found in [21].

4.5 Queries

The instrumentation and visualization system supports queries about architectural concerns and their interactions. Figure 11 expresses our query syntax in Backus Naur form (BNF)[1]. We illustrate the language with examples from the previous case studies.

- From Figure 1, the user displays all the processes that are currently blocked in the virtual memory system using:

  ProcessClass **kindreds -r** *state* = *blocked; blockingSystem* = *VmSystem;* **display**

- From Figure 2, the user disables all the logical memory framework instruments:

  LogicalMemoryFramework **subparts -r deactivate**

- To monitor calls from the file system to the virtual memory at the subsystem-level, the user enters the two queries:

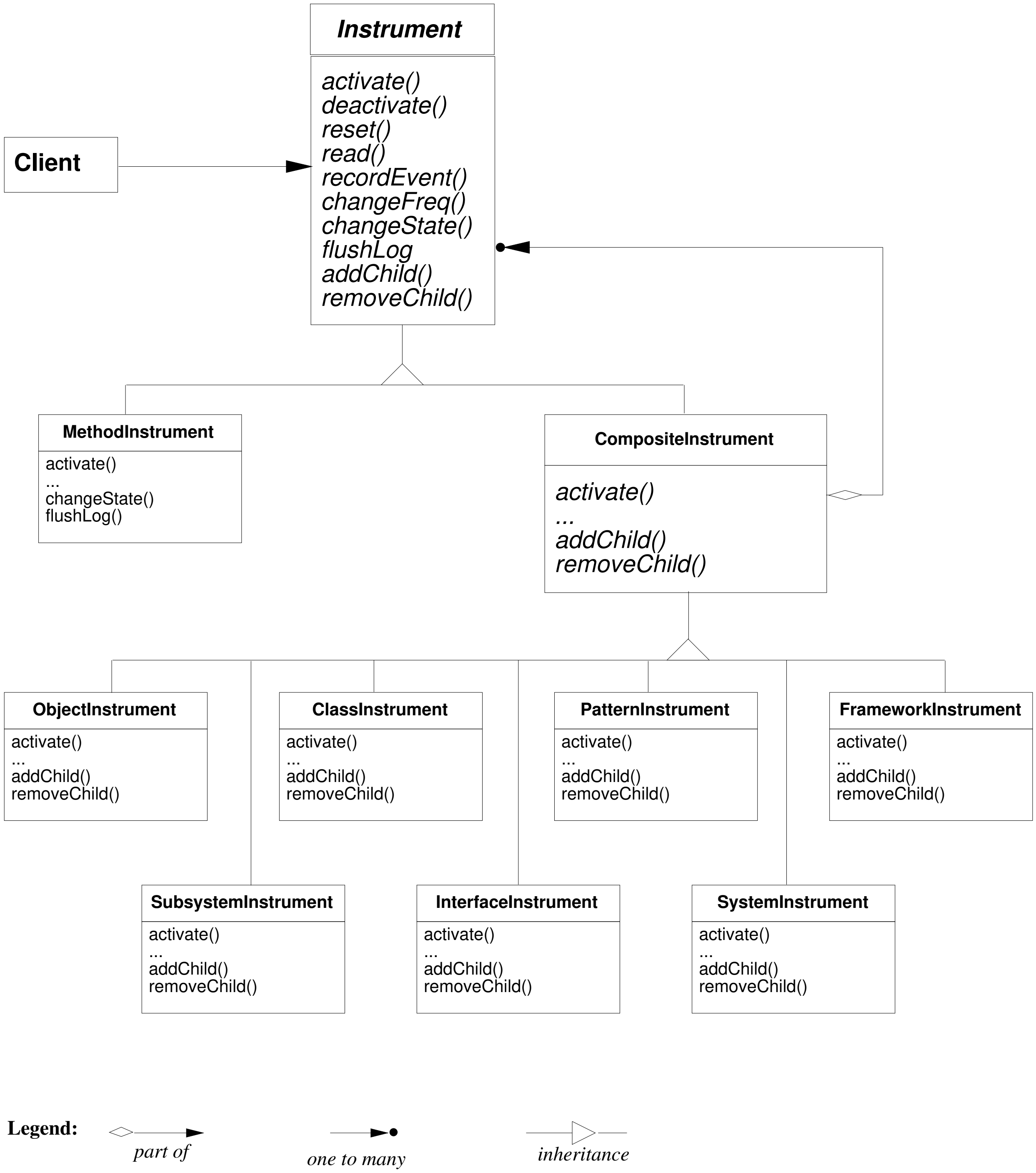

Figure 10: Organizing aggregate and simple instruments for uniform accessibility by the client code.

```
<Query> ::=
      <Component> [<Selector>] <InstrumentOperation>
<Component> ::=
      <System> | <Framework> | <Pattern> | <Class> | <Object> | <Method>
<System> ::=
      system-name
      /* similarly for Framework, Pattern, Class, Object, and Method components */
<Selector> ::=
      <RelatedComponent> [<Per-component-state-descriptor>] |
      <Property-list> | <BasicInstrument>
<RelatedComponent> ::=
      subparts [<Abstraction-level> | -r] | aggregator [<Abstraction-level> | -r]
      | <ClassHierarchy> | <ClassInstances> | <OtherRelation>
<Abstraction-level> ::=
      system-level | framework-level | pattern-level | class-level | object-level
      | method-level | <AbstractionUnit>
<AbstractionUnit> ::=
      <Component>
<ClassHierarchy> ::=
      ancestors [<Depth> | -r] | descendants [<Depth> | -r] | whole-hierarchy
      | children
<ClassInstances> ::=
      kindreds [<Depth> | -r] | members
<Depth> ::=
      integer
<OtherRelation> ::=
      caller [<Abstraction-level>] | callee [<Abstraction-level>]
      | creates [<Abstraction-level>] | deletes [<Abstraction-level>] | etc.
<Per-component-state-descriptor> ::=
      [instrument-state-var <Comparison> value;]+
<Comparison> ::=
      = | ≠ | < | >
<Property-list> ::=
      any | all | <Per-component-state-descriptor>
<BasicInstrument> ::=
      counter | timer
<InstrumentOperation> ::=
      activate | deactivate | display |
      reset | count | etc.
```

Figure 11: A partial description of our query language in Extended BNF notation.

VmSystem **subsystem-level activate**

VmSystem **caller** *FileSystem* **count**

- To display the global counter instrument of the virtual memory system, the user enters:

VmSystem global-counter **display**

In most cases, the user need not explicitly enter a query. The visualizer translates the users selection into the appropriate query. However, we also allow the user to directly enter queries when necessary.

