A Domain-Specific Language Approach to Programmable Networks

Gilles Muller[*], Julia L. Lawall[†], Scott Thibault[‡], Rasmus Erik Voel Jensen[§]

Abstract

Active networks present significant safety, security, and efficiency challenges. Domain-specific languages, *i.e.* languages providing only constructs relevant to a particular domain, provide a solution that balances these constraints. Safety and security can be ensured using verification techniques that exploit the restricted nature of such languages. Strategies have been developed for the compilation of domain-specific languages that provide both portability and efficiency.

This paper presents a survey of work on the PLAN-P domain-specific language for programmable routers. We present the language design, representative experiments that have been carried out using the language, and new compilation strategies. End-to-end performance is typically comparable to that of hand-coded C implementations.

1 Introduction

With the increasing use of the Internet for distributed applications, comes the increasing need to control how application data is transmitted through the Internet. Programmable routers provide a means of adding new network-related functionalities to existing applications, often completely transparently. The introduction of programmability into networks, however,

[*]G. Muller is with the Ecole des Mines de Nantes, Nantes, France (e-mail: `Gilles.Muller@emn.fr`).

[†]J.L. Lawall is with the University of Copenhagen, Copenhagen, Denmark (contact author, e-mail: `julia@diku.dk`).

[‡]S. Thibault is with Green Mountain Computing Systems, Inc., Richmond, VT, USA (e-mail: `thibault@gmvhdl.com`).

[§]R.E. Voel Jensen is with the University of Copenhagen, Copenhagen, Denmark (e-mail: `voel@diku.dk`).

raises a number of issues [24]: (i) *safety*, because distributed systems are complex and difficult to implement and debug; (ii) *security*, because network routers are shared resources; (iii) *portability*, because protocols have to be executed on multiple router architectures; (iv) *performance*, because the execution of active protocols must not degrade the router bandwidth.

Existing active network implementations address these issues to some degree. In *platform specific* solutions [14], active protocols are developed in the C language at the kernel level. Although highly efficient, this approach requires highly skilled programmers, is error-prone, and is not portable. *Java-based* solutions [28, 29] are easier to program and ensure portability. Nevertheless, the efficiency of Java is still an issue. Also, the associated run-time system must check network-specific safety and security properties of active protocols. An alternative approach is offered by *Domain-Specific Languages* (DSLs). A DSL is a language that provides high-level abstractions dedicated to a single domain. These abstractions simplify programming and permit static verification of safety and security properties. The first DSL for active networks was the PLAN language, developed by Hicks *et al.* [11]. Since 1997, we have been working on the PLAN-P language [7, 10, 25, 26], which extends PLAN with features that allow the programming of efficient application-specific programmable network protocols.

This paper

This paper presents a survey of the development of the PLAN-P DSL and describes an illustrative range of experiments that have been carried out using PLAN-P. The design of PLAN-P is particularly geared towards providing both safety and efficiency. Safety is enhanced by the downloading of protocols as source code into routers, enabling router checking of both general and router-specific safety properties, based on knowledge of the behavior of the high-level abstractions of the DSL. Efficiency is provided by JIT compilation of protocols to native code on the router. To eliminate the need to implement an optimizing compiler for every possible router architecture, we generate the JIT compiler automatically from an interpreter, which is both straightforward to implement and highly portable. Overall, we find that the use of PLAN-P permits simple and concise implementations of useful and verifiable network protocols, and that these protocols provide performance competitive with that of hand-coded C solutions.

The main points of this paper are as follows:

- We present an overview of the PLAN-P language and associated run-time system.
- We present a selection of protocols that have been implemented using PLAN-P: a learning bridge, a load balancing web server, and in-network adaptation of MPEG video transmission. These experiments have been reported previously [7, 10, 25, 26], but presented together give an idea of the range of functionalities that can be provided by PLAN-P.
- Our experiments with these protocols show that programmable routers provide a realistic solution to improve network utilization and performance. For example, the PLAN-P learning bridge gives bandwidth identical to that of a C implementation, and the adaptation of MPEG video significantly improves the quality of the video received by a client in a heavily loaded network.
- We present new strategies for the JIT compilation of PLAN-P programs. These strategies both improve the performance and reduce the size of the generated code, without giving up the advantages of an automatically generated compiler. This aspect of the implementation of PLAN-P has not been previously published.

The rest of this paper is organized as follows. Section 2 presents the architecture of the PLAN-P system, the syntax of the language, and a collection of verifiable safety and security properties. Section 3 describes several applications of PLAN-P and gives performance results. Section 4 presents the basic approach to generation of the PLAN-P compiler and some new improvements to this approach. Section 5 describes related work and Section 6 concludes.

2 PLAN-P

The design of PLAN-P is based on the idea of programmable routers located at critical places within the network. Protocols are written in the PLAN-P DSL and downloaded into routers by trusted users, such as LAN administrators and registered service providers. Once loaded, a protocol remains in the router, where it treats all relevant incoming packets. The rest of this section first presents the PLAN-P architecture in more detail, then presents the

PLAN-P DSL, and finally describes some safety and security properties that can be proved for protocols written in this language.

2.1 Architecture

PLAN-P has been implemented as a loadable kernel module for a Sparc running Solaris and for PowerPC running Linux, and as a dedicated PLAN-P OS kernel for a PowerPC.

The architecture of PLAN-P when implemented as a kernel module is illustrated in Figure 1. This architecture is based on IP and allows PLAN-P protocols to be used to build applications based on IP, UDP, or TCP. The PLAN-P system does not require any changes to existing packet formats, and thus PLAN-P routers operate seamlessly within existing networks (i.e., PLAN-P does not require every router to support PLAN-P). PLAN-P protocols are downloaded into the IP/PLAN-P layer on the end-host machines and/or any number of routers in the network. These protocols replace the standard packet processing behavior of the IP layer for certain packets in order to perform application-specific processing.

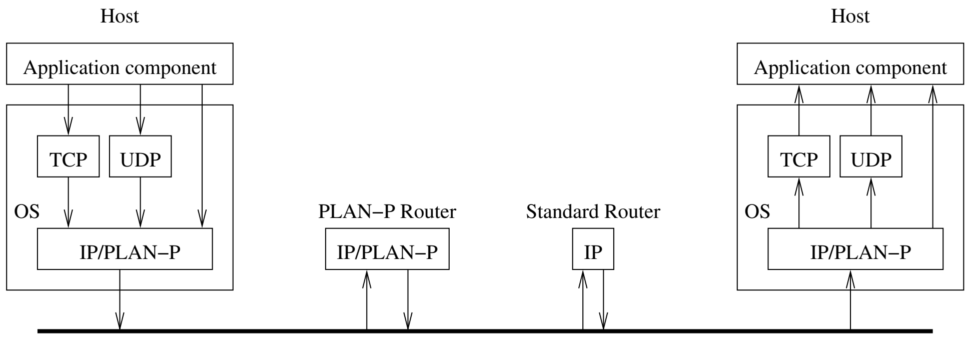

Figure 1: PLAN-P software architecture

The dedicated PLAN-P OS, developed in collaboration with Jean-Philippe Fassino and Jean-Bernard Stefani of France Telecom Research, is based on components designed using the Think software framework [7]. The OS consists only of components that treat IP, UDP, and TCP packets. Incoming packets are forwarded directly to the PLAN-P protocol.

