Virtualizing ARM VFP (Vector Floating-Point) with Xen-ARM

Seehwan Yoo, Sung-bae Yoo, Chuck Yoo *

College of Information and Communication, Korea University, Seoul 136713, South Korea

A R T I C L E  I N F O

Article history:
Received 24 May 2012
Received in revised form 20 May 2013
Accepted 30 September 2013
Available online 11 October 2013

Keywords:
Virtual machine
Vector Floating-Point
Performance

A B S T R A C T

VFP is a Vector Floating-Point unit in ARM processors. It enables ARM processors to handle auxiliary floating-point operations with hardware, which has become an essential part for performance in recent mobile devices. However, mobile virtualization such as Xen-ARM does not support virtual VFP, so floating-point operations are very slow in mobile virtual machine. To overcome the performance limitation of floating-point operations in Xen-ARM virtualization, this paper presents a new virtual VFP so that applications can take the advantage of VFP hardware. With our virtual VFP, Xen-Linux reduces floating point operation latency up to one eighth from the existing software emulation version. In addition, result from mibench with virtual VFP presents 3–4 times higher throughput than that from software emulation. Furthermore, virtual VFP reduces code sizes and enhances power-efficiency.

1. Introduction

With the advent of 3D user interfaces (UI), games, graphics applications in mobile devices, the mobile devices demand more and more processing power. To meet the application demands, recent embedded processors introduced floating-point extensions for enhanced floating-point operations. For example, VFP (Vector Floating Point unit) is a hardware floating-point unit in ARM processors. VFP enables an ARM processor to directly execute floating-point instructions with the help of hardware.

At the same time, virtualization has drawn attention in mobile systems because reliability is one of primary concerns for mobile devices. Recent mobile devices handle private information such as banking account, present location, and business information at work, etc. Such information has to be securely protected from malicious applications. For reliability purpose, several mobile virtualization has been proposed [1–3], and Xen-ARM [2] is one of mobile hypervisors that supports ARM-based mobile devices. In Xen-ARM, multiple guest OSs run concurrently with isolation, so users can safely run trusted, reliable software in a completely separated domain from untrusted software domains. In addition, Xen-ARM presents small footprints (<250 KB), considering limited memory size of mobile devices, and it supports preemptive scheduling for real-time performance.

Although mobile virtualization provides better architecture in terms of reliability, its performance issue is still an obstacle for deployment. To enhance performance for 3D UI, games, and graphics applications, this paper focuses on VFP virtualization. Although VFP could efficiently enhance floating-point performance, it has not been properly addressed in Xen-ARM virtualization. Thus, mobile virtualization such as Xen-ARM is limited to use VFP, and heavily depends upon software emulated floating-point library, so far.

To virtualize VFP, we need to resolve the following problems. First, a Linux guest OS, Xen-Linux, cannot execute some instructions in the FP-handler due to the limited privilege level. Because Xen-Linux kernel is de-privileged in Xen-ARM, Xen-Linux cannot execute some instructions that access VFP registers.

Second, Xen-Linux cannot properly trap VFP instructions. Native Linux performs lazy FP context switching, which enables VFP in an on-demand manner. To lazily switch FP context, Linux disables VFP hardware whenever task switch occurs. Then, Linux enables VFP only when VFP is actually used by application, and switches FP context at the VFP-handler. However, Xen-ARM does not disable VFP although it switches guest OSs. Thus, Xen-Linux cannot properly trap the VFP instructions.

Third, Xen-Linux cannot switch virtual VFP context. This is because the old virtual VFP context might not belong to the current guest OS in virtualization. Note that Xen-ARM strictly prohibits to access the memory of a guest OS from another guest OS.

This paper presents a new virtual VFP with Xen-ARM virtualization. Firstly, we modify VFP instructions within a guest OS so that the Xen-ARM hypervisor can operate with VFP registers on behalf of the guest OS. Secondly, we modified Xen-ARM in order to properly trap VFP instructions. Thirdly, we modified Xen-ARM so that Xen-ARM can asynchronously save the old virtual VFP context of the designated guest OS.

* Corresponding author. Tel.: +82 2 3290 3639.
E-mail addresses: shyoo@os.korea.ac.kr (S. Yoo), sbyoo@os.korea.ac.kr (S.-b. Yoo), hx@os.korea.ac.kr (C. Yoo).

© 2013 Elsevier B.V. All rights reserved.
From Linux microbenchmarks (Imbench) result, we show that virtual VFP speeds up the unit floating point operations up to eight times over software emulated floating-point library. In addition, our virtual VFP presents 3–4 times higher throughput for mibench-FFT (Fast Fourier Transform) benchmark. The result presents near-native throughput of virtual VFP.

This paper is organized as follows. Section 2 briefly introduces VFP of ARM processors, and review ARM processor virtualization with Xen-ARM. Section 3 presents the problems in virtualizing VFP with Xen-ARM. Section 4 describes our virtual VFP, and Section 5 evaluates the virtual VFP. Section 6 exposes other considerations for virtualizing VFP, such as library issue and advanced SIMD support. Finally Section 7 provides some concluding remarks.

