ESP: A Language for Programmable Devices

Sanjeev Kumar, Yitzhak Mandelbaum, Xiang Yu, and Kai Li
Princeton University
(skumar,yitzhakm,xyu,li)@cs.princeton.edu

ABSTRACT

This paper presents the design and implementation of Event-driven State-machines Programming (ESP)—a language for programmable devices. In traditional languages, like C, using event-driven state-machines forces a tradeoff that requires giving up ease of development and reliability to achieve high performance. ESP is designed to provide all of these three properties simultaneously.

ESP provides a comprehensive set of features to support development of compact and modular programs. The ESP compiler compiles the programs into two targets—a C file that can be used to generate efficient firmware for the device; and a specification that can be used by a verifier like SPIN to extensively test the firmware.

As a case study, we reimplemented VMMC firmware that runs on Myrinet network interface cards using ESP. We found that ESP simplifies the task of programming with event-driven state machines. It required an order of magnitude fewer lines of code than the previous implementation. We also found that model-checking verifiers like SPIN can be used to effectively debug the firmware. Finally, our measurements indicate that the performance overhead of using ESP is relatively small.

1. INTRODUCTION

Concurrency is a convenient way of structuring firmware for programmable devices. These devices tend to have limited CPU and memory resources and have to deliver high performance. For these systems, the low overhead of event-driven state machines often makes them the only choice for expressing concurrency. Their low overhead is achieved by supporting only the bare minimum functionality needed to write these programs. However, this makes an already difficult task of writing reliable concurrent programs even more challenging. The result is hard-to-read code with hard-to-find bugs resulting from race conditions.

The VMMC firmware [10] for Myrinet network interface cards was implemented using event-driven state machines in C. Our experience was that while good performance could be achieved with this approach, the source code was hard to maintain and debug. The implementation involved around 15600 lines of C code. Even after several years of debugging, race conditions cause the system to crash occasionally.

ESP was designed to meet the following goals. First, the language should provide constructs to write concise modular programs. Second, the language should permit the use of software verification tools like SPIN [14] so that the concurrent programs can be tested thoroughly. Finally, the language should permit aggressive compile time optimizations to provide low overhead.

ESP has a number of language features that allow development of fast and reliable concurrent programs. Concurrent programs are expressed concisely using processes and channels. In addition, pattern matching on channels allows an object to be dispatched transparently to multiple processes. A flexible external interface allows ESP code to interact seamlessly with C code. Finally, a novel memory management scheme allows an efficient and verifiably safe management of dynamic data.

We reimplemented the VMMC firmware on Myrinet network interface cards using ESP. We found that the firmware can be programmed with significantly fewer lines of code. In addition, since the C code is used only to perform simple operations, all the complexity is localized to a small portion of the code (about 300 lines in our implementation). This is a significant improvement over the earlier implementation where the complex interactions were scattered over the entire C code (15600 lines).

The SPIN verifier was used to develop and extensively test the VMMC firmware. Since, developing code on the network card often slow and painstaking, parts of the system were developed and debugged entirely using the SPIN simulator. SPIN was also used to exhaustively verify the memory safety of the firmware. Once the properties to be checked by the verifier are specified, they can be rechecked with little effort as the system evolves.

The ESP compiler generates efficient firmware. We used microbenchmarks to measure the worst-case performance overhead in the firmware. Based on earlier application studies [17, 5], we expect the impact of the extra performance overhead to be relatively small.

The rest of the paper is organized as follows. Section 2 presents the motivation for a new language. Section 3 describes our three goals and our approach. The next three sections (Sections 4, 5 & 6) describe how ESP meets each of the three goals. Section 4 provides the design of the ESP language. Section 5 describes how the SPIN model-checking verifier can be used to develop and test ESP programs. Section 6 shows how the ESP compiler generates efficient code and presents some performance measurements. Section 7 describes the related work. Finally, Section 8 presents our conclusions.

2. MOTIVATION

Devices like network cards and hard disks often include a programmable processor and memory (Figure 1). This allows the devices to provide sophisticated features in firmware. For instance, disk can support aggressive disk head scheduling algorithms in firmware.

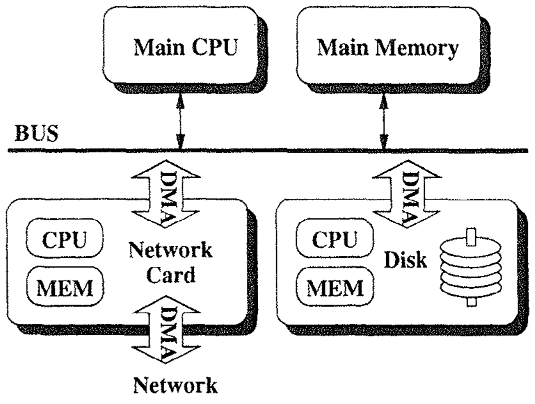


Figure 1: Programmable Devices

The firmware for programmable devices is often programmed using concurrency. Concurrent programs have multiple threads of control that coordinate with each other to perform a single task. The multiple threads of control provide a convenient way of keeping track of multiple contexts in the firmware. In these situations, concurrency is way of structuring a program that runs on a single processor.

Concurrent programs can be written using a variety of constructs like user-level threads or event-driven state machines. They differ in the amount of functionality provided and the overhead involved. However, the programmable devices tend to have fairly limited CPU and memory resources. Since these systems need to deliver high performance, the low overhead of event-driven state machines make them the only choice.

In this paper, we describe a language called ESP that can be used to implement firmware for programmable devices. We were motivated by our experience with implementing the VMMC firmware. We use VMMC firmware as a case study to evaluate the ESP language. In this section, we start with a description of VMMC. Then we examine the problems with event-driven state machines programming in traditional languages like C and motivate the need for a new language for writing firmware for programmable devices.

2.1 Case Study: VMMC Firmware

The VMMC architecture delivers high-performance on Gigabit networks by using sophisticated network cards. It allows data to be directly sent to and from the application memory (thereby avoiding memory copies) without involving the operating system (thereby avoiding system call overhead). The operating system is usually involved only during connection setup and disconnect.

The current VMMC implementation uses the Myrinet Network Interface Cards. The card has a programmable 33-MHz LANai4.1 processor, 1-Mbyte SRAM memory and 3 DMAs to transfer data—to and from the host memory; to send data out onto the network; and to receive data from the network. The card has a number of control registers including a status register that needs to be polled to check for data arrival, watchdog timers and DMA status.

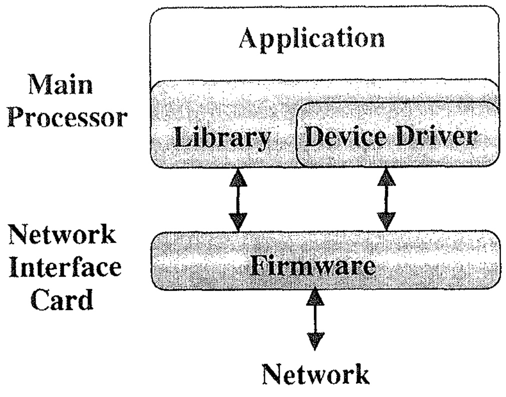


Figure 2: VMMC Software Architecture: The shaded regions are the VMMC components.