5 Performance evaluation

Architecture-aware instrumentation improves upon conventional instrumentation in the following ways:

- It exploits run-time knowledge about the system structure to condense the amount of data it collects, conserving space. Condensation also decreases traffic to the visualizer and the need for data analysis.
- It selectively enables instruments and event sensors in a query-specific manner, reducing the amount of data collected.
- The instrument organization directly reflects the system structure, therefore data about system components is quickly located.

The first two improvements reduce the net amount of data generated by architecture-aware instrumentation. Together, the three improvements reduce the timing overhead of instrumentation. In the following, we substantiate these claims by comparing traditional instrumentation with architecture-aware instrumentation as regards the amount of data generated and the execution overhead introduced by instrumentation. Traditional instrumentation maintains per-method instruments; these instruments may be counting or timing events. In our system, method-level instruments are simply the lowest instrumentation level.

We conducted our performance studies on the *Choices* operating system running on a Sun SPARCStation 2. We collected all the numbers from the virtual memory system. This system comprises 50 classes and 625 object methods. Of the 50 classes, 20 are abstract interfaces visible to the neighboring systems (e.g., the file system).

We ran a distributed remote procedure call (RPC) server as our test application. A client exercises the server by making RPCs that transfer an array of 256Kb each. While this application was executing, we gathered instrumentation data for the query "How many methods of the virtual memory system were called by the file system?" [2]. The instrumentation data packets and execution times were measured over five second intervals.

5.1 Data generation

Figure 12 compares the number of data packets generated by flat instruments and architecture-aware instruments for execution statistics at different levels of detail. All data packets are of the same size. Each packet encodes a component identifier, a cumulative call count, and meta-information needed by the visualizer to unmarshal the packet.

In the performance display graph of Figure 12, the dotted bars depict trace data generated using traditional instrumentation [15, 14, 20, 19, 12, 16]. Here, data packets are generated on a per-method basis; every data packet includes information about the *method* that was called. Consequently, the number of packets equals the number of methods called regardless of the granularity of system modeling.

