Compiling PLAN to SNAP*

Michael Hicks[1], Jonathan T. Moore[1], and Scott Nettles[2]

[1] Computer and Information Science Department
University of Pennsylvania
{mwh,jonm}@dsl.cis.upenn.edu
[2] Electrical and Computer Engineering Department
The University of Texas at Austin
nettles@ece.utexas.edu

Abstract. PLAN (Packet Language for Active Networks) [4] is a highly flexible and usable active packet language, whereas SNAP (Safe and Nimble Active Packets) [11] offers significant resource usage safety and achieves much higher performance compared to PLAN, but at the cost of flexibility and usability. Ideally, we would like to combine the good properties of PLAN with those of SNAP. We have achieved this end by developing a compiler that translates PLAN into SNAP. The compiler allows us to achieve the flexibility and usability of PLAN, but with the safety and efficiency of SNAP. In this paper, we describe both languages, highlighting the features that require special compilation techniques. We then present the details of our compiler and experimental results to evaluate our compiler with respect to code size.

1 Introduction

One of the most aggressive approaches to active networking involves the use of *active packets* (or capsules [16])—packets where the traditional passive header is augmented or replaced with a program. This program is executed as it traverses the network, thus potentially offering per-packet customizability for applications. Indeed, this flexibility has been used to implement a variety of applications, such as application-specific routing [3], transparent redirection of web requests to nearby caches [6], distributed on-line auctions [7], reliable multicast [8], mobile code firewalls [2], and reduced network management traffic [15].

Existing active packet systems have made different tradeoffs between safety, flexibility, usability, and performance. In this paper, we consider two different active packet schemes, PLAN (Packet Language for Active Networks) [4] and SNAP (Safe and Nimble Active Packets) [11]. PLAN is extremely flexible and usable; for example, an active internetwork, PLANet [3], has been constructed in which every packet is a PLAN packet. Furthermore, the PLANet distribution has been downloaded by over 450 users and is actively being used for research. SNAP, on the other hand, was designed to overcome the weaknesses of existing

* This work was supported by the NSF under contracts ANI 00-82386 and ANI 98-13875.

active packet systems. Namely, SNAP offers much stronger safety guarantees and achieves higher performance than other active packet systems, but has a less flexible programming model than PLAN; also, since it is a lower-level language than PLAN, it is harder to use.

The goal of this is to show that we can have the best of both worlds by compiling PLAN into SNAP. Because the two languages differ in their models of active packet execution and indeed in their basic programmatic expressibility, the compilation process is not straightforward. Nonetheless, our compiler allows us to achieve the flexibility and usability of PLAN while attaining the safety and performance of SNAP.

We begin by introducing PLAN and SNAP in Sections 2 and 3 respectively. In particular, we will highlight the differences in programming model and expressibility that we will have to overcome during translation. In Section 4, we then present compilation techniques that allow us to overcome these problems, and experimentally evaluate our compiler in Section 5. Finally, we describe future and related work and conclude in Section 6.

2 PLAN

PLAN [4] is a strongly-typed functional language with syntax similar to Standard ML [9]. PLAN is part of a two-level active networking architecture [1]; namely, active packets provide the control logic and "glue" for tying together and controlling node-resident *services*. In this sense, PLAN programs/packets are similar to Unix shell scripts that provide control over utility functions like `sort` and `grep`.

PLAN supports standard programming features, such as functions and arithmetic, and features common to functional programming, like lists and the list iterator $fold$[1]. A notable restriction is that functions may not be recursive and there is no unbounded looping; this helps guarantee that all PLAN programs terminate.

To support packet transmission, PLAN includes primitives for *remote evaluation*: a user can specify that a computation (function call) take place on a different node. The two main primitives are `OnNeighbor` and `OnRemote`. Both take arguments that describe which function to call, a set of actual arguments, and a node on which to evaluate the call (the *evalDest*). For `OnNeighbor`, the evalDest must be one hop away from the current node, whereas `OnRemote` also accepts a *routFun* argument to specify a particular routing function, which determines the routing on nodes leading to the ultimate evalDest. Both primitives must also be supplied with a resource bound, which acts as a TTL or hop count. The resource bound is decremented on each hop and during execution a program must "donate" some of its resource bound to the `OnRemote` or `OnNeighbor` used to send a new packet.

[1] Intuitively, $fold$ executes a given function f for each element of a given list, accumulating a result as it goes.

```
fun ping(payload:blob, port:int) =
  OnRemote(|reply|(payload,port),getSrc(),getRB(),defaultRoute)

fun reply(payload:blob,port:int) = deliverUDP(payload,port)
```

Fig. 1. PLAN ping

PLAN provides the ability to manipulate programs as data, via a construct known as a *chunk* (short for 'code hunk') [12]. Chunks provide the means for PLAN packets to be fragmented or encapsulated within one another. A chunk has three logical elements: some PLAN code, an entry function name, and actual arguments. Evaluating a chunk results in looking up the named entry function and calling it with the arguments. Note that `OnNeighbor` and `OnRemote` actually transmit chunks.

