ESP: A Language for Programmable Devices

Sanjeev Kumar

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

June 2002

Abstract

This thesis 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 programming 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 model that can be used by a model-checking 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 earlier implementation. We also found that model-checking verifiers like Spin can be used to effectively debug the firmware. Our measurements show that the performance impact on applications of using ESP is small.

Acknowledgments

I would like to thank all the members of my committee: Kai, Ed, Andrew, Randy, and Larry. I have been extremely fortunate to have Kai as my advisor. He gave me a lot of freedom to choose my research agenda. At the same time, he was there to guide me whenever my research stalled. Even when he was busy or away, he made sure that he was available whenever I needed his advice. I would like to thank the other faculty members in the department who keep their doors open to graduate students. JP, Andrew, Larry, Doug, Ed, and Randy have answered numerous technical questions and provided invaluable advice over the years. I would also like to thank our efficient administrative and technical staff. Melissa, in particular, has been very patient, and has ensured that I kept up with the paperwork.

My stay at Princeton has been enormously enriched by the friends I have made here. I would like to thank Dirk, Steve, Rudro, and Patrick. We arrived at Princeton at the same time, and survived the stressful times together. I will always have fond memories of our “Must See TV” and “Survivor” gatherings followed by “Let’s go to Philly” sessions. Other graduate students in the department including Yuanyuan, Tammo, George, Angelos, Stef, and Liviu have engaged me in many lively discussions—both technical and otherwise. I would also like to thank all my undergraduate buddies, too numerous to list here, who have become a large extended family to me here in the US.

I would like to thank my family for their love, encouragement, and support throughout my education. My dad, Naresh, has always been a perfect role model for me. My mother, Shikha, has always emphasized the importance of a good education. My grandparents provided additional warmth and affection while I was growing up. My brother, Praveen, has been a friend and a confidant. I have been lucky to have him nearby since the day I left home for college. My sister, Pooja, has been the baby of the family. Finally, I would like to thank my wife, Sushma, who has been my biggest gain while at Princeton.

This work was supported in part by the National Science Foundation (CDA-9624099, EIA-9975011, ANI-9906704, EIA-9975011), the Department of Energy (DE-FC02-99ER25387), California Institute of Technology (PC-159775, PC-228905), Sandia National Lab (AO-5098.A06), Lawrence Livermore Laboratory (B347877), Intel Research Council, and the Intel Technology 2000 equipment grant.

In memory of my mother,

Geeta Singh

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis shows that a domain-specific language can greatly ease the task of programming devices like network interface cards and hard disks. As devices are getting faster, the overhead of using the main processor to do low-level device management becomes significant. The overhead stems from the cost of interrupt processing and of having to cross multiple buses to reach the device. To address this, some devices are equipped with their own programmable processor and memory. This allows some of the functionality that used to run on the main processor to be offloaded on to the processor on the device.

The task of writing firmware for these programmable devices is challenging for several reasons. First, the programmable devices tend to have limited processing power and memory resources but are required to deliver high performance. Second, they have to constantly process external events and keep track of the progress of multiple events at the same time. Finally, the firmware is trusted by the operating system and can directly write to main memory. A bug in the firmware can corrupt the operating system and crash the entire machine. Therefore, the firmware has to be reliable.

These devices are usually programmed using event-driven state machines in C. Concurrency is an effective way of structuring firmware for programmable devices. And the low overhead of event-driven state machines often makes them the only choice for expressing concurrency in firmware. The ability of C to handle low-level details makes it a popular choice for writing system software.

Using event-driven state machines in C to implement firmware makes an already difficult task of writing reliable, concurrent programs even more challenging. This is because their low overhead is achieved by supporting only the bare minimum functionality needed to write these programs. Although good performance can be achieved, the resulting programs are hard to maintain and hard to debug.

For example, the Virtual Memory Mapped Communication (VMMC) firmware [41] for Myrinet [21] network interface was implemented using event-driven state machines in C. The VMMC architecture delivers high performance communication to applications running at user-level. It allows them to bypass the operating system by using programmable network interface cards. Our experience with the VMMC firmware 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 machine to crash occasionally.

This thesis presents the design and implementation of the Event-driven State-machines Programming (ESP)—a language for programmable devices. Unlike C which forces a tradeoff that requires giving up ease of programming and reliability to achieve high performance, ESP is designed to provide all of these three properties simultaneously.

As a case study, we reimplemented the VMMC firmware using ESP. We compared the new implementation with the earlier implementation in C to evaluate the ESP language. 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 [55] can be used to effectively debug the firmware. Finally, our measurements show that the performance impact on applications of using ESP is small. We have published some of these results [64, 63].

The rest of this chapter is organized as follows. Section 1.1 provides a brief description of programmable devices. Section 1.2 explains the choices available to program these devices. Section 1.3 describes the problems with the traditional approaches to programming these devices in C. Section 1.4 presents the goals, approach and overview of ESP. Section 1.5 describes the related work. Section 1.6 presents the outline for the rest of this thesis.

1.1 Programmable devices

Traditionally, devices implement simple functionality that is usually implemented in hardware. All the complexity is implemented in device drivers running on the main processor. However, as devices get faster, it is increasingly harder for software running on the main CPU to keep up with the devices. This is because the main CPU has to go across the memory and I/O buses to reach the device and incurs several hundreds of cycles for each access. In these situations, better performance can be achieved by implementing some of the functionality on the device instead of on the main CPU [15, 97, 41, 96, 79, 95, 100, 2, 99].

To implement these increasingly sophisticated functionality on devices, these devices are often equipped with a programmable processor and memory (Figure 1.1). Since the processor resides directly on the card, it incurs a much smaller overhead to access the device's resources like DMA engines and device registers.

The code running on the device has to be fast. The processing power and memory on the device tends to be at least an order of magnitude less than the main CPU and main memory. Migrating code from the main CPU to the device involves a tradeoff between

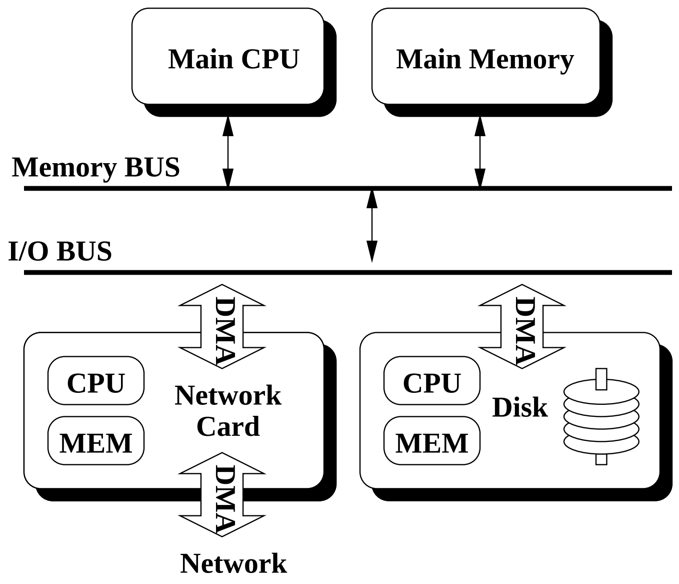

Figure 1.1: A machine with programmable devices

running the code on a faster processor that incurs higher overhead to access the device, and running it on a slower processor that has faster access to the device. The slower the code runs on the device, the smaller the benefit of migrating code to devices.

Devices equipped with general-purpose processors differ a great deal in the amount of processing power and memory they possess. Some of them have a very small processing core that can be used to implement only very simple functionality and that are often programmed in assembly (microengines on the Intel IXP [58]). Other devices [21, 58, 36, 77, 86, 3] have general-purpose processors powerful enough to implement complex tasks but require the code running on them to be fast. This thesis addresses the task of writing firmware for the latter class of devices.

1.2 Programming 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. For instance, network interface cards are required to continuously respond to a variety of events such as requests from the main CPU, messages arriving on the network, and timer interrupt. For better performance, processing of multiple events is overlapped—a DMA might be transferring data to the main memory in response to one event at the same time as a message is being sent out to the network in response to a different event. In these situations, concurrency is used not to exploit parallelism but as an effective way to structure a program that runs on a single processor.

Concurrent programs can be written using a variety of constructs like threads or event-driven state machines. They differ in the amount of functionality provided and the performance overhead involved.

Threads and Coroutines. A concurrent program is composed of a set of threads. Each thread represents a sequential flow of control in the program. The difference between threads and coroutines is that coroutines cannot be preempted. Context switch in coroutines occurs only at well-defined synchronization points in the program. However, in both threads as well as coroutines, each thread has its own stack and can context switch to another thread even when it is in a nested function call. This frees the programmer from having to structure the program without worrying about blocking. However, this convenience comes at a cost. A thread context switch is slower because it involves saving and restoring all the registers including the stack pointer and the program counter. Also, it requires more memory because a separate stack has to be allocated for every thread.

Event-driven state machines. A concurrent program is composed of a set of state machines. Each state machine represents a sequential flow of control in the program. The state machines communicate with each other by sending events. The basic difference between threads and event-driven state machines is that all the state machines in the program share a single stack. The context switch is fast because the only context that needs to be saved and restored on a context switch is the program counter. In addition, the memory requirements are lower because only one stack is needed for the entire program.

Writing event-driven state-machines programs is more difficult. Having to share a stack between the various state machines imposes restrictions on when a state machine can block and transfer control to a different state machine. Since top-level functions are the only points in the program where the stack does not store any useful data (and can therefore be used by a different state machine without having to save any data), state machines are allowed to block only in top-level functions. The programmer has the burden of structuring the program so that it does not block in a nested function call.

In languages like C that do not provide any support for event-driven state machines, the need to share a single stack places additional burden on the programmer. They require state machines to be specified explicitly. The sequential flow of control of a state machine has to be broken up into several functions (handlers). The flow of control between these functions is specified using function pointers (a more detailed description is presented in Section 1.3). This makes event-driven state-machines programs difficult to write, understand and debug.

Since high performance is essential in device firmware, the low overhead of event-driven state machines makes them a compelling choice.

Function	Description
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*

Table 1.1: A typical event-driven state-machines library interface.

1.3 Programming Programmable Devices in C

Libraries to support event-driven state machines in C 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. A typical library interface is presented in Table 1.1.

A program consists of multiple state-machines. Each state machine is specified using a set of handlers.[1] For each state in a state machine, a handler is specified (using `setHandler`[2]) for every event that is expected while in that state. Later, when an event occurs, the corresponding handler is invoked by the library. The handler processes the event, transitions to a different state (using `setState`) and blocks by returning from the handler. The only way a state machine can block is by returning from the handler. A state machine communicates with other state machines by generating events (using `deliverEvent`). All the state machines share a single stack.

In the rest of this section, we use an example to illustrate the problems with programming using event-driven state machines in C.

```
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
  ...
}
```

Figure 1.2: C example (part I): Initialization

```
void handleReq() {                              // Request has arrived
  switch ( reqSM1->type) {
  case SendReq:
    pAddr = translateAddr( reqSM1->vAddr);
    if ( dmaIsFree())
      fetchData();
    else
      setState( SM1, WaitDMA);
    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
}
```

Figure 1.3: C example (part II): Event handlers

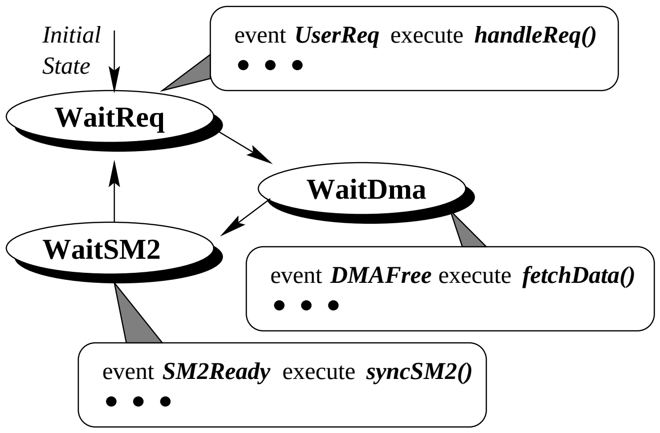
Initial State
event UserReq execute handleReq()
• • •
WaitReq
WaitDma
WaitSM2
event DMAFree execute fetchData()
• • •
event SM2Ready execute syncSM2()
• • •

Figure 1.4: C example which corresponds to code shown in Figures 1.2 & 1.3.

1.3.1 A Programming Example

The C code fragment (Figures 1.2 & 1.3 and illustrated in Figure 1.4) uses the event-driven state-machines programming interface presented in Table 1.1. It 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 user's 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 arrives (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.

[1]A handler is a C function that takes no arguments and returns void.
[2]This is used only during initialization to set up the state machines.

Otherwise, it sets the state of the state machine `SM1` to `WaitDMA` 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 application's 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.

1.3.2 Problems

There are several problems with this approach. First, the code becomes very difficult for a programmer to understand because the code gets fragmented across several handlers. In addition, the use of function pointers (handlers) to specify the state machine makes it difficult for the C compiler to determine the control flow in the state machines and prevents it from effectively optimizing the code. This forces the programmer to hand-optimize the code for better performance.

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. Data is passed between handlers through global variables (e.g. `pAddr`, `sendData`). In addition, state machines communicate with each other using global variables (e.g. `reqSM2`). It is very hard to get the right synchronization to keep from clobbering data.

Third, dynamically allocated data structures have to be managed explicitly. In a concurrent setting, this is hard to implement correctly when several state machines use the data structure 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. 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 collocated 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 state machines). 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.

In summary, while good performance can be achieved when programming in C, these programs are hard to write, maintain and debug.

1.3.3 Case Study: VMMC Firmware

The Virtual Memory-Mapped Communication (VMMC) firmware [41] was implemented using event-driven state machines in C. We use it as a case study throughout this thesis.

The Virtual Memory-Mapped Communication (VMMC) architecture [20] 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 [41] uses the Myrinet [21] Network Interface Cards. Myrinet is a packet-switched gigabit network. The Myrinet network card is connected to the network through two unidirectional links of 160 Mbytes/s peak bandwidth each. The actual node-to-network bandwidth is usually constrained by the PCI bus (133 Mbytes/s) on which the network card sits. The network card has a programmable 33-MHz LANai4.1 processor, 1 Mbyte SRAM memory and three DMA engines to transfer data—one to transfer data to and from the host memory; one to send data out onto the network;

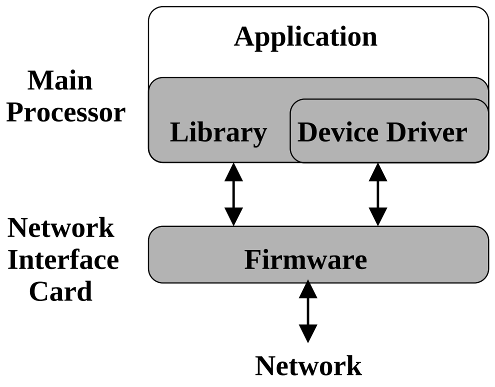

Figure 1.5: VMMC Software Architecture. The shaded regions are the VMMC components.

one to receive data from the network. The card has a number of control registers including a status register that checks for data arrival, watchdog timers and DMA status.

The VMMC software (Figure 1.5) has three 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 software development involved several man years. Most of the bugs encountered were located in the firmware. Consequently, we wanted to reimplement the firmware using the ESP language.

The VMMC software has been extensively used by a number of research projects. Several standard high-level communication libraries including Remote Procedure Call (RPC) [17], Shared Virtual Memory (SVM) [16], Sockets [37] and NX Message Passing [4] have been implemented on top of the low-level API provided by VMMC. Several distributed applications [59, 66] that run on a cluster have also used VMMC as the communication mechanism.

Significant effort [42, 41, 18, 30] has been spent on implementing, maintaining, performance tuning, and extending the functionality of VMMC. Our experience with programming VMMC firmware using event-driven state-machines in C makes it an ideal candidate for a case study in this thesis.

1.4 Event-driven State-machines Programming (ESP)

ESP is a domain-specific language designed to support event-driven state-machines programming on programmable devices. The rest of this section is organized as follows. Section 1.4.1 describes the design goals of ESP. Section 1.4.2 outlines the approach taken by ESP to meet the design goals. Section 1.4.3 presents an overview of the ESP compiler. Section 1.4.4 presents a summary of the evaluation of ESP using the VMMC firmware as a case study.

1.4.1 Goals

The design of ESP was driven by the following goals:

Ease of programming. The language should simplify event-driven state-machines programming by addressing the problems described in Section 1.3.2. It should allow concurrency to be expressed in a concise modular fashion. It should also provide support for dispatch, dynamic memory management, and flexible interface to C.

Extensive testing. Concurrent programs often suffer from hard-to-find race conditions and deadlock. However, the device firmware needs to be reliable because it is trusted by the operating system. A bug in the device firmware can crash the entire machine. ESP should support the use of model-checking verifiers so that the programs can be tested extensively.

Low performance overhead. Good performance is crucial for the firmware running on the programmable devices. ESP should minimize the overhead of using the language

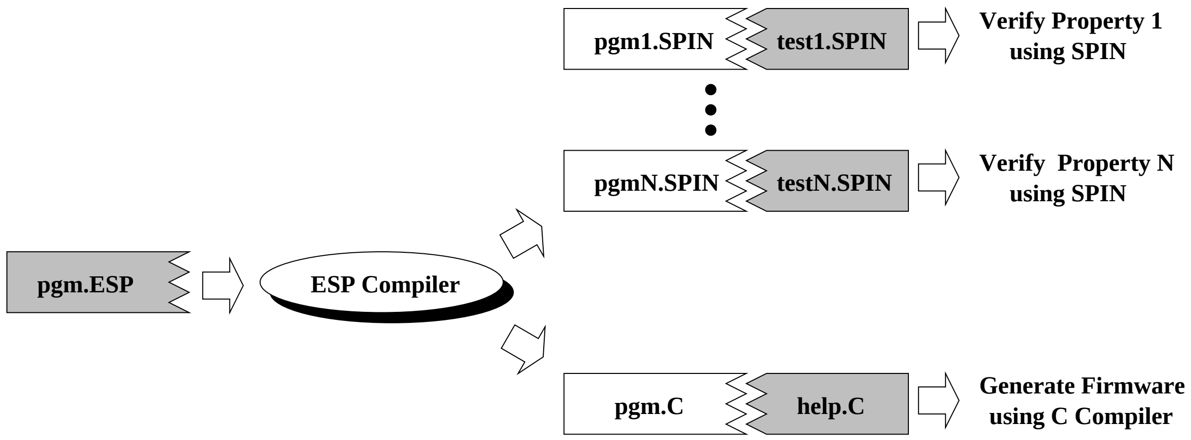
pgm1.SPIN
test1.SPIN
Verify Property 1 using SPIN
pgmN.SPIN
testN.SPIN
Verify Property N using SPIN
pgm.ESP
ESP Compiler
pgm.C
help.C
Generate Firmware using C Compiler

Figure 1.6: ESP Approach. The ESP compiler takes a program written in ESP (`pgm.ESP`) and generates two types of files: models (`pgm[1-N].SPIN`) that can be used by the model checker to extensively test the program; and optimized C code (`pgm.C`) that can be used to generate the firmware. Shaded regions represent code that has to be provided by the programmer.

features provided. It should permit aggressive optimizations to compile concurrent programs to run efficiently on a single processor.

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

1.4.2 Approach

To meet the design goals (Section 1.4.1), ESP takes the following approach (Figure 1.6). First, ESP provides a number of features that simplify the task of programming the firmware (`pgm.ESP`). Then, the ESP compiler generates models (`pgm[1-N].SPIN`) that can be used by a model checker to debug and extensively test the firmware. Finally, the ESP compiler generates optimized C code (`pgm.C`) that can be compiled into firmware that runs on the device.

Other domain-specific languages [14, 28, 12] that use model checkers for testing have focused on expressing the control portion of the program. They leave data handling to be performed externally using C. In contrast, ESP is designed not only to express the control structure in a modular fashion but also to simplify data handling. This helps ESP in meeting all three of its goals. First, the programmer is not burdened by having to separate the data manipulation from the control portion of the program. Second, when the resource constraints prevent the model checker from exhaustively checking the entire program, the ESP compiler can be used to extract abstract (less detailed) models. More detailed models can be used to test local properties of small subsystems while the less detailed models can be used to test global properties of the entire system. Finally, having more of the program expressed in ESP allows the ESP compiler to perform optimizations more effectively.

The ESP language (Section 2.3) uses processes to implicitly express state machines. These processes use channels to communicate with each other. In addition, ESP has a number of language features that simplify the task of writing device firmware.

The ESP compiler can automatically extract models that can be used by a model checker to extensively test the program (Section 3.3). Currently, ESP uses the Spin model checker [55]. Spin is a flexible and powerful verification system designed to verify correctness of software systems (Section 3.3.1). It systematically explores the state space of the system and checks for violations of the specified property. The Spin models (`pgm[1-N].SPIN`) generated by the ESP compiler can be used together with programmer-supplied Spin code (`test[1-N].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.

The ESP compiler uses C as the back-end language and generates optimized C code (Section 4.3). The generated C code (`pgm.C`) can then be compiled together with the C code provided by the programmer (`help.C`) to generate the executable. The programmer-supplied C code implements simple system-specific functionality like accessing device registers to check for network message arrivals. Since device manufacturers usually provide a C compiler to write firmware for their devices, using C as the back-end language makes ESP programs to be portable.

1.4.3 Compiler Architecture

Figure 1.7 presents the overall architecture of the ESP compiler. The compiler was implemented using Standard ML of New Jersey (SML/NJ) and required around 7000 lines of code. The compiler has three main components: a front end (1800 lines of code), a model extractor (1200 lines of code), and an optimizing code generator (4000 lines of code). The front end is responsible for scanning, parsing and type checking and generates a typed abstract syntax tree (details in Section 4.3.1). The model extractor uses the typed abstract syntax tree to generate models (details in Section 3.3.2). The optimizing code generator translates the typed abstract syntax tree more suitable for optimizations, optimizes it, and generates C code (details in Section 4.3).

1.4.4 Case Study: VMMC Firmware

VMMC Firmware (Section 1.3.3) is used as a case study to evaluate ESP. The new implementation that uses ESP is compared with the earlier implementation (which used event-driven state machines in C) to show that ESP meets its three design goals (Section 1.4.1). Detailed evaluation is presented in Sections 2.4, 3.4 and 4.4. A brief summary is presented here.