The solid bars display data collected using architecture-aware instrumentation. The query about aggregate subsystem statistics generates a single data packet from a subsystem level instrument that keeps a running count of the method calls. If we expand the query to find the number of method calls per framework, class, or method, the number of data packets increases proportionally. Thus, we exploit the knowledge of the virtual memory system embedded in the instrumentation system to reduce the trace data size substantially during high level analysis.

5.2 Instrumentation overhead

Architecture-aware instruments reduce system perturbation because (1) they reduce the data and hence the time for data collection, (2) the instrumentation system can selectively enable instruments depending on the components mentioned in a query. We characterize the two situations with Figure 13 and Figure 14 respectively. We consider the execution time for four key operations in the virtual memory system: handling page faults (`repairFault()`), creating address spaces (`domainCreate()`), adding memory objects to address spaces (`addMo()`), and removing memory objects from address spaces (`removeMo()`).

Figure 13 compares the execution times of the four operations with and without instrumentation, and shows how normal execution is slowed down by instrumentation. For example, repairFault() is slowed down by 14% due to method-level (i.e., traditional) instruments, while system-level instruments only slow the operation down by 1%. In

[2]This query helps evaluate the degrees of subsystem cohesion and coupling as in Section 3.1.

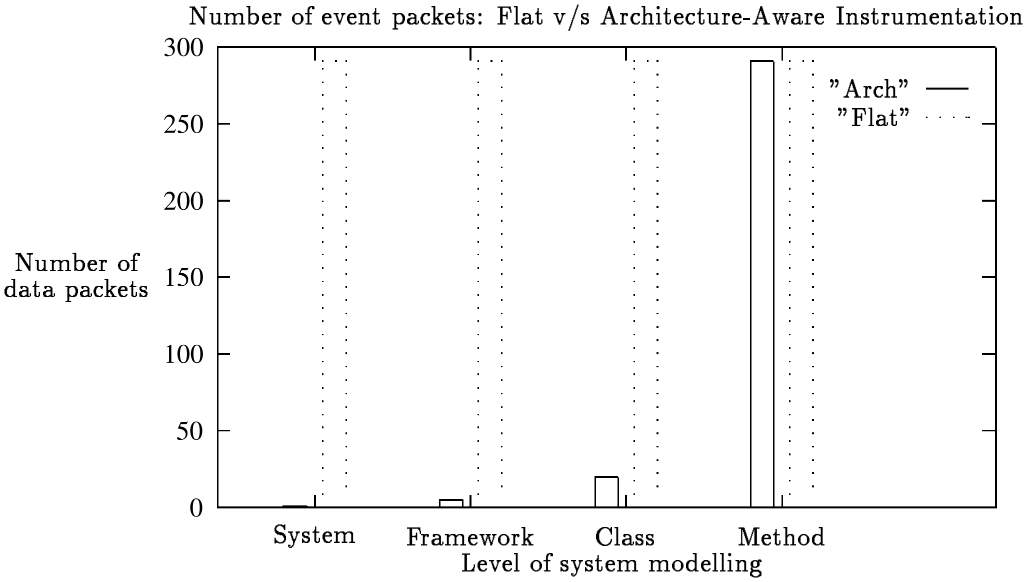

Figure 12: A comparison of flat versus architecture-aware instrumentation in providing cumulative statistics of the *Choices* virtual memory system. Architecture-aware instrumentation dramatically reduces the trace data size as the level of system abstraction increases.

this study, we instrument the system at successively lower instrumentation levels, each time collecting execution perturbation statistics for each operation.

Figure 14 compares the performance of instrumentation that can enable during run-time instruments for a specific framework versus one that cannot. We consider the execution overhead of the same four operations, first with all the method-level instruments of the entire virtual memory system enabled, then with only the method-level instruments of the caching framework of this system enabled. The second case applies to flat instrumentation where selective, framework-based, dynamic control of instruments is difficult because the relevant instruments are difficult to name. Again, it is clear that enabling few instruments reduces system perturbation due to instrumentation.

Remark The above performance studies show that architecture-aware instrumentation is more efficient for supporting architecture-level investigations of running programs. The cost of traditional instrumentation is simply too high when inspecting higher-level events, partly due to poor information abstraction or condensation, partly due to the lack of adaptiveness.

