Optimization Aspects in Network Simulation

Jamal Siadat, Robert J. Walker, and Cameron Kiddle
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
{siadat, rwalker, kiddlec}@cpsc.ucalgary.ca

Technical report 2005-802-33
30 September 2005

ABSTRACT

A primary goal of AOSD in the context of systems software has been to permit improved modularity without significantly degrading performance. Optimizations represent important crosscutting concerns in this context but also a significant challenge due to their fine-grained nature. This paper investigates how well the current state-of-the-art in AOSD can support such optimization aspects, via a case study involving an optimized network simulator, IP-TN. Duplication of optimizations achieved via low-level modifications to IP-TN in C++ have been attempted via aspectization of those optimizations in AspectC++. While comparable run-time performance is achieved with AspectC++ and (un)pluggability is clearly simpler, the effects on comprehensibility are less clear.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Simulation and emulation*; C.4 [**Computer Systems Organization**]: Performance of Systems; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.1.m [**Programming Techniques**]: Miscellaneous—*Aspect-oriented software development*.

General Terms: Design, Experimentation, Languages, Performance.

Keywords: Optimization, performance, network simulation, IP-TN, AspectC++, modularity, fine-grained join points, evaluation.

1. INTRODUCTION

> If you make an optimization and don't measure to confirm the performance increase, all you know for certain is that you've made your code harder to read. [12]

While systems software research has considered ideas from software engineering and programming languages regarding improved modularity (e.g., [19, 10, 37]), issues of improved comprehensibility and evolvability have traditionally been considered far less important than efficiency. As a result, systems software is typically difficult to understand and to modify. Since the early days of research evaluating aspect-oriented software development (AOSD) [36], a key goal has been to determine whether improved modularity and efficient performance can be simultaneously achieved in systems software.

While systems software development can begin from a structurally-clean design, this design can quickly become obscured by the necessary addition of optimizations. Premature optimizations and hard-to-unplug optimizations have the potential to degrade structure and inhibit performance through successions of evolution steps. An AOSD approach to realizing optimizations should be attractive to the systems community because of its potential to separate the development process into two stages: construction of the ideal structure followed by provision of optimizations that can be plugged-in and unplugged as the remainder of the system evolves over time.

Such optimizations can be complex crosscutting concerns for the sake of alternative computation paths [7, 5], or conceptually simpler and finer-grained to eliminate run-time flexibility mechanisms (such as polymorphism). Work-to-date has largely concentrated on the question of whether "non-trivial" crosscutting concerns occur in systems software and whether they can be modularized (e.g., [7, 22, 29]). In contrast, finer-grained optimizations may be more prevalent, more technically challenging for aspect-oriented (AO) programming languages, and less compelling examples of improved modularity. They are not obviously amenable to the support provided by typical AO programming languages (e.g., [29, 34]), as those languages concentrate on granularities at or above the method-level. While approaches to aspect-oriented refactoring focus on provision of join points amenable to the AO programming languages available (such as calls to "hook" methods) [24, 18], adding extra levels of indirection to a program tends to incur performance costs that can be unacceptable in the context of systems software. At the same time, arguing that an aspect consisting of a set of small tweaks exhibits improved modularity could be difficult.

In this paper, we consider how well the current state-of-the-art is capable of reconciling the desires of high performance and improved modularity in a specific systems software context: an optimized network simulator, IP-TN [31]. To assess the effectiveness of AOSD with respect to applying optimizations, two optimizations to the packet buffer mechanism of IP-TN were chosen and implemented in an aspect-oriented language, AspectC++ [33]. The challenges for an aspect-oriented modularization of buffer optimizations in IP-TN include:

- achieving comparable run-time performance;
- improving the localization of the optimizations, and to allow them to be plugged into or unplugged from the base implementation;
- considering the optimizations as after-the-fact modifications to the idealized design, without preplanning and preferably without invasive modification of the base code, in order to evaluate the potential for avoiding premature optimization [12];

- improving the comprehensibility of the optimizations; and
- reproducing the fine-grained weaving capabilities provided via compiler directives, or otherwise achieving the same effect.

We find that while comparable run-time performance can be achieved with existing tool support and (un)pluggability is clearly simpler, the effects on comprehensibility are less clear.

The remainder of the paper is structured as follows. Section 2 describes background information regarding the IP-TN network simulator that we have studied, and two optimizations (one that reduces the number of events without loss of accuracy, and one that reduces the number of events by employing fluid-flow abstraction) that were originally applied therein via traditional methods. Section 3 describes our attempts to provide similar optimizations via aspects written in AspectC++. Section 4 describes our empirical evaluation of the performance of IP-TN to compare the C++ and AspectC++ implementations. Section 5 discusses the results of our study in terms of the lessons learned about AspectC++, and AOSD in general, in this context. Section 6 describes related work. Section 7 concludes and describes future work.

The contribution of this paper is the identification of strengths and weaknesses of an existing AO approach to after-the-fact optimizations in a systems software context.

2. BACKGROUND: THE IP-TN SIMULATOR

Our case study considers how fine-grained optimizations can be achieved via aspects in the Internet Protocol Traffic and Network Simulator (IP-TN) [31]. Therefore, we begin with sufficient detail on the requirements and design of IP-TN so that the purpose of the optimizations can be understood.

IP-TN models IP networks at the network, transport, and application layers of the Open Systems Interconnection (OSI) Reference Model [13]. IP-TN allows for different network protocols and applications to be modelled and tested under various network conditions in a controlled and repeatable environment. An emulation extension to IP-TN, called IP-TNE, allows for real network hosts to interact with the simulator in real-time. This extension enables the direct testing of actual implementations of network protocols and applications.

IP-TN is built on a parallel discrete event simulation kernel, called CCTKit [32]. Network topology and traffic conditions to simulate are specified in input files written in ANother Modelling Language (ANML) [17]. ANML constructs allow for easy reuse and aggregation of components to aid in the construction of complex models. IP-TN consists of approximately 27 kloc of C and C++ source code at its core; all of its extensions and libraries increase the total to the neighbourhood of 100 kloc; the CCTKit simulation kernel consists of an additional 17 kloc.

Figure 1 shows the sequence of steps carried out by IP-TN in the simulation of a network model. First, ANML files describing the model to be simulated are input and processed. Next, the various components of the input network model—including network nodes, links, interfaces, packet buffers, and traffic objects—are constructed. In the initialization phase, the network model components are connected together and mapped to underlying simulation kernel objects. The initial state of all network model and simulation kernel components is also computed at this time. The simulation is then executed with synchronization and advancement of simulation time controlled by the underlying CCTKit kernel. Finally, statistics on the resulting simulation are computed and output, including simulation kernel metrics that measure the performance of the simulator as well as network-oriented metrics that measure the behaviour of the network model and traffic flows.

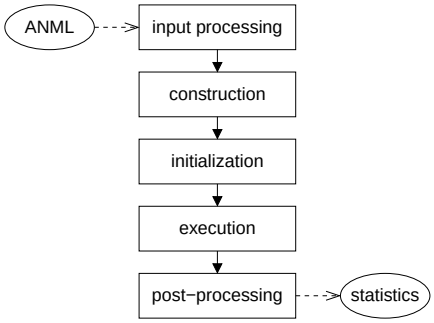

Figure 1: High-level flow graph for IP-TN. Dotted lines indicate dataflow; ellipses are the kind of data involved; solid lines are control flow; boxes are processing steps.

Events are used to model the transmission of packets hop-by-hop from source to destination in the simulated network with queuing, transmission, and propagation delays accounted for. Packets are queued for transmission and receipt by the network nodes in packet buffers. The number of events required to model a single packet transmission from one node to the next (one hop) depends on the implementation of the packet buffers on network node interfaces. The fewer the events required to model a packet transmission, the better the performance that can be achieved by the simulator. Therefore, optimizations on the packet buffers that reduce the number of events during the simulation execution have the potential to improve the execution performance.