2.2 The PLAN-P DSL

The goal of the PLAN-P DSL is primarily to express routing strategies and to update packet contents. PLAN-P protocols rely on a set of *resident services* that provide complex data manipulation operations, and are defined as primitive operations by an underlying resident abstract machine. We now present the features of the PLAN-P DSL, using excerpts of a protocol that provides adaptation for MPEG video distribution as an example. The design of this protocol is considered further in Section 3.1.

A PLAN-P protocol defines a set of *stream* functions that control transmission through the router. Separate streams can be defined for network traffic with distinct characteristics, *e.g.* TCP packets, UDP packets, etc. Named streams can be defined using the IP option mechanism. The MPEG protocol uses two streams: one that manages the installation of new movies and client subscriptions, and another than controls the video distribution. The latter is defined below:

```
typedef struct {
  ip ipheader;
  udp udpheader;
  int ver;
  int seq_no;
  int pictype;
  int frame_no;
  int payload_len;
  blob payload;
} packet;

stream network(packet p) {
 try {
  movie_data m =
   find(movie_table,hashkey(ipDst(p->ipheader)));
  adaptiveForward(m,p);
 }
 handle NoEntry => OnRemote(network, p);
}
```

The name `network` and the parameter type `packet` together imply that this stream is applied to any UDP packet containing at least the five integers required by the definition of the `packet` structure. The stream uses the primitive `find()` to determine whether the destination IP address of the incoming packet is the multicast IP address of a known movie, as recorded in the hash table `movie_table`. If so, the function `adaptiveForward()`, which is defined in the PLAN-P protocol, is invoked on the information `m` about the properties of the movie and the current packet `p` to determine whether the packet should be forwarded. If the destination address is not that of a known movie, `find()` raises a `NoEntry` exception, causing the packet to be forwarded normally.

The PLAN-P run-time system provides a permanent state that persists across the treatment of successive packets. This state allows the treatment of the current packet to be determined by information derived from the treatment of previous packets. The permanent state of the MPEG protocol contains two hash tables, declared as follows:

```
initstate {
 dropped_info hash_table interface_table =
  mkTable(25);
 movie_data hash_table movie_table =
  mkTable(25);
}
```

The `dropped_info` and `movie_data` types are defined by the protocol. The `interface_table` hash table records information about the treatment of packets at each interface. The `movie_data` hash table records the information about the structure of each movie provided when a movie is installed.

The PLAN-P DSL provides network-specific types and library functions that facilitate the development of correct protocols. These include the types of standard packets headers, such as `ip` and `udp`, primitives for extracting components of such packet headers, and the primitives `OnRemote()` and `OnNeighbor()` that, respectively, forward a packet using the standard routing mechanism of the router forward a packet at a specific interface. Our examples also use resident services such as primitives for manipulating hash tables.

Our experience with PLAN-P has shown that it eases both the initial development and maintainability of protocols. Our original protocol for adaptive video treated only a specific movie. Extending this protocol to manage the installation of new movies and to manage client subscriptions required about one day for a programmer familiar with PLAN-P, but having no specialized knowledge of network or kernel programming.

PLAN-P was inspired by the PLAN DSL for programming network diagnostics [11]. Several features of PLAN-P originate with PLAN: the representation of protocols as source code to enable verifications, the view of the DSL as a glue language intended to connect complex operations predefined as resident services, and the original syntax of PLAN-P, which like that of PLAN was based on Standard ML. PLAN-P builds on PLAN, however, in numerous ways. PLAN-P uses a programmable router model, in which protocols are stored on routers, while PLAN uses an active network model, in which protocols are transmitted with their data in packets. The use of the programmable router model in PLAN-P enables PLAN-P protocols to maintain state across the treatment of multiple packets; this feature is needed by all of our examples in Section 3. The safety properties outlined in Section 2.3 are also stronger than those considered for PLAN. Finally, and most importantly, because PLAN execution is based on interpretation, it is much less efficient than PLAN-P execution, which is based on JIT compilation.

The syntax of the PLAN-P DSL presented above is based on that of C and was developed several years after the original syntax based on Standard ML [10]. Our primary motivation for developing a C-like syntax was to make the language more accessible to systems programmers. Nevertheless, as illustrated in Section 4, the imperative style of programming, including in particular the use of global variables rather than the argument passing favored by the Standard ML-like syntax, leads to more efficient compiled protocols as well.

2.3 Safety and security properties

Adding programmability to a system, as in active networks, is a general solution to the problem of providing extensible and adaptable interfaces between systems. A disadvantage of this approach, however, is that it introduces security risks [24]. Security is particularly important in the context of active networks because the network is a widely shared resource. In addition to security, we are also interested in the safety of active network programs. The distributed nature of network programming complicates debugging, thus implying the need for automatic checking of safety properties.

DSLs can provide programmability while at the same time satisfying security and safety demands. The key idea is to restrict the language in a way that the desired properties are satisfied *a priori*, or can be checked automatically by program analysis. A DSL is essentially a glue language that combines domain-specific behaviors, in the form of built-in primitives and constructs, into new functionalities. Since the primitives are semi-fixed, it is reasonable to prove complex properties about them by hand, once. Properties of DSL programs can then be more easily proved in terms of these properties. The remainder of this section presents three properties that can be automatically checked for PLAN-P protocols.

Termination: The PLAN-P language provides only bounded loops and does not provide recursion. These constraints are sufficient to ensure that the treatment of a packet within the code of a PLAN-P protocol always terminates. We must, however, also consider the treatment of a packet within the network. One solution is to maintain a time-to-live counter with each packet. Such a counter is decremented at each hop, and the packet is discarded when the counter reaches 0. This solution risks, however, the unintended discarding of a packet when the estimated time is insufficient.

With PLAN-P, we can use domain-specific knowledge to identify protocols that are guaranteed not to cause looping within the network. Such a proof is based on the following information:

- We assume that network routing tables do not contain loops. This assumption is appropriate because are a interested in proving the correctness of the protocol, not of the network.
- The PLAN-P routing primitive `OnRemote` chooses the next hop of a packet based on network routing tables.
- The PLAN-P primitive `IPDst` extracts the current destination of a packet.

This information is sufficient to identify protocols that always send a packet onwards toward its original destination, which is sufficient to ensure that the packet does not loop in the network. Looping is also not possible if a packet is sent to some destination other than its original destination, but is sent to a different stream, such that there is no mutual dependence between the current stream and the destination stream. Because the `OnRemote` primitive refers to the destination stream explicitly by name, the lack of such a dependency can be determined be straightforward analysis of the protocol.

Guaranteed packet delivery: As for the proof of termination, the proof that packets are guaranteed to be delivered relies both on assumptions about the network and on the understanding of the behavior of the DSL constructs. Here, we assume that the network does not itself lose packets. Given this assumption and the knowledge that it is not possible for a PLAN-P protocol to loop indefinitely, it remains to show that the packet is sent toward the destination on all control flow paths through the protocol. The analysis of explicit control flow constructs, such as conditionals and bounded loops, is straightforward. Control flow is also affected by exceptions; knowledge of the exceptions that can be raised by PLAN-P primitives and resident services is sufficient to check that all exceptions are handled and that all control-flow paths that lead through exception handlers also forward the packet correctly.