Ease of Programming. ESP allows the VMMC firmware to be expressed as a modular concurrent program. Concurrent programs are expressed concisely using processes and channels. In addition, pattern matching on channels allows an object to be

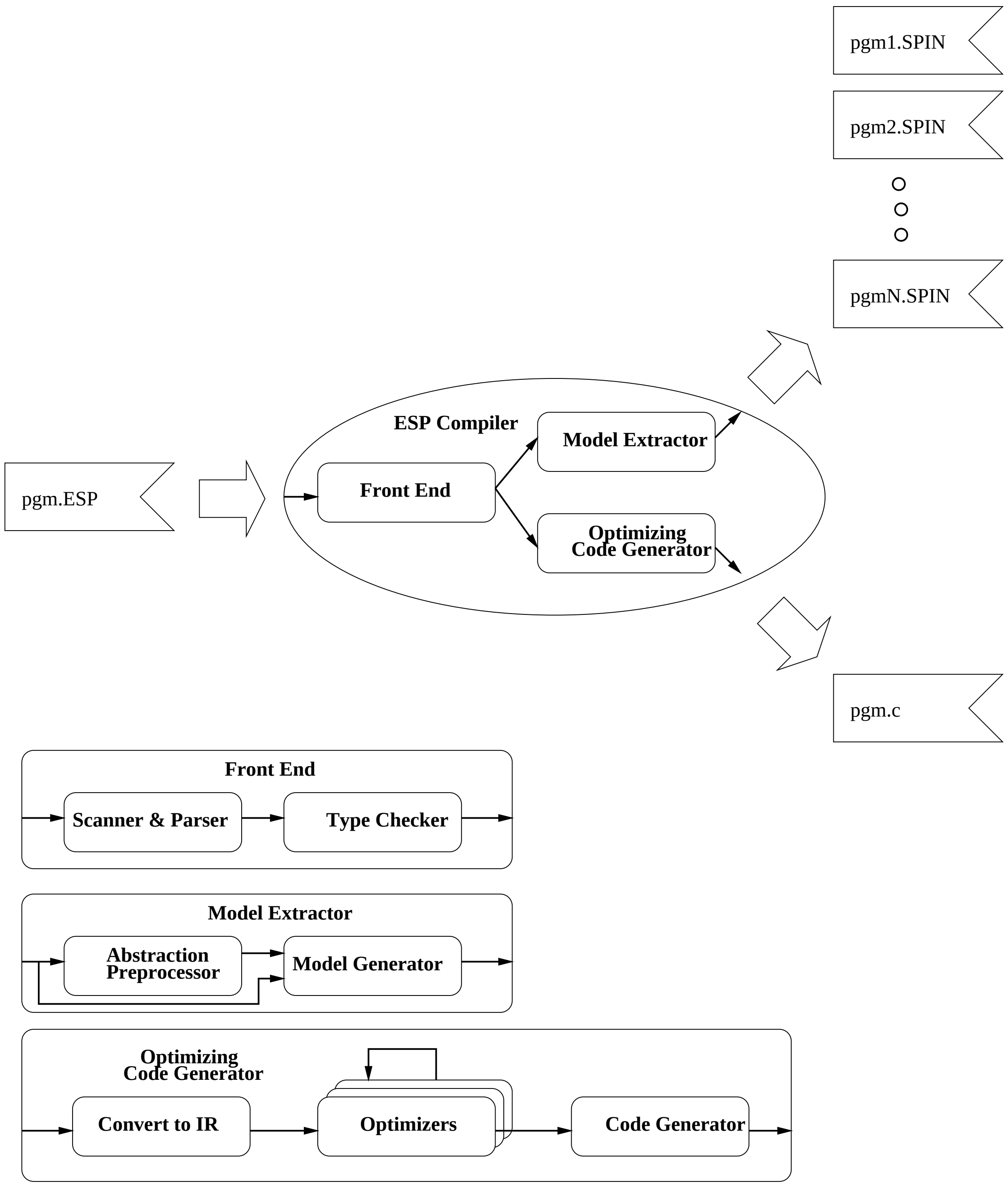

Figure 1.7: ESP Compiler Architecture

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.

The new implementation of VMMC firmware using ESP requires significantly fewer lines of code than the earlier C implementation. The new implementation has 500 lines of ESP code together with around 3000 lines of C code. The C code performs only simple operations like packet marshalling and handling device registers and all the complexity is localized to the ESP code. This is a significant improvement over the earlier implementation where the complex interactions were scattered over the entire C code (15600 lines).

Extensive Testing. The ESP compiler generates Spin models (Section 1.4.2) that were used to develop and extensively test the VMMC firmware using the Spin model checker. Once the properties to be checked by the model checker are specified, the model checker checks them automatically. As the firmware evolves, the properties can be rechecked with minimal programmer effort.

Since developing code on the network card is often slow and painstaking, parts of the firmware were developed and debugged entirely using Spin before being ported to the network card. For instance, the retransmission protocol was implemented using this approach. This greatly speeds up program development. The implementation of the protocol using ESP took two days while the earlier implementation took over ten days.

Spin was also used to exhaustively check the memory safety of the firmware. ESP provides an explicit dynamic memory management interface that is efficient but unsafe. By using Spin to verify safety, ESP provides the benefits of safety without paying the runtime cost of implementing safety through garbage collection.

Spin was able to identify several bugs that would cause the firmware to deadlock. These bugs often triggered in certain rare circumstances and would have been fairly difficult to find using conventional testing methods.

Low Performance overhead. The ESP compiler performs aggressive optimizations and was used to generate efficient VMMC firmware. The performance of the new implementation using ESP was compared with the earlier implementation in C using both microbenchmarks as well as applications.

Microbenchmark measurements show that the performance difference between the firmware implementation using ESP and the earlier implementation using C is usually small. In some cases, the difference is significant. For instance, the ESP implementation involves twice the latency of the C implementation when sending 4 byte messages. However, in this case, the entire difference is the result of the fast paths in the C implementation that is not currently supported by ESP. To obtain a fairer comparison, a version of the C implementation that does not include the fast paths is also used. The measurements show that the ESP implementation performs 0–35 % worse in the latency microbenchmark and 12–25 % worse in the bandwidth microbenchmark.

The performance impact of using ESP on applications is small. We measure the performance of SPLASH2 applications [103] using the two firmware implementations. Our measurements show that the applications run 3.5 % slower on average (10 % in the worst case) when using the ESP version relative to the C version. They also show that the fast paths in the C implementation have little impact on the applications' performance.

1.5 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 designed for programming devices. However, a number of research projects have addressed some of the same problems as ESP albeit in a different context. This section presents a brief survey of the related work. They are discussed in more detail in Sections 2.2, 3.2, and 4.2.

A number of other domain-specific languages have taken an approach similar to ESP's—the compiler generates both models that can be used for debugging using a model checker as well as executable code. However, they have been designed to support event-driven state-machines programming for different domains. Esterel [14] is designed to model the control of reactive systems. Teapot [28] is designed for writing coherence protocols. Promela++ [12] is designed to implement layered network protocols. One of the main differences between ESP and these languages is that these languages focus on the control portion of the program; the data structures have to be handled externally using C. In contrast, ESP is designed to simplify both the control portion as well as the data handling. The Thompson's format [57] is a simple extension to ANSI C to support event-driven state machines. It also allows models to be extracted. However, it provides only minimal support to aid programming.

A vast amount of research has focused on the theory and practice of concurrency. Languages like CSP [52] and Squeak [24] have been designed to explore the use of concurrency to structure programs. However, these languages do not address debugging or efficient code generation. A number of programming languages like Java [8], CML [88, 87], SR [6], Newsqueak [81] provide support for concurrency. However, these languages are designed to be very expressive and incur higher runtime overheads.

1.6 Thesis Outline

The rest of this thesis is organized around the goals ESP set out to achieve. Chapter 2 presents the design of the ESP language and shows that it simplifies the task of writing event-driven state-machines programs for devices. Chapter 3 describes how ESP programs can be tested extensively using a model checker. Chapter 4 describes techniques used to generate an efficient executable from ESP programs and presents performance measurements that show that the impact of using ESP on applications is fairly small. Chapter 5 presents our conclusions.

Appendix A presents the ESP language reference. Finally, Appendix B describes the structure of the VMMC firmware implemented using ESP.

Chapter 2

The ESP Language

The firmware for programmable devices is usually written using event-driven state machines (Section 1.1 and 1.2). This is because event-driven state machines support low-overhead concurrency. The concurrency allows the firmware to keep track of the various events being simultaneously processed by the device. The low overhead of using concurrency primitives allows the firmware running on relatively slow processors to keep up with the high performance devices.

General-purpose languages like C include powerful language features like first-class functions and gotos that can be used to program with event-driven state machines. However, these languages were not designed to simplify event-driven state-machines programming. As a result, event-driven state-machines programs written in C are difficult to implement, debug and maintain. The drawbacks of using C are discussed in detail in Section 1.3.

This chapter presents the design of the ESP language. ESP is a domain-specific language for writing firmware for programmable devices using event-driven state machines. The VMMC firmware is used as a case study to demonstrate that ESP greatly simplifies the task of programming these devices.

The rest of this chapter is organized as follows. Section 2.1 discusses the background. Section 2.2 describes the related work. Section 2.3 presents the design of the ESP language. Section 2.4 presents our experience with using ESP to write VMMC firmware. Section 2.5 suggests directions for future research. Finally, Section 2.6 presents a summary.

2.1 Background

There are two approaches to writing event-driven state-machines programs:

Explicit. In explicit event-driven state-machines programs, a state machine is specified by explicitly specifying its various components: the states that it can be blocked in; the events that it responds to; handlers that specify the code that has to be invoked in response to the events and that transition the state machine to its new state before blocking. Languages, like C, that are not designed for event-driven state-machines programming can support it only using an explicit interface. A typical interface was presented in Section 1.3.

Explicit event-driven state-machines programming is a good match in domains like memory coherence protocols and hardware controllers where the programming task is specified as event-driven state machines. However, the explicit approach results in fragmented code (Section 1.3.2) that makes it less readable. Language features like continuations [28] and buffered channels [12] can be used to alleviate the fragmentation problem.

Implicit. In implicit event-driven state-machines programs, state machines are specified implicitly using processes and channels. Each process implicitly encodes a state machine and represents a sequential flow of control in a concurrent program. A process communicates with other processes by exchanging messages on channels.

Since operations on channels can block, a process has to sometimes suspend waiting for a channel operation to complete. In this setup, each location in a process where it can block represents a separate state of that state machine. Events on channels like message arrival cause a blocked process to resume execution until the next location in the process where it blocks (state transition). Since the different state machines in a event-driven state-machines program have to share a single stack (Section 1.2), the processes have to obey a restriction—a process can block only at the top-level and not in a nested function call. This ensures that a state machine does not have any useful data on the stack while it is blocked and allows the same stack to be reused for the next state machine that is scheduled to run. An example of implicit event-driven state-machines program is presented in Section 2.3.1.

The implicit approach simplifies the programmers task by making event-driven state-machines programming similar to message-passing style concurrent programs. It provides a high-level abstraction and avoids the code fragmentation problem.

2.2 Related Work

This section describes a number of research projects that are related to the ESP language and influenced its design.

Code Generation + Verification. A number of languages [14, 28, 12] have been designed to support event-driven state-machines programming in other domains. They have taken a similar approach of generating efficient executables as well as models that can be used by a model checker. However, they differ from ESP significantly. One of the main differences is that all these languages have been designed to address only the control portion of the program. The complex data structures have to be manipulated externally using the

C language. This design choice is motivated by the fact that the control portion of the program is sufficient to check for some properties like absence of deadlocks and livelocks.

Esterel [14] was designed to model the control of reactive systems. 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. It was used recently to efficiently implement a subset of TCP protocol [25]. 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 language provides some powerful constructs like parallelism at every level. However, to satisfy the synchronous hypothesis, it forces some constraints on valid programs. For instance, each 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. Third, the language is more naturally suited to implementing synchronous hardware than software. It has a notion of a global clock that ticks periodically. Processes communicate with each other emitting broadcast signals. In each clock cycle, there should be a scheduling scenario that allows all the writers to write to a signal before any readers of that signal can read it. 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 [28] is a language for writing coherence protocols that can generate efficient protocols as well as verify correctness. It uses a single state machine (and not a group of communicating state machines) to keep track of the state of a coherence unit (a cache line or a page). The state machine is specified explicitly using a set of handlers similar to the C interface described in Table 1.1. 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 from some of the problems described in Section 1.3.2 when used to implement device firmware. It makes it difficult to write modular code because each state (which is essentially the global state of the program) has to specify a response to all the events that can occur in that state. Teapot also does not provide any support for complex datatypes and dynamic memory management.

Specification languages like Promela [56] (which is used by Spin), Murphi [40], and IOA [47] are used to specify event-driven state machines. These languages are designed to precisely specify and to formally reason about programs. However, they are not designed to be effective programming languages and lack some standard language features. For instance, Promela does not support pointers and dynamic memory management that are necessary for an efficient implementation.

Promela++ [12] is a language designed to implement layered network protocols. It is a nonstrict extension of the Promela language that supports a restrictive form of pointers. The network protocol is implemented as a sequence of layers. The adjacent layers communicate using FIFO queues. Each layer specifies handlers to process the message arriving on its queues. The queues have unbounded buffering. This allows the output operations on the queues to be nonblocking. 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.

The Thompson's format [57] is a simple extension to ANSI C to support event-driven state-machines programming. The program is still written in C using `goto` and `label` statements. However, the extensions provided (identified by the @ prefix) simplify the task of specifying dispatch tables and states. A preprocessor replaces these extensions with the corresponding C code. A model checker was used to check software written in this format [57]. Although this approach simplifies event-driven state-machines programming, it does not address a number of issues including dynamic memory management and communication between the state machines.

Concurrency Theory. Concurrent programs are more difficult to understand than sequential ones. A number of languages [52, 24] have been designed to explore the use of concurrency to structure programs. Several process algebras [72, 53, 73] have been proposed to better understand the nature of concurrency and to formally reason about concurrent programs.

CSP [52] proposes the use of communicating sequential processes (CSP) as a fundamental program structuring method. It shows how CSP programs can be used to implement a variety of constructs including subroutines, datatypes, bounded buffers and semaphores. A CSP program consists of a set of processes that can communicate with each other only by sending synchronous messages (rendezvous communication). Each of the processes is completely sequential. However, CSP does not address some practical issues including complex data types, dynamic memory management, external interface, debugging, and efficient compilation.

Squeak [24] is a concurrent language for implementing graphical user interfaces. Its semantics are formally specified. A Squeak program consists of a fixed number of processes communicating through synchronous channels. The channels are known statically. This allows the programs to be compiled efficiently. However, Squeak does not address a number of features including complex data types, dynamic memory management and debugging.

A number of process algebras or calculi—like CCS [72], CSP [53, 89][1], and π-calculus [73]—identify a small set of primitive operators that are sufficient to express concurrent programs and use algebraic laws to formally reason about them. They focus on concurrent programs with processes that communicate by sending message to each other rather than through shared variables. These algebras serve as foundation for programming languages.

Concurrent Languages. A number of programming languages [8, 87, 6, 81] provide support for concurrency. These languages are very expressive and use threads instead of event-driven state machines to express concurrency. They allow dynamic process and channel creations. They support both message-passing style communication as well as shared-memory communication. However, due to their expressiveness, the runtime overhead incurred is prohibitively high to allow their use to program devices.

Java [8], like most general-purpose programming languages, provides user-level threads to express concurrency.

CML [88, 87] is a concurrent extension of ML [74] that supports first-class synchronous operations. These operations are powerful enough to express a wide variety of communication abstraction including buffered channels, multicast channels, Ada-style rendezvous and futures.

SR [6] is a language for parallel and distributed processing. It supports several virtual machines that communicate using message-passing. Each virtual machine uses a separate address space and can run multiple processes that communicate with each other using both shared memory as well as message passing. This allows an SR program to run on a single parallel machine or to be distributed across several machines.

Newsqueak [81] is a successor to Squeak [24] that includes a type system, dynamic process and channel creation and blocking in functions.

[1]The CSP process algebra is distinct from the CSP programming language [52]

OCCAM [91], a descendant of CSP, is designed to implement concurrent programs that run on a parallel machine. The concurrency operator can be used at any level of the program. This encourages the program to express concurrency so that all the nodes of the parallel machine can be kept busy. OCCAM processes communicate with each other only through synchronous channels. To support this, complex data types like arrays and records are copied during assignment, thereby, avoiding pointer aliasing.

Neuron C [44] is an extension of ANSI C that is designed for a distributed environment. A Neuron C program consists of state machines running on different nodes of a distributed system communicating using channels.

Language Extensions. ESP uses C to handle low-level details like accessing special device registers, dealing with volatile memory and marshalling packets that have to be sent out on the network. Other studies [71, 78, 27] have proposed language extensions to simplify low-level handling. These features are orthogonal to the current design of the ESP language and can be incorporated into the language in the future.

Devil [71] is designed to make it easier to write device drivers. It is an interface description language that specifies the communication with devices using device registers and memory. A compiler automatically checks the consistency of the specification and generates efficient low-level code.

The format of packets used by most protocols cannot be described using the type system of most languages including C and ESP. This is because the packet format includes variable length fields that depend on other fields. Therefore marshalling code has to be written that translates data structures into packet formats before they can be sent out on the network. Since doing this manually is tedious and error prone, some researchers [78, 27] have proposed extending the type system to express the packet formats. Universal Stub Compiler(USC) [78] allows a very precise description of layout of the data structures.

However, it does not support variable-length fields. Packet Types [27] provides a more expressive type system that supports layering of protocols by encapsulation, variable-length fields and optional fields. However, it allows only unmarshalling of packets into C data structures. It does not support marshalling of C data structures into packets.

Automatic Memory Management. Automatic memory management in safe programming languages can be provided either through garbage collection [102] or by allowing explicit memory management using regions [93]. Garbage collection often involves runtime overhead (both in terms of processor overheads as well as additional memory requirement) that are unacceptable on programmable devices. Region-based memory management techniques free memory when exiting dynamic contexts like procedures. This makes them unsuitable for a language like ESP that does not have any dynamic context.

2.3 Language Design

The Event-driven State-machines Programming (ESP) language adopts several structures from CSP [52] and has a C-style syntax. ESP supports event-driven state-machines programming. It is designed to not only allow programs that are easy to write and understand but also use model checkers to simplify debugging (Chapter 3) and generate fast executables (Chapter 4). As a result, the design of the language strikes a balance between the following conflicting goals:

- Provide powerful and flexible language features that support compact, readable event-driven state-machines programs.
- Allow the compiler to extract tractable models that can be used by the model checker. The specification language used for specifying models support limited features. For instance, they usually do not support pointers.

```
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 (C) writer

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});
    free( sendData);
  }
}
```

Figure 2.1: ESP Example that implements the state machine presented in Section 1.3.1.

- Allow the compiler to generate fast executables.

This section discusses the design of the ESP language. A complete description of the language is presented in Appendix A.

Figure 2.1 shows the ESP code that implements the example presented in Section 1.3.1. Figure 2.2 presents some additional code that implements a page table. In this section, we will use code fragments from these figures to illustrate the various language features.

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

Figure 2.2: ESP code to implement a page table.

2.3.1 Processes

Concurrency in ESP is expressed using processes and channels. An ESP program consists of a set of processes communicating with each other over channels.

Processes in ESP implicitly encode state machines—each location in the process where it can block implicitly represents a state in the state machine (Section 2.1). For instance:

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

The above process represents a state machine with 3 states. The first state is the initial state at the start of the process. The second state is when it is blocked waiting on the `in` operation on channel `chan1`. The third state is when it is blocked on the `out` operation on channel `chan2`.

The degree of concurrency supported by the ESP language determines the concurrency overhead that is incurred at runtime. To minimize the concurrency overhead, ESP supports concurrency only at the top level—each process is totally sequential. To reduce the overhead even further, all processes in ESP are static. They cannot be dynamically started at run time. This allows the compiler to optimize the programs more effectively.

2.3.2 Channels

Processes communicate with each other over channels. Messages are sent over the channel using the `out` operation and received using the `in` operation. These operations can block. For instance, a process trying to read on a channel will block till another process writes to the channel. The `alt` statement allows a process to wait on `in` and `out` operations on several different channels till one of them becomes ready to complete.

The `alt` statement in ESP traces its origin to guarded commands [39] that allows a nondeterministic choice in sequential programs. An alternation statement includes a list of guarded commands. A guarded command consists of a statement protected by a guard (a boolean expression). Each execution of an alternation statement nondeterministically chooses one of the statements whose guard evaluates to true and executes it. CSP [52] extends the guarded commands to allow an input operation on the channel to be specified in the guard in addition to a boolean expression. Languages like Amber [23] allow both input and output operations in the guard. This introduces complications in a parallel implementation [62]. However, ESP is targeted to run on a uniprocessor.

The presence of nondeterminism raises the issue of fairness. When multiple guards are true in an alternation statement, if the implementation always chooses one particular guarded command, the other guarded commands can be starved out. This is referred to as *unfairness*. Two types of fairness guarantees can be provided: *weak fairness* and *strong* *fairness* [5]. Weak fairness guarantees that a guarded command will be chosen if the guard remains continuously enabled. Strong fairness guarantees that a guarded command will be chosen if the guard is enabled infinitely often. ESP provides strong fairness. It should be noted that fairness does not imply that each of the enabled guarded statements will be chosen with equal probability.

In ESP, communication over channels is synchronous or unbuffered—a process has to be attempting to perform an `out` operation on a channel concurrently with another process attempting to perform an `in` operation on that 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 over asynchronous (or buffered channels). First, they simplify reasoning about message ordering on channels [7]. This is especially true in ESP that does not impose any structure on the processes—any process can communicate with any other process. For instance, if process A sends a message to process B and then a message to process C, synchronous channels guarantee that process B receives its message before process C. Second, they can be implemented more efficiently than buffered channels. When buffering is required, it can be implemented explicitly by the programmer (Appendix A.8). Third, both bounded and unbounded buffering are problematic [88]. With bounded buffering, the programmer has to still handle the eventuality that a send operation has to block. With unbounded buffering, the memory overflow can result. Finally, asynchronous (buffered) channels increase the size of state space that has to be explored during model checking.

ESP Channels are static and are not first-class objects—they can neither be created dynamically nor stored in variables nor sent over other channels. This design allows the compiler to perform optimizations more effectively. For instance, it eliminates unnecessary allocation associated with pattern matching (Section 4.3).

ESP supports pure message passing communication over the channels. Allowing processes to communicate over shared memory (using shared mutable data structures) would require ESP to provide additional mechanism (like locks) to avoid race conditions.

Two aspects of ESP prevent sharing of data structures. First, ESP disallows global variables. Each variable is local to a single process. Second, objects sent over channels are passed by value. 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.

ESP supports immutable as well as mutable data structures. An immutable object arriving on a channel can be mutated by first applying a cast operation to obtain a mutable version of the object. Semantically, the cast operation causes a new object to be allocated and the corresponding values to be 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.

2.3.3 Data Types and Control Constructs

In addition to basic types like `int` and `bool`, ESP supports complex data types like `record`, `union` and `array`. This distinguishes it from other related domain-specific languages [14, 28, 12]. These languages are designed to express just the control portion of concurrent programs. The complex data structures have to be managed externally in a language like C. Although the control portion of a program is sufficient to check a number of properties using a model checker, it burdens the programmers by requiring them to separate the control and data portions of the program. In contrast, ESP is designed to not only to express the control portion but also manipulation of data structures. This requires ESP to address a number of additional issues like dynamic memory management (Section 2.3.5), mutable shared data structures (Section 2.3.2) and flexible external interface (Section 2.3.6).

ESP does not support recursive data types for two reasons. First, specification languages for model checkers do not support recursive data types. Second, sending recursive data types by-value over channels involves additional run-time overhead.

A process in ESP is completely sequential. ESP provides standard control constructs like `if-then-else` conditional statements and `while` loops found in traditional imperative languages.

ESP does not support functions. Instead, it uses processes to support abstraction [52]. For example, consider the following code fragment from a process which implements a page table which maps virtual addresses into physical addresses (Figure 2.1). 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 look up the mapping and sends a reply back to the requesting process. The variable `ret` identifies the process making the request so that the reply can be directed back to it. The second *case* accepts requests to update the mapping and updates the table.

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

A pair of `out` and `in` operations can be used to mimic the behavior of functions calls that expect return values. For instance:[2]

[2]`@` is a constant that represents the process id of that process (Appendix A).