2. Background and related work

2.1. ARM VFP (Vector Floating-Point)

Recent cortex-a8 processor has FP context that consists of general-purpose registers\(^1\) and system registers \([4]\). General-purpose registers are used to handle floating-point operations, and those registers can be configured to operate with double word size (D0–D31), quad word size (Q0–Q15) or single word size (S0–S31). To use VFP, ARM defines additional instructions such as Floating-point MULTIply (FMUL), or Floating-point DIVision (FDIV). Note that the general-purpose registers are larger in size, so it is better to handle floating-point operations.

Five system registers present hardware feature set, and configure operating modes. Among system registers, FPSR is a floating-point status and control register. VFP registers operate only when FPEXC register’s en bit (FPEXC.en) is set to 1. It includes several bit flags such as Z, N, C (zero, negative), QC (saturation cumulative), NaN (Not-a-Number), RMode (rounding mode), etc. FPEXC is a floating-point exception register. FPEXC has only two-bit field ex and en. ex bit indicates whether there is a floating-point exception such as divide-by-zero. en bit indicates whether VFP is enabled or not. If VFP is not enabled, then any instruction that accesses VFP registers except for FPEXC will generate UNDEF (undefined instruction) exception. Note that FPEXC register is only accessible with ARMs privileged modes (not user mode).

2.2. FP (Floating-Point) context management in Linux

Similarly to ARM context, FP context has to be saved and restored according to the running process. Linux separately manages FP context from process context as shown in Fig. 1. In Fig. 1, processes 1 and 3 use VFP instructions, so they have FP context. On the other hand, process 2 does not use VFP instructions, so it does not have FP context. In addition, the figure shows ARM context switching between processes, and FP context switching between processes. Because process 2 does not have FP context, context switching occurs between process 1 and process 3.

There are two reasons why Linux manages FP context separately from ARM context. First, FP-context switching is more costly than normal ARM CPU context switching. Because FP context includes large register set, so every FP context switch is more costly than that of ARM context switch. Second, VFP is intensively used by the limited number of applications; thus, costly FP context switching is not required in many cases. Thus, Linux lazily saves and restores FP context so that it can efficiently avoid costly FP context switching \([6]\).

Fig. 2 describes an example of lazy FP context switching of Linux. In the figure, Linux firstly disables VFP by clearing FPEXC.en bit when process 1 is switched out. VFP is disabled until the processor executes the VFP instruction. In the figure, since process 2 does not have VFP instructions, process 2 is bypassed. At the first VFP instruction after context switching, the instruction generates UNDEF exception. In the figure, when process 3 executes the first VFP instruction, the instruction generates UNDEF exception. At the time, Linux traps the exception, and checks the source of the exception. In the VFP instruction trap handler, Linux enables VFP so that VFP registers can be used, as shown in the figure. Finally, Linux switches FP context so that process 3 can continue the execution. That is, Linux saves the old FP context for the last thread that executed VFP instruction, and restores the current FP context for the current running thread. In the figure, FP context of process 1 is saved, and FP context of process 3 is restored at this moment. In this way, FP context switching occurs only for the processes that perform VFP instructions.

2.3. Lazy FP context switching in Virtualization

Xen (x86), KVM and L4 are virtual machine monitors (VMM) that support lazy FP context switching. Xen defines lazy FP context switching interface for x86 \([7]\); however, it has not been supported in ARM architecture, so far. Although x86 and ARM could have the same programming interface, the implementation includes architecture-specific code that is largely different between x86 and ARM. KVM (kernel-based virtual machine) is another VMM that is supported by Linux kernel. KVM also considers to support lazy FP context switching \([8]\). However, it also focuses on x86 architecture, not ARM.

Recent study on L4 microvisor presents that L4 microvisor supports lazy FP context switching with ARM \([9]\). The paper presents
basic FP context switching cost in terms of code size and estimated cycle counts; however, virtual VFP performance has not been evaluated with real application workload, which has presented in this paper.

2.4. Xen-ARM: virtualization of ARM processor

Xen-ARM virtualizes an ARM processor. A virtualization system requires at least three privilege levels to run VMM, a guest OS, and application, respectively. Each of them needs to be protected from another, so different privileges are required. However, ARM processors have only two levels. To virtualize an ARM processor, Xen-ARM uses a different protection mechanism is used to overcome the limited privileged levels. The Xen-ARM hypervisor runs with the privileged mode, and guest OS and user applications run with unprivileged mode. To distinguish guest OS from user applications, domain access control in ARM is used. In Xen-ARM virtualization, guest OSs and user applications are specified as different domains, so they are exclusively accessible by the domain access control register.

In addition, a guest OS has to be modified (or paravirtualized) so that all virtualization-sensitive instructions can be trapped by the hypervisor. In an ARM processor, some virtualization-sensitive instructions do not generate exceptions although they are executed at user mode. For example, ARMv5 processors silently ignore all writes to privilege state in CPSR (current program status register). \(^3\) If the instruction is silently ignored, then the hypervisor cannot trap the instruction. Therefore, Xen-ARM adopts paravirtualization techniques that used in Xen such as batched mmu updates to mitigate performance overhead.

Moreover, Xen-ARM simplifies the execution modes by managing context with the only visible registers. ARM processor has complex ARM context than the other architectures such as x86. For example, ARM processor defines several execution modes [10], and each mode has its own dedicated register set (banked registers). When the hypervisor switches the running guest OS, all the banked registers have to be saved and restored accordingly. To do that, the hypervisor has to traverse all the execution modes and save all the banked registers, which makes context-switching between guest OS costly. To virtualize ARM processor, Xen-ARM manages context of a guest OS with the only visible registers because ARM-Linux uses only two modes (usr mode and sys mode) although there are additional execution modes.