Safe packet duplication: Building on the verification strategy for the guaranteed packet delivery property, we can ensure that a protocol does not cause exponential packet duplication within the network if there is at most one occurrence of `OnRemote` or `OnNeighbor` along every control-flow path. This strategy is, however, too conservative, as it does not allow non-exponential behavior such as sending information back to the source at each hop. We thus allow more than one packet to be sent within a given control-flow path when the destination streams are disjoint and there is at most one mutual dependence between the destination streams and the current stream.

These verification strategies are conservative, meaning that they reject all protocols violating the given property, but they also reject some protocols that do not actually violate the property. In some cases, a protocol can be rewritten to make more information available to the verifier, for example by using multiple streams to record the state of a packet rather than encoding this information in state variables. Not all properties, however, are appropriate to all contexts. For example, restricted packet dropping is precisely the goal of the MPEG adaptation protocol presented in Section 3.1. As another example, a given router might want to prevent all packet duplication, rather than allowing safe packet duplication, as described above. Thus, the verifications that are performed depend on the intent of the protocol, which can be described by the service provider who makes the protocol available, and by the needs of the given router.

3 Experiments with PLAN-P

We now describe three of the application areas in which we have used PLAN-P: MPEG video broadcast [10], network routing [25], and web service [26]. All of these examples show that PLAN-P allows realistic functionalities to be implemented in a simple but efficient manner, requiring few or no changes to existing applications.

3.1 MPEG video broadcast

An important feature provided by programmable routers is the ability to observe local network conditions, and adapt routing accordingly, based on application-specific criteria. This form of adaptation responds rapidly to changing conditions, because there is no latency for feedback from the client to the server. Adaptation also respects the heterogeneity of the Internet, because adaptation performed in one router has no effect on clients in other part of the network. In this section, we present an approach to providing adaptation for a MPEG video broadcast system. We have also used PLAN-P to provide audio adaptation [26].

The high bandwidth requirements of video imply that video broadcast must adapt to network conditions, both to provide a reasonable quality for the client and to allow other network traffic to pass through the network. We define an adaptation strategy that uses frame dropping to avoid segment congestion. This strategy uses knowledge of the structure of MPEG video and of the history of the transmission process, as recorded in the permanent state of the PLAN-P protocol, to drop less important and undecodable frames.

MPEG video uses three kinds of frames: I, P, and B frames. Only an I frame contains a complete image and can be decoded independently. Decoding a P frame relies on the result of decoding the previous I or P frame. Decoding a B frame relies on the decoding of the nearest previous and successive I or P frames. An I frame and all of the frames that follow it, up to the next I frame, can be considered as a single unit, known as a *group of pictures* (GOP).

As the network load increases, router-buffer overflow causes packets to be randomly dropped. The goal of the protocol is to avoid such random dropping by pro-actively dropping less important packets and undecodable packets. The dropping strategy is thus based on the importance of the current frame and the available bandwidth. Because the decoding of a complete GOP depends on the decoding of the initial I frame, the adaptation policy never drops such frames. Because the decoding of the remainder of the current GOP depends on the decoding of each P frame, P frames are given high priority as well, and only dropped if the available bandwidth is less than the average bandwidth requirement of a P frame in the given movie. Because a B frame does not influence the decoding of any other frame, B frames have low priority, and are dropped if the available bandwidth is less than the total of the average bandwidth requirement of an I, P, and B frame.

In practice, a frame is too large to be transmitted as a single packet, and must be broken up into a sequence of packets by the server. The PLAN-P protocol thus extends this treatment of frames to the constituent packets. In particular, a packet is unconditionally dropped if a packet of the same frame has been dropped already.

We evaluate the performance of our adaptation policy in terms of three measures: the number of decodable frames received by the client, the amount of bandwidth wasted due to the transmission of undecodable frames, and the processing time within the router. These experiments are carried out using the Wireless LAN environment shown in Figure 2. The streaming server is a Sun 333Mhz Ultra Sparc IIi workstation with 384Mb RAM. The active router is a Sun 167Mhz Ultra Sparc workstation with 320Mb RAM. The client is a 700Mhz Pentium III laptop with 320Mb RAM, and is connected to the network via a Lucent 802.11 wireless Ethernet card. Only one wireless access point is included; more could be added to permit further experiments involving mobility. In order to produce congestion in the active router, we set the wireless access points with the maximum bandwidth of 2Mb/s. The experimental testbed also contains an additional load traffic generator and a client for this load traffic.

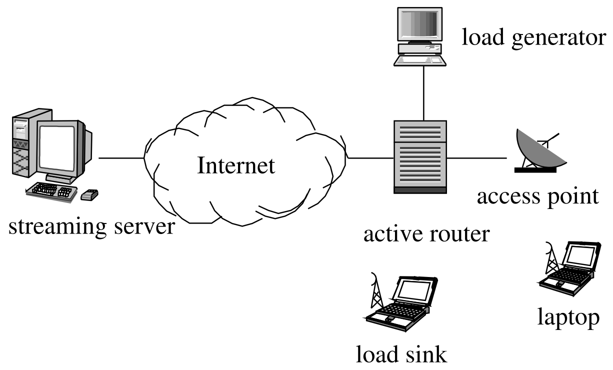

Figure 2: Experimental testbed

Our test data is a 25-second excerpt of *Star Wars*[1] which is transmitted at a rate of 30 frames per second. This excerpt includes scenes with a variety of different properties. The average bandwidth and size of each frame type are shown in Table 1. The thresholds for dropping P and B frames are 100kb/s and 300kb/s, respectively.

[1]http://www.wroclaw.tpsa.pl/users/starwars/mpegnew/astero1.mpg

Table 1: Profile of the Star Wars excerpt

Frame	Ave. BW (kb/s)	Ave. size (kb)	Total frames
GOP	400	71.4	748
I	91	45.7	50
P	137	17.1	200
B	172	8.6	498

Figure 3 shows the effect of adaptation on the percentage of movie frames that are received and decodable by the client. With a background load of 1650kbytes/s, the client can decode fewer than half of the movie frames when there is no adaptation and 70% of the movie frames when adaptation is used. With a very high load, when there is no adaptation, packets of many important frames are lost, and almost no frames are decodable in the client. With adaptation, many I frames are transmitted intact, and about 10% of the movie frames are thus decodable in the client.

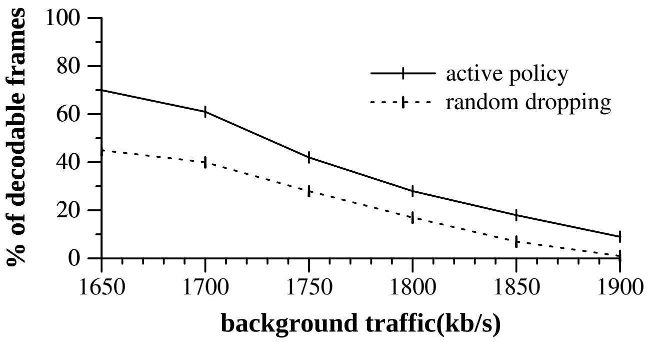

Figure 3: Decodable frame ratio

Transmitting undecodable frames wastes bandwidth that could be used by other applications. As shown in Figure 3, with a background load of 1750kbytes/s, only 40-50% of the movie frames are decodable by the client. Figure 4 shows that without adaptation, the client receives many frames that it cannot decode. In the extreme case, at 10s, a few I frame packets are lost, rendering all packets received during this second undecodable. When adaptation is used, the client can decode almost all of the packets that it receives.