The VMMC software (Figure 2) has 3 components: a library that links to the application; a device driver that is used mainly during connection setup and disconnect; and firmware that runs on the network card. Most of the software complexity is concentrated in the firmware code which was implemented using event-driven state machines in C. The entire system was developed over several years and most of the bugs encountered were located in the firmware. Our goal is to replace the firmware using the ESP language.

2.2 Implementing Firmware in C

Event-driven state machines provide the bare minimum functionality necessary to write concurrent programs—the ability to block in a particular state and to be woken up when a particular event occurs. This makes them fairly difficult to program with. We illustrate event-driven state-machines programming in C with an example. The C code fragment is presented in Appendix A and is illustrated in Figure 3.

A program consists of multiple state-machines. For each state in a state machine, a handler is provided for every event that is expected while in that state. When an event occurs, the corresponding handler is invoked. The handler processes the event, transitions to a different state and blocks by returning from the handler. All the state machines share a single stack.

There are several problems with this approach. First, the code becomes very hard to read because the code gets fragmented across several handlers.

Second, since the stack is shared, all the values that are needed later have to be saved explicitly in global variables before a handler blocks. So data is passed between handlers through global variables (e.g. `pAddr`, `sendData`). In addition, global variables are also used by state machines to communicate with each other (e.g. `reqSM2`). It is very

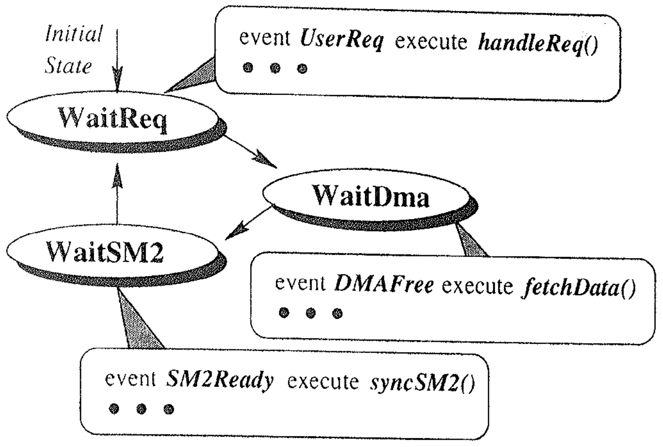

Initial State
event *UserReq* execute *handleReq()*
● ● ●
WaitReq
WaitDma
WaitSM2
event *DMAFree* execute *fetchData()*
● ● ●
event *SM2Ready* execute *syncSM2()*
● ● ●

Figure 3: Programming in C. The code is presented in Appendix A. A state machine is specified using a set of handlers. For each state in a state machine, a list of (event,handler) pairs has to be provided. When an event occurs, the corresponding handler is invoked. A handler is a C function takes no arguments and returns void.

hard to get the right synchronization to keep from clobbering data.

Third, memory to allocate buffers for the data has to be managed explicitly. In a concurrent setting, this is hard to implement correctly when data is used by several state machines before it is eventually freed. Depending on the timing, a different state machine might be the last one to use the data and, therefore, be responsible for freeing. When necessary, explicit reference counts have to be maintained. It is easy to overlook the need for adding reference counts to some data structures and introduce tricky allocation bugs that are hard to find.

Fourth, functions are an inappropriate abstraction mechanism for programming with state machines. This is because a state machine can block only by returning from a handler. As the firmware evolves, there might be a need to block within a function that is not a handler. For instance, in our original implementation, the function **translateAddr** was implemented as a simple table lookup. However, as the firmware evolved, the table became a cache of translations and the entire table was moved to the host memory. This meant that if there was a miss in the translation cache, the translation had to be DMAed from the host memory. But if the DMA was not available, it would need to block. This required extensive rewrite of the code and addition of more states to the state machine. In general, the amount of rewrite is proportional to the nesting depth of the function that wants to block.

Fifth, union datatypes are used extensively in these systems to encode different possible requests. So, a lot of handlers have a `switch` statement to deal with different requests. For instance, an application could request for a message to be sent `SendReq` or to update the virtual to physical translation `UpdateReq`. Since these requests are handled by the same handler `handleReq`, their code had to be colocated even when it makes more sense for these to be implemented in separate modules. A *dispatch* mechanism supported by the language would simplify the implementation.

Finally, hand-optimized fast paths are often built into the system to speed up certain requests. These fast paths rely on global information like the state of the various state machines and their data structures and violate every abstraction boundary. For instance, in VMMC firmware, a particular fast path is taken if the network DMA is free and no other request is currently being processed (this requires looking at the state of multiple DMAs). In addition, the fast path code updates global variables used for retransmission and might have to update the state of several state machines. These fast paths complicate the already complex state-machine code even further.

ESP aims to address these problems without incurring too much performance penalty. As we shall see, the ESP code corresponding to the C code (Figure 3) can be written much more succinctly and readably (Appendix B).

3. GOALS AND APPROACH

ESP is a language designed to support event-driven State-machines programming. It has the following goals:

Ease of development To aid programming, the language should permit the concurrency to be expressed simply. It should also provide support for modularity, dynamic memory management and a flexible interface to C.

Permit extensive testing Concurrent programs often suffer from hard-to-find race conditions and deadlock. ESP should support the use of software verifiers so that the programs can be tested extensively. Currently, ESP uses the SPIN verifier. SPIN [14] is a flexible and powerful verification system designed to verify correctness of software systems. It uses model-checking to explore the state-space of the system.

Low performance penalty These concurrent programs are designed to be run on a single processor. To have low performance overhead, concurrent programs in ESP should permit aggressive compile time optimizations.

In traditional languages, like C, using event-driven state-machines forces a tradeoff that requires giving up ease of development and reliability to achieve high performance. ESP is designed to provide all of these three properties simultaneously.

To meet these design goals, the ESP language is designed so that it can not only be used to generate an executable but also be translated into specification that can be used by the SPIN verifier (Figure 4). The ESP compiler takes an ESP program (`pgm.ESP`) and generates 2 files. The generated C file (`pgm.C`) can then be compiled together with the C code provided by the user (`help.C`) to generate the executable. The programmer-supplied C code implements simple device-specific functionality like accessing device registers. The SPIN file (`pgm.SPIN`) generated by the ESP compiler can be used together with programmer-supplied SPIN code (`test.SPIN`) to verify different properties of the system. The programmer-supplied SPIN code generates external events such as network message arrival as well as specifies the properties to be verified. Different properties of the system can be verified by using `pgm.SPIN` together with different `test.SPIN` files.