Architecture-aware instrumentation, however, is not completely without cost. It increases the size of the application more than flat instrumentation due to the addition of more code for query processing and instrument management. In the above experiment, the *Choices* system executable, compiled with a GNU gcc compiler, was 14.3% bigger (worst case) with architecture-aware instruments, but only 7.8% bigger with conventional instruments. Luckily, most of the code for architecture-aware instruments will remain dormant in the system until explicitly needed. We feel that system size growth is an acceptable tradeoff given the significant benefits to be gained by using instruments that understand the structure of the system, and given the consistent trend of falling memory prices.

6 Related research

Current dynamic visualization tools for object-oriented programs appear to focus on animating class, object, and method interactions[19, 20, 13]. ProgramExplorer[14, 15], Pattern-Lint[23], and OS View[22] are some of the new systems that visualize architecture-level concepts like frameworks and design patterns. However, the granularity of system depiction is still classes, objects, and methods. No system that we know supports dynamic architecture-oriented visualization, where the graphical presentations of the running system directly map to architectural abstractions like frameworks.

Previous research on object-oriented program visualization has concentrated more on trace data organization than on structuring the instruments that gather the data. In ProgramExplorer[14], the data is collected into a Prolog database. In ObjectVisualizer[20], multi-level hash tables organize the data, supporting data accumulation, albeit only on a per-method basis.

Structured instruments were exploited in a limited manner in the *Choices* operating system[4]. Madany uses objectified C++ classes, called first-class Classes, to exam-

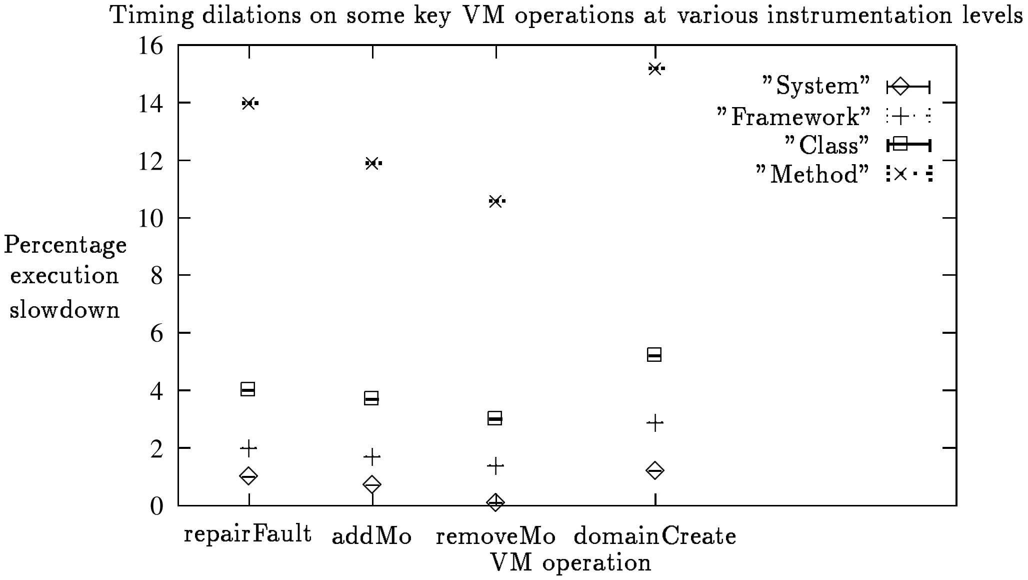

Figure 13: A comparison of the increase in execution time introduced by various instrumentation levels. The higher the instrumentation granularity, the less the timing perturbations.