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

On the other hand, functions that do not expect a return value can be modeled using a single `out` operation as follows:

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

ESP processes are a more appropriate abstraction mechanism than functions for event-driven state-machines programs for two reasons. First, a process can block on a channel while a nested function cannot block (Section 2.1). This avoids problems that arise when a function needs to block on a channel (Section 1.3.2). Second, the process abstraction allows flexibility in scheduling computation. In the last example where no return values are expected in response to the request to update the table, the code to update the table can be postponed to be performed later.

2.3.4 Dispatch on Channels

One of the features of the ESP language is the use of pattern matching to support dispatch efficiently. Pattern matching is used in languages like SML [74] to support expressive switch statements. Other languages [56, 52, 89] have used pattern matching to support dispatch. However, efficiency was not a concern in these languages.

ESP uses pattern matching to support dispatch. A channel in ESP can have multiple processes receiving messages on it. Each receiver specifies a pattern in the `in` operation—only those objects that match the pattern will be accepted by the receiver. An object sent on the channel will be dispatched to the waiting receiver whose pattern matches the object. For example, 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 of the two patterns it matches.

Using pattern matching to support dispatch has several advantages. It frees the process sending a message from having to examine it to determine the receiver. This not only simplifies the sender process but also makes the program more modular. Details about which objects a process is willing to receive is specified where it is more relevant (in the receiver). In addition, this requires the patterns to be specified only once (in the various receivers on the channel) instead of having to replicate dispatch code (in each sender on that channel).

To support pattern matching efficiently, ESP requires that all the patterns used to receive objects on a channel have to be disjoint—an object has to match exactly one pattern. In addition, each distinct pattern can be used by only one process (possibly several times). This allows a channel to be decomposed into a set of ports (Section 4.3.1). A channel can have multiple readers and multiple writers while a port can have multiple writers but only a single reader. In ESP, a channel together with a pattern specifies a port.

ESP does not require all the patterns specified on a channel to be exhaustive (even though this would ensure that each object sent on the channel would have a unique destination) for two reasons. First, the complex data types have several useless patterns (usually with one or more of their fields being `nil`) that would never get matched in a running program. The programmer would have to write an “error” process that received objects of all these types. Second, ensuring that every object sent on a channel has a unique destination does not guarantee that every send on the channel would succeed. A process that has a statement to receive objects might never execute that statement.

2.3.5 Memory Management

The design of the memory management scheme in ESP was driven by two goals. First, the programs should be safe. Bugs stemming from unsafety are difficult to find. The problem is compounded by the fact that these programs are concurrent and run on devices with minimal debugging support. Second, the memory management overhead has to be small.

Memory management schemes fall into two categories: automatic and explicit memory management. On one hand, automatic memory management using garbage collection techniques [102] provides safety but usually involves high overhead both in terms of amount of memory and processing time). On the other hand, explicit memory management involves lower overhead but are hard to program correctly with.

ESP provides a novel memory management scheme that provides safety as well as low overheads. To manage dynamically allocated memory, it provides an explicit `malloc/free`-style interface that incurs low overheads. It ensures safety using a model checker. The only unsafe aspect of ESP is its explicit memory management scheme. The memory allocation bugs can be eliminated using a model checker resulting in a safe ESP program.

The key observation is that allocation bugs are difficult to find because memory allocation correctness is a global property of a program—the property cannot be inferred by looking only at a single module of the program. A programmer has to examine the entire program to make sure that all allocated objects are eventually freed and are not accessed once freed.

To rectify this, ESP makes memory allocation correctness a local property of each process. Section 2.3.2 describes the design choices that ensure that no two processes share any data structure. It should be noted that to support pure message passing-style communication, it would have been sufficient to ensure that no two processes share any mutable data structures. However, to make memory allocation correctness a local property, ESP disallows sharing of even immutable data structures.

Making memory allocation correctness a local property allows the model checker to verify the memory safety of each process separately (Section 3.3.4 and 3.4.2). In addition, it promotes modular programming.

Objects sent over channels are passed by value. i.e. A deep copy of the object is delivered to the receiving process[3]. Objects received over a channel are treated like newly allocated objects and have to be later freed by that process. One possible complication occurs when an object contains multiple links to another object. For instance, the sender sends an array all of whose entries point to the same object. If the deep copy preserved the pointer sharing, the receiver would have to be careful to free the object stored in the object only once. However, if that process later sends a different array whose entries point to different objects, the receiving process would have to free each of the entries. To avoid this, the deep copy of ESP does not preserve any of the pointer sharing in objects being sent over channels. So the receiving process can always perform a recursive free on objects arriving over channels. In addition, the implementation does not have to incur additional runtime overhead to preserve the pointer sharing.

ESP provides a `malloc`/`free`-style interface to manage dynamically allocated memory. The allocation syntax for various types of objects is presented in Section A.7. Two primitives `free` and `rfree` (which performs `free` recursively) allow processes to free the allocated objects.

[3]This is true only semantically. The ESP runtime never has to actually copy the object (Section 4.3.3)

ESP allows dangling pointers (pointers to objects that have been already freed) during program execution. If dangling pointers were not allowed, the program would have to delete all pointers to a given object before that object could be freed. This would require additional bookkeeping and would place unnecessary burden on the programmer. Although ESP allows dangling pointers, it disallows the use of these pointers to access memory. This ensures memory safety. In contrast, the usual approach to ensure memory safety is to reclaim an object only if no pointers point to it. This avoids dangling pointer. The only other approach that we are aware of that provides safety while allowing dangling pointers is region-based memory management [93]. It uses the type system to guarantee that the dangling pointers are not used at run time.

Memory allocation in ESP is a nonblocking operation. In a concurrent program, making memory allocation blocking has some advantages. It allows a memory allocation request in one process that does not find any memory available to block till another process frees up some memory. Although this would lead to better memory utilization, it introduces additional synchronization between the processes. This forces the programmer to treat each allocation as potentially blocking and make sure that it does not cause the program to deadlock.

2.3.6 External Interface

ESP has to support an interface to C code as well as to the testing code written in the specification language of the model checkers (Section 1.4.2). ESP relies on C to implement low-level details like accessing special device registers, dealing with volatile memory and marshalling packets that have to be sent out on the network. In addition, ESP programs have to interact with code written in the specification language of the model checkers (currently, Spin) during debugging and testing (Figure 1.6).

ESP uses channels to interface with both C as well as Spin code. Regular channels have different processes reading from and writing to them. External channels have one or more processes reading (or writing) at one end of the channel while C or Spin code is writing (or reading) at the other end of the channel. External channels, like regular channels, are synchronous. This is different from the traditional approaches that provide an asynchronous interface. Some languages [24, 12] allow C code to be directly embedded in the program while others [14, 28] allow functions that are implemented externally to be invoked. Although Squeak [24] uses primitive channels for some external interaction, these channels are not user definable and are built into the compiler.

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 (Appendix B), the retransmission processes can be dropped and the regular channels used to interact with it can be converted into external channels. Other processes that were using these channels are not affected because they cannot tell the difference between an external channel and a regular channel.

The external interfaces work as follows:

C interface. The C interface is illustrated by the following example.

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

```
        }
```

This declares an external interface `UserReq` that specifies an external writer on channel `UserReqC`. It also specifies a list of function suffix and pattern pairs.

To support a synchronous C interface on this channel, ESP requires two types of functions to be provided: *sync* and *transfer* functions. The sync function has a “IsReady” suffix (`UserReqIsReady`) and returns whether the C code is ready to send data over the channel. The transfer functions are called to transfer data over the channel after the corresponding “IsReady” function has indicated its readiness. Each channel requires a single sync function to be provided. However, it requires a separate transfer function for each pattern (`UserReqSend` and `UserReqUpdate`).

Each data transfer over an external channel involves a call to the sync function followed by a call to one of the transfer functions. The second call has to be performed immediately after the first one. It is not valid for the implementation to first check the readiness of several different channels and then call the transfer functions on all the channels that are ready. This is because invoking one of the transfer functions may cause other external channels that were ready to be no longer ready.

The use of patterns on external channels serves two 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 a 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.

Spin Interface. The generated Spin model has to interface with testing Spin code that is provided to the programmer (Section 1.4.2). Since Spin supports synchronous channels, the ESP channels are simply translated into synchronous Spin channels. The external Spin code can interact directly with the generated Spin model by reading and writing to the appropriate channels.

2.4 Case Study: VMMC Firmware

The VMMC firmware is used as a case study to evaluate ESP (Section 1.3.3). The VMMC firmware was reimplemented using ESP[4] and compared with the earlier implementation that used event-driven state machines in C.

The use of ESP resulted in firmware that was significantly easier to write, understand and maintain. The earlier implementation includes about 15600 lines of C code (Around 1100 of these lines were used to implement the fast paths).[5] In contrast, the new implementation using ESP required 500 lines of ESP code together with around 3000 lines of C code.[6] The C code implements simple tasks like initialization, initiating DMA, packet marshalling and unmarshalling and shared data structures with code running on the host processor (in the VMMC library and the VMMC driver). All the complex state machine interactions are restricted to the ESP code, which uses 8 processes and 19 channels (Appendix B). This is a significant improvement over the earlier implementation where the complex interactions were spread throughout the 15600 lines of hard-to-read code.

ESP addresses the problems that programmers face when programming with event-driven state machines in C (Section 1.3.2). This can be observed by comparing the C code

[4]The implementation supports most of the VMMC functionality [41] (only the redirection feature is currently not supported.

[5]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.

[6]Currently, ESP does not support fast paths.

fragment in Figures 1.2 & 1.3 with the equivalent ESP code fragment in Figure 2.1. ESP addresses the problems as follows:

- ESP programs represent state machines implicitly using processes. This avoids code fragmentation that results from having to specify state machines explicitly in C. In addition, ESP uses processes to implement functions. This avoids the problem that arises when a state machine needs to block in a nested function call.
- ESP processes can communicate only by sending messages to each other over channels. Channels provide well-defined interfaces between different processes. This promotes modular programming, as a process is a modular unit in ESP. In C, the state machines communicated using shared data structures.
- ESP supports dynamic memory to be managed in a modular fashion. Each process independently manages its memory using an explicit `malloc/free`-style interface. For instance, process `SM1` can free `sendData` when it no longer needs it without worrying about whether another process is still using it. Since the explicit interface introduces unsafeness, a model checker is used to ensure safety (Chapter 3).
- ESP uses pattern matching to support dispatch on channels efficiently, so all the code that processes that read on a channel do not have to be collocated. They can be placed in the relevant modules. For instance, code that handle requests on the channel `userReqC` are located in two different processes.
- ESP uses channels to interact with external code that is used to implement low-level device handling in C. The use of channels provides a simple and powerful interface to C code by leveraging features that are already built into ESP channels. These include the ability to wait for events and dispatch to multiple processes.

- ESP currently does not support fast paths. This is a subject for future research (Section 4.5).

2.5 Future Work

ESP is designed to write firmware for a broad class of devices. It does not contain any features specific to VMMC firmware or the Myrinet network card. However, so far ESP has been used to only implement the VMMC firmware for the Myrinet network card. It would be useful to further validate the design of ESP by using it to implement firmware for other devices.

The current design of ESP has left out some language features like recursive data types that were not found to be essential. However, experience with writing firmware for other devices in the future could make a case for adding support for these features in ESP. In addition, some low level device handling capability [71, 78, 27] can be added to ESP. This will reduce the use of C in ESP programs even further.

2.6 Summary

Programming in ESP is significantly easier than programming with event-driven state machines in C. This addresses the first goal that ESP was designed to meet.

ESP allows programs to be written in concise, modular fashion. First, ESP programs consist of processes communicating with each other over synchronous channels. A process encodes a state machine implicitly and comprises a modular unit in a ESP program. Channels provide well-defined interfaces between the processes. Second, dynamic memory is managed explicitly by ESP programs. However, a novel scheme allows each process to independently manage its memory. This not only promotes modular programming but also allows a model checker to be used to verify safety in ESP programs. Third, pattern matching is used to support dispatch on channels efficiently. Finally, channels are used to provide a flexible and powerful interface to C code. Some of the low-level device handling is currently implemented in C.

The VMMC firmware is used as a case study to evaluate ESP. It was reimplemented using ESP and compared with the earlier implementation that used event-driven state machines in C. We found that ESP greatly simplifies the task of writing the firmware. It required about five times less lines of code than the earlier C implementation. The earlier implementation in C required 15600 lines of hard-to-read C code. The new implementation in ESP required 500 lines of ESP code along with 3000 lines of C code. All the complex interactions are contained in the ESP code—the portion written in C implement simple low-level device handling.

Chapter 3

Developing and Testing using a Model Checker

Device firmware has to be reliable, as it is trusted by the operating system. It has the ability to write to any location in the physical memory. A stray memory write resulting from a bug can corrupt critical data structures in the operating system and can crash the entire machine.

Writing reliable firmware for these devices using event-driven state machines is a challenging problem for three reasons. First, concurrent programs are inherently hard to write correctly. Often, they have unforeseen interactions between the different sequential flows of control resulting in race conditions. Second, the problem is compounded in languages like C that are not designed to support event-driven state-machines programming. Event-driven state-machines programs can be written in these languages using an explicit interface (Section 1.3) which requires state machines to be specified explicitly using function pointers. The resulting programs are difficult for the programmer to understand and for the compiler to compile efficiently. To get good performance, the programmer is forced to perform some optimizations manually. This introduces subtle bugs in the program. Third, very limited debugging support is available on the devices.

The earlier implementation of VMMC firmware [41] was implemented using event-driven state machines in C. Our experience was that using event-driven state machines in C was very error-prone and difficult to debug. Even after several man-years spent on debugging the VMMC firmware, we continue to encounter bugs frequently. These bugs are often due to race conditions that occur very infrequently and, are therefore, very hard to find.

Model checking is a promising approach to building reliable concurrent software. Model checkers take a model of the system and explore all possible interleaved executions of the concurrent system. However, since the number of possible executions grows exponentially with the size of the model, abstract models that hide details in the original system are necessary. In addition, often only a fraction of the model can be explored. In spite of these limitations, the systematic search performed by the model checker results in much more extensive testing than traditional methods.

Model checkers require a model of the program to be provided. For model checking to be effective, the model has to be reasonably small. General-purpose languages like C include features like unsafe pointers and recursive data types that make it difficult to extract tractable models automatically. The ESP language has been carefully designed so that the ESP compiler can extract models that can be used for model checking. The VMMC firmware is used as a case study to demonstrate the effectiveness of this approach.

The rest of this chapter is organized as follows. Section 3.1 discusses the background. Section 3.2 describes the related work. Section 3.3 presents the model extraction process using the ESP compiler. Section 3.4 presents our experience with using the Spin model checker to debug VMMC firmware. Section 3.5 suggests directions for future research. Finally, Section 3.6 presents a summary.

3.1 Background

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 in 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.

Model checking verifiers can check for a variety of properties. These properties are traditionally divided into *safety* and *liveness* properties. Safety properties are properties that have to be satisfied in specific global states of the system. Assertion checking and deadlock are safety properties. Assertions are predicates that have to hold at a specified point in one of the state machines. This corresponds to the set of global states where that state machine is at the specified point and the predicate holds. A deadlock situation corresponds to the set of all the global states that do not have a valid next state. Liveness properties are ones that refer to sequence of states. Absence of livelocks is a liveness property because it corresponds to a sequence of global states where no useful work gets done. Liveness properties are usually specified using temporal logics like Linear Temporal Logic (LTL) and Computation Tree Logic (CTL).

The advantage of using model checking is that it is automatic. Given a model 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.

There are two problems with using model checkers. First, the state space to be explored is exponential in the number of processes and the amount of memory used. Therefore 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. Second, the specification language supported by the model checkers provides limited functionality. It is not straightforward to translate concurrent programs written in traditional programming languages into the specification language of the model checkers.

Abstraction is the key to addressing both of these problems. Depending on the property to be verified, a model that captures only the relevant details has to be extracted. For properties involving small subsystems, detailed models can be used. However, for properties involving large subsystems, abstract models have to be used.

Models are usually extracted by hand. This process can be time consuming. In addition, it is hard to be sure that the model accurately captures the actual system. Worse yet, as the system evolves, the model has to be independently updated to reflect the changes. Hence, the use of model checking verifiers is greatly simplified when the models can be extracted automatically[57, 35].

3.2 Related Work

3.2.1 Model Extraction Approaches

Model extraction by hand. Several researchers have verified various aspects of operating systems using model checkers. These efforts involved extracting an abstract model of the system by hand. Spin was used to verify the Interprocess Communication Subsystem in Harmony [26] (a real time operating system) and the RUBIS microkernel [43]. The latter study found that significant effort was needed in extracting the model. Spin was also used to develop and verify a synchronization protocol for Plan 9 [83]. More recently, Spin was used to verify the IPC system of the Fluke OS [94]. All these studies found that the model checking verifier was able to find some hard-to-find race conditions.

Automatic Model Extraction. To avoid the problems with model extraction by hand, some researchers have extracted the models automatically from the source code. Teapot [28] is a domain-specific language for implementing software cache coherence. It extracts a model that can be used by the Murphi model checker [40]. Promela++ [12] is a language for implementing layered protocols. Its compiler generates model that can be used by the Spin model checker. Esterel [14] is a language for specifying synchronous reactive systems and is primarily used in hardware design. The Esterel programming environment includes verification tools like model checkers that can be used to test the programs. Esterel was used to implement a subset of the TCP protocol [25]. They showed that Esterel could be used to generate efficient code. However, they did not report any experience with the verification tools.

In all these cases, the domain-specific language is used to encode the control structure of the program. The rest of the program (data handling) is handled using a different language (typically C). The compiler for these languages extracts a single model that reflects the control structure of the program.

Java PathFinder [50] translates Java programs into Spin models. It handles a significant subset of Java including dynamic object allocation, object reference, exception processing and inheritance. However, it does not handle features like method overriding and overloading. Also, it does not provide a way to abstract details so that a tractable model can be extracted.

Verisoft [48] uses a different approach to perform model checking on a concurrent system. Instead of trying to extract a model, it explores the state space of the system by replacing the scheduler of the concurrent system. By controlling the scheduler, it can force the concurrent program to execute all possible interleavings. This allows it to apply model checking to actual programs written in traditional languages like C (instead of a model). The problem is that it can explore much smaller state spaces because it cannot use some of the optimization techniques used by model checkers like Spin.

Automatic model extraction with support for abstraction. More recent efforts have focused on extracting several abstract models to verify different properties in the system.

FeaVer [57] extracts Spin models from programs written in a C dialect that has simple extensions to support event-driven state machines. The system allows the programmer to specify pairs of C and Spin code patterns. When the C pattern is encountered during translation, the corresponding Spin code is generated. This approach automates the extraction of abstract models. However, the translator does not have any semantic information to check the validity of the translation. The system was used to debug the call processing software for Lucent's Pathstar access server.

Lie et. al. [67] use an approach similar to FeaVer [57] to extract Murphi [40] models from C programs. It requires the programmer to specify two things: a set of patterns that identify the C code that has to be captured in the extracted model, and transformations that translate the identified C code into Murphi code. Unlike FeaVer, it uses program slicing [101, 92] to extract additional code that affects the identified code. However, the standard slicing algorithms have problems with C constructs like pointers, unions and unstructured control flow. Like FeaVer, it cannot check the validity of the generated model.

Bandera [35] allows automatic extraction of finite state models from Java programs. It uses techniques like program slicing [101, 92] and data abstraction to allow more tractable models to be extracted. However, it was used to verify properties in a fairly simple program.

The SLAM project [10, 11] extracts a predicate abstraction to check assertions in sequential programs written in C. A predicate abstraction is a model with only boolean variables that correspond to conditions in the original program. The assertion is checked in the predicate abstraction using a model checker. Since the checker may generate false positives, symbolic execution is used to verify the counterexamples generated by the model checker. If the counterexample is invalid, the predicate abstraction is refined to eliminate the counterexample. This approach has not yet been extended to handle concurrent programs.

3.2.2 Debugging System Software

A vast amount of research has focused on the problem of debugging system software. The techniques used span language design, model checking, compiler analysis, and runtime methods. In this section, we discuss some of the related work in this area.

As described in Section 3.2.1, model checkers have been used to debug system software. Some have focused on debugging programs written in general purpose languages like C, C++ and Java [26, 43, 83, 94, 50, 48, 57, 35]. Others have proposed domain-specific languages that have been designed with model checking in mind [28, 12, 14], and therefore, allow model checking to be more effective.

Meta-level Compilation [32, 46] provides a framework for extending a compiler with application-specific code that can be used to statically check certain properties of that application. It was used to look for bugs in several systems including the cache coherence protocols for the FLASH multiprocessor and the Linux kernel. This technique requires little change to the source code and has been able to find around 500 bugs in these systems. The compiler extensions look for violations of properties like proper buffer allocation and deallocation, and absence of deadlocks. However, these extensions perform only intra-procedural analysis. In some instances, a separate global pass was used to combine data gathered by the intra-procedural analysis of the different functions to check a global property. Since the static analysis is inexact, it can generate false positives. The bugs reported have to be double-checked by the programmer. In addition, the limited scope of intraprocedural analysis can generate false negatives.

Eraser [90] detects data races in multithreaded programs. It instruments the program binary to check at runtime that a lock protects each shared variable access. It does not impose any constraints on the programs, and therefore works on existing programs with little modifications. However, the tool can detect only the data races that occur during the debugging runs; it is the programmer's responsibility to ensure that the program is run with several different inputs so that it is tested thoroughly. In addition, the instrumentation results in a factor of 10 to 30 slowdown in program execution. This can prevent some data races in the program from occurring during debugging.

Programming language features can often prevent an entire class of bugs. For instance, safe programming languages prevent a program from accessing a dynamically allocated object after it has been freed. The Vault [38] language uses an expressive type system to enforce high-level protocols in system software. The type system allows a module writer to specify properties like "a read system call to read from a file can be called only after that file has been opened using the open system call".

Memory allocation bugs are notoriously difficult to find because they usually result in memory corruption that leads to faulty behavior at a location in the program different from the site of the bug. A number of tools [104] help in detecting memory allocation in unsafe languages like C. For instance, the Purify [49] tool inserts code in the executable that check for a number of bugs like invalid indices in array accesses and memory leaks. This allows it to detect error when it happens at run time. However, it is the programmer's responsibility to run the executable with different inputs so as to exercise every possible program path. A different approach [98] to find a more limited class of bugs (buffer overruns) is to formulate the buffer overrun problem as a integer constraints problem and statically check for constraint satisfaction. A limitation of this approach is that it can flag false-positives as well as false-negatives.

The alternate approach to dealing with memory allocation bugs (except memory leaks) is to avoid them by using a safe programming language (Section 2.2).

3.2.3 Symbolic Model Checking

An alternative approach to explicitly exploring the entire state space is symbolic model checking. The basic idea of symbolic model checking [70] is to compactly represent the state space using Binary Decision Diagram (BDD). This allows the model checker to check much larger models. While symbolic model checking is very effective for certain domains like hardware circuits, it often does not perform better than explicit model checkers for software systems.

3.3 Extracting Spin Models

ESP supports the use of model checkers to develop and test ESP programs. Models are extracted automatically from ESP program by the ESP compiler. These can be used to check different properties of the program. The ESP approach differs from the previous efforts as follows:

Domain-specific Language. ESP is designed not only to simplify the task of programming devices but also to make it easier to extract models. General-purpose languages like C++ and Java have language features (complex pointer manipulation, exceptions etc.) that are difficult to translate into the specification language of the model checkers [50, 57, 35].