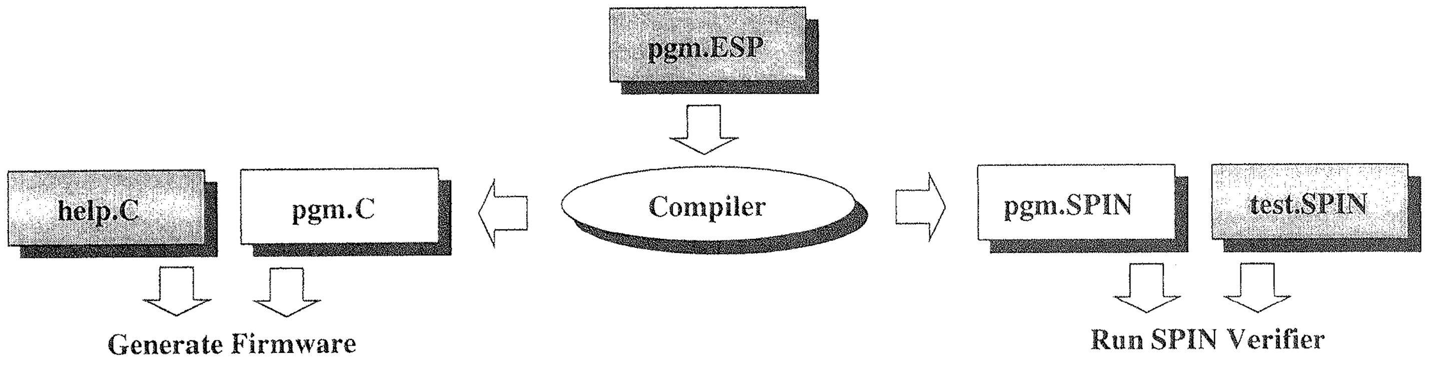


Figure 4: Shaded regions are code provided by the user

4. EVENT-DRIVEN STATE-MACHINES PROGRAMMING (ESP) LANGUAGE

ESP is based on the CSP [13] language and has a C-style syntax. ESP supports Event-driven State-machines Programming. The basic components of the language are processes and channels. Each process represents a sequential flow of control in a concurrent program. Processes communicate with each other by sending messages on channels. All the processes and channels are static and known at compile time.

Appendix B presents the implementation of the example (Section 2.2) in ESP. In this section, we will use fragments from that code to illustrate the various language features.

4.1 Types, Expressions and Statements

ESP supports basic types like `int` and `bool` as well as mutable and immutable versions of complex datatypes like `record`, `union` and `array`.[1] Types can be declared as follows:

```
type sendT = record of { dest: int, vAddr: int, size: int}
type updateT = record of { vAddr: int, pAddr: int}
type userT = union of { send: sendT, update: updateT, ...}
```

ESP does not provide any global variables. All variables have to be initialized at declaration time (New variable declaration is indicated with a $ prefix). Types do not have to be specified when they can be deduced (ESP does a simple type inferencing on a per statement basis). For instance:

```
$i: int = 7;                        // Declare Variable
i = 45;                             // Update Variable
$j = 36;                            // Type inferred
```

ESP provides the common imperative constructs like `if-then-else` statements and `while` loops. However, it does not provide recursive data types or functions. Recursive data types are not supported because they cannot be translated easily into the specification language of the SPIN verifier. Functions are not supported because processes provide a more appropriate abstraction mechanism in a concurrent setting (Section 4.3).

4.2 Channels

Communication over channels are synchronous—a sender has to be attempting a send (using the *out* construct) concurrently with a receiver attempting to receive (using the *in* construct) on a channel before the message can be successfully transferred over the channel. Consequently, both *in* and *out* are blocking operations. Using synchronous channels has several benefits. First, they simplify reasoning about message ordering, especially when processes can have complex interactions. Second, they can be implemented more efficiently than buffered channels. When buffering is required, it can be implemented explicitly by the programmer. Finally, buffered channels increase the size of state-space that has to be explored during verification.

[1]A # prefix indicates a mutable data structure.

The *alt* construct allows a process to wait on the *in/out* readiness of multiple channels. However, for each execution of an *alt* statement, only the actions associated with a single channel are performed. In the case where multiple channels are ready, a single channel is selected. The channel selection algorithm need not be fair (it may favor performance critical channels), but must prevent starvation [20]. The following is a code fragment from a process that implements a FIFO queue. The macros `FULL`, `EMPTY` and `INCR` have the expected functionality. The first alternative accepts new messages and inserts them at the tail of the queue. The second alternative sends the message at the head of the queue and then removes it from the queue. Note that the first alternative is disabled when the buffer is full and second is disabled when the buffer is empty.

```
while {
  alt {
    case( !FULL, in( chan1, Q[tl])     { INCR(tl); }
    case( !EMPTY, out( chan2, Q[hd])) { INCR(hd); }
  }
}
```

One of the features of the language is the use of pattern matching to support dispatch. Pattern-matching is used in languages like ML to provide more expressive switch statements. ESP uses it to support dispatch. Patterns have the same syntax as the one used for allocating unions and records. They can be differentiated based on their position in a statement. They are considered a pattern when they occur in an *lvalue* position and cause allocation when they occur in a *rvalue* position.

```
$sr: sendT = { 7, 54677, 1024};
$ur1: userT = { send |> sr};
$ur2: userT = { send |> { 5, 10000, 512}};
{ send |> { $dest, $vAddr, $size}}: userT = $ur2;
```

In the above code, the first line initializes `sr` to a newly allocated record. The second line initializes `ur1` to a newly allocated union with a valid **send** field[2] that points to the record in **sr**. The third line initializes **ur2** to a newly allocated union with a valid **send** field that points to a newly allocated record. The fourth line has a pattern on the left hand side and pattern matching causes variables **dest**, **vAddr** and **size** to be initialized to 5, 10000 and 512 respectively

Patterns can be specified in an **in** operation. For example, consider process A performs

```
in( userReqC, { send |> { $dest, $vAddr, $size}});
```

to accept only send requests while a process B performs

```
in( userReqC, { update |> { $vAddr, $pAddr}});
```

to accept only update requests. When process C performs

```
out( userReqC, req);
```

the object will be delivered to process A or B depending on which pattern it matches. This frees the process C from trying to figure out the appropriate processes and sending the message to that process. To support this functionality efficiently, ESP requires that all the patterns used on a channel have to be disjoint and exhaustive—an object has to match exactly one pattern. In addition, each pattern can be used by one process only. So, although a channel can have multiple readers and writers, a channel together with a pattern defines a port which can have multiple writers but only a single reader.

Objects sent over channels are passed by value. Since there are no global variables, this ensures that processes can communicate only by sending messages over channels. To support this efficiently, ESP allows only immutable objects to be sent over channels. This applies not only to the object specified in the **out** operation but also to all objects recursively pointed to by that object.

A cast operation allows casting an immutable object into a mutable object and vice versa. Semantically, the cast operation causes a new object to be allocated and the corresponding values to copied into the new object. However, the compiler can avoid creating a new object in a number of cases. For instance, if the compiler can determine that the object being cast is no longer used afterwards, it can reuse that object and avoid allocation.

4.3 Processes

Processes in ESP implement state machines—each location in the process where it can block implicitly represents a state in the state machine.

```
process add5 {
  while( true) {
    in( chan1, $i);
    out( chan2, i+5);
  }
}
```

The above process represents a state machine with 2 states. The first state is when it is blocked waiting on an **in** operation on channel **chan1** and the second when it is blocked on an **out** operation on channel **chan2**.

Processes in ESP are lightweight in that they do not need a stack to run. This is because ESP does not support functions, allowing the local variables of a process to be allocated in the static region. Thus a context switch only requires saving the current location in one process and jumping to the saved location in another.