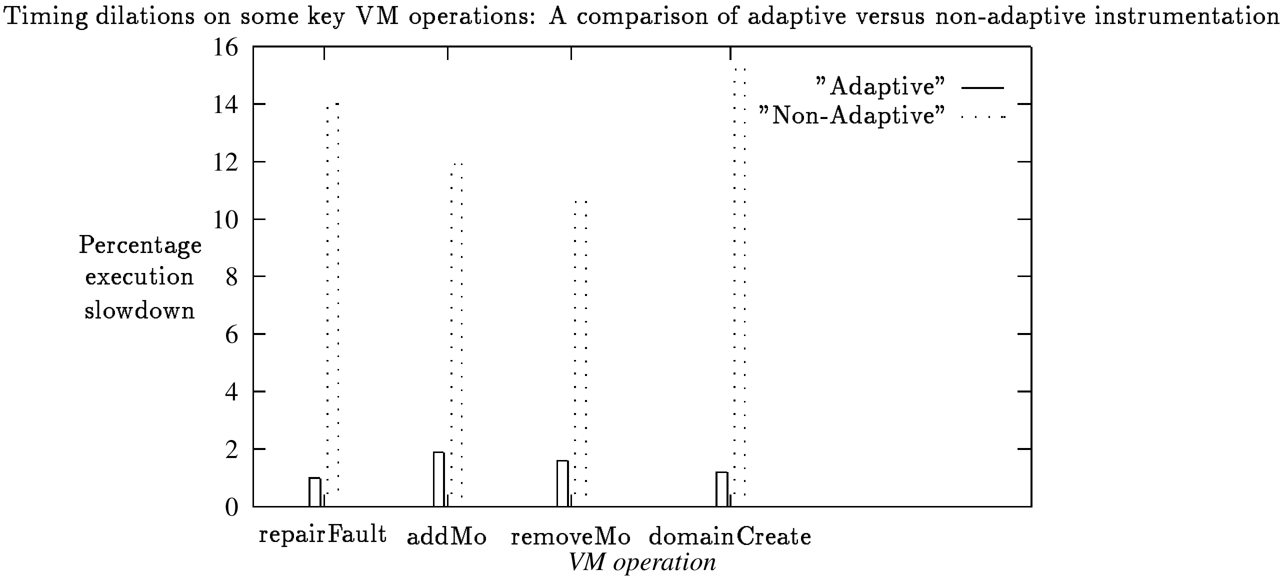

Figure 14: A comparison of adaptive versus non-adaptive instrumentation.

ine the run-time structure and behavior of the operating system. Classes provide run-time type information, enabling the querying of class inheritance and class instance relations among objects. Instruments that are related by inheritance can be controlled in groups dynamically. However, the instrument selection is purely based on class type information[10]. This introspection scheme does not go beyond the class level, nor does it explicitly represent relations among instruments other than Classes. A similar class-based approach is adopted by De Pauw et al.[19].

Snodgrass[25] presents a method for monitoring program execution in which a programmer uses relational algebra queries to track run-time dynamics. The scheme is more tailored for data organization (using a conceptual, temporal database) than instrument organization. Snodgrass' instrumentation is mainly targeted towards distributed debugging and monitoring, not on analyzing the software architecture of the system.

Other related research includes work on computational reflection. While rich structures and meta-level architectures for reflection have previously been defined[18, 11, 6, 26], issues in the explicit representation of gross architectural abstractions appear to be under emphasized because the problems being investigated are different. In this paper, we discussed one way to reify system architectural units and subunits, and showed how to use the reified representations to optimize instrumentation.

7 Conclusions

Software visualization and instrumentation tools have traditionally recognized classes, objects, methods, or functions as the basic units of analysis and animation. However, research in software architecture suggests that the more important reusable components for complex systems are likely to be higher-level aggregates of collaborating classes such as frameworks, design patterns, or subsystems. With the increasing emphasis on composing systems out of large-scale components, programmers need tools that elevate the granularity of analysis for operational software systems to the architecture level.

In this paper, we have demonstrated through real-world case studies that architecture-oriented visualization greatly simplifies structural analysis, behavioral study, and performance measurement of complex object-oriented systems. We have illustrated that this technique is effective in offering scalable views of a large system, and in modeling run-time dynamics hierarchically at varying component levels.

Not only do we advocate the visual examination of working software systems in terms of their conceptual architectural units, but we also introduce a practical instrumentation technique to support such investigations. The technique is called *architecture-aware instrumentation*. A distinctive feature of architecture-aware instrumentation is that it explicitly represents composite architectural structures in a running system and exploits this knowledge to abstract run-time information. Performance studies indicate that instrumentation that optimizes data collection on a per-architectural model basis generates dramatically less trace data and introduces far less overhead than flat, unstructured instrumentation.

For example, a page fault handling operation ran 13% faster with subsystem-aware instruments, an operation to add memory objects to address spaces ran 11% faster, an operation to delete memory objects from address spaces ran 10% faster, and an operation to create address spaces ran 14% faster. This indicates that the system-level perturbation effect of an architecture-aware instrumentation is substantially less than that of traditional instrumentation.