The remainder of this section considers the kinds of buffer optimizations that exist in IP-TN (in Section 2.1) and examines details of their implementations (in Section 2.2).

2.1 Buffer Optimizations

Many different buffer types, including both preemptive and non-preemptive buffers, have been implemented in IP-TN [8, 16]. Only the non-preemptive buffers, which are modelled as fixed size FIFO (first-in-first-out) queues, are considered in this paper.

The most basic and computationally expensive buffer in IP-TN is called the *Simple Buffer*. An optimized buffer that maintains accuracy but reduces the number of events required to model a packet transmission is called the *Standard Optimized Buffer*. Another optimized buffer, which makes use of simulation abstraction techniques to further reduce the number of events but with some loss in accuracy, is called the *Hybrid Buffer*. Further conceptual details of these three kinds of buffer are examined in Sections 2.1.1, 2.1.2, and 2.1.3 respectively.

2.1.1 Simple Buffer

The simplest and most intuitive buffer implementation—it most closely matches buffer implementation in real networks—in IP-TN is the Simple Buffer. When a packet arrives at a particular network node, which is signaled by a packet arrival event, a route lookup is performed to determine the output interface via which the packet must next be sent on route to its destination. If the packet buffer on the corresponding output interface has enough room for the packet, the packet is inserted into the buffer; otherwise, the packet is dropped. If the packet was successfully inserted into the buffer and there are no other packets in the buffer, the packet is sent to the next hop. This is done by generating an arrival event for the

next network node at a time equal to the current time plus the transmission and link propagation delays. Another event must also be generated at the current time plus the transmission delay to check the buffer for newly arrived packets that need to be sent after the current packet has finished transmitting.

Upon receipt of an event to check the buffer for more packets, the next packet is sent following the process discussed above. If the buffer is empty nothing further needs to be done. The Simple Buffer requires two events to model a packet traversing one hop in the network.

2.1.2 Standard Optimized Buffer

The Standard Optimized Buffer is an optimization of the Simple Buffer that reduces the number of events required to model a packet transmission without losing any accuracy. For a non-preemptive FIFO buffer, the arrival time of the packet at the next node can immediately be calculated upon receipt of the packet arrival event. This allows the arrival event for the next node to be generated without the need for an event to check the buffer for more packets.

To accurately keep track of the current buffer usage, a list containing information on all the packets in the buffer is maintained. When a packet arrives at a given network node, the list is "refreshed." Information for packets that have finished transmission are removed from the list, following which the current buffer usage is calculated. If there is room in the buffer, the packet is sent with the appropriate arrival time and information regarding the packet inserted into the list; otherwise, the packet is dropped.

The Standard Optimized Buffer only requires one event to model a packet traversing one hop in the network. In addition to a reduction in the number of events, there can be advantages in parallel execution by allowing neighbouring nodes to safely execute further into the future.

2.1.3 Hybrid Buffer

Both the Simple and Standard Optimized Buffers model each packet individually. The number of events required to model a packet transmission can be further reduced by making use of fluid-based network simulation abstraction techniques [14, 20, 25], in which network traffic is modelled as piecewise-constant bit-rate flows. An event is only generated when there is a change in the rate of a traffic flow. Rate changes may occur due to a change at the traffic source or due to queuing and multiplexing in the network. If many packets are represented by each change in a traffic flow rate, then the number of events can be significantly reduced. Although performance improvement can be achieved with the use of fluid flows, there is some loss in accuracy; this loss has been shown to be acceptable under many cases [25].

The Hybrid Buffer is capable of handling a mixture of detailed packet flows and less detailed fluid flows. It operates in one of three modes depending on the type of traffic it is handling. If the buffer only handles packet flows then it operates in packet mode; if the buffer only handles fluid flows then it operates in fluid mode; and if the buffer handles a mixture of packet flows and fluid flows then it operates in hybrid mode. The operation of the buffer in packet mode is the same as that for the Standard Optimized Buffer. The operation of the buffer in fluid mode is similar to that in the fluid-based network simulation literature [14, 20, 25]. The operation of the buffer in hybrid mode is similar to the fluid mode but with modifications to handle packet-based flows [16].

2.2 Original Buffer Implementations

Since our goal is to provide a comparison of fine-grained optimizations in the original implementation with their aspect-oriented analogues, we describe the design of the original buffer implementations here.

A simplified class diagram of all the buffer implementations and their clients is provided in Figure 2; some names have been shortened and some details have been dropped (e.g., formal parameters, return types, some member variables). Two subclass hierarchies plus some supplementary classes are involved.

All the buffers in IP-TN extend the base class `output_buffer`. This class contains a set of virtual functions with minimal implementations that are overridden by subclasses. Both the Simple and Standard Optimized Buffers are implemented in a single class and are used by two other classes. The Hybrid Buffer is implemented in a single class, with three other classes explicitly calling its member functions. Furthermore, the Hybrid Buffer must be configured at compile time in order to use it. To this end, the base code contains a number of `#ifdef ENABLE_HYBRID` compiler directives specifying the actions to be taken at particular points when this buffer is enabled; Figure 2 attaches notes to several of the classes where these directives are used. The `other_buffer` class is inserted in the diagram to indicate that there are other buffer implementations in IP-TN (i.e., preemptive buffers) that are not discussed in this paper.

The second subclass hierarchy shown in the diagram involves the representation of different kinds of nodes in the simulated network; four of these nodes are of interest here: `net_node`, `ip_base_node`, `ip_service_node` and `interconnect`. When sending or receiving a packet, the interface between a given network node and the corresponding link that the packet must travel through is responsible to create and update information for various buffers. The `buffer_factory` class permits clients to create instances of registered buffer types via call-by-name; `buffer_init_algs` initializes the factory by registering all the buffer types. The `ip_base_node` class is responsible for requesting that a given kind of buffer be created by the `buffer_factory`, according to the specifications provided as input to the simulation in ANML files. In addition, `ip_base_node` determines the appropriate execution path for a packet transmission, based on the buffer type.

Two events are generated per packet transmission if the Simple Buffer is chosen as the default buffer type. For each packet that is processed, the `ip_base_node` class makes a call to the `write_packet_to_buffer()` method in the `simple_buffer` class. This method is responsible for writing a packet to the buffer and sending the packet, if the buffer is empty, by generating a packet arrival event for the next hop. After sending a packet, a second event is generated to call the `copy_next_packet_to_link()` method after the packet has finished transmission. This method sends the next packet in the buffer if there is one.

If the Standard Optimized Buffer is chosen as the default buffer type, only a single event is required to model a single packet transmission. For each packet that is processed, the `ip_base_node` class makes a call to the `update_next_send_time()` method, which generates an arrival event at the next node.

