

# Performance Impact of Using ESP to Implement VMMC Firmware

Sanjeev Kumar, Kai Li

*Abstract*— **ESP is 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.**

**This paper measures the performance impact on applications of using ESP to implement VMMC firmware. It compares the performance of an earlier implementation of VMMC firmware that used C with the new implementation that uses ESP. We find that SPLASH2 applications incur a modest performance hit (3.5 % on average) when using the ESP version.**

**This paper also describes the techniques used by the ESP compiler to optimize the programs. To achieve good performance, the C version required a number of optimizations to be performed manually by the programmer. In contrast, the ESP version was optimized entirely by the compiler.**

*Keywords*— **Programmable devices, Domain-specific languages, User-level Communication**

## I. INTRODUCTION

Device firmware needs to be reliable as well as fast. It has to be reliable because it 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. In addition, the firmware has to be fast because devices are equipped with relatively slow processors but have to keep up with devices operating at Gigabit speeds.

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. For instance, the Virtual Memory Mapped Communication (VMMC) firmware [1] for Myrinet [2] network interface was implemented using event-driven state machines in C. Our experience with the VMMC firmware was that while good performance could be achieved with this approach, it required the programmer to manually perform a number of optimizations. The source code was hard to maintain and debug. The implementation involved around 15600 lines of

C code. Even after several man years of debugging, race conditions cause the machine to crash occasionally.

ESP [3], [4] is 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. Our earlier papers [3], [4] have shown how ESP meets its three goals: ease of programming, ease of debugging, and high performance. However, they used only microbenchmarks to compare the performance of the two versions of the VMMC firmware.

This paper presents the performance impact of using ESP to implement VMMC firmware on applications. SPLASH2 applications are used to compare the performance of the two versions of the VMMC firmware. Our measurements show that SPLASH2 applications incur a modest performance hit (3.5 % on average) when using the ESP version.

This paper also describes some of the techniques used by the ESP compiler to generate efficient code. The effectiveness of the ESP compiler frees the programmer from having to optimize the program manually. This allows ESP programs to be expressed concisely. The ESP version of the VMMC firmware involved only 500 lines of ESP code together with around 3000 lines of C code<sup>1</sup>. This is a significant reduction in the number of lines of code over the C implementation. The complexity of the code is also greatly reduced. This is because the C version required the programmer to manually optimize it.

Ease of programming can help improve the performance of applications. Often, applications can achieve better performance if the device supports a richer interface. 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 [5]. ESP makes it easier to explore and add new features to the device firmware.

The rest of the paper is organized as follows. Section II discusses the related work. Section III presents an overview of the ESP language. Section IV describes how the ESP compiler generates efficient code. Section V measures the performance impact of using ESP to implement the device

Sanjeev Kumar and Kai Li, Department of Computer Science, Princeton University, {skumar,li}@cs.princeton.edu.

<sup>1</sup>C is used to implement only simple low-level operations like packet marshalling and handling device registers. All the complexity in the program is localized to the ESP code.

firmware. It uses both microbenchmarks as well as applications to study the impact. Section VI presents some discussion and future work. Finally, Section VII presents the conclusions.

## II. RELATED WORK

Concurrency in ESP [3] is expressed using processes and channels. Each process in ESP implicitly encodes a state machine. An ESP program consists of a set of processes communicating with each other over channels. The ESP compiler has to compile the concurrent ESP program to run on a single processor.

There are two main approaches to compiling a concurrent program to run efficiently on a single processor: *automata-based* approach and *process-based* approach. The automata-based approach [6], [7], [8], [9] 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 [8], [10] 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 [11], [12] 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 involves a runtime overhead, the overhead is fairly low.

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

Esterel [7] is a synchronous language designed to model the control of concurrent systems. Earlier Esterel compilers [7], [8] 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<sup>2</sup> 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 [12]. Process-based compilers [12] 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.* [12] 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 [7] generates the fastest code but the size of the executable can be 2-3 orders of magnitude larger than the other 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 [14] supports processes and synchronous channels and uses a process-based approach [11] 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 [6] 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.<sup>3</sup>

Filter Fusion [9] 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 [7].