2.1 Example: PLAN ping

Figure 1 shows how to program "ping" in PLAN. Initially we start with a packet whose *evalDest* is set to our ping target, an entry point function `ping`, and actual arguments `payload` and `port`, which are a payload and an application UDP port number, respectively.

When the packet reaches the destination, we evaluate the call to `ping`, which in turn creates a chunk `|reply|(payload,port)` which will invoke `reply` with our payload and port number. We determine our original source via the `getSrc` service, and then use `OnRemote` to cause our new chunk to be evaluated on that node. This spawns a new packet that makes the return trip to our source using `defaultRoute` to determine routing at intermediate nodes. The call to `getRB` returns all of the current packets resource bound and donates it to the new packet.

Finally, when the return packet arrives at the source, we evaluate its chunk and call `reply`, which simply delivers our payload to the application waiting on the appropriate UDP port.

2.2 Advantages and Disadvantages

A key advantage of PLAN is that it is easy to use. Despite the limits on its computational power, it is a high-level language both in its syntax and its features. In particular, the combination of chunks with the remote evaluation provide a powerful set of abstractions that are tailored for packet programming. These powerful features, along with support for general language constructs such as function definitions and calls, are a key part of its flexibility.

Unfortunately, PLAN has some important disadvantages. In particular, transmitting and receiving PLAN requires marshaling and unmarshalling some representation of PLAN programs [3]. This significantly increases the cost of PLAN packets compared to more conventional approaches. Secondly, although PLAN programs are guaranteed to terminate, this guarantee is rather weak. In particular, it is possible for PLAN programs to execute in time and space that is exponential in their length (we will discuss this in more depth later in Section 4). For large packets, this may be essentially no different than unbounded execution.

3 SNAP

SNAP [11, 10] is a second-generation active packet system that was designed, in part, to address other active packet systems' (including PLAN) weaknesses in performance and resource safety. Our main approach in SNAP's design was restricting the expressibility of the packet programming language in return for certain safety guarantees and to simplify and streamline the implementation of a SNAP virtual machine.

The key safety gain of SNAP over PLAN comes from its model of resource usage: SNAP programs use time, space, and bandwidth linearly in the length of the program. The SNAP implementation achieves this by requiring *all branches to go forward*, thus preventing looping and causing the number of instructions executed to be limited by the program's length. This, when combined with the fact that SNAP instructions execute in constant time, means that all SNAP programs run in time linear in their length. Similar restrictions on each instruction's memory and bandwidth use achieve the other bounds.

To enable efficient execution, without unmarshalling, SNAP was designed as a stack-based bytecode language, providing instructions for performing simple arithmetic, environment query, control flow, and packet sends. A full description of the language can be found in [10], but we will highlight here the features that impact our use of SNAP as a compilation target for PLAN.

Being a stack-based language, all computation occurs by removing arguments from the stack, performing the computation, and then pushing the result back on the stack. SNAP includes three main stack manipulation primitives so that programs can properly order their arguments on the stack: `push` v pushes the value v on top of the stack, `pop` removes the top stack item, and `pull` n pushes a copy of the nth stack element onto the stack.

Aside from stack storage, SNAP also includes the notion of a *heap*, in which we can store variable-sized data (such as binary payloads) or data not accessed in a stack-based discipline. The two main primitives for heap manipulation in SNAP are `mktup` n, which creates a new heap-allocated tuple[2] out of the top n stack values, and `nth` n, which extracts the nth element of the tuple pointed to by the top stack value.

There are also a variety of program control flow instructions, such as `jmp` ("jump"), `bne` ("branch if not equal to zero"), `beq` ("branch if equal to zero"), and `paj` ("pop and jump"). Each of these instructions takes a constant immediate

[2] Tuples can be thought of as a small array of values, similar to a `struct` in C.

argument[3] that is a non-negative relative branch offset. In the case of `paj`, this offset is added to the top stack value (again, the resulting offset must be non-negative).

Finally, SNAP contains a variety of packet sending operations. In addition to a more general `send` instruction for doing arbitrary packet sends, SNAP includes a `forw` instruction which compares the current node to a packet's destination address and, if the packet has not yet reached its destination, forwards the packet towards its ultimate destination.

3.1 Example: ping

```
forw       ; move on if not at dest
bne     5  ; jump 5 instrs if nonzero on top
push    1  ; 1 means "on return trip"
getsrc     ; get source field
forwto     ; send return packet
pop        ; pop the 1 for local ping
demux      ; deliver payload
```

Fig. 2. SNAP code for ping.