The Hybrid Buffer handles both packets and advertisements specifying fluid flow rates. A single packet and multiple fluid advertisements can be sent in the same arrival event for the next node. As such, the `ip_service_node` class is extended to direct packets and fluid advertisements that arrive together to the appropriate output interface buffers and to ensure that each of the packets/advertisements at each output interface buffer is processed. The latter is achieved by calling the `process_packets()` method of the `hybrid_buffer` class for the buffer on each output interface. In the `process_packets()` method, calculations are performed to determine any packet loss and changes in fluid flow rates, with an

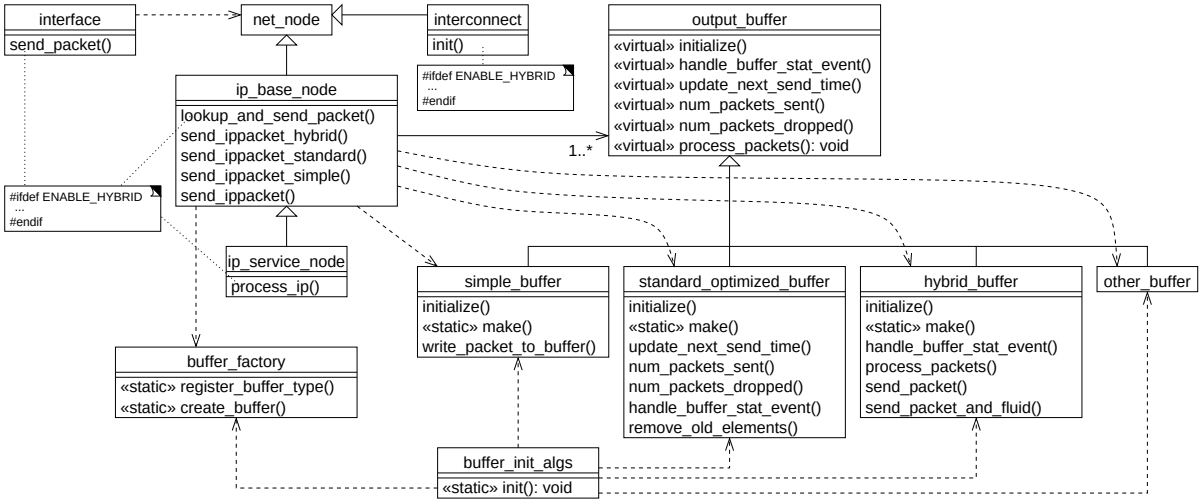

Figure 2: Simplified UML class diagram for the IP-TN non-preemptive buffers. Dashed arrows indicate dependencies; other dotted lines indicate where the compiler directives occur.

arrival event generated to send non-dropped packets and fluid flow rate change advertisements to the next network node.

The remaining methods and classes shown in the diagram matter to our context only inasmuch as they either contain `#ifdef ENABLE_HYBRID` compiler directives, or their functionality must ultimately be provided by an aspect; their details are otherwise beyond this paper.

Note that the Standard Optimized and Hybrid buffer execution paths were not encapsulated exclusively in the `standard_optimized_buffer` and `hybrid_buffer` classes respectively. This can be observed, for example, as the buffer creation and selection mechanisms for each optimization were implemented in separate classes (the `ip_base_node` and `buffer_factory` classes). As shown in Figure 2, the non-modular implementation of these optimizations across several classes indicate crosscutting concerns for each of them. Furthermore, in the Hybrid case the existence of `#ifdef ENABLE_HYBRID` compiler directives indicate an especially scattered implementation of this optimization. We proceed to consider how these optimization crosscutting concerns could be realized as aspects.

3. OPTIMIZATION ASPECTS

Recall the challenges that our aspect-oriented modularization of buffer optimizations must meet: (1) achieving run-time performance comparable to that of the original C++ implementation; (2) improving localization and pluggability of the optimizations; (3) permitting the optimizations to be performed in an after-the-fact manner; (4) improving comprehensibility; and (5) reproducing or mimicking the fine-grained weaving capabilities of compiler directives.

The AspectC++ language [33] was used to implement the aspects, since the base code is in C++. In addition to the goals above were a number of practical obstacles with which we had to cope. The quantity of source code involved in these optimizations was non-trivial to agglomerate into an aspect (the Standard Optimized Buffer and Hybrid Buffer approaches added approximately 0.5 kloc and 1.5 kloc, respectively). IP-TN is written in C++ in a style intended to minimize run-time overhead from levels of indirection (e.g., virtual function usage), so any addition of indirection could impair performance.

The original implementation of IP-TN provides for an ability to select which optimizations should apply on a per-instance basis (the Hybrid Buffer approach additionally required compile-time modifications via compiler directives). This run-time flexibility was provided chiefly for the sake of supporting a single kernel with which to evaluate different simulation approaches. In contrast, we take the approach of providing strict compile-time selection of each optimization for every non-preemptive buffer in an IP-TN simulation. We discuss this issue further in Section 5.5.

The remainder of this section begins with a basic overview of AspectC++, in Section 3.1. A description of the details of each optimization is given, respectively, in Sections 3.2 and 3.3. Further analysis and discussion is left until Sections 4 and 5.

3.1 Overview of AspectC++

The AspectC++ language is largely analogous to the AspectJ language [15]. A class-like `aspect` construct is provided in which can be provided named pointcuts, inter-type declarations, and advice (`before()`, `after()`, and `around()`). Other primitive pointcuts of interest to us include `call()`, `execution()`, `within()`, and `args()`, with analogous interpretations as in AspectJ, while `construction()` is similar to AspectJ's `initialization()`. A reflection mechanism is available via the pointer `tjp`, which is equivalent to AspectJ's `thisJoinPoint`. Within `around()` advice, the original implementation can be resumed via the `proceed()` functions accessible from `tjp`.

Of notable absence is support for fine-grained join points, as is the case with AspectJ; thus, any attempt at accessing fine-grained join points (such as deeply embedded compiler directives) must be made indirectly, possibly by noting a particular sequence of higher-level join points.

Figure 3 illustrates the syntax of AspectC++ for a sample aspect, named `redirecting_opt`. Two named pointcuts, `addition()` and `process()`, and three advices are shown. Note that details of pointcuts, including wildcards such as "`%`" and "`...`", are declared as string literals to simplify the extension to the already complex C++ syntax. The `addition()` pointcut differs from AspectJ, as it names an entire class, `simple_buffer`. The `process()`

```
aspect redirecting_opt {
  pointcut addition() = "simple_buffer";
  pointcut process() =
    "% ip_service_node::process_ip(...)";

  // _state is introduced into simple_buffer
  advice addition(): int _state;

  advice construction(addition()): around() {
    // Body replaces simple_buffer constructor
  }

  advice execution(process()) && args(ipp):
    around(iptn_packet* ipp) {
    // Body replaces impl. of process_ip()

    if(some condition)
      tjp->proceed(ipp);
  }
}
```

Figure 3: Sample AspectC++ source code.

pointcut identifies the `process_ip()` member function on the `ip_service_node` class, ignoring details of the result and formal parameter types. In the first advice, we use this named pointcut to provide inter-type declarations, such as the addition of a member variable, `_state`, to `simple_buffer`. In the second advice, the implementation of the constructor for `simple_buffer` is replaced via `around()` advice on the primitive pointcut `construction()`. In the third advice, `around()` advice again is used to replace the execution of the `process_ip()` member function; an argument passed to this execution is exposed and bound to a formal parameter (`ipp`) for use within the body of the advice. The third advice illustrates the use of `proceed()`.

AspectC++ compilation support is provided via a preprocessor that transforms the source to C++. To understand what known issues exist with the AspectC++ preprocessor, communication with the AspectC++ development team was undertaken; these issues are elaborated upon in Section 5.

3.2 Optimizing the Simple Buffer via the Standard Optimized Buffer Approach

We wished to achieve the Standard Optimized Buffer optimization by altering the functionality of the `simple_buffer` class with an aspect, rather than providing a separate `standard_optimized_buffer` class. Therefore, the aspect to be implemented needed to replicate the functionality of the `standard_optimized_buffer` class, while any explicit reference to the `standard_optimized_buffer` class within the rest of the system had to be removed. Our design is depicted in Figure 4.

We created a single aspect, `standard_opt`, to represent the Standard Optimized Buffer optimization. This aspect made use of four pointcuts: `addition()`, `construction()`, `send()`, and `init()`. Details of these pointcuts follow.

The `addition()` pointcut referred to the `simple_buffer` class, and provided a hook for introducing member functions and variables to that class. In this way, all of the `standard_optimized_buffer` member functions and variables were introduced into the `simple_buffer` class.

The `construction()` pointcut captured invocation of the constructor of the `simple_buffer` class. The constructor was modified via `before` advice to initialize the newly introduced member variables (not shown on the diagram).