Support for Abstraction. Other domain-specific languages [28, 12, 14] extract a single model from the program and use it for model checking. To avoid the state-space explosion associated with very detailed models, these languages have been designed to encode only the control structure of the program. In contrast, the ESP language provides support for both control structure and data manipulation. The ESP compiler uses abstraction to discard some unnecessary details and generate more tractable models.

The ESP compiler (Figure 1.6) generates three types of models. The *detailed* models contain all the details of the ESP program. Usually, these models can be used to check properties of only small subsystems. The *memory-safety* models can be used to check the program for memory allocation bugs. The *abstract* models contain only those details of the ESP program that are relevant to the property being verified. These models can be used to check for system-wide properties like absence of deadlocks. This class of bugs usually involve several different processes. Therefore, these bugs are especially hard to find. Abstract models were used to find several bugs in the VMMC firmware that resulted in deadlocks (section 3.4.3).

The ESP compiler currently generates models that can be used by the Spin model checker. However, the design of ESP is not tied to Spin. The ESP compiler can easily be retargeted to generate models for other model checkers like Murphi [40].

The rest of this section is organized as follows. Section 3.3.1 presents a brief description of Spin and describes why it was chosen for checking ESP programs. Section 3.3.2 describes the ESP compiler stages involved in model extraction. Finally, Sections 3.3.3, 3.3.4 and 3.3.5 describes the procedure used to extract detailed, memory-safety and abstract models respectively.

3.3.1 Spin Model Checking Verifier

Spin [55] 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 the specification language. This would make the state-space 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 allows verification of safety as well as liveness properties. The liveness properties in Spin are 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. In cases where the state space is too big to be explored completely, it can do partial searches. It provides three different modes for state-space exploration. The entire state space is explored in the *exhaustive* mode. For systems with larger state spaces, the *bit-state hashing* mode performs a partial search using significantly less memory. It uses the fact that state spaces are fairly sparse and uses a hash function to obtain a much more compact representation for a state. However, since the hash function can map two states onto the same hash, a part of the state space may not be explored. This technique often allows very high coverage ($>$ 98 %) while using an order of magnitude 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. Thus, 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.

3.3.2 Compilation Stages

The ESP Compiler (Section 1.4.3) compiles the ESP program into optimized C code in several stages (Section 4.3.1). The model extraction can be implemented at any of the intermediate stages. However, the ESP compiler does this very early—right after type checking—for several reasons. First, the Spin specification language does not support pointers. Hence, the translation is much more difficult at the later 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. A disadvantage of this approach is that the model checker cannot catch any bugs introduced by the compiler.

The model extraction takes the abstract syntax tree generated by the type checker and generates Spin models in two stages:

Abstraction Preprocessor. This stage is used only for extracting abstract models and is bypassed during the extraction of detailed and memory-safety models. It drops the details from the abstract syntax tree using the programmer specified abstractions (Section 3.3.5).

Model Generator. This generates Spin models from the abstract syntax tree.

3.3.3 Extracting Detailed Models for Spin

The *detailed* models extracted by the ESP compiler contain all the details from the original ESP program. These detailed models tend to have too much state to be able to perform exhaustive exploration. However, these models are useful while developing and debugging the system using the simulation mode in Spin. They can also be used to check for properties in small subsystems.

The translation of ESP programs into Spin models is fairly straightforward with a few exceptions. The problems in translation arise from the lack of support for pointers and dynamic memory allocation in Spin.

Processes and Channels. ESP processes and channels can be directly translated into Spin processes and rendezvous channels. Figure 3.1 shows a simple ESP process that implements a mutual exclusion lock and Figure 3.2 presents the Spin model generated by the ESP compiler. The entire body of an ESP process can be wrapped in an `atomic` statement in the generated model. This allows Spin to make scheduling decisions only on blocking operations on channels. This is valid because ESP processes cannot communicate by updating shared variables. In the absence of the `atomic` statement, Spin would have to make a scheduling decision after every statement. This would add new intermediate states, thereby increasing the amount of state space to be explored.

Pointers and Dynamic Allocation. Variables in ESP store pointers to data objects. For instance, in Figure 3.3, variables `a1` and `a2` point to the same array object. Since Spin does not support pointers, the above ESP code fragment gets translated into the Spin code shown in Figure 3.4. In the Spin code, variable `a1` and `a2` point to different array objects. Therefore, the assignment `a2 = a1` causes the entire object to be copied over.

```
channel lock: int
channel unlock: int

process mutex {
  $owner = -1;    // No owner
  while ( true) {
    alt {
      case( owner == -1, in( lock, owner)) {} // Do nothing
      case( in( unlock, $p)) {
        assert( p == owner);
        owner = -1;
      }
    }
  }
}
```

Figure 3.1: A ESP program

Although this works for immutable objects, this is insufficient for mutable objects. For instance, in Figure 3.5, an update to `b1` has to be visible to `b2` which will not happen automatically because `b1` and `b2` point to different array objects in the translated Spin code.

To address this, each object is assigned an *objectId* at allocation time. The objectId is stored as an additional field in the object itself. When an object gets copied due to an assignment operation, the objectId field also gets copied. This ensures that all objects in the translated Spin code that share the same objectId represent a single object in the original ESP code. When a mutable object gets updated in ESP code, the translated Spin code includes code to check and update all other objects with the same objectId. The above ESP code fragment gets translated into the Spin code shown in Figure 3.6.

An alternate approach to dealing with the lack of support for pointers and dynamic memory allocation in Spin is to maintain arrays ("heaps") of data objects of each type used in the ESP program. Pointers in the ESP code would then be translated into indices into the array of the corresponding type. An object allocation would allocate an unused index

```
mtype = { OK, DONT_SEND, DONT_RECV};
chan lock [NUM_PROCESSES] = [0] of { mtype, int};
chan unlock [NUM_PROCESSES] = [0] of { mtype, int};

proctype mutex( int pid) {
    int owner;
    int p;
    atomic {
        owner = -1;
        do
          :: 1 -> {               // true
                if                // Nondeterministic conditional
                :: lock[pid] ? eval( (owner==-1) -> OK: DONT_RECV)), owner ->
                        skip      // Do nothing
                :: unlock[pid] ? OK, p -> {
                        assert( p == owner);
                        owner = -1;
                    }
                fi
            }
        :: else -> break          // Never executed
        od
    }
}
```

Figure 3.2: Spin model (detailed) generated from the ESP code shown in Figure 3.1

```
$a1: array of int = { -> 0, 2};      // Allocate
$a2 = a1;                            // Copy pointer
```

Figure 3.3: Code fragment I: Assigning immutable objects

```
typedef intArray {
    int length;
    int contents[MAX_ARRAY];
};

// Variable declaration
intArray a1, a2;
int index;

// $a1 : array of int = -> 0, 2;
a1.contents[0] = 0; a1.contents[1] = 2; a1.length = 2;

// $a2 = a1;
a2.length = a1.length;
index = 0;
do
:: index < MAX -> {
        a2.contents[index] = a1.contents[index];
        index ++
    }
:: else -> break
od;
```

Figure 3.4: Spin code corresponding to the ESP code fragment in Figure 3.3

```
$b1: #array of int = #{ -> 5, 11};  // Allocate
$b3: #array of int = #{ -> 12, 3};  // Allocate
$b2 = b1;                              // Copy pointer
b1[1] = 7;                             // Update
```

Figure 3.5: Code fragment II: Assigning mutable objects

```
typedef intArray {
    int length;
    int objectId;
    int contents[MAX_ARRAY];
};

intArray b1, b2, b3;
int index;

// $b1 : #array of int = # -> 5, 11;
b1.contents[0] = 5; b1.contents[1] = 11; b1.length = 2;
b1.objectId = NEW_ID();

// $b3 : #array of int = # -> 12, 3;
b3.contents[0] = 12; b3.contents[1] = 3; b3.length = 2;
b3.objectId = NEW_ID();

// $b2 = b1;
b2.length = b1.length;
b2.objectId = b1.objectId;
index = 0;
do
:: index < MAX -> {
        b2.contents[index] = b1.contents[index];
        index ++
    }
:: else -> break
od;

// b1[1] = 7;
b1.contents[1] = 7;
if
:: b2.objectId == b1.objectId -> b2.contents[1] = 7
:: else -> skip
fi;
if
:: b3.objectId == b1.objectId -> b3.contents[1] = 7
:: else -> skip
fi;
```

Figure 3.6: Spin code corresponding to the ESP code fragment in Figure 3.5

from that array while a free would release the index for later reuse. An assignment (`b2 = b1`) would simply copy the index from variable `b1` into variable `b2` and an object update (`b2[3] = 7`) would use the index stored in `b2` to update the corresponding object.

This approach would yield a simpler translation than the one taken by the ESP compiler. In addition, it can yield a more compact representation for each state. This is because multiple pointers in ESP program often point to the same object and so the number of different objects that are needed can be less than the number of pointers in the program. However, this approach can result in an increase in the amount of state space that has to be searched by Spin. This is because a state is stored in Spin as a state vector with all the data objects laid out in a particular order. State equivalence is determined by comparing the state vectors for equality. Thus, if an object gets assigned different indices at two different points during a run of the program, it will appear as two different states to Spin even when they represent the same state in the ESP program. The approach taken by the ESP compiler would not suffer from this problem. Therefore, while the approach taken by the ESP compiler might take longer to execute a state transition and use more memory to store each state, it has to search a smaller amount of state space. The former involves linear factors while the latter involves exponential factors.

Arrays. 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 when the size cannot be determined at compile time. The actual length of the array is stored in a separate field in the object.

Unions. Spin does not support union types. Therefore, ESP unions are translated into a record in which only one of its fields is ever valid. An additional field is added that specifies which of the record fields is valid. All invalid fields in the union are zeroed out.

This ensures that although extra memory is being used to store a union object, the amount of state space that has to be searched remains unchanged.

Multiple Instantiations. The ability to run multiple instantiations of the generated Spin model can be very useful during debugging. For instance, a setup where the VMMC firmware on running on a cluster of machines that are communicating with each other can be modeled by running multiple instances of a Spin model extracted from the VMMC firmware and connecting them with Spin code that mimics the network. Cluster-wide properties like absence of deadlocks can then be verified using this approach.

To support this, the ESP compiler generates Spin models that can be instantiated multiple times. Each Spin process in the generated model has to be provided an *instantiationId* when it is started. Each channel in the ESP program is translated into an array of Spin channels and accessed by indexing using the instantiationId.

3.3.4 Extracting Memory-Safety Models for Spin

The *memory-safety* models generated by the ESP compiler can be used to check for memory allocation bugs in the program. These models are essentially detailed models (section 3.3.3) with some additional Spin code inserted to check for validity of memory accesses. Therefore, they contain even more state than the detailed models. In spite of this, these models can be usually used to exhaustively explore the state space for allocation bugs. This is because the memory safety of each individual process can be checked separately using the verifier (Section 2.3.5).

The memory-safety model includes additional code that checks the validity of each object that is accessed. When a new object is allocated, an unused objectId (Section 3.3.3) is assigned to the object. Before every object access, code is inserted in the model to check that the object is live. Array accesses include additional code to check that the array index is within bounds. Union references include code to check that field being accessed is valid. When an object is freed, all objects in the model with that same objectId are marked as invalid by changing the objectId field to -1.

The memory-safety model checks for bugs like accessing an object after it has been freed, double freeing an object, and using an invalid array index. In addition, it can also find most memory leaks in a process. This is because a process in the generated model has a bounded number of objects and the compiler can determine this bound. Arrays are the only source of unbounded allocation in an ESP process, since ESP does not support recursive data types. However, the ESP compiler imposes a bound on the maximum lengths of the arrays during model extraction (Section 3.3.3), thereby bounding the number of objects in the model. By constraining the model to only pick objectIds within this bound, any steady memory leak can be detected. A steady leak will cause the model to run out of objectIds during model checking.

3.3.5 Extracting Abstract Models for Spin

The *abstract* models generated by the ESP compiler omit some of the details that are irrelevant to the particular property being verified. These models can have significantly smaller state than the detailed models and can be used to find bugs in much larger systems.

The ESP compiler makes conservative approximations when generating abstract models. During abstraction, some of the values in the model might become *undeterminable*. For instance, the value of a variable in the model that depended on another variable in the original program that was discarded during abstraction will become undeterminable. The compiler keeps track of these values and makes sure that the abstract model only broadens the scope of model checking [55]. For instance, when the value of a condition in a conditional statement cannot be determined, it can be replaced by a nondeterministic choice in the model. During model checking, both the branches of the conditional statement will be explored. Although this might introduce new deadlocks into the model that do not exist in the program (false positives), all the deadlocks in the program will be present in the model allowing them to be identified during model checking.

The ESP program in Figure 3.7 will be used to illustrate the extraction process. The program implements the following functionality:

- Process `pageTable` translates virtual addresses to physical addresses. It maintains a table that maps virtual page numbers into physical page numbers. It accepts translation requests on channel `reqC` and sends replies on channel `replyC`. Since a region of contiguous virtual memory can map onto a set of noncontiguous physical pages, each request sent on channel `reqC` can yield multiple replies on channel `replyC`. The last reply is identified by the `last` field in the reply.

- Process `transfer` computes a pair of `vaddr` and `size` that identifies a region in virtual memory. It sends a request on `reqC` to translate it into physical addresses. It then receives the physical addresses on channel `replyC` and uses it to transfer data.

- Since other processes might be sending requests on channel `reqC`, the caller field on the channels `reqC` and `replyC` is used to match the replies with the request. `@transfer` is a constant that represents the process id for the process `transfer`.

The rest of this section describes how abstract models are extracted and used in ESP. It starts with a description of the types of abstractions that are supported by the compiler. Then, it discusses the techniques used by the ESP compiler to extract the abstract models. Finally, it uses the example in Figure 3.7 to show how a property, namely the absence of deadlocks, can be verified using an abstract model.

```
#define TABLE_SIZE     100
#define PAGE_SIZE      4096
#define PAGE(a)        ( (a) / PAGE_SIZE)
#define OFFSET(a)      ( (a) % PAGE_SIZE)
#define ADDR(p)        ( (p) * PAGE_SIZE)
type reqT = record of { caller: int, addr: int, size: int}
type replyT = record of { caller: int, last: bool, addr: int, size: int}
channel reqC : reqT
channel replyC : replyT

process pageTable {
  $table : #array of int = #{ TABLE_SIZE -> 0 ... };
  // Omitted: Code to initialize the table
  while ( true) {
    in( reqC, { $caller, $vaddr, $size});
    assert( !OFFSET(vaddr)); // Assumes vaddr is page aligned
    $done: bool = false;
    while( !done) {
      $paddr : int = ADDR(table[PAGE(vaddr)]); // Look up physical address
      $chunk : int = PAGE_SIZE;
      if ( size < PAGE_SIZE)    chunk = size; // Calculate size
      size = size - chunk;
      done = ( size == 0);
      out( replyC, { caller, done, paddr, size}); // Send reply
    }
  }
}

process transfer {
  while ( true) {
    // Omitted: Code that generates values for variables vaddr and size
    out( reqC, { @transfer, vaddr, size});
    $last : bool = false;
    while( !last) {
      in( replyC, { @transfer, last, $paddr, $chunk});
      // Omitted: Code to transfer chunk bytes at address paddr
    }
  }
}
```

Figure 3.7: Another ESP program.

The abstractions to be performed by the compiler have to be specified by the programmer. ESP currently allows the programmer to specify the following two types of abstractions.

Replacing Types. It allows a complex type to be replaced by a much simpler type. This can be done either by specifying an alternative type for the variables individually or by specifying an alternative type in a type declaration. For instance, if the original program contained the following type declaration:

```
type replyT = record of {
  caller: int, last: bool, addr: int, size: int
}
```

then the programmer can specify the following abstraction:

```
replace type replyT = record of { caller: int, last: bool}
```

Currently, ESP requires the replacement type to be a supertype of the original type. Essentially, it allows fields from records and unions to be dropped.

Replacing a complex type by a simpler type can significantly reduce the amount of state in the model. For instance, the code to implement the retransmission protocol accepts packets that are implemented as a union of the different types of packets that have to be sent. However, as the content of the packet makes little difference to the correctness of the retransmission code, the complex datatype representing the packet can often be replaced by a simpler type in the abstract model. The amount of state can also be reduced if the size of the arrays in the model can be reduced. Often, the size of arrays affects only performance and not correctness.

Dropping Variables. Some variables that do not affect the validity of a property being checked can be dropped altogether. For instance, a table that keeps track of the mapping between virtual and physical addresses in the main memory might not be relevant when checking the firmware for deadlocks. The variable `table` in process `pageTable` can be dropped by specifying the following abstraction:

```
drop pageTable $table
```

Once the abstractions have been specified by the programmer, the ESP compiler uses them to generate abstract models. First the compiler performs a type-checking phase during which it determines a type for every expression in the original program (without taking the abstractions into account). Then the model generator phase can apply the abstractions to each of the statements of this fully-typed program independently.

The abstractions specified can cause some of the expressions in a statement to have an indeterminable value. In these situations, the ESP compiler uses nondeterminism to make conservative approximations that strictly generalizes the scope of model checking. The various expressions in a statement can be classified into two classes: *left-exp* and *right-exp*. They are handled as follows:

left-exp. A left-exp is an expression that is used to determine a memory location to which a value will be stored. These expressions appear on the left side of the assignment statements and in `in` operations on channels. Consider the following statements:

```
a = b;
a[i].last = d;
```

where variable `a` has the type

```
type tableT = #array of #record of {
    first: int, last: int
}
```

In the simplest cases, when a left-exp becomes undeterminable, the statement can be simply discarded during model extraction. For instance, if the variable `a` is dropped by the abstraction, the first statement `a = b;` becomes irrelevant and can be discarded. This is because the only side effect of that statement is to the variable `a`. Similarly, if the `last` field is dropped from the type `tableT`, the second statement can be discarded during model extraction. This is because all objects of that type no longer have the `last` field. As a result, the statement has no remaining side effect in the generated model.

The most general case that has to be handled occurs in the second statement when variable `a` or variable `i` is dropped. In this case, the object pointed to by `a[i]` is being mutated, and the change would be visible to any other pointer that was pointing to the same object. To handle this case, the compiler has to determine a list of pointers to which `a[i]` could be aliased. For each of these pointers, the generated model has to include a nondeterministic statement that either updates the object to which it points or does not update that object.

Nondeterministically updating one of a large set of objects can dramatically increase the amount of state space that has to be explored. It can also result in false-positive bugs being introduced into the model. A number of techniques can be used to narrow the list of pointers to which `a[i]` can be aliased. First, only pointers of the same type as `a[i]` have to be considered. Second, only pointers within the same process can be aliased to `a[i]`, since processes in ESP do not share objects. Third, in the case where only variable `i` is dropped, only objects pointed to by an entry in array `a` needs to be considered. Finally, alias analysis can be used to further reduce the list of pointers. If compile-time analysis can determine that the pointer `a[i]` is not aliased to any other pointer, the situation reduces to the simple case where the statement is simply discarded.

right-exp. All expressions that are not left-exp expressions are right-exp expressions. These generate values to be used at various points in the programs. They appear on the right side of the assignment statements, in conditionals of `if` and `while` statements, and in `out` operations on channels.

During abstraction, the value of a right-exp expression can become undeterminable. Ideally, such an expression should be replaced by one that nondeterministically returns a valid value of the type of that expression. This will cause the model checker to try all possible valid values during the state space exploration.

For boolean expressions, only two values are possible and a nondeterministic choice between the two can be made. Therefore, a boolean expression in a conditional statement (like an `if` statement) whose value can no longer be computed is replaced by a nondeterministic statement [55]. During model checking, both the branches of the conditional statement will be explored.

For nonboolean expressions, trying all possible valid values would be computationally very expensive during model checking. It is also usually unnecessary because a small set of values can effectively explore the entire space. However, there is no general way for the ESP compiler to determine the set of values that would be sufficient to cover the entire state space. The ESP compiler relies on the programmer to supply the set of values. For each type in program (except boolean) for which the abstract model needs a nondeterministic value, a channel is generated in the abstract model. When a value is needed, the model performs a read operation on the channel. The programmer is responsible for supplying values on the channel.

```
replace type reqT = record of { caller: int}
replace type replyT = record of { caller: int, last: bool}
drop pageTable $table, $vaddr, $size, $paddr, $chunk
drop transfer $vaddr, $size, $paddr, $chunk
```

Figure 3.8: Abstraction 1

To illustrate the use of abstract models to check for a property, we will use the ESP program in Figure 3.7. We will check for the absence of deadlocks in the program.

The abstraction in Figure 3.8 can be used to check the program for the absence of deadlocks. Using the abstraction, the ESP compiler generates an abstract model (Figures 3.9 and 3.10). The abstraction drops all variables except `caller` and `done` in process `pageTable` and `last` in process `transfer`. It also replaces the types of the two channels. During abstraction, the value of the boolean variable `done` becomes indeterminate because its value depends on the value of the variable `size` that was dropped. The compiler translates the statement

```
done = ( size == 0);
```

into Spin code that nondeterministically assigns either values `true` or `false` to it as follows:

```
if
:: skip -> done = 0
:: skip -> done = 1
fi
```

The Spin model checker can exhaustively explore the entire state space (12 states!) and determine that there are no deadlocks. In contrast, if a detailed model were used, the model checker would have to potentially explore a large number of states (by trying all possible values for `vaddr` and `size`) to determine that there were no deadlocks.

```
mtype = { OK, DONT_SEND, DONT_RECV};
typedef reqT { int nil; int caller; };
typedef replyT { int nil; int caller; bool last; };
chan reqC [NUM_PROCESSES] = [0] of { mtype, reqT };
chan replyC [NUM_PROCESSES] = [0] of { mtype, replyT };

proctype pageTable( int pid) {
  int caller; bool done;
  atomic {
    do
    :: 1 -> {
         reqC[ pid] ? OK, 0, caller;
         done = 0;
         do
         :: ( !done) -> {
              if // Nondeterministically assign a value to done
              :: skip -> done = 0
              :: skip -> done = 1
              fi;
              replyC[pid] ! OK, 0, caller, done
            }
         :: else -> break
         od
       }
    od
  }
}
```

Figure 3.9: Abstract Spin model (Part I). It was generated from the ESP program in Figure 3.7 using the abstraction in Figure 3.8

```
proctype transfer( int pid) {
  bool last;
  atomic {
    do
    :: 1 -> {
         reqC[pid] ! OK, 0, 1;
         last = 0;
         do
         :: ( !last) -> replyC[pid] ? OK, 0, 1, last
         :: else -> break
         od
       }
    od
  }
}
```

Figure 3.10: Abstract Spin model (Part II). It was generated from the ESP program in Figure 3.7 using the abstraction in Figure 3.8

```
replace type reqT = record of { caller: int}
replace type replyT = record of { caller: int}
drop pageTable $table, $vaddr, $size, $paddr, $chunk, $done
drop transfer $vaddr, $size, $paddr, $chunk, $last
```

Figure 3.11: Abstraction 2

Since the compiler makes conservative approximations when generating abstract models, the model checker will not miss a deadlock because of a programmer error in specifying an abstraction. However, a programmer error can cause a spurious deadlock to be flagged. For instance, the abstraction in Figure 3.11 results in a spurious deadlock because the programmer dropped the variable `done` by mistake. Consequently, any bugs detected by the model checker have to be double checked by the programmer.