Integrated Layer Processing (ILP) [15] 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 [16], [17]. 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 processor on the devices. So they are usually performed either on the host processor or by special-purpose hardware engines on the devices.

## III. ESP

ESP [3] is a language for programmable devices. It is designed to meet three goals: ease of programming, ease of debugging, and high performance.

To support ease of programming, ESP allows programs to be expressed in a concise modular fashion using processes and channels. In addition, it provides a number of

<sup>2</sup>The gate-based compilation technique applies to synchronous languages like Esterel and is not applicable to ESP.

<sup>3</sup>In contrast, Esterel programs are deterministic—all possible schedules yield the same result. Therefore, it does not require a random selection at each stage.

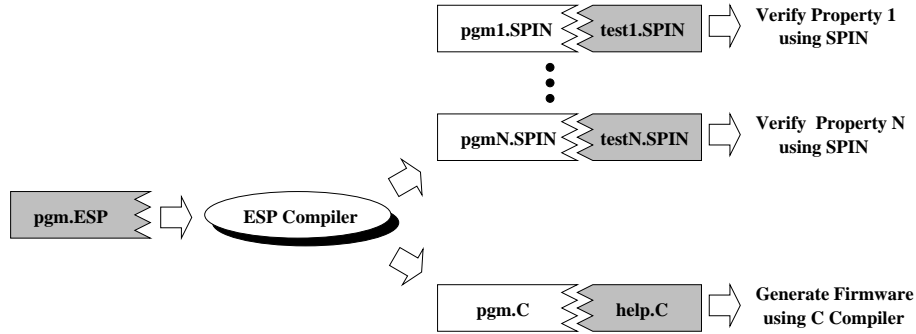


Fig. 1. The ESP compiler generates models (`pgm[1-N].SPIN`) that can be used by the Spin model checker to debug the ESP program (`pgm.ESP`). The compiler also generates a C file (`pgm.C`) that can be compiled into an executable. The shaded regions represent code that has to be provided by the programmer. The test code (`test[1-N].SPIN`) is used to check different properties in the ESP program. It includes code to generate external events such as network message arrival as well as to specify the property to be verified. The programmer-supplied C code (`help.C`) implements simple low-level functionality like accessing special device registers, dealing with volatile memory, and marshalling packets that have to be sent out on the network.

features including pattern matching to support dispatch on channels, a flexible external interface to C, and a novel memory management scheme that is efficient and safe.

To support ease of debugging, ESP allows the use of a model checker like Spin [18] to extensively test the program. The ESP compiler (Figure 1) not only generates an executable but also extracts Spin models from the ESP programs [3], [4]. This minimizes the effort required in using a model checker to debug the program. Often, the ESP program is debugged entirely using the model checker before being ported to run on the device. This avoids the slow and painstaking process involved in debugging the programs on the device itself.

To support high performance, the ESP language is designed to be fairly static so that the compiler can aggressively optimize the programs. In languages like C, event-driven state machines are specified using function pointers. This makes it difficult for the C compiler to optimize the programs. This forces the programmers to hand optimize the program to get good performance. In contrast, ESP is designed to support event-driven state machines. It allows the ESP compiler to generate efficient code.

#### IV. GENERATING EFFICIENT EXECUTABLES FROM ESP PROGRAMS

The ESP compiler uses the process-based approach to generate a sequential code from a concurrent program. 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 other and could starve other processes. 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<sup>4</sup> 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. So 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. So 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

<sup>4</sup>A port can have only a single reader but can have multiple writers. A channel can have multiple readers as well as multiple writers. During compilation, each ESP channel is translated into a group of ports.

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.

The ESP compiler uses C as the back-end language. It compiles a ESP program into a 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. A C compiler is then used to generate an executable program.

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 perform register allocation and to benefit from the optimizations performed by it.

The ESP compiler performs whole program analysis to generate efficient code. The static design of the language allows the compiler to aggressively optimize the program.

The ESP compiler performs some standard optimizations like constant folding, copy propagation and dead code elimination on a per process basis. 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.

At present, ESP does not support fast paths. Fast paths provide better performance to commonly executing paths in the program. 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. In C, the specialized code has to be provided by the programmer. The programmer is responsible for ensuring that the specialized code is functionally equivalent to the corresponding slower path in the program. We are currently exploring ways of using the compiler to generate fast paths in ESP programs.