The `init()` pointcut captured execution of the `initialize()` member function of the `simple_buffer` class. This function is responsible for setting a pointer to the network node interface that this buffer belongs to and for initializing various buffer statistics. It needed to be modified according to the Standard Optimized Buffer approach. To accomplish this, `around` advice on this pointcut was declared that replaced the original functionality with the functionality that had existed in the `initialize` member function of the `standard_optimized_buffer` class (see Figure 2).

Finally, the `send()` pointcut captured execution of the `send_ippacket_simple()` member function on the `ip_base_node` class. In the original implementation, the `send_ippacket()` member function selected one of a set of helper functions based on the buffer type. Thus, `send_ippacket()` was manually modified to remove mention of the Standard Optimized Buffer approach, and the `send_ippacket_standard()` helper function was removed from the `ip_base_node` class. The functionality so lost was then re-inserted as `around` advice on the `send()` pointcut; any attempt at executing `send_ippacket_simple()` resulted instead in the execution of the functionality of the original `send_ippacket_standard()`.

3.3 Optimizing the Simple Buffer via the Hybrid Buffer Approach

Similar to the Standard Optimized Buffer optimization, we wished to realize the Hybrid Buffer optimization by eliminating all mention of the Hybrid Buffer approach in favour of an aspect. Again, the aspect needed to replicate the behaviour of the original `hybrid_buffer` class, and various points in the program had to be advised to utilize the optimization methods so introduced. Our design is depicted in Figure 5.

We created a single aspect, `hybrid_opt`, with a total of 70 advices (not all shown in diagram) applied to 7 pointcuts: `addition()`, `arrival_time()`, `init()`, `send()`, `lookup()`, `process()`, and `hybrid_warning()`. Details of these pointcuts follow.

The `addition()` pointcut serves an analogous purpose as in the `standard_opt` pointcut. It provides a mechanism for introducing new member functions and variables to the `simple_buffer` class. The introduction of this new code effectively results in the transformation of the `simple_buffer` to the `hybrid_buffer` class.

When the packet mode is enabled, the `arrival_time()` pointcut calculates the proper arrival time of a given packet at a new network node. Adjusting the arrival time in the hybrid mode is necessary due to the fundamental assumption that transmission delays are zero in the hybrid mode. An `around` advice implements this adjustment. Although the entire method did not need to be re-written, the aspect code changed the value of some of the local variables that were used in other parts of the method body. These variables were locally defined and thus inaccessible via the available join point constructs. Alternatively, we could have refactored the method by inserting a call to a dummy method and passing it the local state by reference; however, this would add a level of indirection and potentially impede performance.

Similar to the `standard_opt` aspect, the `init()` pointcut captures execution of the `initialize()` member function of the `simple_buffer` class. As noted, this function is responsible for setting a pointer to the network node interface to which this buffer belongs and for initializing various buffer statistics. It needed to be modified to use the Hybrid Buffer approach. To accomplish this, an `around` advice on this pointcut is declared effectively eliminating the original code and replacing it with the functionality that had existed in the `initialize` member function of the `hybrid_buffer` class.

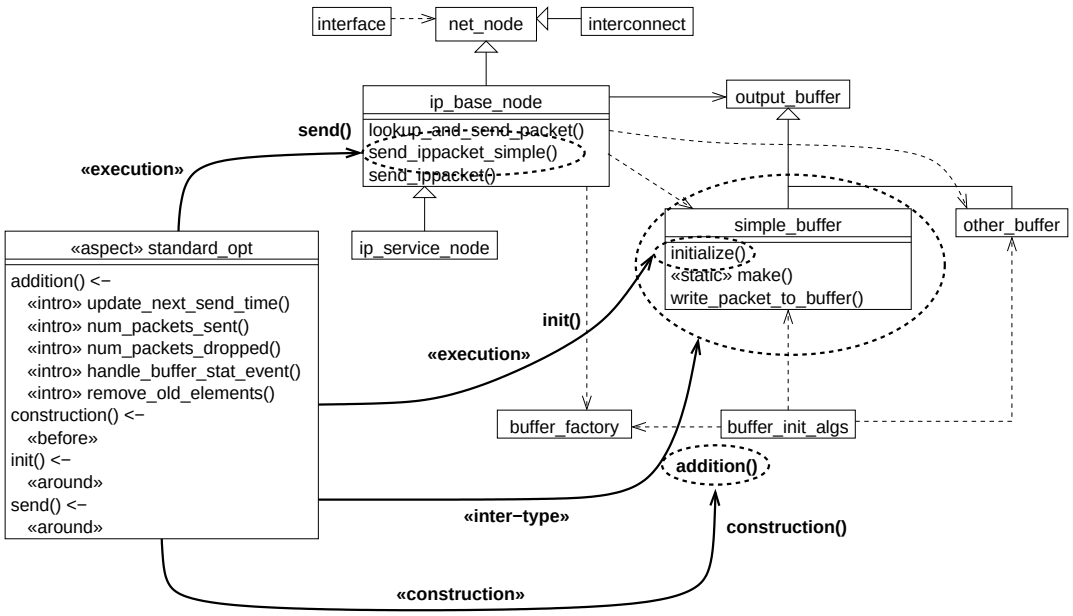

Figure 4: UML class diagram (modified for aspect-oriented model elements) for the AspectC++ realization of the Standard Optimized Buffer approach. The heavy arrows represent pointcuts, and the dotted ellipses to which they point indicate the elements that the pointcuts involve; the name of each pointcut and its kind are shown as annotations on the arrows, in boldface type. Within the aspect itself are listed the advices that apply to each pointcut. See the main text of Section 3.2 for more details.

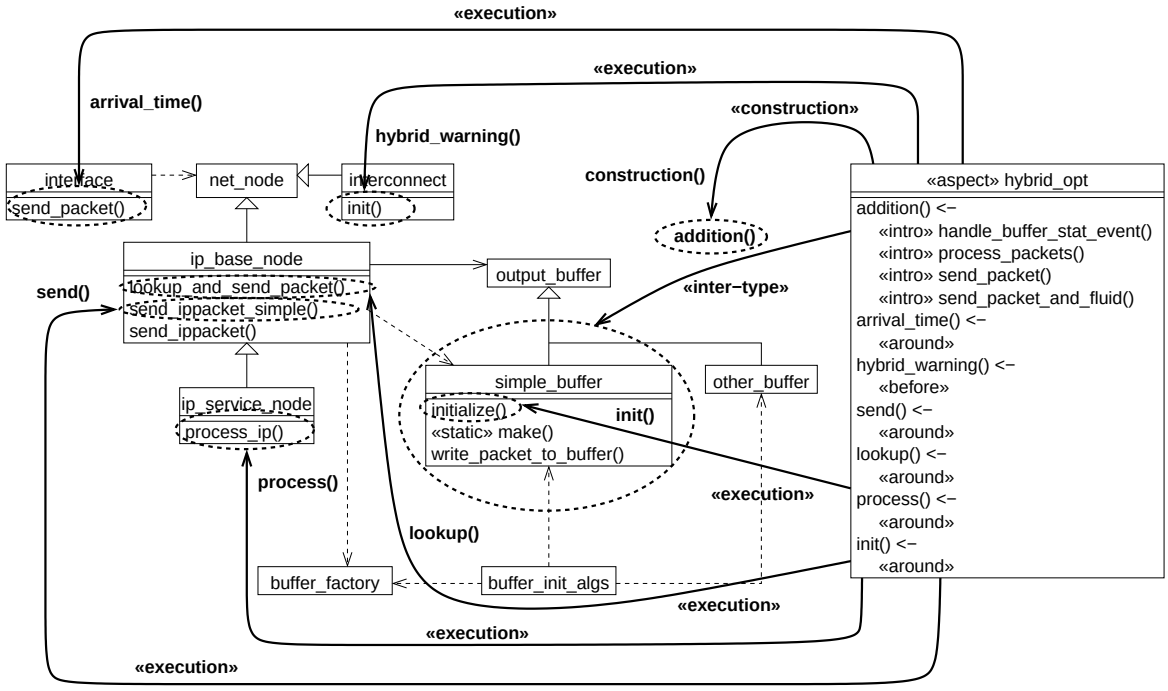

Figure 5: UML class diagram (modified for aspect-oriented model elements) for the AspectC++ realization of the Hybrid Buffer approach. See the main text of Section 3.3 for more details.