Figure 2 shows a ping program written in SNAP assembly language. Assume we send a packet containing this program and a stack of three stack values: [0 :: *port* :: *payload*]. The 0 will be used as a flag to indicate whether we are in the outgoing direction or on the return trip back to the source.

The packet will first forward itself to the destination by executing the `forw` instruction on each intervening node. Upon reaching the destination, the `forw` instruction falls through to the `bne` ("branch if not equal to zero") instruction. Since there is a 0 on top of the stack, the branch falls through as well. We then push a 1 onto the stack to indicate we are ready to return.

Next, the program retrieves the packet's source address via the `getsrc` instruction and then sends itself back towards the source with the `forwto` instruction, thus resetting the packet's destination address to be the original source address. This packet will then forward itself all the way back to the source with `forw`. Upon reaching the source, the `forw` will fall through. Since there is a 1 on top of the stack, the `bne` will branch forward 5 instructions to the `demux` instruction, which delivers our payload to the given port[4].

[3] Our SNAP assembly language allows constant expressions involving code label positions or the location of the current instruction (`pc`, or program counter).

[4] The `pop` instruction is included to allow a node to ping itself (no packet sends result in this case).

3.2 Advantages and Disadvantages

Essentially by design, SNAP's advantages and disadvantages are the converse of PLAN's. SNAP's model of resource usage implies tight control over resources and SNAP retains the advantages of strong typing. The low-level bytecode nature of SNAP allows efficient in-place execution. Unfortunately, SNAP is much harder to understand (and even harder to generate) compared to PLAN. In particular, the severe restrictions on branches (namely, that they may only go forward) make function calls impossible to implement straightforwardly.

4 Compilation

We have seen that PLAN and SNAP have complementary advantages and disadvantages. In fact, what we would really like is an active packet language that is flexible, easy to use, safe (in particular in terms of resource usage), and efficient. We can achieve this blend by noting that the advantages of PLAN over SNAP are mostly notational: PLAN programs are high-level and easy to create, and SNAP's are not. Conversely, SNAP shines in its execution characteristics: it is efficient and can only succinctly express computations that can be executed with limited resources. These observations suggest the strategy of compiling PLAN to SNAP to gain the best of both.

PLAN and SNAP have related computational models, in large part because SNAP's design was informed by our experience with PLAN. However, SNAP departs from PLAN's semantics in a number of non-trivial ways that complicate the process of compiling PLAN programs to SNAP bytecodes. The main difficulty arises due to SNAP's lack of backward branches. We first consider the compilation of PLAN features that would normally require backward branches, and how we compile these features without them. We then consider the mapping between PLAN and SNAP data structures, and finally consider some of PLAN's advanced features.

4.1 Backward branches

To straightforwardly compile certain PLAN constructs would require backward branches, in particular function calls and loops. Of course, backward branches are not permitted in SNAP, so in this subsection, we explain how to compile these language features without them. We then consider a novel way to simulate backward branches by resending a packet to the current router at an earlier evaluation point.

Function Calls For stack-based architectures, a call to a function `f` is compiled as a *jump* to `f`'s address, having first pushed the *return address*, which is typically the address of the instruction following the jump. Upon its completion, the function `f` pops the return address and jumps to it. Unfortunately, since one of these two jump instructions must go backward (either to `f` initially, or in returning to the caller), this approach is prevented by SNAP's semantics. For example, consider the following compilation from PLAN to SNAP:

```
                        | f:    push 1    ;push arg to print
                        |       print     ;print it
                        |       paj 0-pc  ;return to caller
fun f () = print(1)     |
fun g () = f()          | g:    push lab1 ;push return address
                        |       jmp f-pc  ;call f
                        | lab1: paj 0-pc  ;return to caller
```

This compilation will fail at runtime because the `jmp f-pc` instruction will compile to `jmp -4`, which is not allowed. To address this problem, we have explored two compilation strategies for compiling function calls without using backward branches: *reordering basic blocks* and *inlining*.

Approach 1: Reordering Basic Blocks We first considered compiling function calls as usual, but then sorted the resulting basic blocks so that all branches go forward. For example, we can compile the example program above as before, but then sort the basic blocks topologically to prevent backward branches:

```
g:    push lab1 ;push return address
      jmp f-pc  ;call f

f:    push 1    ;push arg to print
      print     ;print it
      paj 0-pc  ;return to caller

lab1: paj 0-pc  ;return to caller
```

We have reordered the function `g` so that its basic blocks are split by the body of `f`. Unfortunately, this approach will not work in general. Consider modifying `g` to call `f` twice:

```
                        | f:    push 1    ;push arg to print
                        |       print     ;print it
                        |       paj 0-pc  ;return to caller
                        |
fun f () = print(1)     | g:    push lab1 ;push return address
fun g () = (f(); f())   |       jmp f-pc  ;call f
                        | lab1: push lab2 ;push return address
                        |       jmp f-pc  ;call f
                        | lab2: paj 0-pc  ;return to caller
```

It is easy to see that `g`'s blocks cannot be rearranged such that all branches go forward. In particular, the block `lab1` must *follow* `f` since it is returned to by `f`, but it must also *precede* `f` since it will jump to `f`.