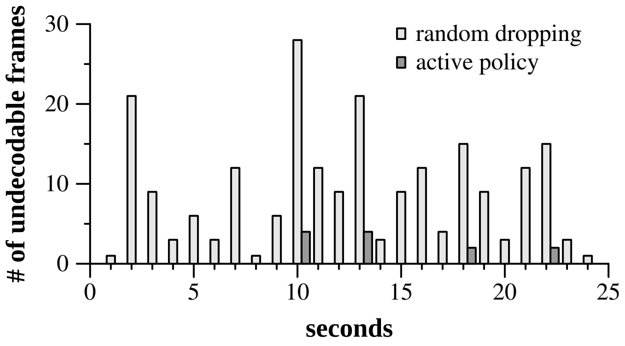

Figure 4: Undecodable frames received by the client (1750kb/s added load)

The cost of executing a PLAN-P program on a router affects the scalability of the approach. Because the policy is applied at every interface, the cost increases linearly in the number of interfaces. On the 167MHz Ultra Sparc workstation used in our experimental testbed, the average cost for a router with only one outgoing interface is 12μs, with little variation according to the available bandwidth. The incremental cost per additional interface is under 5μs. Using the dedicated PLAN-P OS for PowerPC, the average cost to transfer a packet to or from PLAN-P is 8μs and the average treatment time on a router with one outgoing interface is 4μs, for a total treatment time of 20μs per packet. A typical encoding rate used for television is 1.5Mb/s. In our architecture, the server divides frames into packets of at most 1440 bytes (11.52kb) of video data. This strategy implies that the active router receives about 130 packets per second for a single movie. Thus the total treatment cost is 2600μs per movie. This rate allows the treatment of up to 384 movies per second. For one incoming interface and seven outgoing interfaces, the average cost per packet within the protocol is 14μs while the OS cost is 64μs. This performance allows the treatment of up to 98 movies movies per second in the router.

3.2 A learning bridge

A bridge is a network node that connects multiple LANs to form one logical LAN. A simple bridge repeats every received packet on all other LANs connected to the bridge. A learning bridge keeps track of where packets come from in order to determine which LAN a host is connected to; packets for that host are subsequently only repeated on its LAN. This application has also been considered in work on the active network language PLAN [1], and emphasizes the requirement for good performance.

An extract of the PLAN-P implementation of the learning bridge is shown in Figure 5. The declaration of the packet type as containing an Ethernet header followed by some data (described as having type `blob`) implies that the protocol applies to any Ethernet packet. For a broadcast or multicast packet, the function `dumbRepeat` (not shown) is used to forward the packet at all interfaces other than the one at which the packet was received. For a unicast packet, the protocol records the source Ethernet address and the interface at which the packet was received in the permanent storage of the protocol. This is the learning portion. To forward the packet, the protocol checks whether there is any information in the table about the packet's destination. If an entry is found and it has not expired, the packet is forwarded on the interface recorded in the table. Otherwise, the packet is forwarded on all interfaces, as in the broadcast and multicast cases.

We first measure the performance of the learning bridge protocol in a 100Mbs Ethernet network where the bridge and two hosts are all 167 MHz Sun Ultra 1 Model 170 machines with 128Mb of main memory running Solaris 5.5. In this context, the compiled learning bridge protocol is about half as fast as a handwritten C learning bridge implementation. Nevertheless, we achieve the same bandwidth using both the PLAN-P and C implementations, and indeed both implementations achieve the same bandwidth as a direct connection. This is because processing a single packet takes less time than it does to physically transmit the packet.

Despite achieving the maximum bandwidth of the network, the throughput of PLAN-P when running as a Solaris kernel module is substantially less than that of a direct connection. Table 2 compares the throughput of the learning bridge when running as a kernel module in a standard OS with the throughput achieved with the dedicated PLAN-P OS [7]. In these experiments the clients are two 500MHz PowerPC G4 Macintoshes with 256Mb of main memory running Linux 2.2.18 connected by a bridge that is either the 167 MHz Sun Ultra 1 Model 170s mentioned above or a 350MHz PowerPC G4 Macintoshes with 128Mb of main memory running Linux 2.2.18. The results show that much of the degradation in throughput is due to the effect of the Solaris or Linux OS. With the dedicated PLAN-P OS, we achieve almost all of the throughput of a direct connection.

```
typedef struct {
  ethernet header;
  blob data;
} packet;

typedef struct {
  interface i;
  time t;
} packet_info;

stream network(packet p) {
  initstate {
    packet_info hash_table ss = mkTable(100);
  }
  etherhost dst = etherDst(p->header);
  if (isBroadcast(dst) || isMulticast(dst))
    dumbRepeat(p);
  else {
    time t = getTime();
    packet_info tmp =
      new(packet_info, thisInterface(), t);
    insert(ss, hashKey(etherSrc(p->header)), tmp);
    try {
      packet_info entry = find(ss, hashKey(dst));
      if ((t - (entry->t)) < timeLimit)
        OnNeighbor(network, entry->i, p);
      else
        dumbRepeat(p);
    }
    handle NoEntry => dumbRepeat(p);
  }
}
```

Figure 5: Extract of learning bridge

3.3 A load-balancing web server

We construct a load-balancing web server from a gateway, implementing a virtual server, and a collection of ordinary web servers. A client initiates a HTTP request by establishing a TCP connection to the (virtual) server. At this point the gateway selects a physical server for the request and records the relationship between the TCP connection and this physical server in the permanent state of the protocol. Subsequent packets making up the request are routed directly to the selected server. Responses also pass through the gateway. The gateway replaces the source address of each response with the address of the virtual server, so that load balancing is completely transparent to the client. Because no modification is required of either the client code or the server code, this approach provides a high degree

Table 2: Throughput of the learning bridge

bridge	throughput
none	91.5Mbps
PLAN-P/Solaris, 166MHz Sparc	42.0Mbps
PLAN-P/Linux, 350MHz Sparc	65.5Mbps
PLAN-P OS, 350MHz Sparc	87.6Mbps

of configurability. In particular, the protocol can easily be modified to reflect changes in the properties of the cluster of machines or to test new load-balancing strategies. Because ordinary web servers are used, the load-balancing server can easily be moved to a new cluster of machines.

We compare the performance of the PLAN-P load-balancing server to that of a load-balancing server implemented in C, as well as to the performance of HTTP servers without load balancing. For our tests we use a cluster of three Sun Ultra-1 170MHz workstations connected by a 100Mbits Ethernet network. The HTTP servers are running Apache version 1.2.6 [2] with 5 to 10 child processes. One machine is used exclusively as the gateway and the other two as web servers, to separate the cost of load balancing from the cost of serving HTTP requests. Load balancing is done by simply alternating between these two servers. The client machines are also Sun Ultra-1 170MHz workstations and are connected to the cluster by a 10Mbits Ethernet network. The complete web contents are replicated on both physical servers. Measurements are based on a real trace of 80000 accesses. Clients continuously issue requests so as to measure the maximum load that the clustered server can handle.