The `send()` pointcut captures execution of the `send_ippacket_simple()` member function on the `ip_base_node` class. The `send_ippacket()` was manually modified to remove mention of the Hybrid Buffer approach, and the `send_ippacket_hybrid()` helper function was removed from the `ip_base_node` class. This functionality was then restored as `around` advice on the `send()` pointcut; any attempt at executing `send_ippacket_simple()` results instead in the execution of the functionality of the original `send_ippacket_hybrid()`.

The `lookup()` pointcut captures the execution of the `lookup_and_send_packet()` member function in the `ip_base_node` class. When the Hybrid Buffer optimization is enabled, this function must include an extra error check and behaviour to ensure that the proper execution path for sending packets for Hybrid Buffers is followed. In addition, some Simple Buffer behaviour must be eliminated. Although the entire functionality of this method did not need to change, the method lacked natural join point "hooks" to advise. For this reason, we used an `around` advice and rewrote the method entirely. This issue is examined further in Section 5.4.

The `process()` pointcut captures the execution of the `process_ip()` member function in the `ip_service_node` class. When the Hybrid Buffer optimization is enabled, this function is responsible for processing the set of packets and fluid rate change advertisements that are packaged together in the same arrival event. This is achieved in a single `around` advice, where a loop is added around the entire function to process all packets and fluid rate change advertisements. At first glance, it would seem that this additional behaviour could have been achieved via paired `before()` and `after()` advice; however, one of the local values generated in the function itself was of crucial importance to the loop and there was no way of capturing this variable without inserting extra calls to dummy functions. As discussed in the preamble to this section, this alternative design was not considered a viable option as it violates several of our stated goals.

Certain special-purpose functionality (i.e., the provision of interconnects such as switches and hubs) had not been implemented in IP-TN to operate under the Hybrid Buffer optimization. The `hybrid_warning()` pointcut is utilized to inform the user who attempts to use such functionality that it is not supported under the Hybrid Buffer optimization. This is implemented with a simple `before` advice that prints a warning message at the initialization time of the `interconnect` class.

4. EMPIRICAL EVALUATION

To evaluate the run-time performance effects of the buffer optimizations, we performed a small set of characteristic simulations on both the C++ and AspectC++ implementations of IP-TN.

The basic simulation configuration consisted of a tandem network topology, as described in detail by Kiddle and colleagues [16]; this configuration is illustrated in Figure 6. The network model consists of a series of source nodes where traffic flows are generated, a series of sink nodes to where traffic flows are destined, and a set of routers that route the traffic flows to their designated sink. Links from the source nodes to the routers and from the routers to the sink nodes are modelled with a 1 ms propagation delay and 10 Mbps transmission capacity. Links between routers are modelled with a 5 ms propagation delay and a transmission capacity C such that the average network link load is 100%. The size of output buffers on the modelled routers is set such that the maximum queuing delay is 20 ms. We choose 100% average load for these experiments to ensure high buffer activity.

An exponential on/off process is used to model traffic flows. Sources generate traffic at a rate of 5 Mbps in the on state and generate no traffic in the off state. The sojourn times in each state are independently drawn from an exponential distribution such that each source spends 50% of the time in each state on average. This results in an average rate of 2.5 Mbps for each traffic source. The average burst size for each on period is 100 packets of size 576 bytes.

Traffic flows are divided into two categories: foreground flows and background flows. Foreground flows are the traffic flows of interest that are being studied. They traverse the whole network and are always modelled as packet flows. Background flows are traffic flows used to model other network activity and to compete for resources with the foreground flows. They can be modelled as either packet or fluid flows and generally traverse just one router. One background flow also traverses the entire network to study the impact of propagation of rate changes when background flows are modelled as fluid flows. For tests involving the Simple Buffer and Standard Optimized Buffer, all traffic flows are modelled as packet flows. For tests involving the Hybrid Buffer, foreground flows are modelled as packet flows and background flows are modelled as fluid flows.

The network model was originally developed to study the performance and behaviour of the Hybrid Buffer optimization under a wide range of conditions. For the purposes of this paper, we explore a subset of these conditions to provide an indication of how the performance of the AspectC++ and original implementations compare. One foreground flow was used for all experiments, since this would be a common configuration for simulations. To compare performance over several data points the number of background flows was varied from 4 to 32, and the number of routers was varied from 2 to 8.

All tests were performed on a Dell Optiplex GX240 with an Intel Pentium 4 1.60 GHz processor, 256 KB L2 cache, and 1 MB RAM, running Scientific Linux SL release 4.1 (Beryllium) with kernel version 1.3. IP-TN was compiled with g++ (GCC) 3.4.3 using the -O2 optimization flag. The AspectC++ implementation of IP-TN was preprocessed with ac++ version ac-0.9.2 and then compiled with the same configuration of g++. (A small set of tests were also performed using the -O3 optimization flag, but the results were statistically identical to the -O2 optimization results so we do not mention these further.) Each simulation configuration was run 10 times for 600 simulated seconds, with performance results averaged over them.

Figure 7 illustrates the average run time of various buffer optimizations with respect to number of background flows, for configurations with 4 routers; 95% confidence intervals were computed in all cases, but these are too small to be visible on the graphs (the largest of these is ± 9.97 s while most are in the neighbourhood of

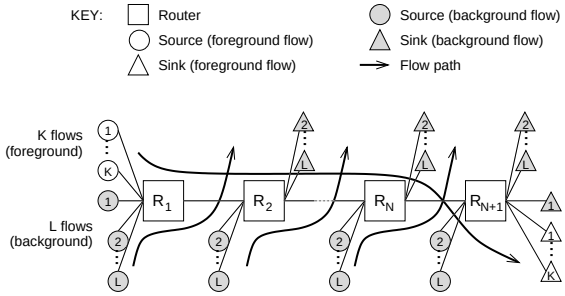

Figure 6: The tandem network model used in the simulation (from [16]).

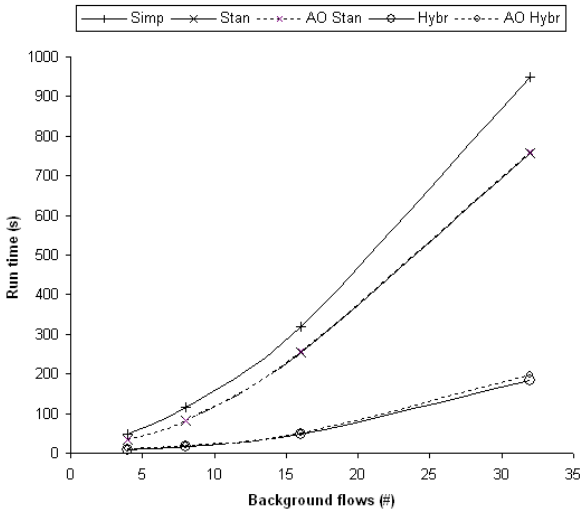

Figure 7: Run-time results for the tandem topology, for configurations with 4 routers and varying number of background flows.