## V. PERFORMANCE

In this section, we compare the performance of the earlier VMMC implementation in C (*vmmcOrig*) with the performance of the new implementation using ESP (*vmmcESP*). Since ESP does not currently support fast paths, we also present the performance of the C 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*).

---

### Local Node

1. Receive the remote write request from the application.
2. Translate Virtual Address to Physical addresses. Since a contiguous region of virtual address can map onto multiple noncontiguous physical pages, large data transfers are broken down into multiple remote write requests each of which sends data from a single page.
3. Use the DMA to transfer data from the host memory to the device memory.
4. Send the packet out on the network. Retransmit it if an acknowledgment is not received before a timeout occurs.

### Remote Node

1. Receive the remote write request on the network.
  2. Arrange for an acknowledgement to be sent later.
  3. Compute the physical address to which the data has to be delivered.
  4. Use the DMA to transfer the data to the host memory.
- 

Fig. 2. Steps involved in a remote send operation. If the size of the data to be delivered is small ( $\leq 32$  bytes), the data to be sent is included along with the request to transfer the data. Therefore, Steps 2 and 3 on the local node are not required for small messages.

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 it. This allows them to isolate and understand the cost of using ESP. However, they represent a worst-case scenario. On the other hand, the actual performance impact is one that is observed by the application. Applications usually exhibit more complex communication patterns than the microbenchmarks. Applications are also less sensitive to the firmware performance.

#### A. Microbenchmark Performance

**Microbenchmarks.** Three microbenchmarks are used to measure 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 *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 *Bidirectional Bandwidth* microbenchmark measures the total bandwidth between two machines when both the machines are sending messages of a particular size simultaneously.

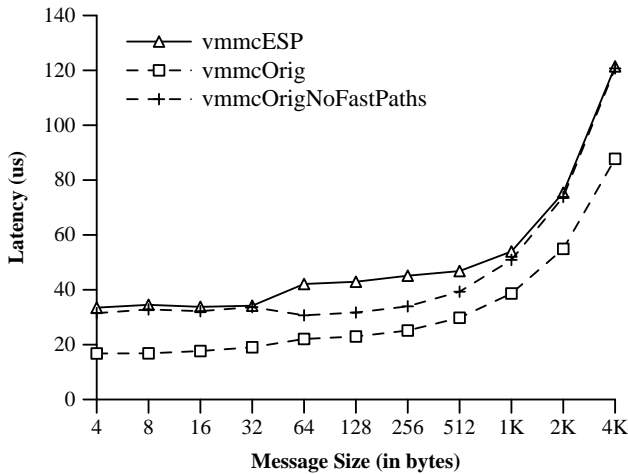


Fig. 3. Latency Microbenchmark

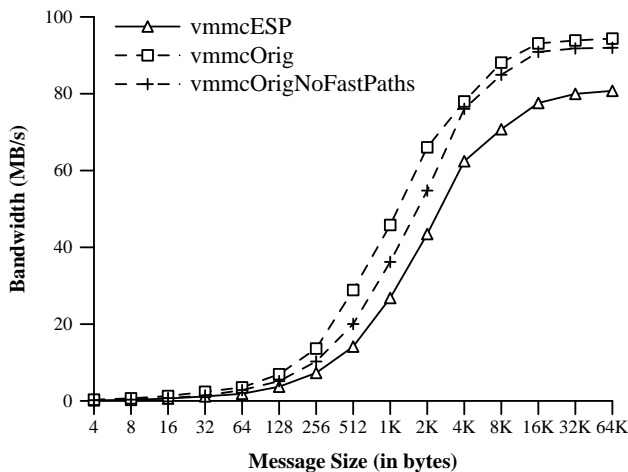


Fig. 4. One-way Bandwidth Microbenchmark

**Platform.** All microbenchmarks measurements use a pair of PC. 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 3, 4 and 5 present the microbenchmark performance. In each case, the x-axis shows the message size. 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 differently (Figure 2). The latter is because the page size is 4Kbytes. Requests that span multiple pages are broken down into multiple requests (Figure 2).