Results are presented in Figure 6. As shown by curves b and c, there is little or no difference in performance between the PLAN-P load-balancing server and the C load-balancing server. Furthermore, the PLAN-P load-balancing server treats up to 85% of the load of two servers with disjoint sets of clients and 1.75 times the load of a single server. Since both servers are saturated, the dropoff in performance as compared to two separate servers shows the impact of introducing a gateway as a contention point.

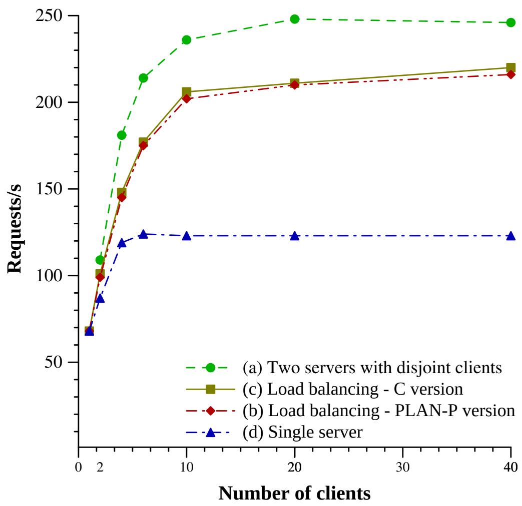

Figure 6: HTTP server performance

3.4 Assessment

These experiments show that PLAN-P can provide adaptation within the network, improved routing within the network, and new network-related functionalities at end-points. Protocols can also combine these features, to provide overall better performance. For example, the MPEG adaptation protocol also performs client management; the protocol uses information about client locations to avoid forwarding multicast data to interfaces at which there are no clients. This selective forwarding further reduces bandwidth usage. All of the protocols are concise (at most about 200 lines, in the case of the MPEG adaptation protocol) and are easily understandable. Nevertheless, although protocols are expressed at a high level, our experiments show that PLAN-P provides good performance, due to the use of JIT compilation. Performance is further improved when using the dedicated PLAN-P OS, which removes the cost of all processing unrelated PLAN-P.

4 Efficient JIT Compilation

Performance is critical for programmable routers, so that the processing used to improve or extend specific applications does not degrade ordinary network traffic. Thus, router protocols must be compiled. Nevertheless, because network routers are heterogeneous, the language implementation must be portable. Furthermore, it must be possible for the router to verify safety and security properties, which for maximum flexibility implies that protocols must be downloaded as source code and compiled on the router. These considerations imply the need for a portable approach to efficient JIT compilation on the router. Our solution is to automatically generate a JIT compiler from an interpreter, using a technique known as *partial evaluation* [12]. In the rest of this section, we present the structure of the PLAN-P interpreter, introduce partial evaluation, and describe some optimizations to the standard partial evaluation process that were needed to obtain good performance.

4.1 Structure of the interpreter

The PLAN-P interpreter is structured in two layers: the lower layer being an abstract machine providing domain-specific mechanisms and the upper layer being an interpreter that processes the language syntax. This structure provides a clean separation between *mechanisms*, implemented by the abstract machine, and *policies*, created by combining these mechanisms in a verifiable way using the language syntax. The lower layer is implemented as a stack machine, and the upper layer is implemented as a recursive traversal of the syntax. A much-simplified example of such an interpreter, providing only addition expressions, is shown in Figure 7. This interpreter structure has also been used in the implementation of a language for specifying video device drivers [27].

4.2 Program specialization

Program specialization is an automatic technique that automatically optimizes a program with respect to partial information about its inputs. Optimizations include aggressive interprocedural constant propagation, loop unrolling, and reduction of conditionals. Specialization of an interpreter with respect to a program has the effect of compilation [8]. We use the program specializer Tempo, which performs specialization at either compile time or run time [5, 6]. Figure 8 shows the result of compile-time specialization of the interpreter of Figure 7 with respect to the expression `3 + 4`. The generic interpretation function `interp` has been replaced by a collection of functions, each implementing a specific use of a single construct. In the context of PLAN-P, compile-time specialization is useful for observing the effect of specialization at the source-code level, but direct generation of executable code, *i.e.* run-time specialization, is needed to minimize the compilation cost.

```
struct cst {
  int val;
};

struct add {
  struct expr *e1;
  struct expr *e2;
};

#define CST 0
#define ADD 1
struct expr {
  int type;
  struct cst *c;
  struct add *a;
};

/* upper layer: syntax processing */
void interp(struct expr *e) {
  if (e->type == CST)
    cst(e->c->val);
  else {
    interp(e->a->e1);
    interp(e->a->e2);
    add();
  }
}
```

```
/* lower layer: abstract machine */
void cst(int i) {
  push(i);
}

void add() {
  int v1,v2;
  v2 = top();
  pop();
  v1 = top();
  pop();
  push(v1+v2);
}
```

Figure 7: An interpreter for arithmetic expressions

To reduce the run-time cost of run-time specialization, Tempo prepares much of the specialization process offline [6, 21]. The compile-time phase begins by identifying *static* constructs, which only depend on the source program and the known inputs, and *dynamic* constructs, which also depend on the unknown inputs. In the context of PLAN-P, the source program is the PLAN-P interpreter and the input that is known at specialization time is the PLAN-P protocol. The unknown input is the packet and the contents of the PLAN-P permanent state. Based on the identification of static and dynamic constructs in the source program, Tempo constructs a dedicated run-time specializer consisting of a *template file* and a *specializer*. The template file contains the dynamic constructs, and is compiled using a standard compiler (`gcc`) to produce object code fragments from which the specialized program is constructed. The specializer contains the static constructs, interspersed with calls

```
void interp_e() {
  interp_e1();
  interp_e2();
  add_e();
}

/* compute 3 + 4 */
void add_e() {
  int v2 = top_v2();
  int v1 = top_v1();
  push_v(v1 + v2);
}

void push_v(int i) {
  stack[1] = i;
}

int top_v2() {
  return stack[2];
}

int top_v1() {
  return stack[1];
}
```

```
/* evaluate 3 */
void interp_e1() {
  cst_3();
}

void cst_3() {
  push_3(3);
}

void push_3(int i) {
  stack[2] = i;
}

/* evaluate 4 */
void interp_e2() {
  cst_4();
}

void cst_4() {
  push_4(4);
}

void push_4(int i) {
  stack[1] = i;
}
```

Figure 8: Specialized interpretation of `3 + 4`

to architecture specific code-generation macros. Generation of specialized code is performed at run-time by applying the specializer to the known inputs. The specializer executes the static constructs, selects and emits compiled templates, and instantiates the emitted code with the values of static subexpressions and branch offsets.

The use of the standard compiler `gcc` to generate the machine code used during the run-time specialization makes Tempo's approach to run-time specialization highly portable. Further portability is provided by the fact that Tempo does no optimization of the code generated by specialization. Indeed, Tempo has been implemented for three different architectures: Sparc, PowerPC, and Pentium. Nevertheless, while the lack of optimization of the specialized code enhances portability, it implies that significant inefficiencies can remain in specialized programs.

4.3 Inlining of specialized code

As illustrated by Figure 8, when an interpreter is organized as a recursive traversal of the syntax, the result of specializing the interpreter can contain many small functions and many function calls. Jumping to and from a function is expensive, and executing the prelude and postlude of each called function represents an additional overhead. Eliminating the jumps is typically as simple as inlining the code at the call site, and deleting the call and return instructions. The prelude and postlude, however, are still needed to pass the arguments, transmit the returned value, and save and restore registers that are modified by the called function but may be needed by the calling context.