In ESP, the processes are used to support abstraction—functions are not supported. For example, consider the following code fragment from a process which implements a page table which maps virtual addresses into physical addresses (Appendix B). The mapping is maintained in the array **table**. When it receives a request to translate virtual address to physical address, it uses the virtual address to lookup the mapping and sends a reply back to the requesting process. The **ret** specifies the process making the request so that the reply can be directed back to that process. The second *case* accepts requests to update the mapping and updates the table.

```
alt {
  case( in( ptReqC, { $ret, $vAddr})) {
    // Request to lookup a mapping
    out( ptReplyC, { ret, table[vAddr]});
  }
  case( in( userReqC, { update |> { $vAddr, $pAddr}})) {
    // Request to update a mapping
    table[vAddr] = pAddr;
  }
}
```

To mimic the behavior of functions that expect return values, a pair of **out** and **in** operations. For instance:[3]

```
out( ptReqC, { @, vAddr});
in( ptReplyC, { @, $pAddr});
```

On the other hand, functions that do not expect a return value can be modeled using an **out** operation

```
out( userReqC, { update |> { vAddr, pAddr}});
```

ESP processes are a more appropriate abstraction mechanism than functions in a concurrent setting because an ESP process can block on an event, while allowing such behavior in a function cannot be done without a stack (Section 2.2). In addition, the process abstraction allows flexibility in scheduling computation. For instance, if no return values are expected (see last example), the code to update the table can be delayed until later.

4.4 Memory Management

Memory allocation bugs are often the hardest to find especially in the context of concurrent programming. However, supporting automatic memory management usually involves too much overhead (both in terms of space and time). On the other hand, explicit memory management with **malloc** and **free** are hard to program correctly with.

ESP provides a novel explicit management scheme to allow efficient but bug free memory management. The key observation is that memory bugs are hard to find because memory safety is, usually, a global property of a concurrent program—memory safety cannot be inferred by looking only at a part of the program. To rectify this, ESP is designed to make memory safety a local property of each process.

When objects are sent over channels, deep copies of the objects are delivered to the receiving process.[4] Hence, there

[2]Exactly one field of a union has to be valid

[3]@ is a constant different for each process (a process id).

[4]This is true only semantically. The implementation never has to actually copy the object.

is no overlap between the objects accessible to different processes. Therefore, each process is responsible for managing its own objects. Bugs in the other processes do not effect it.

ESP provides a reference counting interface to manage memory.[5] At allocation time,[6] the reference count is set to 1. ESP also provides 2 primitives (`link` and `unlink`) to manipulate the reference counts. The `link` primitive increases the reference count of the object while the `unlink` decreases the reference count of the object. If this causes the reference count of an object to become 0, it frees the object and recursively invokes unlink on the objects pointed by it.

ESP is designed so that `link` and `unlink` are the only source of unsafeness in language. However, since the unsafeness is local to each process, the SPIN verifier can be used to verify safety of each process separately. This makes it less vulnerable to state-explosion in the verifier. In fact, the SPIN verifier was able to verify the safety of all processes used to implement the VMMC firmware fairly easily (Section 5.3).

4.5 External Interface

The firmware implementation has to deal with special registers, volatile memory and layout of packets sent/received on the network. ESP addresses this by providing an external interface to interact with C code.

In addition, the specification derived from the ESP code has to interact with some programmer provided SPIN code during verification (Figure 4).

ESP provides a single external interface for both SPIN and C code. It uses the channel mechanism to support external interfaces. This is different from the traditional approaches of either allowing C code to be directly embedded in the program [6, 2] or allowing functions that are implemented externally to be called [3, 8].

Using channels to provide external interfaces has a number of advantages. First, ESP processes often block on external events like arrival of user request or network packets. Using channels allows a process to use the existing constructs to block on external events. Second, external code can also use the same dispatch mechanism built into channels through pattern-matching. Finally, it promotes modularity. For instance, if retransmission is no longer required, the retransmission processes can be dropped and the channels used to interact can be converted into external channels. Other processes that were using these channels are not effected because they cannot tell the difference between an external channel and a regular channel.

A channel can be declared to have an external reader or writer but not both. For example:

```
channel userReqC: userT    // External C writer
interface userReq( out userReqC) {
  Send( { Send |> { $dest, $vAddr, $size}),
  Update( { Update |> $new}),
  ...
}
```

defines a channel with a external writer. The $ prefix in the pattern indicates a parameter to be passed to the C function.

[5]The inability of reference counting to deal with cycles poses no problems to ESP because it does not have circular data structures.

[6]Objects received over channels are treated as newly allocated objects.

Interface to C. To support a synchronous C interface, ESP requires two types of functions to be provided. The first type has a "IsReady" suffix and returns whether the channel has data to send/receive. The second type of function is called after the first one has indicated if it is ready to communicate. So, in the previous example, the following functions have to be provided by the programmer.

```
int UserReqIsReady( void);
void UserReqSend( int *dest, int *vAddr, int *size);
void UserReqUpdate( int **new);
...
```

`UserReqIsReady` should return 0 when it has nothing to send. When it has something to send, it returns a integer that specifies which one of the patterns is ready. A separate function has to be provided for each of the patterns specified. The use of patterns in this context serves 2 purposes. First, it supports dispatch on external channels. Second, it minimizes the amount of allocation and manipulation of ESP data structures that has to be done in C. For instance, by specifying the entire pattern in `UserReqSend`, there is no need for that function to allocate any ESP data structure. `UserReqUpdate`, on the other hand, will have to allocate, correctly initialize and return an ESP record. This can not only introduce allocation bugs in the system but also move the allocation beyond the reach of the ESP compiler, thereby, preventing the allocation from being optimized away.

External `in` channels differ from external out channel in 2 ways. First, the `IsReady` function just returns whether or not the channel is willing to accept data. Then any writer on that channel can write to it. In addition, it does not need to pass pointers since the parameters will not be modified. So, all the parameters have one less level of indirection.

SPIN Interface. Since SPIN has support for channels, external SPIN code can interact directly with SPIN by reading and writing to the appropriate channels.

4.6 Case Study: VMMC Firmware

We have reimplemented the VMMC firmware using ESP. The implementation supports most of the VMMC functionality (only the redirection feature is currently not supported)

The earlier implementation included about 15600 lines of C code (Around 1100 of these lines were used to implement the fast paths).[7]

The new implementation using ESP uses 500 lines of ESP code (200 lines of declarations + 300 lines of process code) together with around 3000 lines of C code.[8] The C code is used to implement simple tasks like initialization, initiating DMA, packet marshalling and unmarshalling and shared data structures with code running on the host processor (in the library and the driver). All the complex state machine interactions are restricted to the ESP code which uses 7 processes and 17 channels. This is a significant improvement over the earlier implementation where the complex interactions were spread throughout the 15600 lines of hard-to-read code.

[7]To make a fair comparison, we counted only those lines of the earlier implementation that correspond to functionality implemented in the new VMMC implementation using ESP

[8]ESP currently does not provide any support for fast paths.

5. DEVELOPING AND TESTING USING A VERIFIER

We have a working prototype of the ESP compiler. It generates both C code that can be compiled into firmware as well as a specification that can be used by the SPIN verifier (Figure 4). In this section, we start with a description of the SPIN model checker. We then describe how ESP code is translated into SPIN specification. Finally, we present our experience with using the SPIN model checker to develop and extensively test the VMMC firmware.

5.1 SPIN Model Checking Verifier

Model checking is a technique for verifying a system composed of concurrent finite-state machines. Given a concurrent finite-state system, a model checker explores all possible interleaved executions of the state machines and checks if the property being verified holds. A *global state* in the system is a snapshot of the entire system at a particular point of execution. The *state space* of the system is the set of all the global states reachable from the initial global state. Since the state space of such systems is finite, the model checkers can, in principle, exhaustively explore the entire state space.