Approach 2: Inlining One way to more generally handle function calls is to eliminate them altogether by inlining. That is, at each call site, we replace a given function call with a copy of its body, with the formal parameters replaced by the actual ones. In general-purpose languages, inlining cannot, in general, remove all calls to recursive functions, since the function's body contains a call to the function itself. However, PLAN does not permit recursive function calls, so all function calls can be removed via inlining.

While inlining solves the problem of backward branches, it can result in code bloat, which is particularly worrisome because code occupies space in a network-bound packet. In our experience, PLAN programs are often written as one or more general functions followed by an 'entry-point' function that provides a sort of 'user interface.' For example, the 'scout' packet described in [3] defines a function `dfs` that performs a depth-first traversal of the network, with an entry-point function `startDFS` to begin the search. Using inlining, the entire contents of `dfs` will be inlined, and then `dfs` will itself be inlined inside of `startDFS`, effectively doubling the size of the packet.

We can reduce code bloat in a number of ways. First, we could 'prune' a packet to eliminate extra code certain to be unneeded by future computations. For example, the `startDFS` function is only invoked by the sender of the packet; therefore, when child packets are transmitted from the sending node, the code for `startDFS` can be eliminated. We perform pruning of this kind in PLANet [3]. However, while performing a run-time analysis to find dead code is reasonably straightforward in PLAN, it is less so in SNAP, and may result in excessive computation.

Second, we could combine inlining with block reordering, using topological sort to split basic blocks when possible, and resorting to inlining otherwise. In general, within a given function `g`, we inline all but one of the calls to some function `f`, made either from `g` *or from some function called by* `g`. The reason for this can be seen by looking at a control-flow graph between the basic blocks of a program. Figure 3 shows the control-flow graphs of two PLAN programs. The basic blocks of each PLAN function are split by function calls; for example, `g`'s block that precedes a call to `f` is labeled `g-fpre`, while the block following it is `g-fpost` (or `g-fpre2` in the case that another call to `f` is made). We can see in the figure that each of these graphs has a loop, implying a backward branch is needed. Furthermore, the backward branch arises due to a second call to `f`, either from `g` in the case of the first program, or from `g`'s child `c` in the second program. There is no way to reorder either program to allow branches to go forward.

To eliminate the backward branch, the second call to `f` is inlined. For the first program we have (`f` is inlined as `print(1)` in the PLAN program):

```
fun f () = print(1);
fun g () = (f(); f())
```

```
fun f () = print(1)
fun c () = f()
fun g () = (f(); c())
```

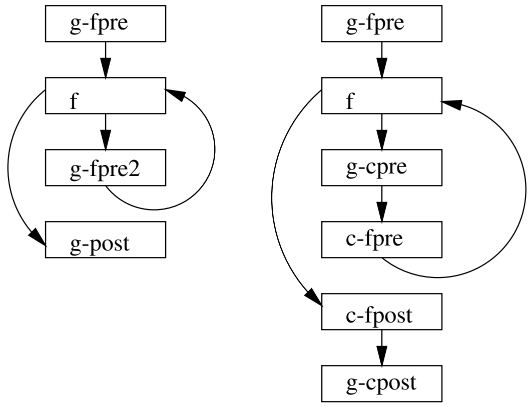

Fig. 3. Control-flow graph of the basic blocks of two PLAN programs

```
fun f () = print(1)
fun g () = (f(); print(1))
```

```
g:    push lab1 ;push return address
      jmp f-pc  ;call f

f:    push 1    ;push arg to print
      print     ;print it
      paj 0-pc  ;return to caller

lab1: push 1    ;push arg to print
      print     ;print it
```

In our current implementation, we implement only the basic inlining approach, but we plan to alter the compiler to employ the hybrid approach described here. As we will see in Section 5, this can result in large code size savings.

Finite Loops Loops require a backward branch to the loop head, and therefore also require more clever compilation. In PLAN, the only looping construct is the list iterator *fold*. Intuitively, *fold* executes a given function f for each element of a given list $[b_1; b_2; ...; b_n]$. The arguments to f include the list element b_i and the current accumulator value a. The value returned by f is supplied by *fold* as the accumulator to the next iteration. Therefore, $fold(f, a, [b_1; b_2; ...; b_n])$ is equivalent to $f(f...f(f(a, b_1), b_2)..., b_n)$. In the PLAN implementation, *fold* as explained above is called `foldl`; there is also a version called `foldr`, such that `foldr`$([a_1; a_2; ...; a_n], b)$ is $f(a_1, f(a_2, (...f(a_n, b)...)))$.