Specialization of a function eliminates static code, and thus typically reduces the number of arguments and registers used by the function definition. We thus add to the specialization process the optimization of the preludes and postludes of specialized functions according to the set of arguments and registers actually used by the specialized definition. Preludes and postludes are simplified in two ways: (i) all processing related to unused registers and arguments is removed, and (ii) registers values are saved in other registers whenever possible, rather than on the stack. These transformations are architecture-dependent and thus add a degree of machine specificity to the specialization process, but the analyses and transformations required do not require knowledge of low-level optimization techniques for the target machine.

Function specialization proceeds as follows. The offline, compile-time phase of Tempo detects the location of the prelude and postlude of each function, as well as the set of registers read and written by each template. The run-time phase generates specialized code for a called function by first generating the prelude, then specializing the body, and finally generating the postlude. Specialization of the body collects the set of registers used by each template included in the specialized definition. Because this information is associated with each template at compile time, the specialization-time collection of this information is very efficient. Treatment of the postlude uses the register usage information collected during processing of the body to create a prelude and postlude optimized according to the specialized function's argument and register usage.

SPARC code resulting from the specialization of the example interpreter with respect to the expression `3 + 4` is shown in Figure 9.[2] The SPARC standard prelude and postlude are each implemented by the single instruction `save` or `restore`, respectively, which save and restore an entire "window" of registers. As shown in Figure 8, the specialized variants of the `interp`, `cst` and `add` functions take no arguments. Furthermore, they use only registers that, by SPARC procedure calling conventions, need not be preserved by a called function. Thus, in each specialized instance of these functions, the standard SPARC prelude `save` is replaced by a `nop` instruction. The functions `push` and `top` are compiled without a prelude in the template file. Thus, no optimization is needed in these cases. To preserve the interface of the specialized code with the calling context, the prelude and postlude of the entry point `init` are not optimized.[3] The `nop` instruction preceding the `mov` (move) instruction in each specialized variant of `cst` is derived from the loading of a small constant (*e.g.*, 3), and is unrelated to the optimizations presented here.

Because space is reserved for the optimized prelude before the operations in the specialized function body are known, it can occur that either too little space is reserved and optimization is not possible, or, more likely, that too much space is reserved and the extra space must be overwritten with `nop` instructions. To address this issue, we reorganize the specializer in two passes such that the first pass evaluates the static expressions and selects templates accordingly, and the second pass emits the chosen templates and instantiates them with the corresponding static values. Such a specializer is awkward to implement by hand,

[2] Specialization generates object code directly. The corresponding assembly code is shown for readability.

[3] The postlude `restore` instruction appears in the delay slot of the `ret` (return) instruction.

```
save %sp, -112, %sp                          init_e
 nop                                         interp_e
 nop                                         interp_e1
 nop                                         cst_3
 nop
mov 3, %o0
sethi %hi(0x30800), %g2                      push_3
st %o0, [ %g2 + 0x244 ]
 nop                                         interp_e2
 nop                                         cst_4
 nop
mov 4, %o0
sethi %hi(0x30800), %g2                      push_4
st %o0, [ %g2 + 0x248 ]
 nop                                         add_e
sethi %hi(0x30800), %g2                      top_v2
ld [ %g2 + 0x248 ], %o0
mov %o0, %l0
sethi %hi(0x30800), %g2                      top_v1
ld [ %g2 + 0x244 ], %o0
mov %o0, %o1
add %o1, %l0, %o0
sethi %hi(0x30800), %g2                      push_v
st %o0, [ %g2 + 0x244 ]
sethi %hi(0x30800), %g2                      top_e
ld [ %g2 + 0x244 ], %o0
mov %o0, %i0
 ret
 restore
```

Figure 9: Code generated for `3 + 4`, with optimized preludes and postludes

because it requires substantial communication between the two phases as to the set of chosen templates and corresponding set of static values. Our implementation uses techniques based on *data specialization* [3, 15, 17] to generate such a specializer automatically from a one-pass specializer.

Figure 10 shows the SPARC code resulting from applying the two-pass optimization during specialization of the example interpreter with respect to the expression `3 + 4`. As compared to Figure 9, the only remaining `nop` instructions are those that derive from the loading of small constants.

```
save %sp, -112, %sp                    init_e
                                       interp_e
                                       interp_e1
nop                                    cst_3
mov 3, %o0
sethi %hi(0x31c00), %g2                push_3
st %o0, [ %g2 + 0x274 ]
                                       interp_e2
nop                                    cst_4
mov 4, %o0
sethi %hi(0x31c00), %g2                push_4
st %o0, [ %g2 + 0x278 ]
                                       add_e
sethi %hi(0x31c00), %g2                top_v2
ld [ %g2 + 0x278 ], %o0
mov %o0, %l0
sethi %hi(0x31c00), %g2                top_v1
ld [ %g2 + 0x274 ], %o0
mov %o0, %o1
add %o1, %l0, %o0
sethi %hi(0x31c00), %g2                push_v
st %o0, [ %g2 + 0x274 ]
sethi %hi(0x31c00), %g2                top_e
ld [ %g2 + 0x274 ], %o0
mov %o0, %i0
ret
restore
```

Figure 10: Code generated for `3 + 4`, with optimized preludes and postludes, generated in two passes

4.4 Performance

Figures 11–13 summarize the effect of these optimizations on the performance of the code generated by compilation of the PLAN-P protocols presented in Section 3: `plearn` is the learning bridge protocol, `web` is the load-balancing web server protocol, and `mpeg` is the adaptive MPEG video distribution protocol. We have implemented these programs using both the C-like PLAN-P syntax, referred to here as CPLAN-P, and the SML-like PLAN-P syntax, referred to here as SMLPLAN-P. The code sizes are similar in both cases: roughly 40 lines for `plearn`, 90 lines for `web`, and 210 lines for `mpeg`. Tests were performed on a 296MHz UltraSPARC-II, a 450 MHz Power PC G4, and a 930 MHz Pentium III.

The optimization of preludes and postludes substantially improves performance on the

SPARC, with speedups of up to 13 times (Figure 11). This benefit is primarily due to reduction in the overhead of the SPARC's register saving mechanism, which for programs with deeply nested recursion such as an interpreter can frequently save a large number of registers to memory. Speedups range from 1.9 times to 3.4 times for the PowerPC and from 1.3 times to 1.6 times for the Pentium. The two-pass implementation gives a small performance improvement in all cases. Due to limitations in the implementation of Tempo for Linux, we do not have a two-pass implementation for the Pentium; instead we use a heuristic that selects a reduced size for the preludes of specialized functions.