The *Latency* microbenchmark (Figure 3) shows that *vmmcESP* is around twice as slow as *vmmcOrig* for 4 byte messages and 38 % slower for 4 Kbyte messages. However, *vmmcESP* is only 35 % slower than *vmmcOrigNoFastPaths* in the worst case (for 64 byte messages) and has comparable performance for 4 byte and 4 Kbyte messages.

The *Bandwidth* microbenchmark (Figure 4) shows that *vmmcESP* delivers 41 % less bandwidth as *vmmcOrig* for

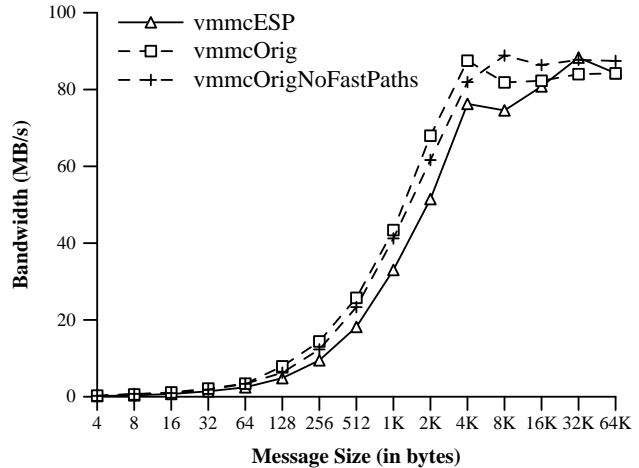


Fig. 5. Bidirectional Bandwidth Microbenchmark

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 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.

The microbenchmark performance shows that *vmmcESP* performs significantly worse than *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.

The *Latency* microbenchmark shows that while the performance of *vmmcESP* is comparable to the performance of *vmmcOrigNoFastPaths* for short messages ( $\leq 32$  bytes), the difference between the two is much larger for 64 byte messages. Some of this difference stems from the fact that there is an extra process involved in *vmmcESP*. In *vmmcOrigNoFastPaths*, a remote send operation (Figure 2) involves two state machines on the local node and two state machines on the remote node. However, in *vmmcESP*, the operation involves three processes (which are essentially state machines) on the local node and two processes on the remote node. The extra process in *vmmcESP* is used to perform the translation of virtual memory addresses into physical memory addresses. Although this extra process is not essential, it leads to a cleaner implementation of the firmware. We are currently investigating techniques that will allow the compiler to reduce or eliminate such overheads. This requires adapting the automata-based approach so that it can be applied selectively.

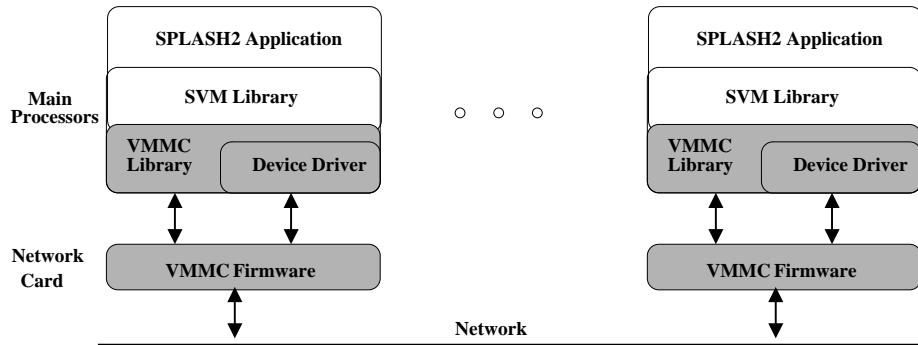


Fig. 6. Experimental Setup

TABLE I  
SPLASH2 APPLICATIONS

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
WaterNsqared	1000 Molecules	14.39	4.04
BarnesSpatial	8192 Particles	103.42	13.59
Volrend	head	314.98	4.76

### B. Application Performance

**Applications.** Figure 6 shows the experimental setup used to run the applications. The SPLASH2 applications [19] run on a cluster of SMP nodes using the VMMC software to communicate. These applications run on top of the Shared Virtual Memory (SVM) [5] library that, in turn, runs on top of the VMMC library. In the common case, the VMMC library bypasses the operating system and directly interacts with the VMMC firmware running on the Myrinet network card. The VMMC device driver that runs inside the operating system is needed for services like connection setup, interrupt processing, and pinning virtual pages in physical memory.