3. Problems in virtualizing VFP with Xen-ARM

Xen-ARM virtualizes an ARM processor in order to run multiple guest OSs with a single physical ARM processor. However, Xen-ARM does not support VFP of ARM processor, so far. This section presents three problems that we faced to virtualize VFP with Xen-ARM.

First, some instructions in FP-handler of Xen-Linux kernel are not permitted to run with Xen-Linux. In Xen-ARM, the Xen-ARM hypervisor runs with privileged mode, and guest OS and user applications run with user mode. However, some ARM instructions within FP-handler of Xen-Linux require privileged level. For example, at the FP-handler, Xen-Linux enables VFP using MRX/MXR (Move Register to eXternal device) instructions that are only permitted for privileged mode. Because Xen-Linux runs with user mode, those instructions cannot be executed at Xen-Linux.

Second, a guest OS would not trap VFP instructions because the hypervisor does not disable VFP when it switches guest OSs. To trap VFP instruction, FPEXC.en bit has to be cleared previously. In native Linux, FPEXC.en bit is cleared at the guest Linux’s switch_to routine that performs context switching between processes.

In Xen-ARM, a guest OS is not aware of the scheduling at the hypervisor. In addition, Xen-ARM does not disable VFP when a guest OS is switched. So, the current guest OS cannot trap the VFP instruction. This incurs a problem specifically when two guest OSs run VFP applications at the same time. Fig. 3 illustrates why VFP is not trapped in Xen-ARM virtual machine. In the figure, VFP is not disabled although the running guest OS is changed from Guest OS 1 to Guest OS 2. Thus, Guest OS 2 cannot trap VFP instructions, and it does not invoke FP-handler, at which FP context switching occurs. That is, FP context switching is not triggered, although the current FP context has to be restored for the current guest OS. Consequently, process 2 would corrupt the context of process 1.

Third, FP context switching occurs asynchronously to the current guest OS, so old FP context might not belong to the current guest OS. Since a guest OS cannot access memory of another guest OS, old FP context cannot be saved from another guest OS.

Currently, Xen-ARM delegates all exception handling to the current guest OS. This is because all exceptions are synchronously generated to the current instruction execution. That is, exception is synchronous to the current task, and accordingly it is synchronous to the current guest OS. So, Xen-ARM redirects the execution to the current guest OS. It works well for most cases because Linux already has corresponding exception handler such as divide by zero, page fault, etc.

However, FP context switching occurs lazily, the exception handler of guest OS should save the old FP context that might belong to another guest OS. This complication comes from the asynchronous FP context switching to the current running guest OS, and this is notable difference from the other exception handling.

Fig. 4 illustrates the situation. In the figure, two guest OSs (Guest OS 1 and Guest OS 2) have two processes. Processes 1 and 2 run within guest OS 1, and processes 3 and 4 run within guest OS 3. Processes 1 and 4 use VFP instructions, and processes 2 and 3 do not use VFP instructions. When process 1 is scheduled out, VFP is disabled. After that, Xen-ARM schedules guest OS 2. Within guest OS 2, process 3 does not use VFP instructions, and process 4 uses VFP instructions. When process 4 executes VFP instruction, guest OS 2 traps the instruction. However, guest OS 2 cannot perform FP context switch because old FP context belongs to guest OS 1. Thus, although guest OS 2 traps VFP instruction successfully, it cannot save old FP context.

4. Virtualizing VFP

We virtualize VFP, overcoming the problems in Section 3. To virtualize VFP, the Xen-ARM hypervisor needs to separate VFP

---

\(^3\) Recent ARM processor such as Cortex-a8 generates UNDEF exception for the same behavior.
state so that each guest OS can have independent and transparent logical VFP. This section describes how we virtualize VFP.

4.1. Virtual VFP design

FP context switching is an essential mechanism that virtualizes VFP hardware. The FP context switching provides logical view of VFP hardware for guest OSs, and guarantees the exclusive access to physical VFP. The context switching between guest OSs are done at the hypervisor.

Our virtual VFP supports lazy FP context switching as well as non-lazy FP context switching. We will explain our implementation with lazy FP context switching and non-lazy FP context switching, one by one. In our lazy FP context switching, the hypervisor provides FP context when a guest OS on demand. Thus, the hypervisor allocates FP context of a guest OS when it actually tries to execute a VFP instruction. That is, FP context is not allocated if a guest OS does not use VFP instruction.

FP context switching also happens when it is necessary. The hypervisor loads and stores FP context among guest OSs that use VFP instructions. Moreover, it happens only when a guest OS explicitly uses VFP instruction. To synchronize FP context with the running guest OS, Whenever the running guest OS is changed, the hypervisor enables VFP so that it can generate trap for later VFP instruction. When a guest OS executes VFP instruction, the hypervisor checks whether the FP context belongs to the currently running guest OS, and perform FP context switching if necessary.