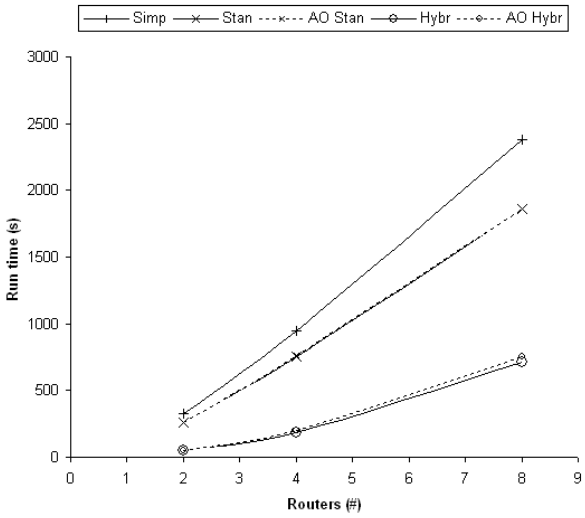

Figure 8: Run-time results for the tandem topology, for configurations with 32 background flows and varying number of routers.

± 1 s). Results for configurations with 2 and 8 routers are similar, so they are not illustrated. Figure 8 illustrates the growth in run time as the number of routers increases, with a constant 32 background flows. Figure 9 illustrates the speedup of each optimization relative to the Simple Buffer approach, for configurations with 4 routers and varying numbers of background flows. In all three figures, the abbreviations are as follows: **Simp** represents the results from the un-optimized, Simple Buffer approach; **Stan** represents the results from the Standard Optimized Buffer approach, as originally implemented, and **AO Stan** from the Standard Optimized Buffer approach realized as an optimization aspect; **Hybr** and **AO Hybr** are the equivalent results for the use of the Hybrid Buffer approach in the original and AspectC++ implementations respectively.

The Simple Buffer shows the worst performance as expected, with the Standard Optimized Buffer showing up to 1.5 times speedup over the Simple Buffer and the Hybrid Buffer showing up to 6.6 times speedup over the Simple Buffer. Simulation run time increases with increasing number of background flows for all of the buffer implementations as more traffic is simulated. This is due to the increased transmission capacity of the links between routers to maintain an average network load of 100% as the number of background flows is increased.

Although the Standard Optimized Buffer requires half as many events to model packet transmissions in comparison to the Simple Buffer, it does not achieve two times relative speedup as might be expected. This is because the execution cost of one kind of event may not necessarily equal that of other kinds of events. Also, not all of the events model packet transmissions; some model the generation of traffic flows. There are an equal number of the latter events for both the Simple Buffer and Standard Optimized Buffer simulation runs.

Relative speedup of the Hybrid buffer implementations increases with increasing number of background flows initially. This is due to the increased percentage of traffic modelled as fluid flows resulting in fewer events simulated. As the number of background flows is increased further, the relative speedup of the hybrid implementations begins to decrease. This is due to what is known as the

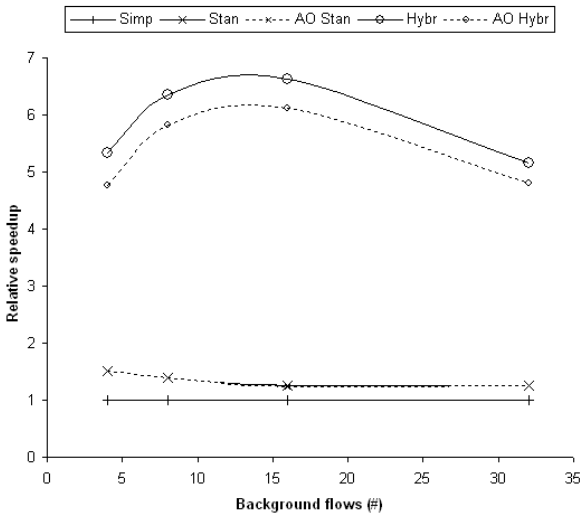

Figure 9: Speedup of optimizations relative to the Simple Buffer approach, for configurations with 4 routers and varying number of background flows.

ripple effect[1] [20, 25]. At congested network nodes a change in the rate of one flow causes changes in the rates of all other active flows. These rate changes propagate down the network and create new rate changes which could result in an explosion of events.

Overall, the AspectC++ implementation performed comparably to the C++ implementation. With the Standard Optimized Buffer approach, the two implementations were generally indistinguishable in their performance. The Hybrid Buffer approach was slightly slower for the AspectC++ implementation, where the speedup of

[1]Not to be confused with the same term in change impact analysis.

the C++ implementation relative to the AspectC++ implementation had a mean value of 1.11 and a standard deviation of 0.05. The key difference between the two optimizations in the AspectC++ implementations was the greater use of `around()` advice for the Hybrid Buffer approach; we consider this issue further in the following section.

5. DISCUSSION

The performance results observed in our empirical study demonstrate that little overhead was added by using the AspectC++ approach. The small performance degradation observed might derive from any combination of three sources: (1) weaknesses in the implementation of the AspectC++ compiler, (2) poor optimization aspect design on our part, or (3) weaknesses in the aspect-oriented model at our disposal. In this section, we proceed to analyze and discuss these issues and others that arise from our case study. The discussion is organized into major issues, as follows: AspectC++ (Section 5.1), localization (Section 5.2), comprehensibility (Section 5.3), fine-grained join points (Section 5.4), and additional aspects and interactions (Section 5.5).

5.1 AspectC++

Inquiring with the AspectC++ development team, Daniel Lohmann informed us[2] about the following known issues.

- The overhead of any advice is mainly determined by the amount of join point context accessed by the advice code. Information such as the pointers returned by `that()`, `target()`, or `args()` are stored in a join point object. They require some stack space and some cycles for initialization. However, the actual calls to the join point objects are in-lined, and so multiple calls to access the same context will not increase overhead.
- The introduction of new methods and variables do not have an impact on the performance relative to the C++ implementation, as they are directly in-lined.
- The `before()` and `after()` advices that we have used lead to no overhead in CPU cycles or stack usage, as they are also directly in-lined.
- On the other hand, `around()` advice uses a specific action object for the implementation of `tjp->proceed()`. This action object occupies some stack space and results in a function pointer call that can not be in-lined. A faster method of implementing `around()` advices is currently being explored.

AspectC++ is not thread-safe. Since IP-TN provides parallel simulation capabilities for the sake of improving performance, AspectC++ would not suffice in its present form to support this functionality. Therefore, we avoided parallel simulation in our empirical evaluation.

Tool support for AspectC++ is not presently at an industrial standard: debugging remains difficult because of the source-to-source transformations performed by the preprocessor; support for files written in C is absent; occasionally incorrect C++ code is generated; and a fairly manual configuration process is necessary to specify the detailed project paths in use. While these issues must be addressed for AspectC++ to be industrially adopted, they are straightforward development issues. Regardless, the tool support is beyond the research prototype stage, and performed well for our application.

[2]Personal communication, 8 August 2005.

5.2 Localization

The clearest advantage that we observed to using aspects is the pluggability factor. Due to the localization of the optimizations, they could be easily incorporated into or removed from IP-TN. In our implementation we were careful to leave the base code intact and unaware of the aspect. This enabled us to have compile-time selection of the desired optimization without the need to scatter and tangle code wrapped in compiler directives.

One question is how much of this effect could be achieved through the modularization mechanisms available in C++, which are less stringent than those in Java. For example, if an aspect merely consisted of a set of complete member functions that happened to be scattered amongst several classes, C++ would permit these to be collected into a single file. However, the majority of the crosscutting behaviour that we observed in our case study consisted of finer-grained elements within member functions. As such, many advices that did not involve inter-type declarations were needed, and C++ could not help to separate and localize these.