Finally, we can introduce a deadlock in the program by replacing the line

```
$done: bool = false;
```

by the line

```
$done: bool = ( size == 0);
```

This will cause a deadlock if the `size` specified on channel `reqC` is 0. The model checker will find the bug using either of the two abstractions.

3.4 Case Study: VMMC Firmware

The earlier C implementation of VMMC firmware contains bugs even after several man-years have been spent on developing and debugging it. Therefore, the VMMC firmware was reimplemented using ESP (Section 2.4). The Spin model checker was used extensively to develop and test the new VMMC firmware.

The model checker was used 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.

Description	ESP Program	Abstraction Specification	Generated Model	Type of Model	Test Code
Retransmission Protocol	197	-	454	Detailed	73
Safety of `reliableSend`	230	-	1198	Safety	45
Safety of `reliableRecv`	152	-	664	Safety	41
Safety of `localReq`	172	-	742	Safety	67
Safety of `remoteReq`	167	-	882	Safety	85
Safety of `remoteReply`	177	-	715	Safety	104
Absence of Deadlocks	453	108	2202	Abstract	128

Table 3.1: Sizes (in lines) of the various files used to debug the VMMC firmware. The second column shows the size of the portion of program relevant for the particular model. The third column shows the number of lines required to specify the abstraction. The fourth column shows the size of the model generated by the ESP compiler. The fifth column shows the type of the model generated. The last column shows the number of lines of Spin test code that was required.

As described earlier, Spin test code (test[1-N].SPIN in figure 1.6) has to be written to check for different properties. The code not only specifies the property to be verified but also simulates external events such as network message arrival. This size of the test code is usually fairly small. Abstract models further simplify the task of writing the test code because they require smaller data structures to be sent over channels. Table 3.1 presents the sizes of the test code and the abstraction specifications that had to be written to debug VMMC firmware. It also shows the size of the models generated by the ESP compiler.

Each test code has to be written only once but can be used repeatedly to recheck the system as the software evolves. Since the models are extracted automatically, rechecking the software requires little programmer effort.

The Spin model checker was very useful in developing and testing the VMMC firmware. The rest of this section describes some of the situations where the model checker was effectively used and summarizes our experience with using the model checker.

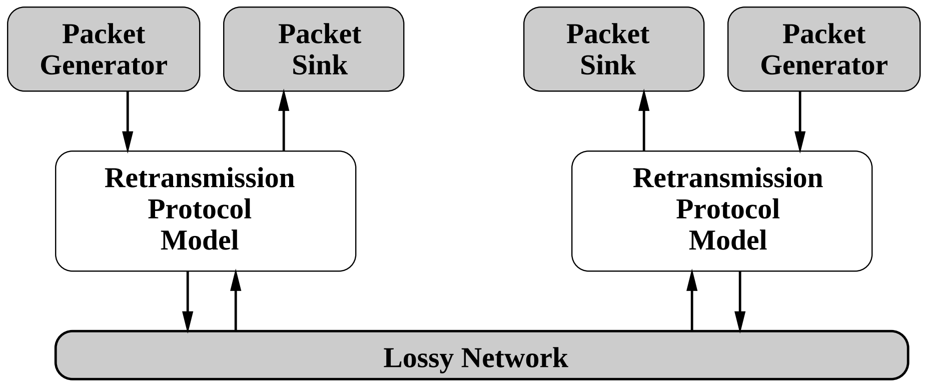

Figure 3.12: Setup used to debug the retransmission protocol. The shaded regions represent Spin code provided by the programmer.

3.4.1 Implementing Retransmission Protocol

VMMC firmware implements a simple well-known retransmission protocol to reliably deliver packets even when packets are dropped by the networking hardware. The protocol implemented was a sliding-window protocol with piggyback acknowledgement. The code that implemented the retransmission protocol was developed and debugged entirely using the Spin model checker before being ported to run on the network interface card.

The retransmission subsystem (which involved two processes) was developed separately. The ESP compiler extracted a detailed model that was used for debugging. The setup is shown in Figure 3.12. Two instances of the model were used to mimic two machines communicating over the network. The programmer provided test code to drive the model. First, for each instance of the model, a process (packet generator) constantly generates packets that have to be reliably sent over the network. A separate process (packet sink) accepts all network packets that it is given. The packet generator process includes a sequence number in every packet. The packet sink process checks to see that all messages sent are received exactly once and in the correct order by examining the sequence numbers. Second, a pair of processes implements a lossy network. To simulate lost packet, the network process nondeterministically drops a packet.

Process Name	No. of States		Time	Memory Used
	Stored	Matched	(in sec)	(in MBytes)
`reliableSend`	11118	316725	67.6	34.45
`reliableRecv`	124	220	0.1	25.13
`localReq`	167	61	0.1	25.30
`remoteReq`	2315	3510	0.9	26.87
`remoteReply`	8565	7312	2.3	30.55

Table 3.2: Checking for memory safety in the VMMC firmware using Spin. In each case, the entire state space was explored in the exhaustive mode in Spin. The *stored* column shows the number of unique states encountered while the *matched* column shows the number of states encountered that had already been visited before.

The Spin simulator was used to debug the retransmission protocol. As explained earlier (Section 3.3.1), the simulator is fairly effective in finding bugs. The new implementation took two days to be written, debugged, and then ported to run successfully on the card. The earlier implementation (that used C) had taken over ten days to develop and debug. This demonstrates that ESP greatly simplifies the task of programming the firmware compared to C. The comparison of the effort required in the two implementations is fair because the protocol implemented was a standard one that is described in textbooks [80]. Two different graduate students performed the two implementations. This discounts the possibility that the second implementation was faster because of the experience gained during the first implementation.

3.4.2 Verifying Memory Safety

ESP takes a novel approach to providing memory safety (Section 2.3.5). Instead of supporting safety through garbage collection, ESP supports an explicit `malloc/free`-style interface to support dynamic memory management. Although this interface is unsafe, model checking can be used to verify memory safety. To allow this, ESP is designed so that the memory safety is a local property of each process. This allows memory safety of each process to be verified separately without running into state-space explosion.

The ESP compiler generates memory-safety models that include code to check the validity of memory accesses. These models were used to verify memory safety of each of the processes in the VMMC firmware. Since each process can be checked separately, the models were small enough to be exhaustively explored using the Spin model checker. The ESP model generated catches not only bugs due to invalid memory accesses but also most of the memory leaks.

Table 3.2 presents the amount of state that had to be explored to check for memory safety in each of the ESP processes in VMMC firmware.[1] For every process, the entire state space could be explored using exhaustive search mode in Spin. The biggest process (`reliableSend`) required 67.6 seconds of processor time and 34.45 Mbytes of memory.

The memory safety bugs in the VMMC firmware had already been eliminated by the time the ESP compiler was modified to support the memory-safety models. Spin was used to check an earlier version of the firmware that had an allocation bug. The verifier easily identified the bug. To further check the effectiveness of using the memory-safety models, a variety of memory allocation bugs were inserted manually in the program. These bugs either access objects after they were freed or use an invalid array index or introduce memory leaks. Spin was able to quickly find the bug in every case.

3.4.3 Checking for Absence of Deadlocks

System-wide deadlocks are often a result of complex interactions in the program and can be difficult for programmers to find. Therefore, the use of model checking to find these bugs is important. We used an abstract model (section 1.4.2) to check for deadlocks in the firmware

[1]The processes not listed in the table did not involve any dynamic allocation.

because state-space explosion made it difficult to use detailed models. However, even with the abstract model, an exhaustive search was not possible, so only partial searches were performed.

Even with the partial search, Spin found seven bugs in the firmware. The first bug was due to a circular dependency involving 3 processes that resulted in a deadlock. Once identified, the deadlock was avoided by eliminating the cycle.

The second bug involved a situation when the sliding window in the retransmission protocol was full and, therefore, not accepting any new messages to be sent to the network. This eventually led to no new data packets being accepted from the network. Since incoming messages were delivered in FIFO order, an explicit acknowledgement message that could unlock the system was trapped behind a data packet resulting in a deadlock. To fix this problem, packets have to be dropped occasionally to allow the explicit acknowledgements to get through.

Two other bugs uncovered were similar to the bug that was discussed in the example in Section 3.3.5. They would result in deadlocks if the application requested a zero-byte data transfer.

Model checking allows bugs to be uncovered early in the debugging process. This is highlighted by the fact that several bugs found by Spin would not have been discovered using conventional testing as long as our VMMC implementation was used on all the machines in the network. These bugs could only be triggered when the firmware was used to communicate with other VMMC implementations that were either malicious or buggy. Since the VMMC architecture requires the firmware on other machines to be untrusted, these are bugs that have to be fixed.

The remaining bugs discovered involved receiving unexpected messages or not receiving expected messages. The first bug involved receiving acknowledgments with invalid acknowledgement numbers. This was fixed by first checking for the validity of

Spin Search Mode	Limiting Resource	No. of States		Time (hr:min:sec)	Memory Used (in MBytes)
		Stored	Matched		
Exhaustive	Memory	117351	265492	0:01:24	268.35
Bit-state Hashing	CPU Time	22574700	77165900	3:57:30	167.92

Table 3.3: Checking for the absence of deadlocks in the VMMC firmware using Spin. In both cases, the state-space exploration could not be completed because of resource constraints. The *stored* column shows the number of unique states encountered while the *matched* column shows the number of states encountered that had already been visited before.

the acknowledgement numbers before using them. The second bug involved receiving an unexpected import reply message. These messages are usually received in response to an import request. An unexpected reply would deadlock the system. The problem was fixed by adding code that discarded these unexpected messages. The final bug involved not receiving a reply to an import request. We had been aware of this bug but had not fixed it yet. Including a timeout will eliminate this bug.

Further state-space exploration using Spin did not uncover any more bugs. As stated earlier in this section, resource constraints prevented Spin from exploring the entire state space. Table 3.3 presents the amount of state space that could be explored using the resources available. In the exhaustive mode, Spin had to abort the search after 84 seconds because it ran out of memory. In the bit-state hashing mode, Spin ran for 3 hours and 57 minutes before the search was terminated by the user.

3.4.4 Discussion

Partial searches are fairly effective in finding bugs in the concurrent programs. This is because the state machine being explored is usually much larger than necessary; each state of the minimal state machine is represented multiple times. Techniques like abstraction and optimizations like partial-order reduction [54] try to eliminate some of this redundancy.

However, significant redundancy remains because the size of the state space is exponential in the size of the model. For instance, a variable `i` whose value ranges from one to ten but has no bearing on the property can result in each state of the minimal state machine being explored ten times. Even a partial search that explores a small fraction of the state space can cover a significant fraction of the minimal state machine.

The objectIds are a source of unnecessary increase in state space to be explored in models generated by the ESP compiler. The problem stems from the fact that a given object in the program can get assigned different objectIds depending on the scheduling decisions made prior to its allocation. The result is that a single "state" manifests itself as several different states in the state space. This problem can be easily solved by adding a new feature to the Spin model checker. Spin would have to allow some variables in the model to be marked as *store-only*. These variables would be a part of each stored state but would not be used when comparing two states for equality. Then, the objectIds could be marked as store-only. In the absence of the store-only feature in Spin, several optimizations can be made to alleviate the problem. First, objectIds are necessary for all objects only in memory-safety models. In detailed and abstract models, the objectIds are necessary only for the mutable objects. In these models, the ESP compiler can choose to not assign objectIds to immutable objects. Second, a separate objectId table can be used for each type in each process, since two pointers can point to the same object only if they have the same type and belong to the same process. This will reduce the number of different objectIds a given object can get assigned.

The model checker is effective in catching subtle bugs from race conditions. The implementation of VMMC firmware in ESP was designed to avoid the bugs encountered in the earlier implementation in C. In addition, the ESP language allowed the complex interactions in the system to be implemented concisely ($<$ 500 lines). Therefore, it was surprising when the model checker uncovered several bugs that could deadlock the system.

This highlights the limitations of careful code inspection by the programmer and the benefits of using tools like model checker that explore the various possible scheduling scenarios systematically.

The model checker catches bugs early in debugging process. In the earlier VMMC firmware implementation, we encountered new bugs every time we tried a different class of applications or ran it on a bigger cluster. The Spin model checker caught several bugs in the VMMC firmware implementation that would not have been detected by traditional debugging methods until much later (Section 3.4.3).

No bugs were encountered in the new version of the VMMC firmware while running the applications. In contrast, one of the applications still does not run to completion using an earlier version of the VMMC firmware implemented using C.

3.5 Future Work

One interesting direction for future research is to explore the tradeoff of using a radically different model checker like Verisoft [48] (Section 3.2.1). Verisoft does not require a model to be provided. Instead it takes a concurrent program and explores its state space by manipulating the scheduler. It can be used to model check programs written in any language including C. However, since it treats the concurrent program as a black box, it cannot perform certain optimizations that are performed by model checker like Spin. As a result, the size of the state space it can explore is smaller.

The use of Verisoft would allow more freedom to the language designer. Since a model is no longer necessary, the design of the language would no longer be constrained by the specification language of the model checker. Even when a model is not needed, the compiler can aid the model checker in a number of ways. In addition to generating an executable that is optimized for efficient execution, it can generate other executables that allow the model checker to more effectively explore the state space. It can also use techniques described in this chapter to generate executables that check for memory safety or use abstraction. This is different from the way Verisoft is currently used—the same executable is used for model checking as well as for regular execution.

3.6 Summary

The Spin model checker can be used to develop and extensively test ESP programs. This satisfies the second goal that ESP was designed to meet.

The use of model checker was greatly simplified because the models were automatically extracted from the ESP programs by the ESP compiler. Automatic model extraction not only increases our confidence that the model accurately reflects the program but also allows the program to be rechecked with little effort whenever changes are made to it.

ESP uses the model checker throughout the development process. Traditionally, model checkers are used to find the hard to find bugs from a mostly-debugged system. However, the ESP programs are often developed and debugged entirely using the model checker. Once debugged, they are run on the devices. This approach avoids the slow and painstaking process of debugging the firmware by running it on the device.

VMMC firmware is used as a case study to evaluate the effectiveness of using a model checker to develop and extensively test device firmware. The new implementation of the VMMC firmware in ESP used the Spin model checker. Our experience is summarized below.

The ESP compiler extracts three types of models. They are:

Detailed. A detailed model retains all the details in the original ESP program. It is used during the early debugging stages using the simulator mode in the Spin model checker. It can also be used to exhaustively check small subsystems for bugs.

A detailed model was used to develop and debug the retransmission protocol in the VMMC firmware using the Spin simulator. It was developed entirely using Spin before it was ported to run on the network card. The entire development process took 2 days. This is significantly better that the earlier implementation in C which took around 10 days.

Memory Safety. A memory-safety model allows the Spin model checker to verify the memory allocation correctness of the ESP program. The novel design of ESP allows the model checker to check each process independently to ensure the memory safety of the entire program. Checking the various processes one at a time makes the model less vulnerable to state-space explosion.

The Spin model checker was able to perform an exhaustive exploration of the state space of each of the processes in the VMMC firmware. It identified a bug in an earlier version of firmware. It also found all the memory allocation bugs—like double freeing, accessing an object after it has been freed and memory leaks—that were deliberately inserted into the firmware.

Abstract. The abstract models generated are more compact and tractable because they omit some details in the ESP program that are irrelevant to the property being verified. The compiler uses programmer annotations to generate conservative abstract models. The models are conservative in that they retain all the bugs in the original ESP programs. However, they might generate false positives.

Abstraction was essential for obtaining models that could be used for identifying bugs that result from violation of system-wide properties. State-space explosion prevented the use of detailed models to check for system-wide properties like absence of deadlocks in the VMMC firmware [64]. The abstract models were used successfully to identify seven bugs in the firmware that could cause it to deadlock.

State-space explosion is a constant problem when using model checkers. Often only a partial state-space exploration is possible due to resource constraints. In these cases, it is impossible to be sure that all the bugs in the system have been identified. However, even a partial systematic search by the model checker results in more extensive testing than traditional testing methods and can be invaluable in debugging concurrent systems.

Chapter 4

Generating Efficient Code

Good performance is crucial to device firmware—the performance of the device firmware determines the fraction of the hardware performance that can be delivered to the applications.

Device firmware can achieve good performance by using event-driven state machines in C but requires significant programming effort (Section 1.3). Usually, function pointers are used to encode state machines in C. This makes it difficult for C compilers to effectively optimize these programs. As a result, the programmer is forced to manually perform some of these optimizations. This not only requires significant programmer effort but also introduces bugs into the programs.

ESP compiler can compile the ESP programs to generate efficient code. ESP provides a number of language features to simplify event-driven state-machines programming (Chapter 2). However, these features are designed so that the compiler can effectively compile them. This frees the programmer from having to manually optimize the programs.

VMMC firmware is used as a case study to measure the performance impact of using ESP. Microbenchmarks as well as applications are used to compare the performance of the two implementations of the firmware: the new implementation that uses ESP, and the earlier implementation that used C.

The rest of the chapter is organized as follows. Section 4.1 presents existing techniques used to compile a concurrent program to run efficiently on a single processor. Section 4.2 discusses the related work. Section 4.3 describes the techniques used by the ESP compiler to generate efficient code. Section 4.4 measures the performance impact of using ESP instead of C to implement the VMMC firmware. Section 4.5 discusses compiler optimizations that we intend to explore in the future to further improve the performance of the ESP programs. Section 4.6 summarizes the chapter.

4.1 Background

There are two main approaches to compiling a concurrent program to run efficiently on a single processor: *automata-based* and *process-based* approach. The automata-based approach [24, 14, 31, 84] essentially treats each process in the concurrent program as a state machine and combines all the state machines in the program to generate a single global state machine. The global state machine does not contain any concurrency and can be translated directly into sequential machine code. The advantage of this approach is that all the concurrency is compiled away and the program incurs no runtime overhead to support concurrency. The code generated is extremely fast. However, the global state machine generated can be, in the worst-case, exponential in the size of the individual state machines. Some optimization techniques [31, 25] alleviate the code blowup problem by identifying and eliminating some of the duplicated code. Still, the code blowup remains exponential in the worst-case.

The process-based approach [82, 45] generates the code for the different processes separately and dynamically context switches between them. Since these processes are essentially state machines, only a small amount of state (just the program counter register) needs to be saved and restored during a context switch. The stack and the other registers are used only temporarily and do not have any useful state that needs to be saved before a context switch. Although the process-based approach has a runtime overhead, the overhead is fairly low.

4.2 Related Work

A number of concurrent languages have compilers that compile a concurrent program to run efficiently on a single processor.

Esterel [14] is a synchronous language designed to model the control of concurrent systems (Section 2.2). Earlier Esterel compilers [14, 31] used the automata-based approach to generate code. More recently, gate-based compilers [13] have been implemented. They avoid the code blowup that occurs using the automata-based compiler but incur a runtime overhead. The gate-based compilers translate[1] an Esterel program into a synchronous circuit and then generate code from the circuit. However, the translation of an efficient synchronous hardware circuit into efficient software is nontrivial and involves runtime overheads [45]. Process-based compilers [45] have also been implemented for Esterel. However, they can handle only a subset of valid Esterel programs—those in which a valid schedule for the concurrent Esterel program can be determined statically.

Edwards *et. al.* [45] evaluates the tradeoff of using each of the three approaches—automata-based approach, gate-based approach, and process-based approach—for compiling Esterel programs. As expected, the automata-based compiler [14] generates the fastest code but the size of the executable can be 2–3 orders of magnitude larger than the other

[1]The gate-based compilation technique applies to synchronous languages like Esterel and is not applicable to ESP.

approaches. The gate-based compiler [13] generates fairly compact code but can be 4–100 times slower than the automata-based compiler. The process-based approach generates code that is only twice as slow as the automata-based approach but yields the smallest executables.

Newsqueak [81] (Section 2.2) supports processes and synchronous channels and uses a process-based approach [82] to generate sequential code. Some of the techniques used in the implementation are similar to those used in ESP. However, context switches and rendezvous are more expensive operations in Newsqueak.

Squeak [24](Section 2.2) uses the automata-based approach to generate sequential code. It considers all possible interleavings of the concurrent program. At each stage, one of the unblocked processes is executed for one step. A random number generator is used to select a process when multiple processes are ready for execution.[2]

Filter Fusion [84] uses the automata-based approach to fuse filters. A concurrent program is expressed as a sequence of filters where only adjacent filters communicate with each other. A sequential program is obtained by successively fusing pairs of adjacent filters into a single filter using a technique similar to that used in Esterel compilers [14].

Integrated Layer Processing (ILP) [33] is an implementation technique for improving the performance of layered network protocols. The protocol is implemented as a sequence of layers in which each layer manipulates the data in the packet and hands it to the next layer. ILP reduces the number of data accesses by combining the packet manipulation loops of different layers into one or two integrated processing loops [1, 22]. ILP is appropriate for layers that manipulate the data portion of the large packets (like checksum computation and encryption). However, performing operations that need to examine entire packets are too computationally expensive to be performed on the slow processor on the devices.

[2]In contrast, Esterel programs are deterministic—all possible schedules yield the same result. Therefore, it does not require a random selection at each stage.

These operations are usually performed either on the host processor or by special-purpose hardware engines on the devices.

4.3 Generating Optimized Code

The ESP compiler compiles an ESP program into optimized C code (Figure 1.6). It uses the process-based approach to generate a sequential code from a concurrent program. To perform whole-program analysis, the compiler requires the entire program to be available during compilation.

The ESP compiler uses C as the back-end language. It compiles an ESP program into one large C function which looks like an assembly program. Each statement in the function performs simple operations like three-operand arithmetic operation or a transfer of control to a different part of the function using a `goto` statement.

Using C as the back-end language has several advantages. First, it makes the ESP compiler portable to different devices with little effort since most device vendors provide a C compiler for their device. Second, the ESP compiler can rely on the C compiler to do register allocation and benefit from optimizations performed by it.

4.3.1 Compilation Stages

The ESP compiler (Figure 1.7) was implemented using SML/NJ. The different modules that are involved in code generation are as follows:

Scanner & Parser. This module reads in an ESP program and builds an Abstract Syntax Tree (AST).

Type Checker. The type checker traverses the AST and checks for type correctness in the program. Since ESP allows types to be unspecified when they can be inferred, the type checker also performs simple type inferencing on a per-statement basis.

Convert to IR. The high-level AST is converted into a low-level intermediate representation (IR) that is more suitable to performing optimizations.

The channels (in the AST) are translated into ports (in the IR). A channel can have multiple readers and multiple writers while a port can have multiple writers but only a single reader. In ESP, each reader on a channel has to use a separate disjoint pattern to read from it (Section 2.3). Therefore, each channel represents a group of ports where each port is identified by the pattern being used by the reader.

All patterns are eliminated during this translation of the channels into ports. The sender on a channel matches the data to be sent against all the possible patterns and sends the values to the appropriate port. This eliminates the unnecessary allocation associated with pattern matching. For instance, if a sender allocated a record with two fields to send it over a channel and the receiver uses a pattern to receive the two fields, the ESP compiler avoids the allocation by sending unboxed values over the channel.

Each operation (`in` and `out`) on a channel is broken down into 2 operations in the IR. The `syncOp` operation synchronizes the sender and receiver. The `transferOp` operation transfers the data from the sender to the receiver. Data to be sent is computed after the `syncOp` operation. This avoids unnecessary computation when a process is waiting on multiple channels using the `alt` statement. 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.