For an example, Fig. 5 explains lazy FP context switching among multiple guest OSs. In Fig. 5, there are three guest OSs, and each guest OS runs two processes. Guest OS 1 and guest OS 3 have processes that use VFP (processes 1 and 6), and guest OS 2 does not have any process that uses VFP. In the figure, process (ARM) context is switched by the scheduling of the hypervisor as well as guest OSs. On the other hand, FP context switching is not triggered by the scheduling of neither guest OSs nor the hypervisor. In lazy FP context switching scheme, some processes (processes 2–5) do not have FP context, and FP context switching is performed on-demand, among the processes that explicitly use VFP instructions. Thus, in the figure, FP context is switched only between processes 1 and 6. At the time, Xen-ARM saves the FP context of Guest OS 1, and restores the FP context of guest OS 3. By this way, virtual VFP efficiently reduces the number of unnecessary FP context switching.

Note that the hypervisor saves and restores the FP context that the guest OS remembers, not saving and restoring the FP context of specific processes. That means a guest OS might perform additional FP context switching. For example, in Fig. 6, Xen-ARM performs context switching between guest OS 1 and guest OS 2 (the scheduling sequence is OS 1 to OS 2 to OS 1, and the execution sequence of processes is processes 1 to 2 to 3 to 2 to 4 to 1 – left to right in Fig. 6). When guest OS 1 switches process, from processes 1 to 2, VFP is disabled. Then, hypervisor switches the guest OS from guest OS 1 to guest OS 2. Because guest OS 2 uses VFP, FP context is switched from FP context A to FP context B. Remember that FP context A is disabled. After the completion of guest OS 2, guest OS 1 is scheduled again by the hypervisor. The guest OS schedules another process 4 that also uses VFP, which generates VFP trap. Then, the hypervisor restores FP context A of guest OS 1. To save and restore FP context, the hypervisor has to enable VFP hardware, and restores all live registers. The FP context A belongs to process 1, and it is disabled; thus, the hypervisor resets VFP state to disabled, and returns the control back to the guest OS 1. Then the guest OS 1 triggers an additional FP context switch from FP context A to FP context C so that process 4 can use its own FP context C. Similarly, when process 1 is rescheduled, then guest OS 1 would perform FP context switching.

Our virtual VFP also supports non-lazy FP context switching. In non-lazy FP context switching, Xen-ARM saves and restores virtual VFP context at every guest OS switching. To save and restore FP context at every guest OS switch, Xen-ARM has to temporarily enable VFP even though VFP is not used by the guest OS. Then, Xen-ARM has to save and restore VFP context synchronous to the guest OS execution. In addition, Xen-ARM has to prepare FP context for all the guest OSs regardless of whether the guest OS uses VFP instruction or not.

4.2. Implementation

4.2.1. Modification at Xen-ARM

To support virtual VFP, the Xen-ARM hypervisor has to reset the control bit when the hypervisor scheduler changes the running guest OS. In the VM context switching routine, Xen-ARM invokes

Fig. 4. A guest OS cannot save old FP context of another guest OS.

Fig. 5. Lazy FP context management with Xen-ARM
the vfp_context_switching function. In the function, FPEXC.en bit is cleared by executing mxr/mxr instructions. Note that at both scheduling routine, FP context is not stored. Instead, simply FP unit is disabled for generating the later trap.

The actual FP context switching is triggered by the trapping of VFP instruction. To trap the execution of VFP instructions, the hypervisor exception handler has to be modified. In the undefined instruction exception handler, the Xen-ARM hypervisor inspects the exception registers in order to check whether the exception is due to the VFP instruction execution. If it is, the hypervisor checks whether FPEXC.en bit is cleared or not. If the bit is clear, Xen-ARM performs FP context switching by calling vfp_lazy_scheduling.

To switch FP context, Xen-ARM firstly enables VFP unit because FP registers are inaccessible when the VFP unit is disabled. To save the old FP registers, and to restore new FP registers, Xen-ARM has to enable the VFP unit, at the first hand.

Once it is enabled, Xen-ARM now has live VFP registers. Then, it checks the current FP registers belong to which guest OS. It is normally, the guest OS that lastly executed VFP instruction. Thus, Xen-ARM checks whether the currently running guest OS is different from the guest OS that has the FP context. If they are different, Xen-ARM stores FP context on a separate stack of the designated guest OS. To save FP context, we use STC instruction that moves VFP register values to memory, and we use LDC instruction for loading a new FP context.

To switch FP context, Xen-ARM firstly enables VFP unit because FP registers are inaccessible when the VFP unit is disabled. To save the old FP registers, and to restore new FP registers, Xen-ARM has to enable the VFP unit, at the first hand.

4.2.3. Modification at guest OS

To trap VFP instructions, Linux has a notification call chain, thread_notifier, that is invoked whenever a thread event is triggered. The vfp_notifier is registered at system initialization. Then, vfp_notifier is automatically invoked when a Xen-Linux performs context switching. In the routine, the guest OS invokes a hypercall to disable VFP. Then, the hypervisor disables VFP unit by clearing the FPEXC.en bit. We define the following macros in Listing 1 in order to maximize the reusability for the existing code base.

Listing 1: VFP macro

```
#define asm (_vfp_,_var_) asm ("mrc p10, 7, %0, "vfpreg (_vfp_),cr0, 0 @ fmrx %0," #_vfp_ "="r"(_var_):"cc":"_v:"__v;)
#define fmrx (_vfp_) (HYPERVISOR_vfp_op (CMD_FMRX, _vfp_##_XEN, _var_))
#define fmxr (_vfp_,_var_) (HYPERVISOR_vfp_op (unsigned int cmd, unsigned long val))
```

When a guest Linux makes a hypercall, Xen-ARM executes the corresponding mxr/mxr instructions. In the function signature, cmd specifies the direction of the operation: whether it copies value from coprocessor registers to memory or from memory to coprocessor registers. In addition, cpnum specifies VFP specific operation number such as FPEXC, FPSID in order to distinguish the target VFP control registers. Finally, the hypercall receives the value to assign the register.