This work opens several avenues for future research. One interesting approach involves merging our system with a code refactory that would automate design repair: use visualization to identify problems, and then use the refactory to correct them. Another possibility employs our instrumentation techniques to gather statistics about component interactions and use them in an optimizing compiler. We are using the system in a project to build customizable infrastructures for distributed objects, where the system will be extended for visualization of distributed systems. Another area of interest concerns the integration of our system with programming tools like low-level debuggers and code browsers, making visualization a daily staple in programming. We expect that the use of pervasive instrumentation will lead to improved user interfaces and program views, especially three-dimensional views using the emerging virtual reality technology.

Acknowledgments: We thank Amitabh Dave, John Coomes, Ashish Singhai, Howard C. Huang, and the anonymous OOPSLA referees for their helpful comments.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[2] Marla J. Baker and Stephen G. Eick. Visualizing Software Systems. In *Proceedings of the 16th International Conference on Software Engineering*, pages 59–67, 1994.

[3] Grady Booch. *Object-Oriented Analysis and Design With Applications* . The Benjamin/Cummings Publishing Company, Inc., 1994.

[4] R. H. Campbell, N. Islam, D. Raila, and P. Madany. Designing and Implementing Choices: An Object-Oriented System in C++. *Communications of the ACM*, pages 117–126, September 1993.

[5] L. Peter Deutsch. Design Reuse and Frameworks in the Smalltalk-80 Programming System. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability*, volume II, pages 55–71. ACM Press, 1989.

[6] Brian Foote and Ralph E. Johnson. Reflective Facilities in Smalltalk-80. In *OOPSLA 89, Conference Proceedings*, pages 327–335. ACM, 1989.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.

[8] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of the 2nd ACM SIGSOFT*, pages 175–188, December 1994.

[9] Philip Haynes, Tim Menzies, and Robert F. Cohen. Visualisations of large object-oriented systems. Technical Report TR 95-4, Monash University, Melbourne, Australia, August 1995.

[10] J. A. Interrante and M. A. Linton. Run-time Access to Type Information in C++. In *Proceedings of the USENIX C++ Conference*, San Francisco, California, April 1990.

[11] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[12] D. R. Kohr, S. Zhang, M. Rahman, and D. A. Reed. Object-Oriented Parallel Operating Systems: A Performance Study. In *Scientific Programming*, 1994.

[13] C. Laffra and A Malhotra. HotWire:A Visual Debugger for C++. In *Proceedings of the USENIX C++ Conference*, pages 39–54, 1994.

[14] Danny B. Lange and Yuichi Nakamura. Interactive Visualization of Design Patterns Can help in Framework Understanding. In *OOPSLA*, 1995.

[15] Danny B. Lange and Yuichi Nakamura. Program Explorer: A Program Visualizer for C++. In *Usenix Conference on Object-Oriented Technologies*, pages 39–54, June 1995.

[16] Rahman M. Choices Instrumentation Support. Technical report, University of Illinois-Urbana Champaign, 1992.

[17] Peter W. Madany, Roy H. Campbell, and Panos Kougiouris. Experiences Building an Object-Oriented System in C++. In *Technology of Object-Oriented Languages and Systems Conference*, Paris, France, March 1991.

[18] Pattie Maes. Concepts and Experiments in Computational Reflection. In *OOPSLA 87, Conference Proceedings*, pages 147–155. ACM, 1987.

[19] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the Behavior of Object-Oriented Systems. In *OOPSLA*, October 1993.

[20] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Modelling Object-Oriented Program Execution. In *Proceedings of the 8th European Conference on Object-Oriented Programming, ECOOP'94. Lecture Notes in Computer Science*, October 1994.

[21] Mohlalefi Sefika. *Design Conformance Management of Software Systems: An Architecture-Oriented Approach*. PhD thesis, University of Illinois at Urbana-Champaign, July 1996.

[22] Mohlalefi Sefika and Roy H. Campbell. An Open Visual Model For Object-Oriented Operating Systems. In *Fourth International Workshop on Object Orientation in Operating Systems*, Lund, Sweden, August 1995.

[23] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring Compliance of a Software System With Its High-Level Design Models. In *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, March 1996.

[24] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[25] R Snodgrass. A Relational Approach to Monitoring Complex Systems. *ACM Transactions of Computer Systems 6(2):157-196*, May 1988.

[26] Yasuhiko Yokote, Fumino Teraoka, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. The Muse Object Architecture: A New Operating System Structuring Concept. Technical Report SCSL-TR-90-012, Sony Computer Science Laboratory Inc., October 1990.