High-level control constructs like `while` loops etc. are translated into low-level control constructs like `goto`. All high-level data constructs like `record` and `array` are translated into a uniform low-level object representation. However, the processes in the ESP program are kept separate, so the IR consists of a set of processes. Each process has simplified control and data constructs and communicates with other processes by sending data on the ports.

Optimizers. The optimizers perform standard optimizations like constant folding, copy propagation and dead-code elimination on a per-process basis. Each optimizer expects an IR as an input and generates an optimized IR, so the various optimizers can be applied a number of times in sequence.

Although most C compilers also perform these optimizations, the ESP compiler cannot rely on the C compiler to effectively perform the optimizations on the generated C code. This is because all the processes are combined into a single C function during code generation. The semantic information lost during code generation makes it hard for the C compiler to perform these optimizations effectively.

Code Generation. The ESP compiler uses the process-based approach (Section 4.1) to combine all the processes into a single C function. It uses the process-based approach to avoid the exponential code blowup associated with automata-based compilers (Section 4.2). In addition, unlike Esterel, all legal ESP programs can be compiled using the process-based approach.

4.3.2 Scheduling

The run-time system performs nonpreemptive scheduling; context switches are performed during blocking operations (reading from and writing to channels). The runtime system maintains a *ready list* of processes that are ready to execute. When there are no ready processes, it executes an idle loop. The idle loop polls for messages on external channels on which processes are blocked. When a message becomes available, it unblocks the corresponding process and restarts it by jumping to the location where it had blocked. The process then executes till it reaches a channel operation. At this point, it has to synchronize with another process to complete the channel operation before it can continue. If more than one choice is available, the scheduler picks one of those processes randomly and performs the channel operation. At this point, both the synchronizing processes are ready to execute. This scheduling policy picks one of these two processes to continue execution and inserts the other one into the ready list. When an executing process blocks, the next process in the ready list is chosen to execute. This is repeated till there are no processes left in the ready queue. Then the execution returns to the idle loop.

To avoid starvation, ESP uses a simple FIFO scheduling policy. The only distinguishing feature is that after a synchronization operation, it always chooses the process receiving on the channel to continue. The sending process is inserted into the ready list. At first glance, this might appear to introduce starvation. For instance, two processes can repeatedly send messages to each messages to each other could starve other processes out. However, this cannot happen because after a synchronization operation, the sending process will be queued behind other ready processes.

External channels require additional care to avoid introducing starvation. First, for internal channels, the runtime system maintains the invariant that each port can have either reader or writers but not both blocked on it. If a writer arrives at a synchronization point and finds a reader waiting on the port, the writer can deduce that there are no other writers waiting on that port. Hence, the writer does not have to check for other writers before synchronizing with the reader. However, it is difficult to maintain this invariant for the external channels since an external event can cause the external end of the port to become ready for synchronization at any instant. Therefore, an additional check has to be performed to ensure fairness on external channels.

Second, a message on an external channel could get ignored for long periods of time. New external messages are detected at two locations. A running process checks for the availability of new messages on channels; if none are available, it blocks on the channel. Subsequently, when the control reaches the idle loop, the idle loop checks for new messages on each of the external channels. The problem with this is that if one or more processes are continuously receiving external messages, the control will not return to the idle loop. As a result, other processes that blocked on other external channels do not get restarted even when new messages are available on them. To avoid this problem, the ESP scheduler periodically returns the control to the idle loop even if the ready queue is not empty.

4.3.3 Memory Management

From the programmer's perspective, each process has its own set of objects that have to be managed separately for each process (Section 2.3). Each process allocates its objects and frees them (using `free` and `rfree`). Objects sent over the channels are deep copied before being handed to the receiving processes. Therefore, objects arriving over a channel are treated like newly allocated objects that have to be later freed by that process.

The implementation uses a reference-counting scheme to manage the objects. Although semantically, processes do not share objects, the implementation shares objects between processes for efficiency—copying objects is computationally expensive. The runtime system maintains reference counts to keep track of the number of processes sharing the object. Recursive increment and decrement operations on cyclic data structures require additional bookkeeping to avoid infinite loops. However, ESP does not allow cyclic data structures, and allows these operations to be implemented efficiently.

Normal allocation causes objects to be allocated and their reference count initialized to one. When an object is sent over a channel, the reference count of the object is recursively incremented (thereby avoiding the expensive deep copy operation) before giving it to the receiving process. When a process frees an object, its reference count is decremented by one. The object is actually deallocated only when all the processes have freed it and the reference count is zero.

The deep copy performed when a data structure is sent over a channel does not preserve pointer sharing (Section 2.3.5). This has two benefits. First, it allows the copying semantics to be implemented efficiently; a simple recursive increment of reference count suffices. An object that is pointed to multiple times within the data structure will have its reference count incremented multiple times. Second, it allows the correctness of the memory allocation to be a local property of each process. If pointer sharing were preserved, the receiving process would need to know about the sharing to check that the data structure was correctly freed. However, it cannot determine the sharing because pointer comparisons are not allowed in ESP. The example in Figure 4.1 illustrates the problem with copying semantics that preserves pointer sharing.

A cast of an immutable object into a mutable object can require copying the object. This is because a program can detect object sharing by mutating it at one location and observing the change at another location. However, the cast operation is fairly uncommon in ESP programs. In addition, the copying is not always necessary. The copy can be often avoided when a cast is necessary but the program is written carefully (to allow the compiler to optimize it). For instance, if the reference count of the immutable object is one (no other process is holding that object) and the object is freed immediately after the cast, the compiler can avoid the copy and use the same object.

Several design choices in the ESP language allow the implementation to share objects while providing the illusion of the disjoint set of objects. First, only immutable objects can be sent over channels. Therefore, the program cannot detect that the object is being shared by mutating it in one process and observing the change in another process. Second, objects

```
type entryT = record of { v: int}
channel shareC: array of entryT


process process1 {
  in( shareC, $a);
  assert( length(a) == 2);
  free( a[1]);
  $b = a[0];
}


process process2 {
  $p1: entryT = { 11 };
  $p2: entryT = { 13 };
  out( shareC, { -> p1, p2});
}


process process3 {
  $p: entryT = { 5 };
  out( shareC, { -> p, p});
}
```

Figure 4.1: An example to illustrate the problems with copying semantics that preserves pointer sharing. Process `process1` expects an array of two elements on the channel shareC. Once it receives it, the process frees one of the entries and proceeds to use the other entry. If process `process2` sends an array on the channel, `process1` would execute correctly because the two entries point to different objects. However, if process `process3` sends an array on the channel, `process1` will try to access the record after it has freed it resulting in an error.

```
type entryT = record of { v: int}
channel countC: array of entryT


process processA {
  $p1: entryT = { 2 };
  $p2: entryT = { 3 };
  $a = { -> p1, p2};
  out( countC, a);
  $v1 = p1.v;
}


process processB {
  in( countC, $b);
  free( b[0]);
}
```

Figure 4.2: An example that shows that the traditional reference counting scheme is not sufficient for ESP.

cannot be compared for pointer equality. This prevents the program from comparing the pointer for two different objects and detecting that the implementing is using the same object to represent both of them. Finally, ESP does not support recursive data types and therefore the program cannot have cyclic data structures. This means that recursive reference count increments does not have to deal with infinite loops due to cyclic data structures and can therefore be implemented cheaply.

Traditional reference counting schemes maintain the counts on the objects differently from the one used by ESP. In the traditional scheme, the reference counts are incremented only at the root and decremented recursively only when the reference count of the object becomes zero. Our earlier paper [64] suggested that this would be sufficient for ESP too. It turns out that this is not sufficient. Consider the example in Figure 4.2. Till the point when the objects are sent over the channel `countC`, both schemes would have kept the same reference counts on all the objects; each of the three objects (pointed to by the variables `p1`, `p2`, and `a`) would have a reference count of one. However, on performing the send operation on the channel `countC`, the traditional scheme will increment the reference count of only the array object while the ESP scheme will increment the reference count of each of the three objects. If the scheduler chooses to schedule the process `processB` first, then the `free` statement will be executed. With the traditional scheme, this will cause the reference count of the object pointed to by `p1` to go to zero, thereby freeing the object. This will generate an error when the process `processA` is scheduled to run and it tries to access the variable `p1`. With the ESP scheme, the reference count of the object pointed to by `p1` will be decremented from two to one, and so the object will not be freed. This allows the process `processA` to later access it.

4.4 Case Study: VMMC Firmware

In this section, we compare the performance of the earlier VMMC implementation (*vmmcOrig*) with the performance of the new implementation using ESP (*vmmcESP*). Since ESP does not currently support fast paths[3], we also present the performance of the earlier implementation with the fast paths commented out (*vmmcOrigNoFastPaths*). This allows us to separate the actual cost of using ESP (the difference between *vmmcESP* and *vmmcOrigNoFastPaths*) from the benefit of using fast paths (the difference between *vmmcOrig* and *vmmcOrigNoFastPaths*).

Microbenchmarks as well as applications are used to measure the three implementations of VMMC: *vmmcOrig*, *vmmcOrigNoFastPaths*, and *vmmcESP*. On one hand, the microbenchmarks measure specific aspects of the communication like latency and bandwidth by stressing them. This allows them to isolate and understand the cost of using ESP.

[3]Section 4.5 examines the problem of supporting fast paths in ESP.

However, they represent a worst-case scenario. On the other hand, the actual performance impact is one that is observed by the application. The applications usually exhibit more complex communication patterns than the microbenchmarks. They are also less sensitive to the firmware performance.

4.4.1 Microbenchmark Performance

Microbenchmarks. Three microbenchmarks are used to measure the performance of the performance of the three implementations of ESP. Each microbenchmark involves running it on two different machines that communicate with each other using VMMC.

The *Latency* microbenchmark measures the latency of sending a message of a particular size between two machines. This is measured using a simple pingpong program that sends a message back and forth between the two machines. The latency is computed as:

$$\frac{\text{RoundTripTime}}{2}$$

The *Bandwidth* microbenchmark measures the bandwidth that can be achieved between two machines when sending messages of a particular size. This is measured by using a program on one machine to continuously send messages of that size to a program on the second machine that is repeatedly receiving the messages. The bandwidth is computed as:

$$\frac{\text{NumMessagesSent} \times \text{MessageSize}}{\text{TimeElapsed}}$$

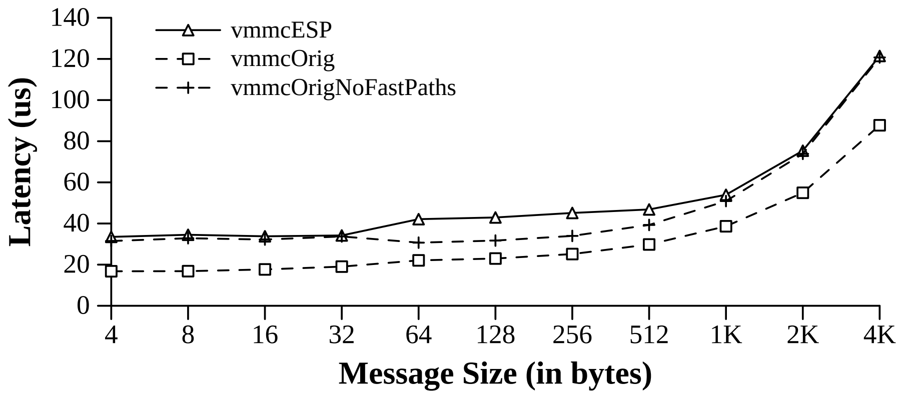

Figure 4.3: Latency microbenchmark

The *Bidirectional Bandwidth* microbenchmark measures the total bandwidth between two machines when both the machines are sending messages of a particular size simultaneously. The bidirectional bandwidth is computed as:

$$\frac{(\text{NumMessagesSent} + \text{NumMessagesReceived}) \times \text{MessageSize}}{\text{TimeElapsed}}$$

Platform. All microbenchmarks measurements use a pair of PCs. Each PC has a 300 MHz Pentium processor, 128 MB memory and a Myrinet network interface card with a LANai 4.x 33 MHz processor and 1 MB on-board SRAM memory. The nodes are directly connected to each other using a Myrinet cable. The PCs run Windows NT 4.0.

Performance. Figures 4.3, 4.4 and 4.5 present the microbenchmark performance.[4] In each case, the x-axis shows the message size.

The *Latency* microbenchmark (Figure 4.3) shows that *vmmcESP* is around twice as slow as *vmmcOrig* for 4 byte messages and 38 % slower for 4 Kbyte messages. However,

[4]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.

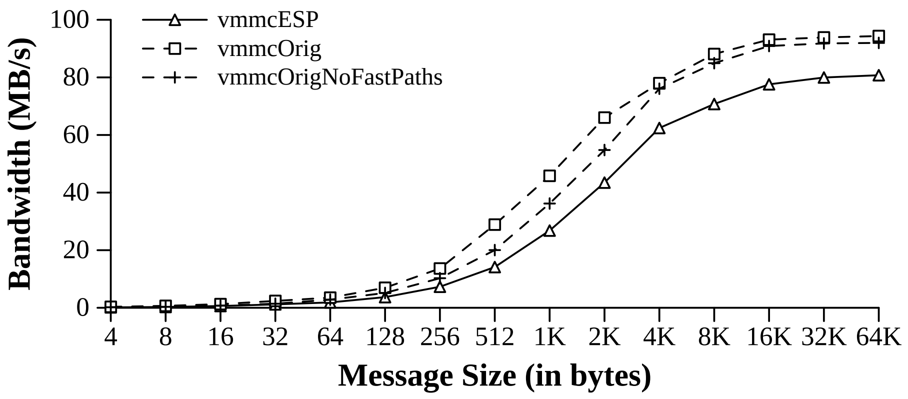

Figure 4.4: One-way bandwidth microbenchmark

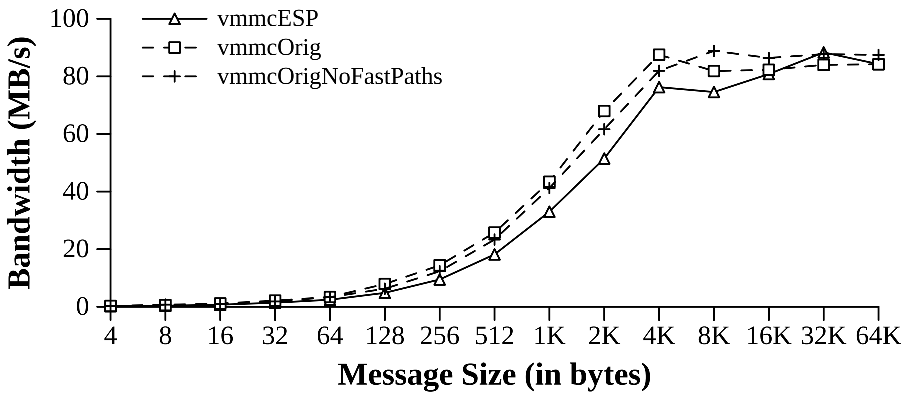

Figure 4.5: Bidirectional bandwidth microbenchmark

vmmcESP is only 35 % slower than *vmmcOrigNoFastPaths* in the worst case (for 64 byte messages) and has comparable performance for 4 byte and 4 Kbyte messages.

The *Bandwidth* microbenchmark (Figure 4.4) 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 *Bidirectional Bandwidth* microbenchmark (Figure 4.5) 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.

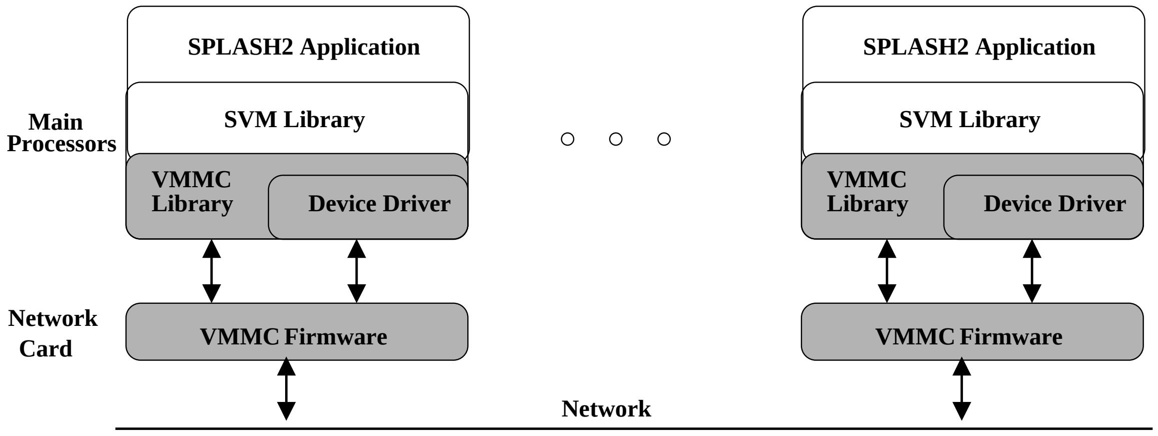

Figure 4.6: Experimental setup

Application	**Problem Size**	**Uniprocessor Execution Time (in seconds)**	**Base Speedups**
FFT	1 Million Points	3.97	2.95
LUContiguous	2048 x 2048 Matrix	139.48	9.53
WaterSpatial	15625 Molecules	118.09	5.62
WaterNsquared	1000 Molecules	14.39	4.04
BarnesSpatial	8192 Particles	103.42	13.59
Volrend	head	314.98	4.76

Table 4.1: SPLASH2 Applications

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 fast paths. Also, the fast paths are more effective when the communication pattern is simpler. 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 microbenchmarks, the firmware has to deal with only one type of message at a given instant.

4.4.2 Application Performance

Applications. Figure 4.6 shows the experimental setup used to run the applications. The SPLASH2 applications [103] run on a cluster of SMP nodes using the VMMC software to communicate. These applications run on top of the Shared Virtual Memory (SVM) [18] library that, in turn, runs on top of the VMMC library. The VMMC software architecture is discussed in Section 1.3.3.

The applications in the SPLASH2 suite [103] are parallel applications that use shared-address space to communicate with each other. The versions of these applications used in this study have been restructured to perform well on a cluster of loosely connected nodes [60]. The restructuring involved simple changes to decrease the amount of communication (by padding, aligning and reordering fields in the data structures) and synchronization (by algorithmic changes to reduce the amount of locking used). The SPLASH2 applications and the corresponding problem sizes used are listed in Table 4.1.

The Shared Virtual Memory (SVM) [18] library provides a shared-address space abstraction in software on a cluster whose nodes cannot access each other's physical memory directly. Since the VMMC implementation that used ESP (*vmmcESP*) currently implements only the VMMC interface described in [41], we use a version of the SVM library that uses only that VMMC interface.[5]

Platform. All applications measurements were made using a cluster of four SMP PCs. Each PC has four 200 MHz Pentium processors, 1 GB memory and a Myrinet network interface card with a LANai 4.x 33 MHz processor and 1 MB on-board SRAM memory. The nodes are connected by a Myrinet crossbar switch. The PCs run Windows NT 4.0.

[5]Some of the other extensions [18] to VMMC that were proposed to further improve the performance of SVM are currently not supported.

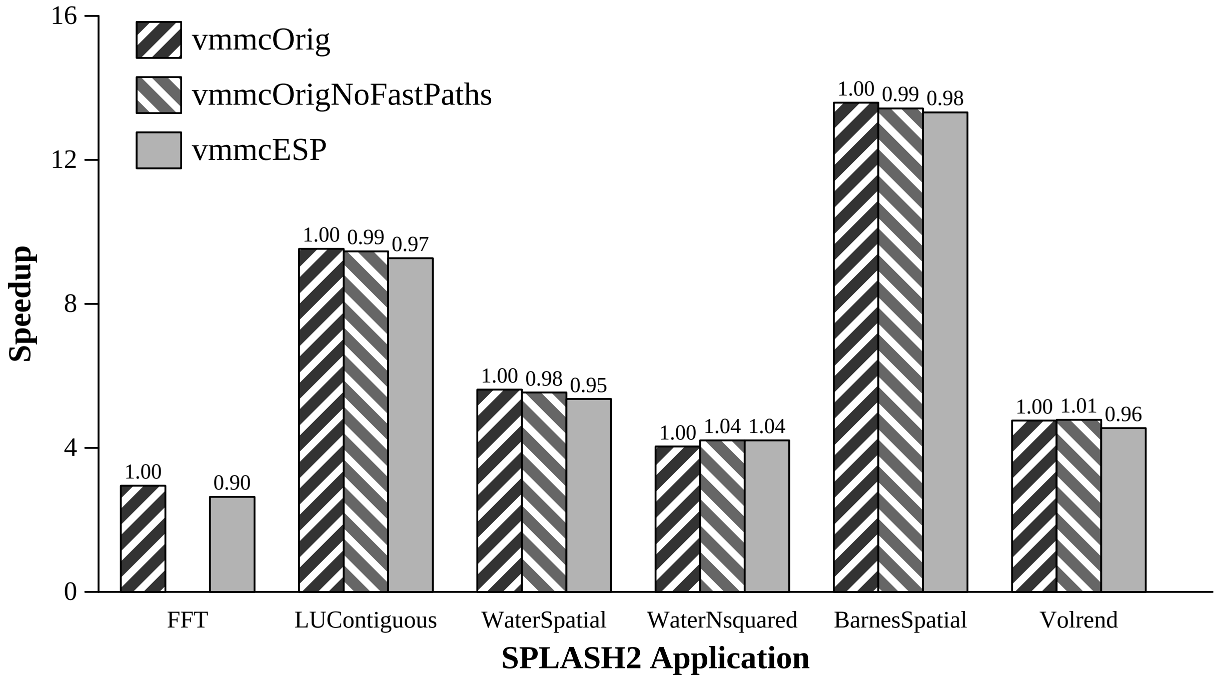

Figure 4.7: SPLASH2 Application Performance: Speedup on 16 processors (four 4-way SMP nodes). The number on the top of each bar shows the relative speedup compared to *vmmcOrig*.

Performance. The performance of the SPLASH2 applications is presented in Figure 4.7. The Y-axis shows the application speedup when running on 16 processors. As before, the performance of the applications is shown for each of the three versions of VMMC:[6] *vmmcOrig*, *vmmcOrigNoFastPaths*, and *vmmcESP*.

The performance hit of using ESP is the difference between the performance of the applications when using the *vmmcOrigNoFastPaths* and the *vmmcESP* versions. Figure 4.7 shows that the average performance difference[7] between the two versions is 3.5 %. Only two applications incur more than 5 % performance hit: FFT (10 %) and Volrend (5 %).

The performance benefit to the applications of the fast paths in *vmmcOrig* is the difference between the performance of the applications when using the *vmmcOrig* and the *vmmcOrigNoFastPaths* versions. Figure 4.7 shows that the fast paths have little impact on these applications. The largest benefit is observed in WaterSpatial (2 %).

4.4.3 Discussion

Applications incur a smaller performance hit compared to microbenchmarks for two reasons. First, 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 VMMC performance. Second, applications are often fairly insensitive to certain communication parameters like latency and bandwidth and more sensitive to other parameters like host overhead and interrupt costs [69, 19, 18]. The VMMC firmware does not affect the latter parameters as these are determined by code (OS, VMMC device driver and library) running on the host CPU.

[6]The FFT performance when using the *vmmcOrigNoFastPaths* version is not shown because a bug in that implementation prevents that application from running to completion

[7]For FFT, we use *vmmcOrig* to conservatively approximate *vmmcOrigNoFastPaths* performance.