4.2.2. Additional VFP hypercall

In addition to virtual VFP context switching, Xen-ARM has to provide an additional interface for VFP operation at guest OS. ARM processors provide mxr/mxr instructions in order to control VFP registers, which are only executed in privileged mode. In Xen-ARM virtualization, a guest OS is de-privileged, and runs in the user mode. Because Xen-Linux does not have enough privilege level to execute those instructions, Xen-Linux has to make a hypercall so that the Xen-ARM hypervisor can perform those instructions on behalf of Xen-Linux. The additional hypercall HYPERVISOR_vfp_op has the following signature.

- `asm linkage int HYPERVISOR_vfp_op (unsigned int cmd, unsigned int cpnum, unsigned long val)`

To trap VFP instructions, Linux has a notification call chain, thread_notifier, that is invoked whenever a thread event is triggered. The vfp_notifier is registered at system initialization. Then, vfp_notifier is automatically invoked when a Xen-Linux performs context switching. In the routine, the guest OS invokes a hypercall to disable VFP. Then, the hypervisor disables VFP unit by clearing the FPEXC.en bit. We define the following macros in Listing 1 in order to maximize the reusability for the existing code base.
5. Evaluation

5.1. Single guest OS case

In this section, we evaluate the performance of virtual VFP with Xen-ARM. Our hardware platform uses cortex-a8-based ARM processor with 512 MB main memory. We used Xen-ARM version that require paravirtualization.

In addition, we use Linux version 2.6.29. For Xen-Linux, we patched Linux so that it can run with Xen-ARM. In addition, we set the timer tick as 10 ms in order to minimize the overhead from high timer tick resolution.

To evaluate the performance of virtual VFP, we use three kinds of benchmark applications: Imbench [11], mibench [12], and x264 [13]. Imbench [11] is a popular Linux micro-benchmark that includes floating-point operations. It measures unit operation latency of addition, multiplication, and division with float, double data types, respectively. For additional workload, mibench is used to evaluate overall throughput. Mibench [12] is a benchmark suite for embedded systems, which consists of automotive, consumer electronics, network, office, security, and telecommunication applications. Among various application workloads, telecom-FFT (Fast Fourier Transform) is used. It generates FFT translation table, which performs considerable floating point operations. x264 [13] is a widely used video encoder for H.264/MPEG-4 AVC format. To compare the performance of our virtual VFP with software-emulated library (soft-fp), software has to be compiled with two different versions: (1) with vfp (2) with soft-fp.

At first, we compare the result from Imbench’s lat.ops. The benchmark measures the execution time for the following operations: float add/mul/div, double add/mul/div. The first and the second column show results with a Xen-Linux guest OS. The first column is the results from imbench when virtual VFP is used, and the second column is the results when soft-fp is used. The third column shows the result from native VFP. The results are presented in Table 1.

In Table 1, the result shows that our virtual VFP enhances the latency of unit floating-point operations. In the second row, latency for multiplication is reduced from 23.10 to 15.26 ns. In the first row, latency for division is reduced from 13.49 to 13.06 ns.

Table 1 Performance of virtual VFP.

<table>
<thead>
<tr>
<th>Type</th>
<th>Virtual VFP</th>
<th>Virtual soft-fp</th>
<th>Native VFP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Env.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>float</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>13.49</td>
<td>35.06</td>
<td>13.40</td>
</tr>
<tr>
<td>mul</td>
<td>15.26</td>
<td>23.10</td>
<td>15.02</td>
</tr>
<tr>
<td>div</td>
<td>50.16</td>
<td>126.51</td>
<td>49.64</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>13.62</td>
<td>48.42</td>
<td>13.39</td>
</tr>
<tr>
<td>mul</td>
<td>16.78</td>
<td>38.49</td>
<td>16.55</td>
</tr>
<tr>
<td>div</td>
<td>87.00</td>
<td>695.97</td>
<td>85.72</td>
</tr>
</tbody>
</table>

Table 2 shows the number of ARM context switching and FP context switching. We count the execution time in Xen-Linux is reduced from 267.4 to 84.6 ms. For medium size and for large size FFT table calculation, the execution times are reduced from 6619.8 to 1910.2 ms and from 47,859.8 to 13,204.6 ms, respectively.

For native Linux, mibench execution times are also reduced with the help of VFP. The execution times are reduced from 267.7 to 82.8 (small-FFT); from 6626.8 to 1892.6 (medium-FFT); and from 48,104.8 to 13,222.2 (large-FFT), respectively.

We can approximately derive performance gain by using virtual VFP over using soft-fp from the following Eq. (1).

\[
\text{Performance gain} = \frac{\text{Throughput with VFP}}{\text{Throughput with soft-fp}} = \frac{\text{Execution time with soft-fp}}{\text{Execution time with VFP}}.
\]

5.2. Two guest OSs case

Virtual VFP can be shared among virtual machines, so multiple guest OSs can take advantage of virtual VFP. To evaluate multiple VM case, we run two guest OSs, dom0 and dom1. dom0 runs mibench-FFT that used in the previous experiment. In the meanwhile, dom1 also runs long mibench-FFT as a background workload so that virtual VFP can be shared among dom0 and dom1. We count the number of context switching to present the behavior of our virtual VFP, and measure the execution time to show the performance of our virtual VFP.