We can eliminate the presence of `fold` from PLAN programs by *unrolling* its computation, revealing the calls to the iterator function. However, the number of times that the iterator is called depends on the length of the list, which in general cannot be known until runtime. Therefore, the compiler unrolls the loop a fixed number of times, as specified by the user. For example:

```
fun foo(l) =
  let fun sub(i,j) = (i-j)
      val a = foldl(sub,0,l) in
    print(a)
  end
```

```
fun foo(l) =
  let
  fun sub(i, j) = (i - j)
  val a = let
    val _list = l
    val _acc = 0 in
    if (_list <> []) then
      let
        val _acc = sub(_acc, (hd _list))
        val _list = (tl _list) in
        if (_list <> []) then
           ...
        else _acc
      end else _acc
    end
  in print(a)
  end
```

The result of unrolling the `fold` is that calls to the iterator function `sub` have been made explicit. These function calls can then be eliminated by the techniques described above.

There are two disadvantages to unrolling folds. For purposes of discussion, assume that u is the number of times the loop is unrolled, and n is the length of the list, known at runtime. The first problem is that when $u > n$, the packet program occupies more space than is needed, thus unnecessarily reducing the amount of useful payload that can be carried. The second, more serious problem occurs when $u < n$, meaning that the loop was not unrolled enough times to process the entire list, potentially leading to incorrect behavior.

There are a number of ways to deal with the second problem, when $u > n$. First, we could ignore the fact that we did not complete the loop processing. This approach makes sense when incomplete list processing leads to only degraded, rather than incorrect service. Alternatively, we could signal an error by throwing an exception if we complete processing without reaching the end of the list. This exception will either exit the packet scope and halt the packet (see §4.3, below), or it could be handled within the packet by wrapping the `fold` with a `try` ... `handle` block.

Typical PLAN programs use loops only on short lists of addresses or devices, so we expect to be able to choose $u \leq n$ in most cases. We look at the effect of unrolling on code space in Section 5.

Simulating Backward Branches Though not possible with straightforward computation, a SNAP program can effectively branch backwards by sending a packet to the current router with an earlier entry point, at the cost of one resource bound unit. At the SNAP level, a backward branch can be encoded as follows:

```
g: ...       ; some instructions
   jmp g-pc  ; backwards branch
```

```
g: ...       ; some instructions
   here      ; local address
   getrb     ; current rb
   push -1   ; reuse entire stack
   push g    ; entry point
   send      ; branch backwards
   exit      ; kill current packet
```

Essentially, a backward branch is a `send` instruction, having the following arguments: the local host address, the total current resource bound, the total current stack, and the address of desired program location. We need to follow the `send` with an `exit` instruction, since `send` creates a new packet, while the current packet continues to execute.

Selective use of backward branches can improve both the problems of code bloat and incomplete list processing. For example, we could choose not to inline calls to large functions, relying on backward branches instead. Similarly, we could compile `fold` to branch backwards after u unrollings of the iterator function if list processing is not yet complete. We hope to experiment with employing backward branches in the compiler, though we have not yet done so. With selective use of backward branches we can trade resource bound for smaller packet sizes and more straightforward semantics.

4.2 Data-structures

In addition to primitive data types like integers, characters, and floats, PLAN supports arbitrary-length, homogeneous *lists* and fixed-length, heterogeneous *tuples*. SNAP supports similar primitive types, as well as arbitrarily-sized blocks of pointers which are essentially untyped tuples. Tuples compile one-to-one from PLAN to SNAP, and list elements in PLAN map to pairs, in the style of Scheme, where the first element contains the data, and the second element contains a pointer to the next element, or `0` to terminate the list.

4.3 Advanced Features

PLAN supports a number of advanced language features, including exceptions, per-packet *routing functions*, and *chunks*. In this section, we explain how we compile these features.

Exceptions PLAN supports exceptions in the style of ML (which is similar to Java). As a small example, consider the following PLAN code:

```
try
  if x = 1 then raise Exit
  else print(x)
handle Exit =>
  print("caught Exit")
```

The code raises an exception `Exit` when the variable `x` is equal to 1; this exception is caught by the surrounding *handle block*, which prints a message. If not predefined by some service, `Exit` must be declared by the user earlier in the program. Exceptions can also be raised by services and network primitives, like `OnRemote`.