The advantage of using model checking is that it is automatic. Given a specification for the system and the property to be verified, model checkers automatically explore the state space. If a violation of the property is discovered, it can produce an execution sequence that causes the violation and thereby helps in finding the bug.

The disadvantage is that the state space to be explored is exponential in the number of processes and the amount of memory used (for variables and data structures). So the resources required (CPU as well as memory resources) by the model checker to explore the entire state space can quickly grow beyond the capacity of modern machines.

SPIN [14]. It is a flexible and powerful model checker designed for software systems. SPIN supports high-level features like processes, rendezvous channels, arrays and records. Most other verifiers target hardware systems and provide a fairly different specification language. Although ESP can be translated into these languages, additional state would have to be introduced to implement features like the rendezvous channels using primitives provided in that specification language. This would make the state explosion problem worse. In addition, the semantic information lost during translation would make it harder for the verifiers to optimize the state-space search.

SPIN supports checking for deadlocks and verifying simple properties specified using assertions. More complex properties, like absence of starvation, can be specified using Linear Temporal Logic (LTL).

SPIN is an on-the-fly model checker and does not build the global state machine before it can start checking for the property to be verified. So, in cases where the state space is too big to be explored completely, it can do partial searches. It provides 3 different modes for state-space exploration. The entire state space is explored in the *exhaustive* mode. For larger systems state spaces, the *bit-state hashing* mode performs a partial search using significantly less memory. The *simulation* mode explores single execution sequence in the state space. A random choice is made between the possible next states at each stage. Since it does not keep track of the states already visited and could explore some states multiple times while never exploring some other states. However, the simulation mode in SPIN usually discovers most bugs in the system. Most simulators are designed to accurately mimic the system being simulated. So, hard to find bugs that occur infrequently on the real system also occur infrequently on the simulators. The SPIN simulator is different in that it makes a random choice at each stage and is, therefore, more effective in discovering bugs.

5.2 Translating ESP into SPIN Specifications

The ESP code can be translated into the SPIN specification at various stages of the compilation process. The ESP compiler does this very early—right after type checking—for several reasons. First, the SPIN specification language does not support pointers. So, the translation is much more difficult at the latter stage because it would require the compiler to carry some of the type information through the transformations on the intermediate representations. Second, the addition of temporary variables during the compilation increases the size of the state space that must be explored. The one disadvantage is that any bugs introduced by the compiler cannot be caught by the verifier.

The ESP compiler generates SPIN specification that can instantiate multiple copies of the ESP program. This is achieved easily in SPIN by using an array of every data structure. Then each instance can access its data by using its instantiation id. The ability to run multiple copies of a ESP program under SPIN allows one to mimic a setup where the firmware on multiple machines are communicating with each other.

The translation into SPIN specification is fairly straightforward with a few exceptions. These stem from the lack of pointers and dynamic allocations. While ESP allows the size of the arrays to be determined at run time, SPIN requires it to be specified at compile time. This problem is addressed by using arrays of a fixed maximum size. This size can be specified per type.

Another problem arising from the lack of pointers in SPIN is dealing with mutable data types. For instance,

```
$a1: #array of int = #{ 5 -> 0, ...};  // Allocate
$a2 = a1;                              // Copy pointer
$a2[3] = 7;                            // Update
```

Here, an update to a2 has to be visible to a1. Since, SPIN does not support pointers, different memory is allocated for a1 and a2 and an assignment causes the entire structure to be copied. This causes a problem with mutable data structures because an update to one structure a2 has to be visible in the other a1. We address this by assigning an *objectId* to all objects at allocation time. So, when objects get copied, the objectId also gets copied. Later, when a structure is updated, we update the all structures with the same objectId. Although, this may appear very inefficient, it does not increase the state-space that has to be explored and, therefore, does not significantly impact the verifiability of the system.

Memory safety of each individual process can be verified independently using the verifier (Section 4.4). To verify memory safety, we maintain a table that maps the objectId of the objects to reference count. Before each object access, the compiler inserts an assertion to verify that the object is live. The objectId is reclaimed when the reference count falls to 0 and the object is freed. One positive side-effect of having to use fixed size reference count table is that the verifier can often catch memory leaks. This is because a memory leak can cause the system to run out of objectIds during verification.

5.3 Case Study: VMMC Firmware

The motivation for using a verifier is to allow more extensive testing than achievable with conventional methods. In the earlier VMMC implementation, we encountered new bugs every time we tried a different class of applications or ran it on a bigger cluster. The state-space exploration performed by verifiers allows more extensive testing.

We used SPIN throughout the development process. Traditionally, model checking is used to find hard-to-find bugs in working systems. However, since developing firmware on the network interface card involves a slow and painstaking process, we used the SPIN simulator to implement and debug it. Once debugged, the firmware can be ported to the network interface card with little effort.

As explained earlier (Figure 4), the programmer has to supply some test code (*test.SPIN*) for each property to be checked. The code not only specifies the property to be verified but also simulates external events such as network message arrival. The test code is usually less than 100 lines each. Once written, these can be made part of the testing suite and used to recheck the system whenever changes are made to it.

We have successfully used the SPIN verifier in a number of situations. They include:

Development of Retransmission Protocol. The retransmission protocol (a simple sliding window protocol with piggyback acknowledgement) was developed entirely using the SPIN simulator. The SPIN test code used was 65 lines. Once debugged, the retransmission protocol was compiled into the firmware. It ran successfully on the network card without encountering any new bugs. The retransmission protocol in the earlier implementation required about 10 days to get a working version. Since we developed our code using SPIN, it required 2 days.

Checking Memory Safety. Since memory safety is a local property of each process, each process can be checked separately for memory safety. To verify the memory safety of the biggest process in the firmware required 40 lines of test code. The entire state space was 2251 states and could be explored using exhaustive search mode in the SPIN verifier. It took 0.5 second to complete and required 2.2 Mbytes of memory. It should be noted that an exhaustive search would not only catch all the memory safety bugs but also some memory leaks. The result is a safe system that does not incur the overhead of garbage collection.

The firmware had been debugged by the time our memory safety verifier was developed. So we ran the verifier on an earlier version of the system that had a bug. The bug was identified by the verifier. We also introduced a variety of memory allocation bugs that access data that was already freed or introduce memory leaks. The verifier was able to find the bug in every case.

State-space explosion prevented us from checking for system-wide properties like absence of deadlocks. We are currently working on extracting more abstract models so that the state-space search is more tractable. This has allowed us to find several bugs in the firmware that can cause deadlocks [15].

6. GENERATING EFFICIENT FIRMWARE

As described earlier (Figure 4), the ESP compiler uses C as backend and generates C code that can be used to generate the firmware. In this section, we describe the ESP compiler and then compare the performance of the new VMMC implementation using ESP with the earlier implementation.

6.1 ESP Compiler

Processes. The ESP compiler requires the entire program for compilation. It does whole-program analysis and generates one big C function that implements the entire concurrent program. One approach is to treat each process as an automaton and to combine them to generate one large automaton [3, 18]. Although this approach provides zero-overhead context switching, it can result in exponential growth in code size [11]. The ESP compiler takes a simpler approach. It generates the code for the processes separately and context switches between them. Since these processes are essentially state machines, the stack does not have to be saved during a context switch—only the program counter needs to be saved and restored. This has a fairly low overhead and involves only a few instructions.