5.2.1. Number of context switching

Firstly, we compare the number of ARM context switching and FP context switching under the following five different configurations. The first configuration consists of two domains, dom0 and dom1. Both dom0 and dom1 uses virtual VFP. The second configuration also consists of two domains, dom0 and dom1. In the configuration, dom0 runs mibench with VFP, and dom1 runs mibench with soft-fp. The third configuration is similar to the second configuration except that Xen-ARM supports non-lazy context switching. The fourth configuration runs two mibench within a native Linux. In the configuration, we run two mibench with VFP over a single native Linux. Finally, the fifth configuration runs two mibench, one with VFP and the other with soft-fp, over a single native Linux.

Table 3 shows the number of ARM context switching and FP context switching under various configurations. In the table, ARM CS stands for the number of ARM context switching, and FP CS stands for the number of FP context switching. We count the

---

Footnote: There are two Xen-ARM versions, one supports PV (paravirtualization), and the other supports HVM (hardware virtual machine).
number of context switching (ARM CS and FP CS) at Xen-ARM and Xen-Linux.

In the table, with the first configuration, ARM CS (Xen-ARM) is similarly presented with FP CS (Xen-ARM). This means that FP context switching mostly happens when guest OSs are switched. This is because both dom0 and dom1 use virtual VFP. Whenever a guest OS is switched, the guest OS requires its own FP context, so Xen-ARM performs FP context switching. In addition, ARM CS (Xen-Linux) is small because dom0 runs only mibench. FP CS (Xen-Linux) in the table is zero because only mibench uses virtual VFP, so there is no FP context switching within the Xen-Linux.

In the second configuration, the number of ARM context switching at Xen-ARM is as small as the first configuration. This is because dom0 uses virtual VFP, and the execution time is similar to that of the first configuration. On the other hand, there is no FP context switching in the second configuration. This is because Xen-ARM uses lazy FP context switching. Because only dom0 uses virtual VFP, VFP is only enabled with dom0, and no FP context switching is required. Note that when soft-fp is used, no VFP registers are used, so no FP context saving and restoring is required. Similarly to the first configuration, ARM context switching at Xen-ARM is small, and no FP context switching is observed within Xen-Linux.

In the third configuration, results are similar to that of the first configuration. In non-lazy FP context switching, FP context has to be switched whenever Linux switches two mibench processes. However, the numbers of context switching is much larger than that in the first configuration. This difference comes from the scheduling granularity of Xen-ARM and Linux. In the first configuration, Xen-ARM schedules guest OS, dom0 and dom1, whereas Linux schedules two mibench processes in the third configuration. Xen-ARM schedules guest OSes every 30 ms, whereas Linux schedules processes every 10 ms. Therefore, native Linux performs three times more context switching, in average.

The fifth configuration in the table shows the similar number of ARM context switch with the fourth configuration. However, there is no FP context switch because only one mibench uses VFP.

5.2.2. Execution time with multiple guest OS

Secondly, we measure the execution time of mibench to present the performance of virtual VFP. At dom0, we measure the execution time of mibench under diverse configurations. The results are averaged over 250 runs of execution. Time unit is presented in milliseconds.

The first, the second, and the third configuration consist of two domains, dom0 and dom1. In the first configuration, both dom0 and dom1 runs mibench with virtual VFP. In the second and the third configuration, dom0 runs mibench with virtual VFP, and dom1 runs mibench with soft-fp. The only difference between the second configuration and the third configuration is the use of lazy FP switching. The second configuration uses lazy FP context switching, but the third configuration does not. In the fourth configuration, both dom0 and dom1 run mibench with soft-fp. In the fifth

<table>
<thead>
<tr>
<th>Table 2</th>
<th>mibench performance of virtual VFP (milliseconds).</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT param. (wave, length)</td>
<td>Small (4, 8 K)</td>
</tr>
<tr>
<td>Xen-Linux v. VFP</td>
<td>84.6</td>
</tr>
<tr>
<td>Xen-Linux soft-fp</td>
<td>267.4</td>
</tr>
<tr>
<td>native Linux VFP</td>
<td>82.8</td>
</tr>
<tr>
<td>native Linux soft-fp</td>
<td>267.7</td>
</tr>
</tbody>
</table>

5.2.2. Execution time with multiple guest OS

Secondly, we measure the execution time of mibench to present the performance of virtual VFP. At dom0, we measure the execution time of mibench under diverse configurations. The results are averaged over 250 runs of execution. Time unit is presented in milliseconds.