In PLAN, exceptions are treated as strings. For example, when the handler above catches an exception, it performs a string compare on the exception's name. String compares are not permitted in SNAP, because they are non-constant-time operations, so exceptions are essentially integers. As a result, we map PLAN exceptions to SNAP integers during compilation. As in PLAN, SNAP exceptions can be raised by services and network primitives, like `send`, in that when something goes wrong, an exception is returned rather than a value of the expected type. For example, if `send` fails due to lack of resource bound, it will return an exception. Because service exceptions must have global identity (and therefore must always map to the same integer), we store the mapping in a configuration file used by the compiler.

We can compile exceptions to SNAP in a straightforward manner for two reasons: (1) handlers always occur later than where an exception is raised (so raising an exception will not result in a backward branch), and (2) the handler for a (potentially) raised exception is known at compile time. In most languages, separate compilation does not permit handlers to be known compile time, but our compiler essentially translates whole programs.

We compile exception-raising as follows. First, we observe that exceptions are raised in two ways: by the `raise` command, and by services like `hd`; each of these is handled differently. For the expression `raise` e, we move the exception e into a well-known stack location, and then jump to the closest handler (whose address is known at compile time); if no handler exists, we halt the program.[5] For services or primitives that may return exceptions, we check the returned value of that service using the `ISX` instruction to see if it is an exception, and if so, jump to a handler. Handle-blocks are compiled to look for the exception in the well-known stack location. If the exception does not match the one in the `handle` statement, the code jumps to the surrounding handler.

Chunks Translating chunks to SNAP is straightforward: a chunk becomes a SNAP 2-tuple consisting of an entry offset (relative to the top of the code block) and another tuple containing the initial stack; the code part is elided (it is shared with the current packet's code). The entry offset is not the address of the code block itself, but instead some "unmarshalling" code that precedes that block. This piece of code expects the tuple containing the arguments to be on top of the stack, so that it can extract the arguments into the stack positions expected by the actual code block. For example, the following piece of code defines a function `f`, and a function `make_fchunk` that makes a chunk out of `f`:

[5] In PLAN, an uncaught exception is turned into a packet that is sent to the sender. Adding this feature to SNAP is future work.

```
                          | f_chunk:     pull 0       ; copy args tuple
                          |              nth 2        ; get the second arg
                          |              pull 1       ; copy the tuple
                          |              nth 1        ; get the first arg
                          |              store 2      ; store in first slot
                          | f:           add          ; add the two args
                          |              print        ; print the result
fun f (i,j) =             |              exit         ;
  print(i+j)              |
fun make_fchunk () =      | make_fchunk: push f_chunk ; chunk addr
  |f|(1,2)                |              push 1       ; arg 1
                          |              push 2       ; arg 2
                          |              mktup 2      ; combine the args
                          |              mktup 2      ; make the chunk
                          |              exit         ;
```

Chunks are ultimately used by the network primitives, `OnRemote` and `OnNeighbor`, which are are described below.

Network Primitives `OnRemote` and `OnNeighbor` map to SNAP's `send` and `hop` primitives. Like their PLAN equivalents, the SNAP primitives require the user to specify a destination address, some resource bound to donate, and some code to execute. In SNAP, the code part is specified in two parts: an address in the code segment, and an initial stack, copied from the top of the current stack. Unlike `OnRemote`, `send` does not require a routing function argument; this is because SNAP programs execute on every hop and perform their own routing. Mapping routing functions to SNAP is described below.

Compiling the network primitives is fairly straightforward. In the case that the chunk argument is a literal, the compiler uses the literal's actual arguments and code address as arguments to `send` or `hop`. If a non-literal is used, then the compiler extracts the chunk preamble address as the code pointer (the first element of the pair) and the argument tuple as the stack (the second element).

Routing Functions As discussed, unlike SNAP, PLAN programs do not evaluate on every active router they traverse, but on the evaluation destination only. On the intervening PLAN nodes, a packet-specified "routing function" is evaluated instead, which determines the next hop and forwards the packet there.

We implement routing functions in SNAP code as a preamble to the main program. This preamble checks if the packet has reached its destination, and if not looks up the next hop using the appropriate SNAP primitive or service function, and forwards the packet. If the packet has arrived at its destination, the preamble jumps to the actual entry point, stored on the top of the stack. For the `defaultRoute` routing function, which is the one most often used in our existing PLAN programs, this preamble is two instructions: the `forw` instruction, followed by a `paj` ("pop and jump"). Note that the routing function code is always earliest in the packet, to assure that the `paj` will result in a forward branch.