The generated code has an idle loop that polls for messages on external channels. When a message is available, it checks to see if a process is waiting for that message. If there is, it restarts that process by jumping to the location where the process was blocked. The process then executes till it reaches a synchronization point. If one or more processes are blocked waiting to synchronize, it picks one randomly and completes the message transfer. At this point, both the synchronizing processes can continue executing. ESP currently uses a simple stack-based scheduling policy. This scheduling policy picks one of these two processes to continue execution and adds the other one to the ready queue (queue of processes that are waiting to execute). The processes are executed non-preemptively. When the running process eventually blocks, the next process in the ready queue is executed. This is repeated till there are no more processes to run and the program returns to the idle loop.

The ESP compiler performs some of the traditional optimizations like copy propagation and dead code elimination on each process separately before combining them to generate the C code. Although, the C compiler also performs these optimizations, the semantic information lost when the processes are combined to generate the C code makes it hard for the C compiler to perform these optimizations effectively.

Channels. One way of implementing channels is to have a set of queues (one for each pattern used on the channel) that writers can wait on.[9] This approach makes *alt* fairly expensive. This is because, before blocking on an *alt* statement, the process has to be added to multiple queues (one for each *case* in the *alt*). When it is later unblocked, it has to

[9]Although there can be multiple readers on a channel, there can only be one reader per-pattern on a channel. So a queue is not needed for the readers.

be removed from all these queues (which can require looking through the queue since it might be in the middle of the queue).

The ESP compiler takes a different approach. It uses a bit-mask per process—one bit for every channel the process may block on. Blocking at an *alt* statement requires simply setting the right bit mask for the process, while unblocking requires zeroing out all the bits. This approach can have two problems. First, checking if a channel has a writer now requires checking the bit masks of multiple processes (as opposed to just checking the corresponding queue). However, since each process uses only a few bits (much fewer than 32), the bit masks for several processes can be colocated on a single integer at compile time. Colocating the right processes can reduce the number of different masks to be checked to 1 or 2. Second, we lose the FIFO ordering of the queues, and extra effort must be made to avoid introducing starvation. However, most of the time only one other process is waiting. No extra overhead is incurred in the common case.

Another simple optimization that helps *alt*'s performance is postponing as much computation as possible until after the rendezvous. For instance, if an object has to be allocated before being sent over the channel, the allocation is postponed so that the allocation does not happen if one of the other alternatives succeeds.

Messages on channels. Semantically, messages sent over the channels require deep copies to be handed to the receiving processes. However, the implementation can simply increment the reference count of the objects to be sent over channel and just send pointers to those objects. This works because only immutable objects can be sent over channels.

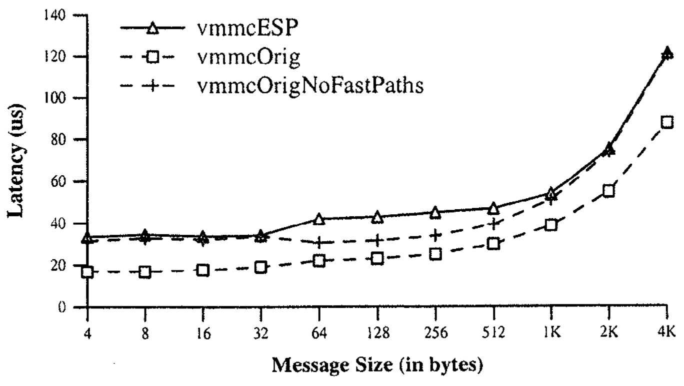
The ESP compiler also avoids some unnecessary allocation associated with pattern matching. For instance, if a process wants to send more that one value over a channel, it has to put it in a record. If the receiving process is using a pattern to access the components, the compiler can avoid allocating the record. This is possible because the static design of the language allows the compiler to look at all the patterns being used to receive messages on a channel along with all the senders on that channel.

6.2 Case Study: VMMC Firmware

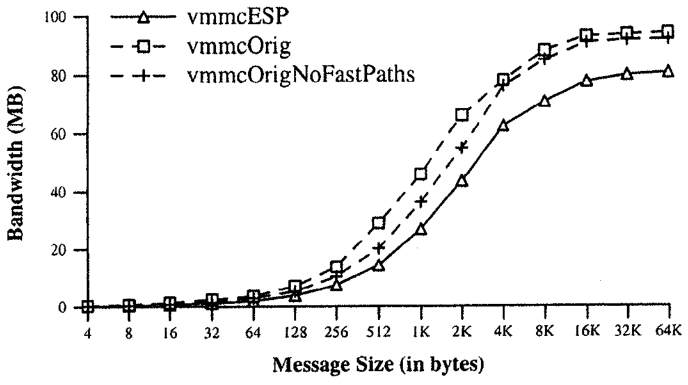
Figure 5 compares the performance of the earlier VMMC implementation (*vmmcOrig*) with the performance of the new implementation using ESP (*vmmcESP*) using 3 microbenchmarks. In addition, we also present the performance of the earlier implementation with the fast paths disabled (*vmmcOrigNoFastPaths*). The ESP implementation currently does not implement fast paths.

The first microbenchmark measures the latency of messages of different sizes between applications running on 2 different machines. This is measured by running a simple pingpong application that send messages back and forth between 2 machines. Figure 5(a) shows that *vmmcESP* is around twice as slow as *vmmcOrig* for 4 byte messages and 38 % slower for 4 Kbyte messages. However, *vmmcESP* is only 35 % slower than *vmmcOrigNoFastPaths* in the worst case (for 64 byte messages) but has comparable performance for 4 byte and 4 Kbyte messages.

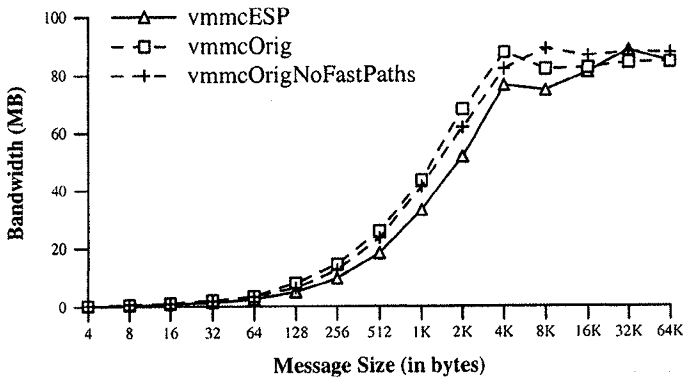
The second microbenchmark measures the bandwidth between two machines for different message sizes. In this case, an application running on one machine continuously sends



(a) Latency



(b) One Way Bandwidth



(c) Bidirectional Bandwidth

Figure 5: Microbenchmarks Performance. The graphs have some discontinuities at the 32/64 byte boundary as well as at 4/8Kbyte boundary. The former is because small messages of 32 bytes and less are handled separately as a special case. The latter is because the page size is 4Kbytes.

data of particular size to the second machine which simply receives it. Figure 5(b) shows that *vmmcESP* delivers 41 % less bandwidth as *vmmcOrig* for 1 Kbyte messages and 14 % for 64 Kbyte messages. However, *vmmcESP* is only 25 % slower than *vmmcOrigNoFastPaths* for 1 Kbyte messages and 12 % for 64 Kbyte messages.