5.3 Comprehensibility

The execution path of the buffer optimizations in the original implementation was difficult to understand due to its scattering and tangling. The buffers were instantiated via a call-by-name interface provided in the `hybrid_factory` class. The `ip_base_node` class implemented a selection mechanism to choose the appropriate buffer optimization and to make use of the functionality provided in the buffer classes via polymorphic function calls. The scattered and tangled nature of the Hybrid Buffer optimization is even more pronounced, due to the use of several code fragments wrapped in `#ifdef ENABLE_HYBRID` compiler directives. Implementing these optimizations as aspects clearly increased the locality of the concerns, and a specific representation of which code was involved in a given optimization.

Nevertheless, it is less clear that the comprehensibility of the optimizations was improved overall. Each aspect represented its corresponding optimization in its totality, resulting in large aspect implementations (approximately 0.5 kloc and 1.5 kloc respectively for the Standard Optimized Buffer and Hybrid Buffer optimizations). Individual advices tended to represent detailed behavioural modifications that were difficult to interpret in isolation. The execution paths that were difficult to understand in the original system remain obscure, as the optimizations did not encapsulate them completely, merely the portions that were modified; reference to the base code continued to be necessary.

Several arguments could be made regarding these observations. Perhaps optimizations do not represent sufficiently cohesive concerns to be reasonable candidates for aspectization. However, the buffer optimizations we have encountered here do crosscut and have specific purposes ascribed to them. Thus, they represent crosscutting concerns, and as such, fall under the claims of improved modularization made by AOSD. Given that we only separated two optimization aspects, but many other crosscutting concerns exist within IP-TN, perhaps any shortcomings that exist are due to the scattering and tangling of these other concerns. However, one would expect separation of some crosscutting concerns to provide partial benefit, as has been claimed elsewhere with other crosscutting concerns. In contrast, it is not clear that comprehensibility has changed significantly in our situation, either positively or negatively.

5.4 Fine-Grained Join Points

As we have alluded to earlier, the support provided by AspectC++ for capturing natural fine-grained join points in the IP-TN system

is insufficient. The `within()` primitive pointcut can be used to capture function calls made from within specific member functions, but this does not suffice when a function is called several times within a specific member function and only one of these calls is to be advised.

Furthermore, situations where compiler directives are used to insert unbalanced parentheses at seemingly arbitrary locations within the body of a function (e.g., to enclose a set of existing statements within the true-block of a new if-statement) remain difficult to cope with, especially when local state must be exposed. While in most contexts, the code to be enclosed could be refactored into a helper function and local state exposed in a call to this function, inserting such extra levels of indirection can undo the benefit to be gained from an optimization.

The simplest alternative in our context was to apply `around()` advice to the function to be modified, copy-and-paste the original implementation, and insert the additional behaviour into the copy. In our case study, following this procedure resulted in situations where the function to be advised consisted of $>$100 loc, and the behaviour to be added could be implemented in only a line or two. This is, of course, not a desirable approach to take in general as code replication leads to difficulties in debugging and evolution.

Better support for fine-grained join points might be necessary to successfully support systems software fully.

5.5 Additional Aspects and Interactions

Many crosscutting concerns can be observed within IP-TN that represent targets for modularization using an aspect-oriented approach. Preemptive buffers and their corresponding optimizations could also be implemented in an AspectC++ and their performance could be measured and compared. In addition to being a simulator, IP-TN has extensions that let it function as an emulator. This means that it can communicate with real hosts in real networks. The implementation of the emulator extensions spans across more than 20 files. An emulation aspect could modularize this implementation to a great extent. Such potential aspects remain future work to investigate.

In Section 3, we described how the original implementation of IP-TN allows different buffer optimizations to be applied to individual buffer instances within a given simulation, whereas our compile-time approach applied the same optimization to every buffer. Advanced simulation scenarios can benefit from such a fine-grained optimization, to improve performance where detailed simulation is not really needed. It is not obvious how the original design of multiple, optimized buffer classes could be improved upon without degrading performance with aspect instances. This remains a difficult question for future work.

While we have noted that the Hybrid Buffer optimization is implemented within code enclosed in compiler directives, we have not mentioned that `ENABLE_HYBRID` is but one of over 25 such flags used by IP-TN and the underlying simulation kernel. Not all combinations of these flags are meaningful, but there are over 100 potentially useful variations of the simulation kernel flags alone. Ideally, one should be able to provide one aspect to correspond to each flag; however, the interactions between these is non-trivial, so providing cohesive but correct aspect implementations for all of them might be challenging for aspect interaction research.

6. RELATED WORK

Some research has previously considered the role of modularity in operating systems (OS), using non-AO mechanisms. Levin and colleagues constructed Hydra [19], an OS that explicitly separated mechanism concerns from policy concerns to permit user-level specification of policy while the implementation of those policies was provided safely within the kernel. Various others followed this trend towards attempts at combining efficiency with separation of concerns (e.g, [27, 4, 2, 10]). Denys and colleagues provide a recent survey [9]. Yokote attempted to provide an OS, Apertos, in which computational reflection was key to providing customizability [37]; unfortunately, performance was so poor that the OS community has largely turned their backs on the ideas.

Work has been done in identifying and aspectifying emergent crosscutting concerns in systems contexts. Coady and colleagues developed an AspectC prototype and used it to modularize prefetching in the FreeBSD operating system across several versions [6, 7]. They have expanded their work to modularize page daemon activation, disk quotas, and blocking in device drivers [5]. Barreto, Åberg and their colleagues have heavily used AO approaches in developing the Bossa OS kernel [3, 1], with particular attention paid to crosscutting concerns involving temporal patterns. Mahrenholz and colleagues used AspectC++ to modularize interrupt synchronization in the PURE operating systems [22]. They have also created an AspectC++ implementation of program instrumentation which is used in the PURE operating systems for debugging and monitoring purposes [23]. Ségura-Devillechaise and colleagues [30] created a prototype system (μDyner) for the dynamic weaving of AOP in a running C program; they used μDyner to implement web cache prefetching policies and dynamically place them in web caches. Similar to the previous work in traditional approaches to structuring OSes, Lohmann and colleagues describe the connection between non-functional and architectural properties in the domain of OS product lines [21]. None of these approaches addresses the conceptually simpler but technically more problematic optimization aspects with which we are concerned here.

Various work has considered alternative design approaches for OSes to take advantage of AO properties. Schwanninger and colleagues propose an alternative architectural approach to deal with crosscutting concerns in systems contexts [29]. A framework for weaving aspects into real-time operating systems was proposed by Park and colleagues [26], where the aspects have been designed in a hierarchical fashion. Tešanović and colleagues propose a design strategy that decomposes real-time systems into components and aspects [35]. A component-based embedded real-time database system (COMET) has been developed and real-time policies are incorporated into this system as aspects. In contrast to these ideas, convincing the systems community to adopt fundamentally different design approaches so that optimizations can be conveniently modularized would be difficult; a demonstration of detailed improvement would be necessary rather than a demonstration that an AO design approach is merely possible—lessons from Apertos must be remembered.

Surprisingly little work can be publicly found that addresses systems performance issues in the AOSD community. Coady and colleagues simulated limited micro-benchmarks to give an indication of likely performance penalties, should their AspectC ever develop sufficiently [5]; this is not a perfect indicator of end-to-end performance. Schult and Polze discuss speed versus memory usage in components [28]. Lohmann has indicated to us[3] that the AspectC++ development team is working on a micro-benchmark suite and hope to publish on it in the near future.

Limited work has addressed the need for join points at the sub-method-level granularity. In the early days of the formation of the AOSD community, discussions about this finer granularity occurred but seem to have been pushed aside, perhaps for practi-

[3]Personal communication, 8 August 2005.

cal issues (one must start somewhere, after all) or philosophical ones. Murphy and colleagues point out the difficulty in separating crosscutting concerns that are tangled within basic control structures [24]. Schwanninger and colleagues note the limited scope and poor quality of available tools, hinting at their inability to cope effectively [29]. Engel and Freisleben question the lack of join points for capturing native code execution [11]. Sullivan and colleagues have recently argued that existing pointcut mechanisms in AspectJ were inadequate to provide true obliviousness, and promoted the introduction of explicit hooks in the base code [34]. While such explicit hooks are reasonable short-term workarounds in most contexts, they seem inappropriate here. Designing up-front for optimizations raises the spectre of premature optimization, wasted effort, and weaker code. Invasively modifying the base code after-the-fact raises the question of the usefulness of the aspect-oriented modularization: if manual optimization is only partially eliminated, it is unclear that partial benefit will be obtained.