4.4 Discussion

Because PLAN programs of length n can consume $O(x^n)$ time and space (for some x), while SNAP programs of length n are bounded in time and space by $O(n)$, we can infer that compiling PLAN to SNAP must result in forbidding certain PLAN constructs or in altering the length of the program. In fact, our compiler does both. For execution time, we *expand* problematic operations, namely function calls and loops, so that they require more space in the program. For example, the following PLAN program runs in time $O(2^n)$:

```
fun f () = (print(1); print(1))
fun g () = (f (); f())
fun h () = (g (); g())
```

That is, the program has $n = 3$ lines, defines $2 * n = 6$ function calls (2 per function), but invokes these functions $2^n = 8$ times dynamically. To compile this program to SNAP, we inline each call:

```
fun h () = (print(1); print(1); (* f() *)
            print(1); print(1); (* f() *) (* g() *)
            print(1); print(1); (* f() *)
            print(1); print(1)) (* f() *) (* g() *)
```

The result is that we have expanded the program so that it defines the same number of function calls that it invokes dynamically. A similar expansion takes place when unrolling `fold`.

PLAN can also allocate memory on the order of $O(x^n)$ for some x. One way is to structure the program as above, but have the leaf nodes (*i.e.* the calls in `f`) perform constant-time allocation. We deal with these programs by expansion. However, PLAN supports some operators that permit non-constant space allocation. For example, the following program allocates $O(2^n)$ memory blocks:

```
fun h () =
  let val x = [1;1]
      val x2 = x @ x
      val x3 = x2 @ x2 in
    x3
  end
```

That is, we double the length of the list for each line in the program by using the *append* operator `@`. We could similarly concatenate the same string to itself using the *concatenation* operator `^`. Translations of these operations are not straightforward, because they depend on the size of their arguments, so we forbid their translation.

In PLAN, polymorphic equality (the `=` operator) is *structural*. This means that the contents of compound data-structures, like lists and tuples, are recursively compared, which necessarily requires non-constant time. SNAP supports

	size (B)		Ratio
Program	PLAN	SNAP	(SNAP/PLAN)
`deliver`	406	284	0.70
`devinfo`	586	1624	2.77
`getNeighbors`	123	80	0.65
`getRoutes`	117	80	0.68
`multiprint`	810	1856	2.29
`ping`	300	204	0.68
`ping_pong`	268	184	0.69
`pingtime`	556	384	0.69
`query_gc`	100	72	0.72
`traceroute`	519	336	0.65
Median	353	244	0.69

Fig. 4. Code size experiments. We present the wire format of the given PLAN program, that of the SNAP program output by our compiler, and the ratio of SNAP size to PLAN size.

physical equality, meaning that if two arguments are considered equal only if they share the same identity (*i.e.* occupy the same region in memory). Clearly, physical equality implies structural equality, but not the other way around. Therefore, during the translation, we map PLAN's equality operator to the SNAP equality operator, but signal a warning if the arguments are polymorphic or are known not to be of primitive type.

5 Experimental Analysis

SNAP has already been demonstrated to run faster than PLAN [11]. Furthermore, network transit overheads tend to dominate overall application performance, rather than per-node processing overheads. For example, the compiler's output for the PLAN ping code presented in Figure 1 is only marginally slower than the tightly hand-tuned 7 instruction SNAP ping in Figure 2.

As such, the most important characteristic of our compiler with respect to performance is code size. As code size becomes bigger, an application must pay the overheads of transmitting the code around in its packets. To make matters worse, larger code segments leave less room for useful payload, decreasing application throughput.

In this section, we evaluate how compilation affects the resulting wire size of several PLAN programs. In most cases, the SNAP programs generated by the compiler are 30% smaller than their original PLAN versions. For those PLAN programs that are not linear in their resource usage, we see a an increase in code size.

5.1 Code Size Experiments

We ran our compiler on ten programs selected from the PLANet [3] distribution; they represent a variety of different networking tasks, from simple payload delivery (`deliver`) to information gathering (`devinfo, getNeighbors, getRoutes`) to multicast (`multiprint`) to simple network diagnostics (`ping`, `traceroute`).

Figure 4 contrasts the resulting packet sizes for the native PLAN wire format for each program with the packet size for the corresponding SNAP program produced by our compiler. Generally, the resulting SNAP programs for straight-line execution are 30% smaller than their PLAN equivalents. For two of the programs, `devinfo` and `multiprint`, the resulting SNAP programs are actually larger than the PLAN originals. A closer analysis of these pathological cases reveals a great deal about the importance of various optimizations in our compiler.

One first order effect is code bloat from loop unrolling; both programs iterate over all devices present on a given node. By default, the compiler unrolls `fold` five times. To understand the effect of unrolling, we parameterized our compiler not to unroll all of the `fold` operators, relying instead on simulating backward branches with `send`, as discussed in the previous section. In this case, the resulting SNAP program sizes were 728 bytes and 576 bytes respectively, giving SNAP/PLAN ratios of 1.24 and 0.71. This restores the typical 30% improvement for the multicast example, but not for `devinfo`.