The applications in the SPLASH2 suite [19] 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 [20]. 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 I.

The Shared Virtual Memory (SVM) [5] 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 [1], we use a version of the SVM library

that uses that VMMC interface.<sup>5</sup>

**Platform.** All applications measurements were made using a cluster of four SMP PC. 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.

**Performance.** The performance of the SPLASH2 applications is presented in Figure 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:<sup>6</sup> *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 7 shows that the average performance difference<sup>7</sup> between the two versions is 3.5 %. Only one application incurs more than 5 % performance hit: FFT (10 %).

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 7 shows that

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

<sup>6</sup>The FFT performance when using the *vmmcOrigNoFastPaths* version is not shown because a bug in that implementation prevents FFT from running to completion

<sup>7</sup>For FFT, we use *vmmcOrig* to conservatively approximate *vmmcOrigNoFastPaths* performance.

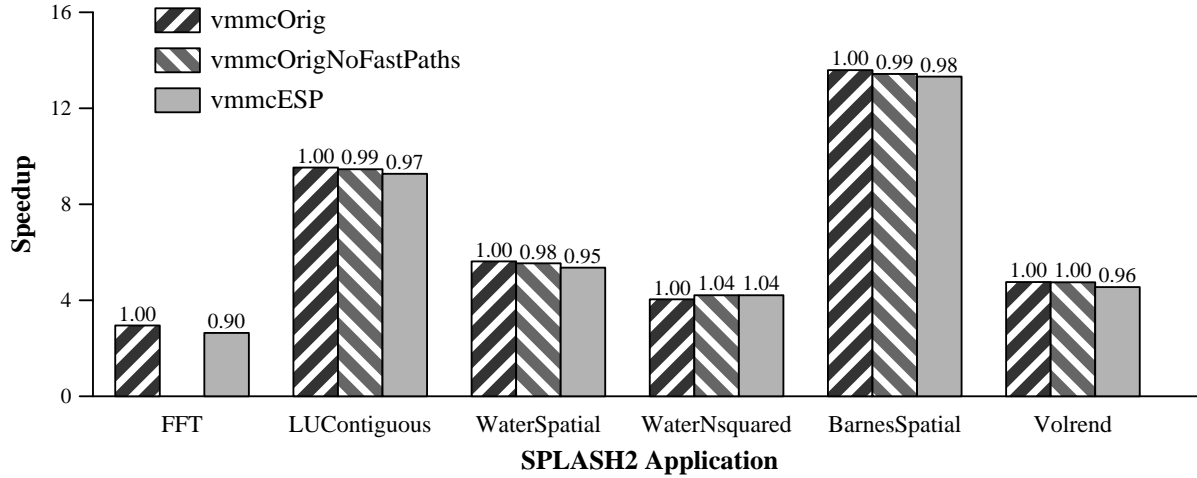


Fig. 7. SPLASH2 Application Performance: Speedup on 16 processors (four 4-way SMP nodes). The number on the top of each bar shows the relative speedup compared to *vmmcOrig*.

TABLE II  
APPLICATION CHARACTERISTICS.

The table shows the number of requests of each type that the firmware has to process for each of the applications. (Two applications have not been shown here because we have not been able to run them with the instrumented firmware.) A *Send* request is a request to transfer data from the current node to another node. A *Fetch* request is a request to fetch data from another node to the current node. Since *Send* requests of up to 32 bytes are treated differently on the local node, the *Send* requests are broken down into *Small* ( $\leq 32$  bytes) and *Rest*. The *Request Origin* column indicates whether the request originated at the local node or at a remote node. The remaining columns list the numbers of requests handled by each of the four nodes along with the total.