7. CONCLUSIONS

For the systems software community to adopt aspect-oriented approaches, it must be a clear win for them. Run-time performance must be maintained while other properties—such as (un)pluggability and comprehensibility of optimizations—are improved.

We have examined how well current AOSD state-of-the-art can achieve these goals, by considering fine-grained optimizations in the context of an optimized network simulator. Two optimizations in the original C++ implementation were mimicked in AspectC++, in an attempt to apply the optimizations in an after-the-fact manner. Base code was kept oblivious to the presence of the optimizations, both to avoid invasive modification to support the optimizations and to avoid the addition of levels of indirection.

The performance of the AspectC++ implementation was comparable to that of the C++ implementation overall. A small degradation was observed to be correlated with frequent use of `around()` advice, which is a known target for improvement by the AspectC++ development team. The ability to plug and unplug the optimizations is clearly improved in the AspectC++ implementation, due to localization; however, it is not clear that the aspect solutions provide better comprehensibility, as they are large and refer to fine-grained details of the base code that are difficult to abstract away. We found that improved support for fine-grained join points would be helpful, to improve the ability to maintain performance without copy-and-paste programming.

A cleaner design for the base code is another possible route forward, but given the complexity of the system and the large numbers of tightly interacting optimizations present, this remains an idealization that requires a major research undertaking to evaluate fully. In the meantime, while these smaller-scale results will not convince the systems software community to adopt AOSD techniques, they should be encouraging for AOSD researchers that aspect-orientation in the context of optimized systems software remains a viable target.

8. ACKNOWLEDGMENTS

We thank Reid Holmes, Mark McIntyre, and Kevin Viggers for their comments on early drafts, and Carey Williamson and Anirban Mahanti for initial discussions on the topic. This work was supported in part by an NSERC Discovery Grant.

9. REFERENCES

[1] R. A. Åberg, J. L. Lawall, M. Südholt, and G. Muller. Evolving an OS kernel using temporal logic and aspect-oriented programming. In *Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD*, 2003. Position paper.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, 1992.

[3] L. P. Barreto, R. Douence, G. Muller, and M. Südholt. Programming OS schedulers with domain-specific languages and aspects: New approaches for OS kernel engineering. In *Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD*, 2002. Position paper.

[4] D. D. Clark. The structuring of systems using upcalls. In *Proc. ACM Symposium on Operating Systems*, pages 171–180, 1985.

[5] Y. Coady and G. Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In *Proc. International Conference on Aspect-Oriented Software Development*, pages 50–59, 2003.

[6] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. Structuring operating system aspects. *Communications of the ACM*, 44(10):79–82, 2001.

[7] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Joint Proc. European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 88–98, 2001.

[8] R. Curry, R. Simmonds, and B. Unger. Modelling differentiated services in conservative PDES. In *Proc. 2003 Winter Simulation Conference*, pages 658–666, 2003.

[9] G. Denys, F. Piessens, and F. Matthijs. A survey of customizability in operating systems research. *ACM Computing Surveys*, 34(4):450–468, 2002.

[10] P. Druschel, L. L. Peterson, and N. C. Hutchinson. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Proc. IEEE International Conference on Distributed Computer Systems*, pages 512–520, 1992.

[11] M. Engel and B. Freisleben. Using a low-level virtual machine to improve dynamic aspect support in operating system kernels. In *Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD*, 2005. Position paper.

[12] M. L. Fowler. Yet another optimization article. *IEEE Software*, 19(3):20–21, 2002.

[13] Information technology—open systems interconnection—basic reference model: The basic model. Standard ISO/IEC 7498-1, International Organization for Standardization, 1994.

[14] G. Kesidis, A. Singh, D. Cheung, and W. Kwok. Feasibility of fluid event-driven simulation for atm networks. In *Proc. IEEE Global Telecommunications Conference*, pages 2013–2017, 1996.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. European Conference on Object-Oriented Programming*, pages 327–353, 2001.

[16] C. Kiddle, R. Simmonds, C. Williamson, and B. Unger. Hybrid packet/fluid flow network simulation. In *Proc. 17th*

Workshop on Parallel and Distributed Simulation, pages 143–152, 2003.

[17] C. Kiddle, R. Simmonds, D. K. Wilson, and B. Unger. ANML: A language for describing networks. In *Proc. Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 135–141, 2001.

[18] R. Laddad. Aspect-oriented refactoring series: Part 1—overview and process. URL [accessed 18 August 2005]: `http://www.theserverside.com/articles/article.tss?l=AspectOrientedRefactoringPart1`, 2003.

[19] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proc. ACM Symposium on Operating Systems Principles*, pages 132–140, 1975.

[20] B. Liu, D. Figueiredo, Y. Guo, J. Kurose, and D. Towsley. A study of networks simulation efficiency: Fluid simulation vs. packet-level simulation. In *Proc. Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 3, pages 1244–1253, 2001.

[21] D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat. On the configuration of non-functional properties in operating system product lines. In *Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD*, 2005. Position paper.

[22] D. Mahrenholz, O. Spinczyk, A. Gal, and W. Schröder-Preikschat. An aspect-oriented implementation of interrupt synchronization in the PURE operating system family. In *Proc. 5th ECOOP Workshop on Object Orientation and Operating Systems*, 2002.

[23] D. Mahrenholz, O. Spinczyk, and W. Schröder-Preikschat. Program instrumentation for debugging and monitoring with AspectC++. In *Proc. 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 249–256, 2002.

[24] G. C. Murphy, A. Lai, R. J. Walker, and M. P. Robillard. Separating features in source code: An exploratory study. In *Proc. International Conference on Software Engineering*, pages 275–284, 2001.

[25] D. Nicol, M. Goldsby, and M. Johnson. Fluid-based simulation of communication networks using SSF. In *Proc. European Simulation Symposium*, pages 270–274, 1999.

[26] J. Park, S. Kim, and S. Hong. Weaving aspects into real-time operating system design using object-oriented model transformation. In *Proc. IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 292–298, 2003.

[27] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[28] W. Schult and A. Polze. Speed vs. memory usage: An approach to deal with contrary aspects. In *Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD*, 2003. Position paper.

[29] C. Schwanninger, E. Wuchner, and M. Kircher. Encapsulating crosscutting concerns in system software. In *Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD*, 2004. Position paper.

[30] M. Ségura-Devillechaise, J.-M. Menaud, G. Muller, and J. L. Lawall. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In *Proc. International Conference on Aspect-Oriented Software Development*, pages 110–119, 2003.

[31] R. Simmonds, R. Bradford, and B. Unger. Applying parallel discrete event simulation to network emulation. In *Proc. 14th Workshop on Parallel and Distributed Simulation*, pages 15–22, 2000.

[32] R. Simmonds, C. Kiddle, and B. Unger. Addressing blocking and scalability in critical channel traversing. In *Proc. 16th Workshop on Parallel and Distributed Simulation*, pages 17–24, 2002.

[33] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *Proc. International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific)*, pages 53–60, 2002.

[34] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *Joint Proc. European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 166–175, 2005.

[35] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Aspects and components in real-time system development: Towards reconfigurable and reusable software. *Journal of Embedded Computing*, 1(1), 2004.

[36] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy. An initial evaluation of aspect-oriented programming. In *Proc. International Conference on Software Engineering*, pages 120–130, 1999.

[37] Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 414–433, 1992.