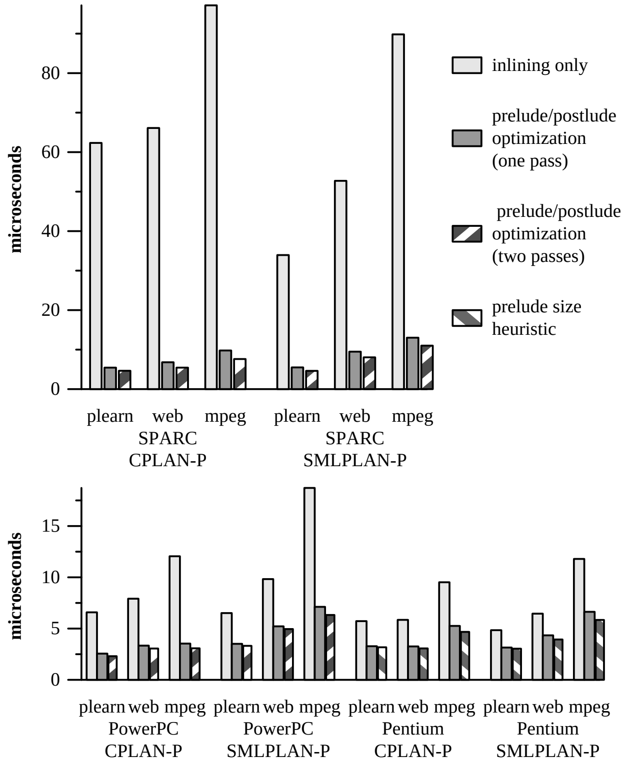

Figure 11: Execution time per packet (microseconds)

The optimization has the greatest impact on code size for the PowerPC, reducing code size up to 75%. On this architecture, the original prelude and postlude are implemented as a sequence of instructions, and the optimization eliminates the need for almost all register saves. On the SPARC, the one-pass implementation removes most postludes and the two-pass implementation then further removes most preludes. The total reduction in code size is up to 45%. For the Pentium, there is little code size reduction using the basic one-pass

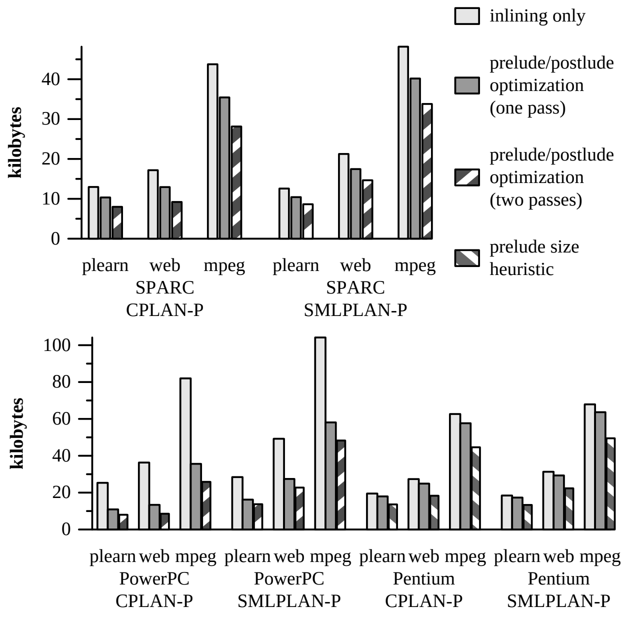

Figure 12: Size of the specialized code (kilobytes)

approach, but the use of the heuristic reduces the size of the function preludes, and thus gives a significant overall size reduction, particularly for the most complex of the protocols, `mpeg`.

Each degree of optimization requires more bookkeeping, and thus can be expected to increase the specialization time, as is the case for the SPARC (Figure 12). For the PowerPC, the one-pass optimization decreases specialization time, probably because much less code is generated. The optimizations have negligible effect on the cost of specialization on the Pentium.

5 Related work

The work most closely related to PLAN-P is that of the PLAN project [11]. As compared to PLAN, we have extended the language to allow the implementation of application protocols and to provide stronger safety and security properties, and we have developed an implementation based on compilation rather than interpretation, for greater efficiency. A follow-up to PLAN is the SNAP framework, which provides a byte-code language for active packet programming. The primary emphasis of SNAP is on constraining resource usage. In

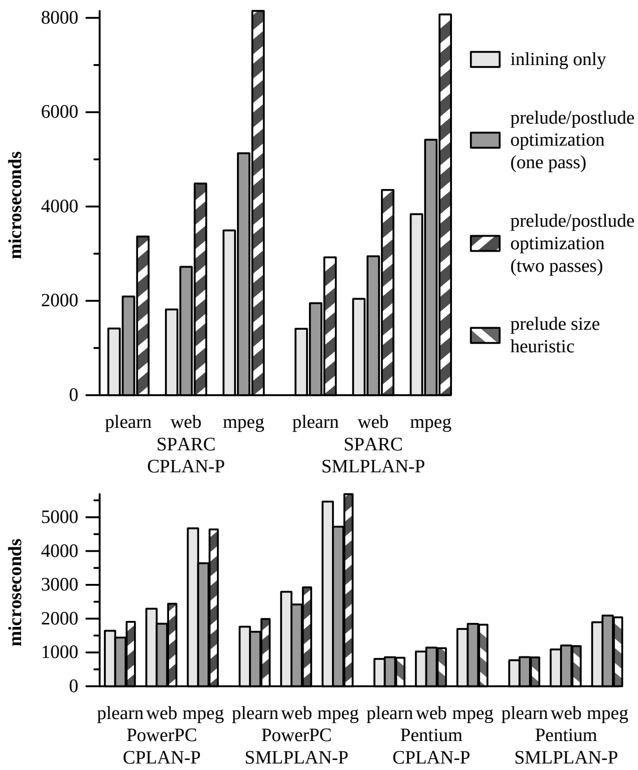

Figure 13: Specialization time (microseconds)

particular the language allows only forward jumps, which implies that for protocols that do not use resident services, resource usage is proportional to the packet size. SNAP protocols are also significantly more efficient than the corresponding PLAN protocols. Nevertheless, interpretation is still used, unlike PLAN-P, which is compiled to native code.

ANTS is an active network approach in which Java code is distributed with packets [28]. Code caching is used to reduce the code distribution cost. Security relies on signatures that ensure that code comes from a trusted source and on the execution of protocols within a sandbox that limits access to critical resources. Preliminary results show that ANTS is significantly slower than comparable C code, but many of the inefficiencies, such as execution in user space, are not intrinsic to the ANTS model. The PAN active network system uses many of the ideas of ANTS, but emphasizes performance over safety [22]. PAN runs in either user space or, like PLAN-P, as a loadable kernel module. Protocols can be distributed as either native Pentium object code or Java byte code. When used with native Pentium code, overall performance is comparable to that of a non-active node. PLAN-P also executes native code, and we obtain similar results in the learning bridge experiment (Section 3.2). Nevertheless, PLAN-P can perform safety checks that are not possible with a pure native-code solution.

SmartPackets is an active network approach geared toward network management tasks [23]. The primary emphasis is on minimizing the size of protocols, so that a useful protocol can be stored within a single packet. Protocols are written in a restricted version of C that, like the PLAN-P DSL, contains network-specific types and primitive functions. Protocols are compiled to a dedicated assembly language and distributed on the network in this form with an accompanying authenticity certificate. Downloaded protocols are then interpreted by a virtual machine, thus providing portability. In contrast, PLAN-P analyzes the downloaded code itself, and can thus safely generate and execute native code.

Mascolo *et al.* propose to use XML both to define languages for programmable networks and as the concrete syntax of programs written in these languages [18]. The emphasis is on dynamic update, both of previously deployed programs and of the languages themselves. Moore and Nettles have compiled a survey of approaches to adding programs to packets, and assess the safety, efficiency, and flexibility of these approaches [20].