The first, the second, and the third configuration consist of two domains, dom0 and dom1. In the first configuration, both dom0 and dom1 runs mibench with virtual VFP. In the second and the third configuration, dom0 runs mibench with virtual VFP, and dom1 runs mibench with soft-fp. The only difference between the second configuration and the third configuration is the use of lazy FP switching. The second configuration uses lazy FP context switching, but the third configuration does not. In the fourth configuration, both dom0 and dom1 run mibench with soft-fp. In the fifth

<table>
<thead>
<tr>
<th>Table 3</th>
<th>Number of ARM/FP context switching.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configurations</td>
<td>CS type</td>
</tr>
<tr>
<td>dom0: v.VFP, dom1: v.VFP</td>
<td>ARM CS (Xen-ARM)</td>
</tr>
<tr>
<td></td>
<td>FP CS (Xen-ARM)</td>
</tr>
<tr>
<td></td>
<td>ARM CS (Xen-Linux)</td>
</tr>
<tr>
<td></td>
<td>FP CS (Xen-Linux)</td>
</tr>
<tr>
<td>dom0: v.VFP, dom1: soft-fp (lazy FP switching)</td>
<td>ARM CS (Xen-ARM)</td>
</tr>
<tr>
<td></td>
<td>FP CS (Xen-ARM)</td>
</tr>
<tr>
<td></td>
<td>ARM CS (Xen-Linux)</td>
</tr>
<tr>
<td></td>
<td>FP CS (Xen-Linux)</td>
</tr>
<tr>
<td>dom0: v.VFP, dom1: soft-fp (non-lazy FP switching)</td>
<td>ARM CS (Xen-ARM)</td>
</tr>
<tr>
<td></td>
<td>FP CS (Xen-ARM)</td>
</tr>
<tr>
<td></td>
<td>ARM CS (Xen-Linux)</td>
</tr>
<tr>
<td></td>
<td>FP CS (Xen-Linux)</td>
</tr>
<tr>
<td>Native: VFP + VFP</td>
<td>ARM CS (Linux)</td>
</tr>
<tr>
<td></td>
<td>FP CS (Linux)</td>
</tr>
<tr>
<td>Native: VFP + soft-fp</td>
<td>ARM CS (Linux)</td>
</tr>
<tr>
<td></td>
<td>FP CS (Linux)</td>
</tr>
</tbody>
</table>
configuration, we run two mibench with VFP over a single native Linux to compare performance of virtual VFP with native VFP.

The result is shown in Table 4. The first row shows the results with the first configuration. With the first configuration, VFP is used by both dom0 and dom1. To preserve consistency, Xen-ARM switches virtual VFP context of dom0 and dom1, back and forth. In the table, time is doubled from single VM case in Table 2. This is because ARM CPU is virtualized, and CPU utilization is shared with dom1.

The second row of the table shows the results from the second configuration. VFP is enabled only when dom0 runs; thereby dom0 runs mibench with the help of hardware. As shown in the table, the execution time is comparably small as that in the first row.

The third row shows the results without lazy FP context switching. The result shows that medium-FFT and large-FFT execution times are slightly smaller than using lazy FP context switching in the second configuration. This is because of the complex relationship between unnecessary FP context saving (in non-lazy FP switching) and long exception handling (in lazy FP switching).

Without lazy FP context switching, every guest OS switching introduces overheads of saving and restoring FP context regardless of the use of VFP. On the other hand, when using lazy FP context switching, exception handling becomes longer to switch FP context. Despite the result, it is generally better to use lazy FP context switching because unnecessary FP context switching happens to all guest OSs that is not related with VFP. This is why lazy FP context switching is popularly used in Linux.

The fourth row shows the results with the fourth configuration. In the fourth configuration, dom0 cannot take advantage of VFP hardware; thus, the execution time is largely increased. Compared with the first configuration, the execution time is reduced when we use virtual VFP from 535.2 to 169.9 for small-FFT, from 13,284.3 to 3839.1 for medium-FFT, and from 95,991.3 to 26,530.2 for large-FFT, respectively. The performance gain ranges from 3.1 to 3.6.

The fifth row shows the results from a native Linux that runs two mibench with VFP. We can compare the result with that in the first row. Compared with the first row in the table, execution over a native Linux is slightly faster than that over a Xen-Linux. These differences (less than 1% in large-FFT case) are due to virtualization overhead.

The virtualization overhead is mainly due to TLB and cache flush at the guest OS switching. In native Linux, when a process is switched, then TLB and cache flush occurs only for the application region. However, with Xen-ARM, when a guest OS is switched, the entire TLB and cache flush occurs because all Xen-Linux have the same virtual memory layout. Thus, the switching cost between guest OSs is larger than the switching cost between processes. Therefore, results with Xen-Linux (in the first configuration) is worse than native Linux (in the fourth configuration) although the number of context switching at Xen-ARM is only one third of native Linux. The performance gain of VFP over soft-fp in native Linux are 3.22 for small-FFT, 3.49 for medium-FFT, and 3.65 for large-FFT, respectively. Note that the performance gain from virtual VFP is comparable to native VFP over soft-fp.

In summary, virtual VFP significantly reduce the execution time of application. The results show that virtual VFP also has significant performance enhancement under multiple guest OS environment.

With Xen-Linux, performance gain ranges 3–4, which implies that virtual VFP achieves 3–4 times higher throughput (with regard to floating-point CPU performance) over soft-fp with marginal virtualization overhead.

5.3. Context switching overhead and cache/TLB impact

Our virtual VFP introduces additional context switching time due to FP context management that did not exist in the current Xen-ARM. Because context switching time is usually very small, we measure the context switching time using performance monitoring unit that can accurately measure the hardware events at CPU cycle-level accuracy.

We measure the cycle count of context switching function of Xen-ARM. The results are averaged from thousands of context switches. Without our virtual VFP, the context switching time of Xen-ARM is 4656 cycles. With our virtual VFP, the context switching time is 7284 cycles when lazy switching is applied. The result shows that the context switching time increases about 50% due to save and restore general-purpose VFP registers. Note that in this case, VFP registers are saved and restored at every context switching even though any application does not use VFP hardware.