The final microbenchmark measures the total bandwidth between two machines for different message sizes in a different scenario. In this case, applications on two machines continuously send data to each other simultaneously. Figure 5(c) shows that *vmmcESP* delivers 23 % less bandwidth as *vmmcOrig* for 1 Kbyte messages but similar performance for 64 Kbyte messages. Also, *vmmcESP* is 20 % slower than *vmmcOrigNoFastPaths* for 1 Kbyte messages but similar performance for 64 Kbyte messages.

The microbenchmark performance shows that *vmmcESP* performs significantly worse that *vmmcOrig* in certain cases (latency of small messages). However, most of the performance difference is due to the brittle fast paths. Also, the performance difference is significantly less in the bidirectional bandwidth microbenchmark where the firmware has to deal with messages arriving on the network as well as the host at the same time. In the other two microbenchmark, the firmware has to deal with only one type of message at a given instant.

The microbenchmarks represent the worst case scenario. The impact of the performance difference on real applications should be much smaller [17, 5] for a number of reasons. First, the *vmmcOrig* numbers represent the performance of some hand-optimized fast paths in the system. These fast paths tend to be fairly brittle and applications often fall off the fast path. While some applications [16] (which repeatedly send very large messages) that have very simple communication patterns benefit from the fast paths, a lot of applications do not. SVM applications [4] experience a lot of contention in the network and the actual latency measured by the different applications varied between 3 times to 10 times slower than the microbenchmarks numbers for small messages. So, for most applications, the *vmmcOrigNoFastPaths* is a more accurate representative than *vmmcOrig* when comparing performance with *vmmcESP*.

Second, the microbenchmarks represent applications that spend 100 % of their time communicating, while most real applications spend only a fraction of their time communicating and are, therefore, less sensitive to firmware performance [17, 5].

Finally, we plan to implement more aggressive optimizations that should decrease the performance gap. For instance, data-flow analysis is currently performed on a per process basis. We plan to extend data-flow analysis across processes.

7. RELATED WORK

Devices are usually programmed using event-driven state machines in languages like C, and sometimes, in assembly. We are not aware of any other high-level language for programming network devices.

Concurrency Theory. A number of languages like CSP [13] and Squeak [6] have been designed to gain better understanding of concurrent programming. Both of these languages support processes communicating with each other. However, they were not designed with efficient implementation in mind.

Concurrent Languages. A number of languages like CML [19], Java [1] and OCCAM [20] support concurrency. CML [19] provides first-class synchronous operations. OCCAM [20] was designed to implement concurrent programs that run on a parallel machine. Java [1], like most other programming languages, provides user-level threads to express concurrency. All these systems are fairly expressive and hard to be compiled efficiently for devices.

Code Generation+Verification. A number of other languages [3, 8, 2] have taken a similar approach of generating efficient executables as well as specifications that can be used by a verifier. However, they differ from ESP significantly.

Esterel [3] was designed to model the control of synchronous hardware and has been used recently to efficiently implement a subset of TCP protocol [7]. It adopts the *synchronous hypothesis*—the reaction to an external event is instantaneous—and ensures that every reaction has a unique, and therefore, deterministic reaction. This makes the programs easier to analyze and debug. The esterel programs can be compiled to generate both software and hardware implementations. However, using esterel to implement device firmware has several drawbacks. First, the reactions are not instantaneous in practice. For instance, if a DMA becomes available while an event was being processed, it cannot be used to process the current event. The "DMA available" event would be registered on the next clock tick and would be then available for use. This results in inefficient use of the DMA. Second, the synchronous hypothesis forces some constraints on valid programs. For instance, every iteration of a loop has to have a "time consuming" operation like signal emission. In addition, this constraint has to be verifiable by the compiler. This disallows simple loops that initialize an array. Finally, the language is designed to encode only the control portion of the program. The data handling has to be performed externally using the C interface. This forces some of the complex tasks including memory management to be implemented in C.

Teapot [8] is a language for writing coherence protocols that can generate efficient protocols as well as verify correctness. It uses a state machine to keep track of the state of a coherence unit (a cache line or a page). The state machine is specified using a set of handlers similar to the C interface described in Section 2.2. However, they use continuations to reduce the number states that the programmer has to deal with. While this approach works well when applied to coherence protocol, it suffers for some of the same problems described in Section 2.2 when used to implement device firmware. Teapot also does not provide any support for complex datatypes and dynamic memory management.

Promela++ [2] is a language designed to implement layered network protocols. The adjacent layers communicate using FIFO queues. Although, the layered framework works well for writing network protocols, they are too restrictive for writing firmware code where the different modules have much more complex interactions. Also, they do not provide any support for dynamic memory management.

Software Testing. Some systems [12, 9] have been successful in finding bugs in existing software written in traditional languages like C. Verisoft [12] does this by modifying the scheduler of the concurrent system to do a state-space exploration. Meta-level Compilation [9] attempts to verify system-specific invariants at compile time. However, these systems do not simplify the task of writing concurrent programs.

8. CONCLUSIONS

We have presented the design and implementation of ESP—a language for programmable devices. ESP has a number of language features that allow development of compact and modular concurrent programs. ESP programs can be developed and debugged using the SPIN model-checking verifier. The compiler automatically generates SPIN specifications from ESP programs. Once debugged, ESP programs can be compiled into efficient firmware that runs on the programmable device.

We have reimplemented VMMC firmware for the Myrinet network interface cards using ESP. Our main conclusions are the following:

- Programming event-driven state machines can be fairly easy with the right language support. We found that the firmware can be programmed with significantly fewer lines of code. In addition, since C code is used only to perform simple operations, all the complexity is localized to a small portion of the code (about 300 lines in our implementation). This is a significant improvement over the earlier implementation where complex interactions were scattered over the entire C program (15600 lines).
- Model-checking verifiers like SPIN can be used to extensively test the firmware. However, state-space explosion limits the size of the models that can be checked. SPIN was used to develop and debug a retransmission protocol. The new implementation took around 2 days (compared to the earlier implementation which took around 10 days). SPIN was also used to exhaustively check the memory safety on the firmware.
- The performance overhead of using ESP is relatively small. Our microbenchmarks measurements indicate that most of the performance difference with the earlier implementation of VMMC is due to brittle fast paths that rarely benefit applications. Based on earlier application studies [17, 5], we expect the impact of the extra performance overhead to be relatively small.

9. ACKNOWLEDGEMENTS

We would like to thank Rudrajit Samanta, Tammo Spalink, Daniel Wang, Dirk Balfanz and the anonymous reviewer whose comments have helped improve the paper.

10. REFERENCES

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, Third Edition.* Addison-Wesley Publications, 2000.

[2] A. Basu, T. von Eicken, and G. Morrisett. Promela++. A language for correct and efficient protocol construction. In *Infocom*, 1998.

[3] G Berry and G. Gonthier The ESTEREL synchronous programming language : design, semantics, implementation. *Science of Computer Programming*, 19(2), 1992.

[4] A. Bilas, C. Liao, and J. Singh. Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems. In *International Symposium on Computer Architecture*, June 1999.

[5] A. Bilas and J. P. Singh. The effects of communication parameters on end performance of shared virtual memory clusters. In *SC97 conference*, Nov 1997.