Some applications can be made to run significantly faster by adding the right functionality on the network interface card. For instance, a similar set of SPLASH2 applications observed a 37% increase in performance when additional network support was added to VMMC to avoid asynchronous protocol processing in the SVM library [18]. In this respect, ESP can help improve applications' performance by making it easier to experiment with and to add new functionality to the firmware.

The fast paths implemented in *vmmcOrig* have a small impact on the application performance even though they have a significant benefit in the microbenchmarks for several reasons. The two reasons presented in the previous paragraph apply in this case too. In addition, fast paths are often fairly brittle. They are often designed to provide better performance in simple situations that are expected to occur very frequently. However, communication intensive applications cause high contention in the network card with complex communication patterns. As a result, the fast paths do not get taken very often in the applications. This is indicated by the measurements reported in an earlier study [18] that found the actual message latency measured when running the different applications varied between three times to ten times slower than the microbenchmarks measurements for small messages. Section 4.5.1 presents a general discussion of fast paths: how more robust fast paths can be built and how they can be supported in ESP.

4.5 Future Work

4.5.1 Fast paths

Fast paths provide better performance to commonly executing paths in the program (Figure 4.8). Modular programs are more reusable, readable, and maintainable. However, they

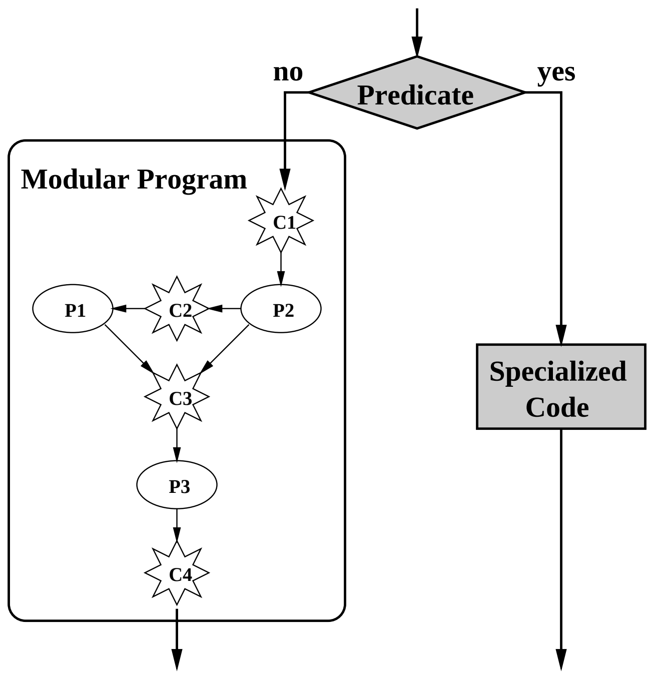

Figure 4.8: A fast path bypasses a modular program. Blocks representing the fast path components are shaded.

usually incur additional overhead when the program execution crosses a module boundary. Fast paths avoid this overhead in the common cases.

A fast path consists of two components: A predicate that identifies a common case, and specialized code that is optimized to efficiently handle the common case. The specialized code corresponds to the execution path in the modular program that would have been taken in the absence of the fast path. It is composed of the code fragments extracted from several different modules. This allows fast paths to avoid module-crossing overheads. This also makes them more amenable to compiler optimizations.

Languages like C do not provide support for fast paths. However, they allow fast paths to be implemented manually by the programmer—the programmer can insert a predicate in the program to check for the common case and transfer control to the specialized code. The programmer is responsible for providing specialized code that is equivalent to the corresponding path in the program. This violates modularity and makes the fast paths error prone.

ESP currently does not support fast paths. Since ESP enforces modularity, it prevents fast paths from being implementing manually by the programmer. To support fast paths in ESP, the following three questions need to be answered.

How to select fast paths? For fast paths to be effective, the fast paths have to be executed a significant percentage of time, i.e., the fast paths have to be common.

There are two main approaches to selecting the fast paths. First, the commonly executed paths can be identified by profiling the running program. Past research on path profiling [9, 65] has focused on sequential programs, and need to be extended to work for concurrent programs. The advantage of this approach is that fast paths are identified during the actual running of the program. The disadvantage is that some of the scheduling choices made (like the order in which the external events are processed) influence which paths are more common. This is especially true for a language like ESP because additional scheduling decisions are made to schedule the various processes in the concurrent program. The right scheduling policy can help the fast path be executed more often.

Second, since the programmer has some intuition on how the program behaves, the programmer can annotate the program to specify the fast paths and provide hints for the right scheduling policy. They also provide the programmer more control over program execution. Some languages [25, 12] provide limited annotation capability—they allow the conditional statements to be tagged to indicate whether it usually evaluates to true or false. The problem with program annotations is that it places additional burden on the programmer. In addition, the programmer intuition is often wrong.

The approach that is likely to work best for ESP is probably a combination of the two approaches. The programmer can specify the fast paths and provide hints on the right scheduling policy while program profiling can be used to check that the annotations were useful.

How to generate fast paths? Once a fast path is identified as a pair of predicate and path through the program, the ESP compiler has to extract the path. This is a challenging problem for several reasons.

First, it is hard to statically determine that the predicate specified always corresponds to the path specified. One option is to insert checks in the code to verify it at run time, and thereby, paying a runtime cost. The alternative is to trust the specification and risk introducing bugs in the program. A model checker can be used to check the specification in this case.

Second, the fast path involves code that is extracted from different processes and thereby makes some scheduling decisions. The ESP compiler has to ensure that it does not introduce starvation in the program.

A number of research projects [85, 75, 68] investigate the use of fast paths in sequential programs.

The Synthetix project [85] manually generated fast paths in the HP-UX operating system. They show that file system calls like read can be speeded up by generating specialized code for it based on invariants and quasi-invariants that are available when the file is opened.

The Scout operating system [75] makes paths an explicit abstraction mechanism to improve resource allocation and scheduling decisions. It uses compiler optimizations like outlining, cloning, and path inlining improve the performance of the fast paths [76]. However, since the paths are dynamically created, the compiler cannot always optimize these paths. To address this, the compiler generates optimized code for some paths. When a path is created at runtime, if the runtime system can determine that optimized code is available for that path, it uses the optimized code.

Formal methods can be used to build optimized fast paths [68] in the Ensemble network architecture [51]. A protocol stack consists of a sequence of protocol layers. The NuPRL [34] system was used to semiautomatically extract a fast path from the protocol stack. It ensures that the fast path generated is provably semantically equal to the protocol stack when the fast path predicate holds.

How to support robust fast paths? Our experience with fast paths in VMMC firmware shows that it is difficult to build robust fast paths. One often ends up with fast paths that help simple applications like the microbenchmarks but have little impact on applications. The more specific the fast path predicate, the more specialized and efficient fast path will be. However, it also means that the predicate will be satisfied less often. It is often difficult to identify the right point in the tradeoff.

One possible solution to this problem is to build a number of fast paths with increasing degrees of specialization. This would mean that the program could benefit from the more aggressive fast paths when the predicate holds but would still benefit from the less aggressive fast paths when they do not. Unlike when programming in C, building a number of fast paths in ESP would require only a small amount of programmer effort because of the compiler support for fast paths.

4.5.2 Compiler Optimizations

A number of compiler optimizations can further improve the performance of the generated code. First, the data-flow analysis can be extended to perform interprocess analysis—it currently analyzes each process separately. These optimizations should be fairly effective for ESP programs for two reasons: the channels in ESP are static (Section 2.3.2); processes communicate only using channels.

Second, the automata-based approach (Section 4.1) can be adapted to perform *process inlining* selectively. The automata-based approach combines all the processes into a single process, thereby avoiding process context switching overheads. However, it results in significant increase in code size. In contrast, process inlining would combine only some of the processes. For instance, when a process is used to implement a function (Section 2.3.3), process inlining would inline that process into each of the calling processes.

4.6 Summary

The ESP compiler compiles the ESP programs into code that incurs low performance overhead at runtime. This meets the third goal that ESP was designed to meet.

The ESP compiler compiles a concurrent ESP program to run efficiently on a single processor. It uses a process-based approach to compile the concurrent program to run on a single processor. A context switch is fairly lightweight and only involves saving and restoring the program counter. The language design allows the compiler to aggressively optimize the programs. The language is fairly static—all the processes and channels are know at compile time. This allows the compiler to implement pattern-matching over channels efficiently. Although messages sent over channels are passed by value, the runtime system uses a reference counting allocator to avoid actually copying the messages. Creating a copy of the messages on each message send operation would be inefficient.

VMMC firmware was used as a case study to measure the performance impact of using ESP instead of C to write device firmware. The performance of the new implementation that uses ESP is compared with the earlier implementation that used C. The performance is measured using both microbenchmarks as well as applications.

Microbenchmarks measurements show that the performance difference between the firmware implementation using ESP and the earlier implementation using C is usually small. In some cases, the difference is significantly high. For instance, the ESP implementation involves twice the latency of the C implementation when sending 4 byte messages. However, in this case, the entire difference is the result of the fast paths in the C implementation that is not currently supported by ESP. To obtain a fairer comparison, a version of the C implementation that does not include the fast paths is also used. The measurements show that the ESP implementation performs 0–35 % worse in the latency microbenchmark and 12–25 % worse in the bandwidth microbenchmark.

The performance impact of using ESP on applications is small. We measure the performance of SPLASH2 applications using the two firmware implementations. Our measurements show that the applications run 3.5 % slower on average (10 % in the worst case) when using the ESP version relative to the C version. They also show that the fast paths in the C implementation have little impact on the applications' performance.

Chapter 5

Conclusions and Future Directions

This thesis presents the design and implementation of ESP—a domain-specific language for programmable devices. ESP was designed to meet the following three goals:

Ease of Programming. ESP should make it easier to write device firmware using event-driven state machines. It should allow the firmware to be expressed concisely and in a modular fashion. It should minimize the use of C by allowing the bulk of the program to be written in ESP. It should support dispatch, dynamic memory management, and a flexible external interface to C.

Extensive Testing. Device firmware is trusted by the operating system and can write to any address in the physical memory. A bug in the firmware can compromise the integrity of the entire machine. As a result, the firmware needs to be extensively tested to ensure reliability. This is challenging for two reasons. First, the use of concurrency makes it difficult to find bugs resulting from race conditions. Second, very limited debugging support is available on these devices.

Low Performance Overhead. The effectiveness of the device firmware depends on the speed with which it can respond to events occurring on the devices. ESP should minimize the overhead incurred when using the language features it provides. In addition, it should allow aggressive compiler optimization to generate fast executables.

Event-driven state machines is C meets only one of these three goals listed above. Traditionally, event-driven state machines in C are used to develop firmware for these devices. For instance, the earlier implementation of VMMC firmware was programmed using event-driven state machines in C. Our experience with VMMC firmware showed that event-driven state machines in C meets only the performance goal. The programs are hard to write, maintain and debug. This is because C forces state machines to be specified explicitly. In addition, since the C compiler cannot optimize these programs effectively, the programmer is forced to perform these optimizations manually. This further degrades the readability of the code. This also introduces subtle bugs into the program. Even after several man-years spent on debugging the VMMC firmware, we continue to encounter bugs frequently.

ESP meets all three of its goals. It has a number of language features that allow development of compact and modular programs. The Spin model checker is used to develop and extensively test the programs. It uses models that are generated automatically by the ESP compiler. Once debugged, ESP programs can be compiled into efficient device firmware using the ESP compiler.

The VMMC firmware is used as a case study to evaluate ESP. The VMMC firmware was reimplemented using ESP and compared against the earlier implementation that used C. In the rest of this chapter, we present our conclusions and discuss directions for future research.

5.1 Language Support for Ease of Programming

ESP is designed to simplify event-driven state-machines programming. State machines are specified implicitly using processes. The processes communicate with each other by sending messages on channels. ESP provides a number of novel language features. First, it provides an explicit `malloc/free`-style interface to dynamically allocate memory. To make it easier to perform the allocation correctly, ESP allows each process to manage its allocation independently. This allows a model checker like Spin to exhaustively check for allocation bugs in the program. Second, pattern matching is used to support dispatch on channels efficiently. Third, channels are used to provide a flexible and powerful interface to C.

The ESP compiler was implemented using Standard ML of New Jersey (SML/NJ) and required around 7000 lines of code.

The VMMC firmware was reimplemented using ESP. It required 500 lines of ESP code together with 3000 lines of C code. The portion of the program written in C was used to implement simple low-level operations like accessing device registers and volatile memory. In contrast, the earlier implementation in C required 15600 lines of C code. The ESP implementation was significantly easier than the C implementation. It required a factor of five fewer lines of code. In addition, all the complex interactions in the ESP implementation were confined to the 500 lines of ESP code. In the earlier implementation in C, the complex interactions were scattered over the entire program.

A number of things remain to be done. First, ESP can be extended to support low-level operations [71, 78, 27]. It currently uses C to perform these operations. Second, ESP has so far been used to write firmware for a single device (Myrinet network interface card). Experience with writing firmware for other devices is necessary to further validate the effectiveness of ESP.

5.2 Extensive Testing using a Model Checker

Model-checking verifiers like Spin can be used to develop and extensively test ESP programs. The ESP compiler automatically extracts Spin models for use by the model checker. This greatly reduces the effort required in using the model checker. This allowed the model checker to be used not just to find subtle deadlocks late in the development process—it was used for initial development as well. This allows the programmer to avoid the slow and painstaking process of developing the firmware on the device itself. To deal with the state-space explosion problem, the ESP compiler generates several different models that can be used for different purposes: detailed models are usually used in the simulation mode during the initial development; memory-safety models are used to check for allocation bugs; abstract models are used to check system-wide properties like absence of deadlocks.

The Spin model checker was used to develop and debug the new implementation of VMMC firmware in ESP. Spin was used to implement a retransmission protocol. The new implementation in ESP took around 2 days (compared to the earlier implementation in C which took around 10 days). Spin was also used to exhaustively check the firmware for memory allocation bugs. It found an allocation bug in an early implementation of the firmware. It also found all the allocation bugs that were manually inserted to verify its effectiveness. Finally, Spin was used to identify seven bugs in the firmware using abstract models. These bugs could cause the firmware to deadlock.

One interesting direction for future research is to explore the tradeoff of using a radically different model checker like Verisoft [48]. Verisoft does not require a separate model to be provided—it uses the concurrent program itself as the model and explores the state space by controlling the scheduler at runtime. This would allow more freedom in the language design.

5.3 Generating Efficient Code

The ESP compiler compiles ESP programs to generate efficient firmware. A context switch is fairly lightweight because the only context that needs to be saved and restored is the program counter. The design of the language facilitates aggressive optimizations. It allows a reference-counting allocator to be used to implement “sending messages over channel by-value” semantics without actually creating a copy of the message. In addition, since the processes and channels are known at compile time, pattern matching over channels is implemented very efficiently.

Measurements using VMMC firmware as a case study indicate that the performance impact of using ESP instead of C to write device firmware is small. Since ESP does not currently support fast paths, the performance difference between the two implementations can be broken down into two components: impact of not supporting fast paths in ESP, and impact of using ESP instead of C. For microbenchmark, measurements show that the ESP version performs 0–35 % worse in the latency microbenchmark and 12–25 % worse in the bandwidth microbenchmark. For SPLASH2 applications, using ESP results in a performance hit of 3.5 % on average. The measurements also indicate that the fast paths in the C implementation have little impact on application performance even though they sometimes make a significant impact on the microbenchmarks (latency of 4 byte messages is improved by a factor of two due to the fast paths).

A number of compiler optimizations can be added to the ESP compiler to further improve the performance of the generated code. The data-flow analysis can be extended to perform inter-process optimizations. In addition, optimizations like process inlining can avoid the process context-switch overheads in some cases. Finally, ESP compiler can provide support for fast paths. However, this will require research to answer several challenging problems. First, the compiler would have to identify the fast paths in the concurrent program. Second, the compiler will have to extract the fast path and optimize it. Third, the fast paths have to be robust so that they benefit not only the microbenchmarks that have simple communication patterns but also the applications.

Appendix A

The ESP Language Reference

This appendix describes Event-driven State-machines Programming (ESP)—a language for programmable devices. ESP is designed to simplify the task of implementing reliable high-performance firmware for these devices.

A.1 Lexical Issues

Identifier: An identifier is a sequence of letters, digits and underscores starting with a letter. Identifiers in ESP are case sensitive. In this appendix, the symbol `id` represents an identifier.

Keywords: A keyword is a reserved identifier that has a special meaning in ESP. In this appendix, a keyword is shown in bold.

Comments: ESP supports C++ style comments. Comments are enclosed between '/*' and '*/' or between '//' and the end of the line.

A.2 Notation

This appendix presents the grammar of ESP using the following notation:

```
[ x ]⋆          ⇒  0 or more x
[ x ]+          ⇒  1 or more x
[ x ]?          ⇒  0 or 1 x
[ x ],+         ⇒  1 or more x separated by ','
( x | y | z )   ⇒  x or y or z. For e.g.,
                         P  →  ( x | y | z )
                    is the same as
                         P  →  x
                            →  y
                            →  z
```

A.3 Program

```
program        →  [ dec ]⋆

dec            →  typeDec
               →  channelDec
               →  processDec
               →  interfaceDec
```

A program consists of a sequence of type, channel and process declarations. The concurrent program specifies a set of processes that communicate with each other using channels.

A.4 Data Types

The syntax of types and type declarations in ESP is

```
typeDec        →  type id = ty
```

```
ty          →  int
            →  bool
            →  id
            →  [ # ]? record of { tyFields }
            →  [ # ]? union of { tyFields }
            →  [ # ]? array of ty

tyFields    →  [ id : ty ],+
```

Named Types: Two built-in types `int` and `bool` are predefined. Other named types can be defined using previously defined (or built-in types). Each type declaration creates a new type that is type-incompatible with all other types. ESP does not support recursive types, so ESP programs cannot have any circular data structures.

Records and Unions: Record and union types specify a sequence of named fields. All the fields of a nonnil record are valid. Exactly one field of a nonnil union is valid. The order of fields in records and unions are significant.

Only the valid field of a union can be read. Writing to a different field of a mutable union invalidates all other fields.

Arrays: An array is an indexed list of objects of a particular type. The length of the array is not specified in the type but is determined during array allocation at run time.

Mutable Types: ESP supports both mutable and immutable version of records, arrays and unions. A '#' is used to indicate that the type is mutable.

Examples: In the following code, `sendT` and `updateT` are immutable record types, `userT` is an immutable union type and `tableT` is a mutable array type.

```
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}
type tableT = #array of int
```

A.5 Channels

Channels in ESP are declared as follows:

```
channelDec    →  channel id : ty
```

Channels provide the only way for processes to communicate with each other. Channels are synchronous, i.e, there is no buffering on the channels. Also, only immutable types can be specified for a channel.

A.6 Processes

Processes in ESP are specified as follows:

```
processDec    →  process id compoundStmt
```

A process represents a sequential flow of control and is specified by the `compoundStmt`. Since the processes do not need a stack, they implicitly encode a state machine. Process cannot be dynamically created.

A.7 Expressions

```
expr          →  constant
              →  lValue
              →  primOp ( lValue )
              →  allocation
              →  expr : ty
              →  expr :: ty
              →  expr bOp expr
              →  uOp expr
```

```
constant      →  numericConst
              →  @ [ id ]?
              →  ( true | false )
              →  nil

allocation    →  [ # ]? { [ expr ],+ }
              →  [ # ]? { id |> expr }
              →  [ # ]? { expr -> expr ... }
              →  [ # ]? { -> [ expr ],+ }

bOp           →  ( <= | < | == | != | >= | >  )
              →  ( & | | | << | >>  )
              →  ( && | || )
              →  ( + | - | * | / | % )

uOp           →  ( ! | ˜ | + | -)

primOp        →  length
```

Expression: An expression is a side-effect-free (other than allocation) piece of code that evaluates to a value.

Constant: `numericConst` are integer constants. `@ SM1` is an integer constant that uniquely identifies process `SM1`. When the optional process name is left out, it defaults to the current process. `true` and `false` are boolean constants. `nil` is a constant that is valid for arrays, records and unions.

Allocation: A record is allocated by specifying the values for its fields in order. A union is allocated by specifying the valid fields and the value for that fields separated by `'|>'`. An array can be allocated two ways and is identified by `'->'` in the expression. First, a size and a value (obtained by computing the specified expression once) is provided that is used to initialize all the entries. Second, all the entries are specified. A mutable version of a data structure can be allocated using the '#' prefix.

Note that a data structure cannot be allocated and dereferenced in the same expression. For instance, `{->1,3,2}`[2] is invalid.

Types: The type of any expression can be specified using the ':' separator.

Type Cast: An object can be cast to a different type using the cast operation (specified using '::'). Not all casts are valid. ESP allows a mutable object to be cast into an immutable object and vice versa. A cast operation causes allocation and returns a semantic deep copy of the object being cast (as described in Section A.9). The programmer is responsible for freeing objects allocated during the cast.

Operators: Operators in ESP have the same precedence and associativity as in C [61]. However, ESP disallows pointer comparisons.

```
lValue       →  id
             →  lValue . id
             →  lValue [ expr ]
             →  lValue : ty

lValueNew    →  $ id

pattern      →  lValue
             →  lValueNew
             →  pattern : ty
             →  constant
             →  _
             →  { [ pattern ],+ }
             →  { id |> pattern }
```

l-value: An l-value is an expression that specifies a memory location that may be read and written. As in C, the '.' is used to access fields of records and unions and '[]' is used access an array entry. A '$' prefix specifies a new variable declaration in the current scope.

Pattern: A pattern provides a convenient way to access the components of a data structure. When the pattern matches the data structure, the l-values in the pattern are assigned the corresponding values in the data structure. The rest of the pattern is used for pattern matching. The components that correspond to a '_' in the pattern are ignored. A pattern should have at least one l-value.

As in ML, patterns and allocations use similar syntax. They can be distinguished based on their position in a statement. They are considered a pattern when they occur in a l-value position and cause allocation when they occur in a r-value position. For instance:

```
$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[1] 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.

In ESP, patterns are also used to support dispatch on channels (Section A.8).

New variables: '$' declares a new variable in the current scope. All variables have to be initialized during variable declaration.

A.8 Statements

```
stmt        →  pattern = expr ;
            →  assert ( expr ) ;
            →  procName ( lValue ) ;
            →  if ( expr ) stmt else stmt
            →  while ( expr ) stmt
```

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

```
              →  break ;
              →  compoundStmt
              →  chanOp ;
              →  alt { [ case ( expr , chanOp ) stmt ]+ }

compoundStmt  →  { [ stmt ]⋆ }

chanOp        →  in ( id , pattern )
              →  out ( id , expr )

procName      →  free | rfree
```

Standard Statement: ESP supports the standard statements like assignment, `if-then-else`, `while` and `break` statements supported by most imperative languages.

Simple Type Inferencing: ESP allows the type of an expression to be unspecified when it can be inferred from its context in the statement. For instance, types do not have to be specified in the following variable declarations.

```
$i = 36;                          // int
$a = #{ -> 5, 4, 1};              // #array of int
```

However, the following statements need to specify the type.

```
$b: #array of int = nil;
$r: record of { count: int, init: bool} = { 5, false};
```

Operations on Channels: Processes can write to a channel using the `out` statement and read from the channel using the `in` statement. A process specifies a `pattern` when reading from a channel. Then only objects that match the pattern will be delivered to that process.

A channel can have multiple writers and multiple readers. However, the processes reading from a particular channel have to satisfy two properties. First, the set of patterns specified on the channel have to be disjoint. This means that every object sent on the channel will match at most one pattern. Second, each distinct pattern can be used by only one process (possibly several times).

The `alt` statement allows a process to wait on `in` and `out` operations on several different 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. 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.