Application	Request Origin	Request Type	Node 1	Node 2	Node 3	Node 4	Total
LUContiguous	Local	Send (Small)	1590	1577	1577	1575	6319
		Send (Rest)	5	5	4	2	16
		Fetch	13147	7224	7258	7287	34916
	Network	Send	1580	1581	1579	1596	6336
WaterSpatial	Local	Send (Small)	14725	16067	727	1151	32670
		Send (Rest)	5067	5091	235	229	10622
		Fetch	49135	44838	47137	42540	183650
	Network	Send	462	1773	19697	21361	43293
BarnesSpatial	Local	Fetch	39719	46219	45021	52691	183650
		Send (Small)	17538	17074	17811	321	52744
		Send (Rest)	2035	2144	2140	20	6339
	Network	Fetch	1817	1893	1861	745	6316
WaterNsquared	Local	Send	606	227	219	58033	59085
		Fetch	799	687	753	4077	6316
		Send (Small)	5545	6407	6217	7296	25465
	Network	Send (Rest)	2714	2736	3097	3152	11699
Fetch		2653	2846	3700	4111	13310	
Send		9241	9489	9442	8992	37164	
WaterNsquared	Network	Fetch	3712	3251	3167	3180	13310

the fast paths have little impact on these applications. The largest benefit is observed in WaterSpatial (2 %).

Table II presents the number of requests of each type that have to be handled by the device firmware. The numbers suggest that the sensitivity to the applications to the firmware performance is correlated to the amount of remote fetch operations performed by it. In WaterSpatial and LUContiguous, they account for over 80 % of the traffic. These two applications also appear to be more sensitive to the firmware performance.

This can be due to one of two reasons. First, applications might be more sensitive to the performance of remote fetch than to the performance of remote send. This could be because it is more likely that a process is waiting for the data that is delivered by remote fetch than by a remote write. In the case of a remote fetch, the process requesting the data often waits for it to arrive. In the case of remote write, the data will be used by a process on a different node. It might not use it until much later. Second, the two versions of the firmware might have a significantly different performance for remote fetch operations under contention. We plan to investigate this in the future.

## VI. DISCUSSION AND FUTURE WORK

The SPLASH2 applications incur a modest performance hit (3.5 % on average) when using ESP to implement VMMC firmware. The 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 are more sensitive to other parameters like host overhead and interrupt costs [21], [22], [5]. The VMMC firmware does not affect the latter parameters as these are determined by code running on the host CPU (in the Operating System, the VMMC device driver, and the VMMC library).

The microbenchmarks involve one process running on each node while the applications run four processes on each node of the cluster. However, this does not account for the difference between the performance of the microbenchmarks and the performance of the applications. This is because all four processes on a node share a single queue to communicate with the firmware. As a result, the firmware is not even aware of the fact that it is dealing with multiple processes.

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 [5]. In this respect, ESP can help improve applications' performance by making it easier to experiment with and to add new functionality to the firmware.

Unlike the C version, the ESP version of the VMMC firmware did not require the programmer to manually perform any optimizations. This is because the ESP language is designed to support event-driven state machines and allows the compiler to optimize the programs effectively. We are considering a number of compiler optimizations that can further improve the performance of the generated code. First, the data-flow analysis can be extended to perform interprocess analysis. Second, the automata-based approach can be adapted so that it can be applied locally. This will allow the compiler to control the size of the generated code.

The fast paths implemented in *vmmcOrig* have a very small impact on the application performance even though they have a significant benefit in the microbenchmarks for several reasons. First, as explained earlier, applications are inherently less sensitive than microbenchmarks to the communication subsystem's performance. Second, 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 [5] that found the actual message latency for small messages when running these applications to vary between 3 times to 10 times longer than the latency measured by the microbenchmarks. Currently, we are exploring ways of supporting robust fast paths in ESP.

## VII. CONCLUSIONS

The ESP compiler compiles a concurrent ESP program to run efficiently on a single processor. It uses the process-based approach to switch between the various processes. A context switch is fairly lightweight and only involves saving and restoring the program counter. In addition, the language design allows the compiler to aggressively optimize the programs to generate efficient code.

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 measured 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.

ESP can indirectly help improve the performance of the applications. Device firmware is difficult to implement—debugging is slow and painstaking, and optimizing programs manually is error prone. As a result, adding new functionality to the device firmware requires significant programmer effort. However, applications' performance can sometimes be improved by adding new functionality to the device firmware [5]. ESP makes it easier to experiment with and to add new functionality to device firmware.