If we furthermore apply the topological sorting of basic blocks as discussed earlier, we can trim the resulting size of the `devinfo` SNAP program to 544 bytes, resulting in a ratio of 0.92. Now at least, the SNAP program is smaller than the original PLAN version, but not by much. A quick perusal of the SNAP program reveals that most operations are (often redundant) stack management operations. Hand-tuning to eliminate cases like a `push` immediately followed by a `pop` results in a new program size of 476 bytes (a ratio of .81), much closer to the usual.

In the end, however, the compiled code must respect SNAP's linear resource restrictions. As they are, PLAN's `multiprint` and `devinfo` programs consume *non*-linear resources, since they iterate over the list of devices. Naive compilation results in code blowup proportional to a (conservative) bound on that iteration. We have shown that using backward branches, at the cost of one resource bound per branch, essentially results packet sizes proportional to the original PLAN packets. In other words, we can trade resource bound for compactness.

Of course, resource bound cannot be used to freely consume resources. In SNAP, resource bound is limited to 256 units per packet, which must be shared among that packet's progeny. The result is that the compiler must use resource bound for backward branches only sparingly, or run the risk of losing packets when they run out of resource bound.

6 Future Work and Conclusions

This work is part of larger project to build a "second-generation" active internetwork, called FASTnet. While FASTnet currently uses PLAN as its packet language, our goal is to use SNAP instead, relying on the compiler to seamlessly convert PLAN programs written by users into SNAP programs used by the network. FASTnet is implemented in TAL/Popcorn [14, 13], and takes advantage of dynamic code updating [5] to allow the system to evolve dynamically over time. Adding a SNAP byte-code machine into FASTnet should be straightforward.

To seamlessly incorporate SNAP into FASTnet requires more work on the compiler. The most important requirement is to improve the compiler's generated code quality, particularly in terms of code compactness. As we mentioned in Section 5, there are a number of optimizations that could net significant reductions in code size. We have already identified that topological sorting of basic blocks and careful arrangement of subexpressions to reduce stack reordering operations could result in substantial savings.

We also need to gain more experience in using PLAN programs with nonlinear resource consumption, such as the `devinfo` program from the previous section. Furthermore, we need to understand better what strategy to apply when a loop is not unrolled enough times to complete its task. We ultimately want the compiler to use reasonable heuristics to strike a balance between consuming resource bound and unrolling loops.

While PLAN is flexible and highly usable, SNAP is resource-safe and efficient. We have shown in this paper that compiling PLAN to SNAP allows us to gain the benefits of both languages, but requires us to overcome some significant challenges to map PLAN to SNAP's limited execution model. Initial performance measurements show that for simple programs, the compiler produces compact SNAP code that is approximately 30% smaller than the original PLAN code; programs that use iteration result in larger code sizes, owing in general to SNAP's resource usage model.

References

1. D. Scott Alexander, William A. Arbaugh, Michael Hicks, Pankaj Kakkar, Angelos Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The SwitchWare active network architecture. *IEEE Network*, 12(3), 1998.
2. M. Hicks and A. D. Keromytis. A secure PLAN. In *International Working Conference on Active Networks (IWAN)*, June 1999.
3. M. Hicks, J. T. Moore, D. S. Alexander, C. A. Gunter, and S. Nettles. PLANet: An Active Internetwork. In *IEEE INFOCOM*. IEEE, March 1999.
4. Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, September 1998.
5. Michael Hicks, Jonathan T. Moore, and Scott M. Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2001.
6. U. Legedza and J. Guttag. Using network-level support to improve cache routing. In *Proceedings of the 3rd International WWW Caching Workshop*, June 1998.

7. U. Legedza, D. Wetherall, and J. Guttag. Improving the Performance of Distributed Applications Using Active Networks. In *IEEE INFOCOM*. IEEE, March 1998.
8. L. Lehman, S. Garland, and D. Tennenhouse. Active Reliable Multicast. In *IEEE INFOCOM*. IEEE, March 1998.
9. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
10. Jonathan T. Moore. Practical active packets. Dissertation Proposal, University of Pennsylvania, February 2001.
11. Jonathan T. Moore, Michael Hicks, and Scott Nettles. Practical Programmable Packets. In *IEEE INFOCOM*, pages 41–50. IEEE, April 2001.
12. Jonathan T. Moore, Michael Hicks, and Scott M. Nettles. Chunks in PLAN: Language support for programs as packets. In *Proceedings of the 37th Annual Allerton Conference on Communication, Control, and Computing*, September 1999.
13. Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. Atlanta, May 1999.
14. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.
15. 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*, 18(1), February 2000.
16. D. J. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *IEEE OPENARCH*. IEEE, April 1998.