```
channel chan1: int
channel chan2: int

process buffer {
  $Q = #{ SIZE -> 0, ...  };
  $tl = 0;
  $hd = 0;

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

Namespaces: There are four separate namespaces in ESP, one for each of types, channels, processes and variables.

Scope Rules: Types, channels and processes are defined only at the top-level scope. Only variables can be defined in nested scopes. A variable can be redefined in a nested scope but not in the same scope. However, all identifiers can be accessed from the same or a deeper-nested scope.

ESP does not support global variables, so variables cannot be defined at the top level. ESP programs are lexically scoped. New nested scopes are introduced by

- Compound statements.
- Body of the `while` loop.
- Each branch of the `if` statement.
- Each `case` of the `alt` statement introduces a pair of nested scopes. The first nested scope includes the `chanOp`. The second nested scope is nested inside the first and is introduced by the body of that `case`.

In the following example, variables `i` and `b` declared in the same scope.

```
channel c1: bool
process p1 {
  while ( true) {
    $i = 0;
    in( c2, $b);
  }
}
```

The following code illustrates the nested scopes introduced by an `alt` statement.

```
channel c2: bool
channel c3: int

process p2 {
  while ( true) {
    $b = true;                    // Defining b.
    alt {
      case ( in( c3, $j)) {}
      case ( in( c2, $b)) { // Redefining b in a nested scope.
```

```
                $c = b;           // b is accessible here.
                $b = 5;           // Redefining b in a nested scope.
                $k = j;           // Error. j not accessible here.
              }
            }
          }
        }
```

A.9 Memory Management

Object Allocation: Objects are allocated using the syntax described in Section A.7.

Objects on Channels: The receiving process receives a deep copy of the object that was sent to it over the channel. The deep copy performed does not preserve any pointer sharing that was present in the sent object. Objects received over a channel are similar to a newly allocated object and have to be freed by the receiving process.

Casting Objects: A deep copy of the object (like objects on channels) is returned by the cast operation.

Freeing Objects: The set of objects accessible to each process is disjoint. Each process is responsible for freeing its own objects using `free` and `rfree` operations. Each object has to be freed exactly once. The `free` operation frees just the object being specified. The `rfree` operation recursively frees the object being specified. It should be noted that `rfree` does not check to see if an object has already been freed. So, if an array was allocated and all entries point to a single record, `rfree` will free that record multiple times. This is an error and can cause the program to crash at runtime.

Anonymous Allocation: An anonymous allocation is one that the programmer never acquired a pointer to. The only place this can occur is when an object is allocated in an `out` statement to be sent over the channel. A process is not responsible for freeing anonymous allocation.

C interface: C functions implementing `in` operation on channels might be passed ESP objects. Similarly, C functions implementing `out` operation on channel might need to allocate and return a new object. To continue to have access to these objects even after a function has returned, the function has to invoke the `link` library function provided by the ESP library. This will prevent the ESP runtime from freeing the objects while the C code is still using it. In addition, `unlink` and `alloc` library functions allow the C code to relinquish linked objects and allocate new objects respectively. C functions are not allowed to mutate the ESP objects.

A.10 External Interface

```
interfaceDec   →  interface ( ( in | out ) id ) cInterface
cInterface     →  { [ id ( pattern ) ],+ }
```

External channels provide a synchronous interface to external code written in C or Spin. An external channel has ESP processes reading (or writing) from it while C code writes (or reads) values on that channel. An external channel specifies exactly one interface. For example:

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

defines a channel with an external writer.

Interface to C. The `cInterface` is used to specify the interface with C code. It includes a list of `id` and `pattern` pairs. The `$` prefix in the pattern indicates a parameter to be passed to the C function. The patterns in the `cInterface` are not fully general patterns (Section A.7)—an `lValue` is not permitted in these patterns.

For an external channel, ESP requires two types of C 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. 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 an integer that specifies which one of the patterns is ready. Depending on the return value, one of the two functions (`UserReqSend` and `UserReqUpdate`) will be invoked.

External `in` channels differ from external `out` channel in two ways. First, the `IsReady` function just returns whether or not the channel is willing to accept data. Then any C 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 fewer level of indirection.

Interface to Spin. Spin code interacts with ESP code by directly reading and writing on the `userReqC` channel.

Appendix B

VMMC Firmware in ESP

This chapter provides an overview of the implementation of VMMC firmware using ESP. The implementation of the VMMC firmware required 500 lines of ESP code together with around 3000 lines of C code. The ESP code uses 8 processes and 19 channels. Figure B.1 presents a schematic representation of the code. Table B.1 and Table B.2 provide a brief description about the processes and channels respectively. Details about the functionality implemented by the firmware are discussed in [41, 42].

The following example illustrates the interactions between the various processes and channels. Consider a request by an application on machine A to remote fetch data from machine B. This should cause some data on machine B to be read and transferred back to machine A. This will involve the following steps:

1. On machine A, process `localRequest` receives the application's request and generates a remote fetch request packet to be sent to machine B. The packet is sent on channel `C7`. Details necessary to process the reply to the packet is sent on channel `C6`.

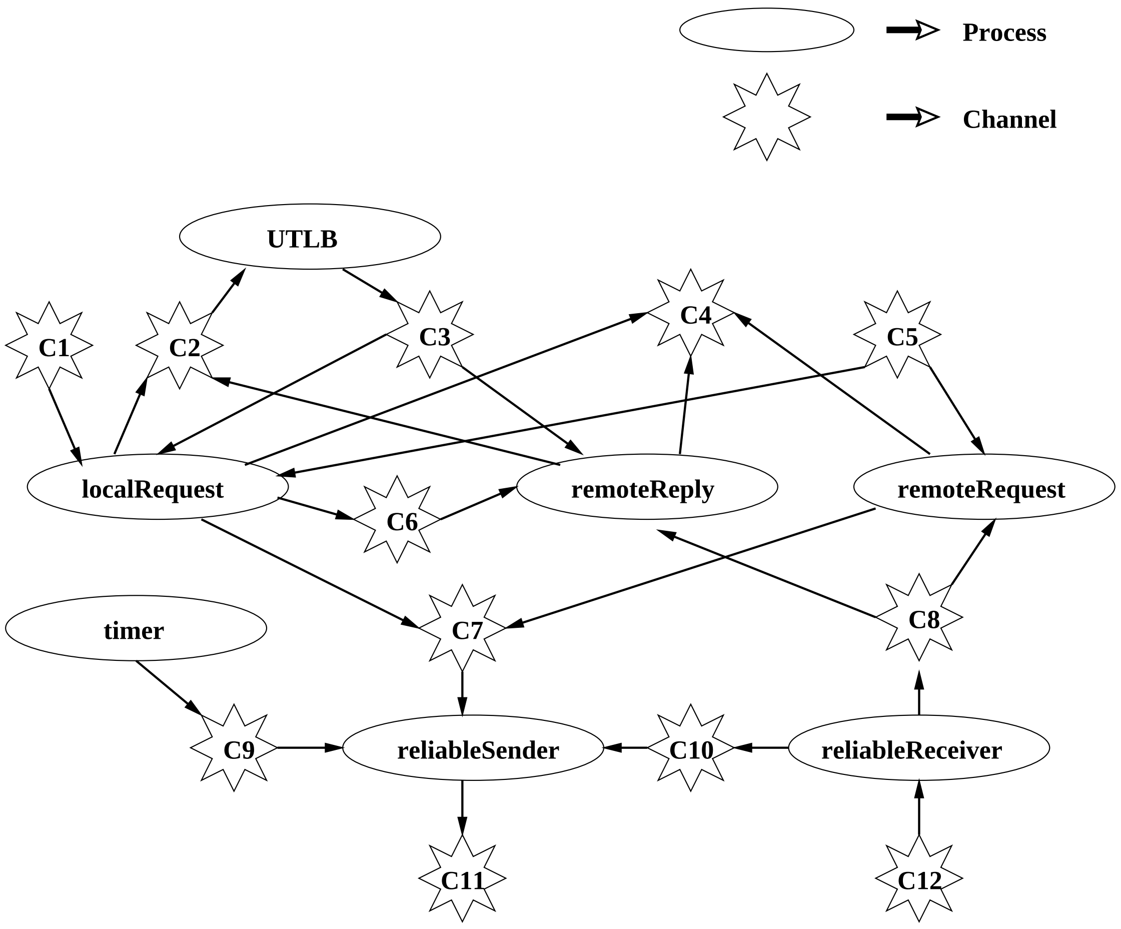

Figure B.1: VMMC Firmware using ESP. The channels that have only readers or only writers are external channels that are used to interface with C. One process (`ETLB`) and seven external channels (`C13`-`C19`) have been omitted from this figure for simplicity.

Process	Description
localRequest	Handles requests from the application to send/receive data.
remoteRequest	Handles requests arriving on the network to send/receive data.
remoteReply	Handles replies arriving on the network in response to a request sent.
UTLB	Translates virtual addresses to physical addresses [29].
ETLB	Translates virtual addresses to physical addresses for exported regions.
reliableSender	A component of the retransmission protocol that accepts packets to be sent on the network.
reliableReceiver	A component of the retransmission protocol that accepts packets arriving on the network.
timer	Generates timeouts for the retransmission protocol.

Table B.1: Description of the processes used

Channel	Description
C1	Requests from the application to send/receive data.
C2	Requests to the UTLB.
C3	Replies from the UTLB.
C4	Request to the DMA to read/write data.
C5	Data that has been read by the last DMA request.
C6	Relevant state pertaining to a request sent that requires a reply.
C7	Packet to be sent out to the network reliably.
C8	Packet received from the network reliably.
C9	Acknowledgement send/receive timeouts.
C10	Details of the last packet received/acknowledged.
C11	Packet to be sent out to the network.
C12	Packet received from the network.
C13	Request to lookup the export table.
C14	Result of the export table lookup.
C15	Periodically generates a clock tick for use by the timer.
C16	Returns a node identifier for the network card.
C17	Request to update the import table.
C18	Requests to the ETLB.
C19	Replies from the ETLB.

Table B.2: Description of the channels used

2. Process `remoteReply` reads the details about the request on channel `C6` and waits for the reply.

3. Process `reliableSender` reads the packet on channel `C7` and sends it to the network by writing it to channel `C11`. It then waits for an acknowledgement. Processes `reliableSender` and `reliableReceiver` implement a simple sliding window protocol with piggyback acknowledgement.

4. If a timeout occurs, process `timer` sends a timeout message on channel `C9`.

5. If a timeout message arrives on channel `C9` before an acknowledgement is received, process `reliableSender` resends the packet.

6. On machine B, process `reliableReceiver` receives the packet, checks to see if it has a valid sequence number and sends it out on channel `C8`. If the packet includes a piggyback acknowledgement, it send it to process `reliableSender` on channel `C10`.

7. Process `remoteRequest` reads the remote fetch request from channel `C8`. It looks up the export table using channels `C13` and `C14` to validate the request and determine the physical address of the requested data. It uses channels `C4` and `C5` to access the DMA engine and transfer data from the host memory to the network card. It then composes a reply packet that includes the data and sends it on channel `C7`.

8. Process `reliableSender` reads the packet on channel `C7` and sends it to the network using channel `C11`.

9. On machine A, process `reliableReceiver` receives the reply packet, checks to see if it has a valid sequence number and sends it out on channel `C8`.

10. Process `remoteReply` receives the reply packet on channel `C8`. It uses channels `C2` and `C3` to communicate with process `UTLB` to translate the virtual addresses to

physical addresses. It uses channels `C4` to access the DMA engine to write the data received into the applications memory. It then notifies the application that the remote fetch operation was successfully completed using channel `C4`.

Bibliography

[1] M. B. Abbott and L. L. Peterson. Increasing Network Throughput by Integrating Protocol Layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, Oct. 1993.

[2] A. Acharya and M. U. amd Joel Saltz. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 1998.

[3] Adaptec. *Adaptec RAID Controller*, 2001.

[4] R. D. Alpert, C. Dubnicki, E. W. Felten, and K. Li. Design and Implementation of NX Message Passing using SHRIMP Virtual Memory Mapped Communication. In *Proceedings of the International Parallel Processing Symposium*, Honolulu, Hawaii, April 1996.

[5] G. R. Andrews. *Concurrent Programming*. Benjamin/Cummings Publishing Company, 1991.

[6] G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings, 1993.

[7] G. R. Andrews and F. B. Schneider. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys*, 15(1):3–43, 1983.

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

[9] T. Ball and J. Larus. Efficient Path Profiling. In *IEEE Micro*, Paris, France, December 1996.

[10] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *Proceedings of the Conference on Programming Languages Design and Implementation*, Snowbird, Utah, June 2001.

[11] T. Ball and S. K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In *Proceedings of the International Spin Workshop*, Stanford University, August 2000.

[12] A. Basu, T. von Eicken, and G. Morrisett. Promela++: A Language for Correct and Efficient Protocol Construction. In *Proceedings of the IEEE Infocom*, San Francisco, California, March 1998.

[13] G. Berry. *The Constructive Semantics of Pure Esterel*. Draft 3, 1999.

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

[15] R. A. F. Bhoedjang, T. Rühl, and H. E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, 1998.

[16] A. Bilas. Improving the Performance of Shared Virtual Memory on System Area Networks (Thesis). Technical Report TR-586-98, Princeton University, Department of Computer Science Department, 1998.

[17] A. Bilas and E. W. Felten. Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface. *Journal of Parallel and Distributed Computing. Special Issue on Workstation Cluster and Network-based Computing*, 1997.

[18] A. Bilas, C. Liao, and J. Singh. Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. In *Proceedings of the International Symposium on Computer Architecture*, Atlanta, Georgia, May 1999.

[19] A. Bilas and J. P. Singh. The Effects of communication Parameters on End Performance of Shared Virtual Memory Clusters. In *Proceedings of the Supercomputing Conference*, San Jose, California, November 1997.

[20] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the International Symposium on Computer Architecture*, Chicago, Illinois, April 1994.

[21] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[22] T. Braun and C. Diot. Protocol Implementation Using Integrated Layer Processing. In *Proceedings of the SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Cambridge, Massachusetts, September 1995.

[23] L. Cardelli. Amber. *Combinators and Functional Programming Languages Lecture Notes in Computer Science*, 242, 1985.

[24] L. Cardelli and R. Pike. Squeak: A Language for Communicating with Mice. *Computer Graphics*, 19(3):199–204, July 1985.

[25] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating Efficient Protocol Code from an Abstract Specification. In *Proceedings of the SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Stanford, California, August 1996.

[26] T. Cattel. Modeling and Verification of a Multiprocessor Realtime OS Kernel. In *Proceedings of the Conference on Formal Description of Techniques (FORTE)*, Berne, Switzerland, October 1994.

[27] S. Chandra and P. J. McCann. Packet Types: Abstract specifications of network protocol messages. In *Proceedings of the SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Stockholm, Sweden, August 2000.

[28] S. Chandra, B. E. Richards, and J. R. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *Proceedings of the Conference on Programming Languages Design and Implementation*, Philadelphia, Pennsylvania, May 1996.

[29] Y. Chen, A. Bilas, S. N. Damianakis, C. Dubnicki, and K. Li. UTLB: A Mechanism for Address Translation on Network Interfaces. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 1998.

[30] Y. Chen, S. N. Damianakis, S. Kumar, X. Yu, and K. Li. Porting a User-Level Communication Architecture to NT: Experiences and Performance. In *Proceedings of the USENIX Windows NT Symposium*, Seattle, Washington, July 1999.

[31] M. Chiodo, P. Guisto, A. Jurecska, L. Lavagno, H. Hsieh, K. Suzuki, A. L. Sangiovanni-Vincentelli, and E. Sentovich. Synthesis of Software Programs for Embedded Control Applications. In *Proceedings of the Conference on Design Automation*, San Francisco, California, June 1995.

[32] A. Chou, B. Chelf, D. Engler, and M. Heinrich. Using Meta-level Compilation to Check FLASH Protocol Code. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, Massachusetts, November 2000.

[33] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of the SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Philadelpia, Pennsylvania, September 1990.

[34] R. L. Constable, S. F. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. J. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.

[35] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R. Shawn, and L. Hongjun. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the International Conference on Software Engineering*, Limerick, Ireland, June 2000.

[36] Cyclone Microsystems. *Intelligent I/O Controllers*, 2000.

[37] S. N. Damianakis. Efficient Connection-Oriented Communication on High-Performance Networks (Thesis). Technical Report TR-582-98, Princeton University, Department of Computer Science Department, Apr. 1998.

[38] R. DeLine and M. Fahndrich. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the Conference on Programming Languages Design and Implementation*, Snowbird, Utah, June 2001.

[39] E. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, 1975.

[40] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Washington, 1992.

[41] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Proceedings of the Hot Interconnects Symposium*, Stanford, California, August 1997.

[42] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *Proceedings of the International Parallel Processing Symposium*, Geneva, Switzerland, April 1997.

[43] G. Duval and J. Julliand. Modeling and verification of the RUBIS μ-Kernel with Spin. In *Proceedings of the International Spin Workshop*, Montreal, Quebec, October 1995.

[44] Echelon Corporation. *Neuron C Reference Guide*, 2001.

[45] S. A. Edwards. Compiling Esterel into sequential code. In *Proceedings of the Conference on Design Automation*, Los Angeles, California, June 2000.

[46] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, San Diego, California, October 2000.

[47] S. J. Garland, N. A. Lynch, and M. Vaziri. *IOA: A Language for Specifying, Programming and Validating Distributed Systems*. Laboratory for Computer Science, MIT, 1997.

[48] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[49] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the USENIX Winter Technical Conference*, San Francisco, California, January 1992.

[50] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[51] M. Hayden. The Ensemble System. Technical Report TR98-1662, Computer Science Department, Cornell University, 1998.

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

[53] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[54] G. Holzmann and D. Peled. An Improvement in Formal Verification. In *Proceedings of the Conference on Formal Description of Techniques (FORTE)*, Berne, Switzerland, October 1994.

[55] G. J. Holzmann. The Spin Model Checker. *IEEE Transaction on Software Engineering*, 23(5):279–295, May 1997.

[56] G. J. Holzmann. *Promela Online Reference*. http://cm.bell-labs.com/cm/cs/what/spin/Man/Intro.html, 2001.

[57] G. J. Holzmann and M. H. Smith. A Practical Method for Verifying Event-Driven Software. In *Proceedings of the International Conference on Software Engineering*, Los Angeles, California, May 1999.

[58] Intel Corporation. *IXP1200 Network Processor Data Sheet*, 2000.

[59] D. Jiang, B. O'Kelly, X. Yu, S. Kumar, A. Bilas, and J. P. Singh. Application Scaling under Shared Virtual Memory on a Clusters of SMPs. In *Proceedings of the International Conference on Supercomputer*, Rhodes, Greece, June 1999.

[60] D. Jiang, H. Shan, and J. Singh. Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-coherent Multiprocessors. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, Las Vegas, Nevada, June 1997.

[61] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[62] R. Kieburtz and A. Silberschatz. Comments on "Communicating Sequential Processes". *ACM Transactions on Programming Languages and Systems*, 1(2):218–225, 1979.

[63] S. Kumar and K. Li. Performance Impact of Using ESP to Implement VMMC Firmware. In *Proceedings of the Workshop on Novel Uses of System Area Networks (SAN-1)*, Cambridge, Massachusetts, February 2002.

[64] S. Kumar, Y. Mandelbaum, X. Yu, and K. Li. ESP: A Language for Programmable Devices. In *Proceedings of the Conference on Programming Languages Design and Implementation*, Snowbird, Utah, June 2001.

[65] J. Larus. Whole Program Paths. In *Proceedings of the Conference on Programming Languages Design and Implementation*, Atlanta, Georgia, May 1999.

[66] 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.

[67] D. Lie, A. Chou, D. Engler, and D. Dill. A Simple Method for Extracting Models from Protocol Code. In *Proceedings of the International Symposium on Computer Architecture*, Goteborg, Sweden, June 2001.

[68] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building Reliable, High-Performance Communication Systems from Components. In *Proceedings of the Symposium on Operating Systems Principles*, Kiawah Island Resort, South Carolina, December 1999.

[69] 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 *Proceedings of the International Symposium on Computer Architecture*, Denver, Colorado, June 1997.

[70] K. McMillan. Symbolic model checking—An approach to the state explosion problem (Thesis). Technical Report CMU-CS-92-131, School of Computer Science, Carnegie Mellon University, 1992.

[71] F. Merillon, L. Reveillere, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for Hardware Programming. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, San Diego, California, October 2000.

[72] R. Milner. A Calculus of Communicating Systems. *Lecture Notes in Computer Science*, 92, 1980.

[73] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[74] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[75] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[76] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley. Analysis of Techniques to Improve Protocol Processing Latency. In *Proceedings of the SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Stanford, California, August 1996.

[77] Nortel Networks. *Alteon Network Adapter*, 1998.

[78] S. O'Malley, T. Proebstring, and A. B. Montz. USC: A Universal Stub Compiler. In *Proceedings of the SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, London, August 1994.

[79] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of the Supercomputing Conference*, San Diego, California, December 1995.

[80] L. Peterson and B. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers, 1996.

[81] R. Pike. Newsqueak: A language for communicating with Mice. Technical Report TR143, AT&T Bell Laboratories, Computing Science Dept, 1989.

[82] R. Pike. The implementation of newsqueak. *Software, Practice and Experience*, 20(7):649–660, 1990.

[83] R. Pike, D. Pressoto, K. Thompson, and G. Holzmann. Process sleep and wakeup on shared-memory multiprocessors. In *Proceedings of the EurOpen Conference*, Tromso, Norway, May 1991.

[84] T. A. Proebsting and S. A. Watterson. Filter Fusion. In *Proceedings of the Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996.

[85] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Proceedings of the Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, December 1995.

[86] Ramix Inc. *Ramix Ethernet Controllers*, 2001.

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

[88] J. H. Reppy. Higher-order Concurrency. Technical Report TR-92-1285, Computer Science Department, Cornell University, 1992.

[89] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[90] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[91] SGS-Thomson Microelectronics, Bristol. *OCCAM 2.1 Reference Manual*, 1995.

[92] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[93] M. Tofte and J.-P. Talpin. Implementation of the Typed Call-by-Value lambda-Calculus Using a Stack of Regions. In *Proceedings of the Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994.

[94] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: A practical tool for OS implementors. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, Cape Cod, Massachusetts, May 1997.

[95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *Proceedings of the Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, December 1995.

[96] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[97] T. von Eicken and W. Vogels. Evolution of Virtual Interface Architecture. *IEEE Computer*, 31(11):61–68, 1998.

[98] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities . In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, California, February 2000.

[99] S. Walton, A. Hutton, and J. Touch. High-Speed Data Paths in Host-Based Routers. *IEEE Computer*, 31(11):46–52, 1998.

[100] R. Y. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log Based File Systems for a Programmable Disk. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, February 1999.

[101] M. Weiser. Program slicing. *IEEE Transaction on Software Engineering*, 10:352–357, 1984.

[102] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management*, St. Malo, France, September 1992.

[103] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[104] B. Zorn. *Debugging Tools for Dynamic Storage Allocation and Memory Management*. http://www.cs.colorado.edu/homes/zorn/public_html/MallocDebug.html, 2001.