Additional compiler optimizations should further improve the performance of the ESP programs. First, the data-flow analysis can be made more effective by extending it to perform interprocess optimizations. Second, the automata-based approach can be adapted to combine processes selectively. Finally, support for robust fast paths can be added to ESP.

#### ACKNOWLEDGMENTS

The authors would like to thank Angelos Bilas for providing the Shared Virtual Memory library.

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.

#### REFERENCES

- [1] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li, "VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication," in *Hot Interconnects*, 1997.
- [2] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su, "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, 1995.
- [3] Sanjeev Kumar, Yitzhak Mandelbaum, Xiang Yu, and Kai Li, "ESP: A Language for Programmable Devices," in *Programming Languages Design and Implementation*, 2001.
- [4] Sanjeev Kumar and Kai Li, "Using Model Checking to Debug Network Interface Firmware," in *Submitted for Publication*, Available at <http://www.cs.princeton.edu/~skumar/>.
- [5] A. Bilas, C. Liao, and J.P. Singh, "Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems," in *International Symposium on Computer Architecture*, 1999.
- [6] Luca Cardelli and Rob Pike, "Squeak: A Language for Communicating with Mice," *Computer Graphics*, vol. 19, no. 3, pp. 199–204, July 1985.
- [7] G. Berry and G. Gonthier, "The ESTEREL synchronous programming language: design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, 1992.
- [8] Massimiliano Chiodo, Paolo Guisto, Attila Jurecska, Luciano Lavagno, Harry Hsieh, Kei Suzuki, Alberto L. Sangiovanni-Vincentelli, and Ellen Sentovich, "Synthesis of Software Programs for Embedded Control Applications," in *Design Automation Conference*, 1995.

- [9] Todd A. Proebsting and Scott A. Watterson, "Filter Fusion," in *Principles of Programming Languages*, 1996.
- [10] Claude Castelluccia, Walid Dabbous, and Sean O'Malley, "Generating Efficient Protocol Code from an Abstract Specification," in *SIGCOMM*, 1996.
- [11] R. Pike, "The implementation of newsqueak," *Software, Practice and Experience*, vol. 20, no. 7, pp. 649–660, 1990.
- [12] Stephen A. Edwards, "Compiling Esterel into sequential code," in *Design Automation Conference*, 2000.
- [13] G. Berry, *The Constructive Semantics of Pure Esterel*, Draft 3, 1999.
- [14] Rob Pike, "Newsqueak: A language for communicating with Mice," Tech. Rep. TR143, AT&T Bell Laboratories, Computing Science Dept., 1989.
- [15] David D. Clark and David L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," in *SIGCOMM*, 1990.
- [16] Mark B. Abbott and Larry L. Peterson, "Increasing Network Throughput by Integrating Protocol Layers," *IEEE/ACM Transactions on Networking*, vol. 1, no. 5, pp. 600–610, Oct. 1993.
- [17] Torsten Braun and Christophe Diot, "Protocol Implementation Using Integrated Layer Processing," in *SIGCOMM*, 1995.
- [18] Gerard J. Holzmann, "The Spin Model Checker," *IEEE Transaction on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997.
- [19] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *International Symposium on Computer Architecture*, 1995.
- [20] D. Jiang, H. Shan, and J. Singh, "Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-coherent Multiprocessors," in *Principles and Practice of Parallel Programming*, 1997.
- [21] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson, "Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture," in *International Symposium on Computer Architecture*, 1997.
- [22] Angelos Bilas and Jaswinder Pal Singh, "The Effects of communication Parameters on End Performance of Shared Virtual Memory Clusters," in *Supercomputing Conference*, 1997.

**Sanjeev Kumar** is a PhD student in the Department of Computer Science at Princeton University. He received a BTech in Computer Science and Engineering from the Indian Institute of Technology (Madras) in 1993 and an MS from Indiana University in 1995.

**Kai Li** is a professor in the Department of Computer Science at Princeton University. He received his PhD from Yale University in 1986. He is a member of IEEE Computer Society and a Fellow of the ACM.