[6] L. Cardelli and R. Pike. Squeak: a language for communicating with mice. *Computer Graphics*, 19(3):199–204, July 1985.

[7] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating efficient protocol code from an abstract specification. In *SIGCOMM*, 1996.

[8] S. Chandra, B. E. Richards, and J. R. Larus. Teapot: Language support for writing memory coherence protocols. In *Programming Language Design and Implementation*, 1996.

[9] A. Chou, B. Chelf, D. Engler, and M. Heinrich. Using meta-level compilation to check flash protocol code. In *Architectural Support for Programming Languages and Operating Systems*, 2000.

[10] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: efficient support for reliable, connection-oriented communication. In *Proceedings of Hot Interconnects*, Aug. 1997.

[11] S. A. Edwards. Compiling esterel into sequential code. In *Design Automation Conference*, 2000.

[12] P. Godefroid. Model checking for programming languages using VeriSoft. In *Principles of Programming Languages*, Paris, France, 1997.

[13] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[14] G. J. Holzmann. The SPIN model checker. *IEEE Transaction on Software Engineering*, 23(5):279–295, May 1997.

[15] S. Kumar. ESP: A language for programmable devices. *Ph.D. thesis, Dept. of Computer Science, Princeton University*, In Preparation.

[16] K. Li, H. Chen, Y. Chen, D. W. Clark, P. Cook, S. Damianakis, G. Essl, A. Finkelstein, T. Funkhouser, A. Klein, Z. Liu, E. Praun, R. Samanta, B. Shedd, J. P. Singh, G. Tzanetakis, and J. Zheng. Early experiences and challenges in building and using a scalable display wall system. *IEEE Computer Graphics and Applications*, 20(4):671–680, 2000.

[17] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *International Symposium on Computer Architecture*, 1997.

[18] T. A. Proebsting and S. A. Watterson. Filter fusion. In *Principles of Programming Languages*, 1996.

[19] J. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[20] B. SGS-Thomson Microelectronics. *OCCAM 2.1 Reference Manual.* 1995.

APPENDIX

A. C EXAMPLE

We present a code fragment that illustrates the task of programming with event-driven state machines in C. It uses a typical event-driven state-machines programming interface in C which includes the following functions:

setHandler(sm,s,e,f) Sets function *f* to be the handler for event *e* when the state machine *sm* is in state *s*.

setState(sm,s) Moves state machine *sm* to state *s*.

isState(sm,s) Checks if state machine *sm* is in state *s*.

deliverEvent(sm,e) Deliver event *e* to state machine *sm*.

The C code fragment presented in this section implements the following functionality. The state machine SM1 is responsible for handling requests from applications. On receiving a request to send data, it DMAs the data from the users memory onto the network card and hands it over to state machine SM2 (which is responsible for sending it over the network). Then, SM1 waits for the next request. While processing the send request, SM1 might need to block if the DMA is busy or if SM2 is not ready to accept the request.

During initialization, the handlers for different events are set up and the state machine is initially in state WaitReq. When a request from the user arrivers (event UserReq). the corresponding handler **handleReq** is triggered. Since the user specifies virtual address of the data, it is first translated into physical address by calling function `translateAddr` that performs a table lookup. Then, it checks if the DMA is available. If it is, it calls **fetchData** directly. Otherwise, it sets the state of the state machine SM1 to **WaitForDMA** and blocks. In this case, **fetchData** will be called when the DMA becomes available (because it is the handler)

When **fetchData** is invoked, it DMAs the data from the applications memory onto the network card by calling dmaData(). Then, it checks to see if the state machine SM2 is ready to accept data. If it is, it calls syncSM2 directly. Otherwise, it sets the state of the state machine SM1 to **WaitSM2** and blocks. In this case, **syncSM2** will be called when SM2 is finally ready to accept data.

When **syncSM2** is invoked, the request is handed over to SM2 by updating global variable **reqSM2**. Then an event SM1Ready is delivered to SM2. This will eventually cause the corresponding handler in SM2 to be invoked. Finally, it sets the state of SM1 to **waitReq** and waits for the next request.

```
enum StateMachineT { SM1, SM2, ...};
enum StateT { WaitReq, WaitDMA, WaitSM2, WaitSM1, ...};
enum EventT { UserReq, DMAFree, SM2Ready, SM1Ready, ...};
enum UserReqT { SendReq, UpdateReq, ...};

ReqSM1 *reqSM1;
ReqSM2 *reqSM2;
int pAddr, *sendData;

main() {
  ...
  // Initialize state machine SM1
  setHandler( SM1, WaitReq, UserReq, handleReq);
  setHandler( SM1, WaitDMA, DMAFree, fetchData);
  setHandler( SM1, WaitSM2, SM2Ready, syncSM2);
  setState( SM1, WaitReq);    // Initial State
  ...
}
```

```
void handleReq() {     // Req has arrived
  switch ( reqSM1->type) {
  case SendReq:
    pAddr = translateAddr( reqSM1->vAddr);
    if ( dmaIsFree())  fetchData();
    else               setState( SM1, WaitForDMA);
    return;  // Block State machine
  case UpdateReq:
    updateAddrTrans( reqSM1->vAddr, reqSM1->pAddr);
  ...
}

void fetchData() {     // DMA is available
  sendData = dmaData( pAddr, reqSM1->size);
  if ( isState(SM2,WaitSM1))  syncSM2();
  else                        setState( SM1, WaitSM2);
}

void syncSM2() {        // SM2 is ready for next request
  reqSM2->data = sendData;
  reqSM2->dest = reqSM1->dest;
  deliverEvent( SM2, SM1Ready);
  setState( SM1, WaitReq);    // Wait for next request
}
```

B. ESP EXAMPLE

This section presents ESP code fragment that is used to illustrate different aspects of the ESP language throughout this paper. It implements some of the same functionality described in Appendix A.

```
type dataT = array of int
type sendT = record of { dest: int, vAddr: int, size: int}
type updateT = record of { vAddr: int, pAddr: int}
type userT = union of { send: sendT, update: updateT, ...}

channel ptReqC: record of { ret: int, vAddr: int}
channel ptReplyC: record of { ret: int, pAddr: int}
channel dmaReqC: record of { ret: int, pAddr: int, size: int}
channel dmaDataC: record of { ret: int, data: dataT}
channel SM2C: record of { dest: int, data: dataT}
channel userReqC: userT   // External (aka C) writer

process pageTable {  // virtual to physical address mapping
  $table: #array of int = #{ TABLE_SIZE -> 0, ...};
  while ( true) {
    alt {
      case( in( ptReqC, { $ret, $vAddr})) {
         // Request to lookup a mapping
        out( ptReplyC, { ret, table[vAddr]});
      }
      case( in( userReqC, { update |> { $vAddr, $pAddr}})) {
         // Request to update a mapping
        table[vAddr] = pAddr;
      }
    }
  }
}

process SM1 {
  while ( true) {
    in( userReqC, { send |> { $dest, $vAddr, $size}});
    out( ptReqC, { @, vAddr});
    in( ptReplyC, { @, $pAddr});
    out( dmaReqC, { @, pAddr, size});
    in( dmaDataC, { @, $sendData});
    out( SM2C, { dest, sendData});
    unlink( sendData);
  }
}
```