Keller *et al.* [14] have also developed a video broadcast service based on active networks. Their approach is based on wavelet encoding of the video. Their active network is platform-specific and programming must be done in C at kernel level. Therefore, programming protocols is error-prone and requires highly-skilled engineers. The lack of portability of this solution further complicates protocol testing and deployment.

Our learning bridge experiments showed the performance advantages of a dedicated active network OS as compared to the implementation of network facilities with a conventional OS. Janos is a Java-oriented OS for active network nodes. The emphasis of this work is to provide the safety and security by using the OS to restrict the operations that an active protocol can perform. Protocols are implemented as Java bytecode, running on a restricted virtual machine that follows the ANTS model [29, 28]. Peterson *et al.* have developed an OS interface for active routers that emphasizes efficient management of OS resources related to packet forwarding. This interface has been implemented for Janos, as well as for Scout [19] and Xok [13]. Although NodeOS is directed towards active networks, in which packets contain both code and data, the programmable router architecture used by PLAN-P could also benefit from the packet filtering and resource management facilities provided by NodeOS.

The balance between performance and portability provided by PLAN-P relies on the use of run-time program specialization to automatically generate compiled code based on an interpreter. Several other related approaches to run-time code generation have been developed. The DyC run-time specialization system uses information collected at compile time to perform limited specialization-time optimizations within individual blocks of generated code [9]. Nevertheless, the provided optimizations have been found not to be very applicable to specialization of recursive interpreters, which are typically implemented using very small blocks of code. The 'C language allows the the programmer to implement programs that generate code at run-time. Because the programmer must explicitly construct the code that is generated at run time, this approach is not practical for implementing an interpreter for a realistic DSL.

DSLs have already been used in the design of OS sub-systems to allow programmers to specify new policies. Examples are TEAPOT [4], for writing DSM coherence protocols, and HiPEC [16], for writing page-replacement policies. While HiPEC relies on a bytecode interpreter, which is less efficient than a compiler, TEAPOT is implemented with a compiler, which is more difficult to maintain. In earlier work, we have proposed GAL [27], a language for designing device drivers. GAL was implemented using the same interpreter/specialization framework as PLAN-P, but specialization was done at compile time rather than at run time.

6 Conclusion

In this paper, we have presented the design and implementation of the PLAN-P language for developing protocols for programmable routers. The focus of PLAN-P is on providing both safety and efficiency. Safety is provided by the use of a DSL, which restricts the computations that can be expressed by a PLAN-P protocol, and by downloading protocols as source code into the router, where they can be checked for both general and router-specific safety properties. Efficiency is provided by compiling this source code to native code on the router using a dedicated compiler. The compiler is generated automatically from a straightforward interpreter, thus enhancing the portability of the approach.

Our experiments show that PLAN-P provides a simple approach to implementing effective adaptation policies and useful network-related extensions to existing applications. The protocols are concise and require little network expertise to develop and maintain. In general, we obtain end-to-end performance comparable to that of hand-written C implementations.

References

[1] D.S. Alexander, M. Shaw, S.M. Nettles, and J.M. Smith. Active bridging. In *Proceedings of the 1997 SIGCOMM Conference*. ACM Press, 1997.

[2] Apache. HTTP server project. URL: http://www.apache.org.

[3] G. J. Barzdins and M. A. Bulyonkov. Mixed computation and translation: Linearisation and decomposition of compilers. Preprint 791 from Computing Centre of Sibirian division of USSR Academy of Sciences, p.32, Novosibirsk, 1988.

[4] S. Chandra, B. Richards, and J.R. Larus. Teapot: Language support for writing memory coherence protocols. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 237–248, 1996.

[5] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann,

editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.

[6] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the* 23^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, FL, USA, January 1996.

[7] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A software framework for component-based operating system kernels. In *2000 USENIX Annual Technical Conference*, pages 73–86, Monterey, CA, June 2002.

[8] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprinted from Systems · Computers · Controls 2(5), 1971.

[9] B. Grant, M. Mock, M. Philipose, C. Chambers, and S.J. Eggers. The benefits and costs of DyC's run-time optimizations. *ACM Transactions on Programming Languages and Systems*, 22(5):932–972, 2000.

[10] D. He, G. Muller, and J.L. Lawall. Distributing MPEG movies over the Internet using programmable networks. In *The 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 161–170, Vienna, Austria, July 2002.

[11] M. Hicks, P. Kakkar, J.T. Moore, C.A. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, volume 34(1) of *ACM SIGPLAN Notices*, pages 86–93. ACM, June 1998.

[12] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.

[13] M.F. Kaashoek, D.R. Engler, G.R. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 52–65, St-Malo, France, October 1997.

[14] R. Keller, S. Choi, D. Decasper, M. Dasen, G. Fankhauser, and B. Plattner. An active router architecture for multicast video distribution. In *Proceedings of IEEE INFOCOM 2000*, pages 1137–1146, Tel Aviv, Israel, March 2000.

[15] J. Lawall. Implementing circularity using partial evaluation. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects, Second Symposium, PADO 2001*, volume 2053 of *Lecture Notes in Computer Science*, pages 84–102, Aarhus, Denmark, May 2001. Springer.

[16] Chao Hsien Lee, Meng Chang Chen, and Ruei Chuan Chang. HiPEC: High performance external virtual memory caching. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 153–164, 1994.

[17] K. Malmkjær. Program and data specialization: Principles, applications, and self-application. Master's thesis, DIKU University of Copenhagen, August 1989.

[18] C. Mascolo, W. Emmerich, and H. De Meer. An XML based programmable network platform. In *Proceedings of the ICSE Workshop on Software Engineering and Mobility*, Toronto, Canada, May 2001.

[19] A.B. Montz, D. Mosberger, S.W. O'Malley, L.L. Peterson, and T.A. Proebsting. Scout: A communications-oriented operating system. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[20] J.T. Moore and S.M. Nettles. Towards practical programmable packets. Technical Report MS-CIS-00-12, Department of Computer and Information Science, University of Pennsylvania, May 2000.

[21] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization : Implementation and experimental study. In *International Conference on Computer Languages*, pages 132–142, Chicago, IL, May 1998. IEEE Computer Society Press. Also available as IRISA report PI-1065.

[22] E.L. Nygren, S. Garland, and M.F. Kaashoek. PAN: A high-performance active network node supporting multiple mobile code systems. In *Proceedings of IEEE OpenArch 1999*, pages 78–89, New York, NY, March 1999.

[23] B. Schwartz, A.W. Jackson, W.T. Strayer, W. Zhou, R.D. Rockwell, and C. Partridge. Smart packets: applying active networks to network management. *ACM Transactions on Computer Systems (TOCS)*, 18(1):67–88, 2000.

[24] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[25] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, Indiana, October 1998.

[26] S. Thibault, J. Marant, and G. Muller. Adapting distributed applications using extensible networks. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 234–243, Austin, Texas, May 1999. IEEE Computer Society Press.

[27] S. Thibault, R. Marlet, and C. Consel. Domain-specific languages: from design to implementation – application to video device drivers generation. *IEEE Transactions on Software Engineering*, 25(3):363–377, May–June 1999.

[28] D.J. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

[29] D.J. Wetherall, J. Guttag, and D.L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH*, April 1998.