The context switching time with lazy fp switching is 5652 cycles, which is much smaller than non-lazy fp context switching scheme. The reason is that not all general-purpose VFP registers are saved and restored at every context switching. Therefore, the context switching overhead can be efficiently mitigated if we apply lazy fp context switching.

The overall result shows that our FP context switching increases overall context switching time. Although lazy FP context switching is used, the context switching time increases about 1000 cycles. That is mostly due to the overhead of touching VFP logic. Although VFP is not used actively, the hypervisor has to check whether the VFP is touched or not, and disables the VFP unit if it is enabled.

Regarding the cache and TLB effect, when a guest OS or the hypervisor switches ARM context, cache and TLB are flushed, accordingly. That is, cache and TLB are always synchronous to the currently running guest OS. Even though we use lazy FP context switching, no additional TLB and cache flush is required.

### Table 4

<table>
<thead>
<tr>
<th>Configurations</th>
<th>Lazy</th>
<th>small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>dom0 v. VFP</td>
<td>√</td>
<td>169.9</td>
<td>3839.1</td>
<td>26,530.2</td>
</tr>
<tr>
<td>dom1 v. VFP</td>
<td></td>
<td>172.9</td>
<td>3844.9</td>
<td>26,535.3</td>
</tr>
<tr>
<td>dom0 v. VFP</td>
<td>√</td>
<td>176.1</td>
<td>3834.8</td>
<td>26,474.1</td>
</tr>
<tr>
<td>dom1 soft-fp</td>
<td></td>
<td>535.2</td>
<td>13,284.3</td>
<td>95,991.3</td>
</tr>
<tr>
<td>dom0 soft-fp</td>
<td></td>
<td>168.4</td>
<td>3806.0</td>
<td>26,408.7</td>
</tr>
</tbody>
</table>
because they are already synchronous to the running guest OS. Thus, our FP context switching has limited impact on cache and TLB.

Yet, we further investigate the VFP impact on cache and TLB. Note that those overheads are difficult to measure statically because those overheads are very differently presented according to the run-time behavior. Thus, we separately observe the VFP data and instruction effect by measuring cache and TLB misses. As a representative application, we run FFT that is used in the previous section. To distinguish FFT-vfp and FFT-soft-fp, we compile the same application with two different options, enforcing enabling and disabling vfp hardware, respectively. FFT-soft-fp and FFT-vfp run with three parameter sets: small, medium, and large. During the application execution, we measure the miss count of cache and TLB for instruction and data, respectively. The result is shown in the following figures.

Fig. 8 shows the instruction cache and TLB misses, and Fig. 9 shows the data cache and TLB misses. Kindly note that Y-axis is log scale in both figures. In Fig. 8, cache and TLB misses show different trends regardless of the calculation parameters. In the figure, for instruction cache column, all dark gray bars are higher than light gray bars. This means that soft-fp presents more instruction cache misses than VFP. In soft-fp implementation, floating point emulation is implemented as C and assembly functions; thus, soft-fp requires more instructions to perform the same floating point operation than that using VFP. As the data set increases, the total number of executed instruction also increases; and the difference in ICache miss becomes larger.

On the other hand, instruction TLB column in the figure shows that all dark gray bars are lower than light gray bars. This means that soft-fp presents less (or similar) instruction TLB misses than VFP. The result is because soft-fp has smaller working set. To emulate floating point operations, the FFT-soft-fp extensively uses software library that runs with small loops, which is helpful for TLB management.

Fig. 9 shows the results with data cache and TLB. In Fig. 9, cache and TLB shows similar trends according to the parameters given to the application. For small parameters, dark gray bars in the figure are lower than light gray bars. This means that soft-fp presents less cache and TLB misses than VFP. A reason is that FFT-vfp presents different execution behavior from FFT-soft-fp. To maximize the advantage of hardware SIMD, FFT-vfp aggressively uses VFP registers. That is, when beginning the VFP operation, the application executes vpush/vpop instruction, storing all live registers, and loads data from memory (usually stack) to VFP registers at a time. Then, it calculates arithmetic operations using the VFP registers. After the calculation, it stores back the result values from VFP registers to memory. On the other hand, soft-fp loads and stores only live register in an on-demand manner. Thus, VFP leads to more DCache misses from those stack operations (push/pop) than soft-fp. Moreover, a compiler can optimize the number of memory operations of soft-fp, using register renaming, reordering the instruction streams, which is not possible with VFP registers.

On the other hand, for large parameters in Fig. 9, dark gray bars are higher than light gray bars. This means that VFP presents less cache and TLB misses than soft-fp. A reason is that soft-fp has much longer execution time; thus it leads to more context switching. The number of context switching observed during soft-fp (large) is about 300 times, and that during vfp (large) is about 170 times. Because DCache is flushed during the context switching, it would lead to more DCache miss.

5.4. Code size and energy efficiency

For embedded systems, code size and energy efficiency are also important criteria because most embedded systems have limited physical resources such as memory size, battery, etc. Our virtual VFP enhances performances not only in terms of execution time but also in terms of code size and energy efficiency. To measure the binary code size, we statically build mibench with math and libc libraries. The binary code size of mibench with VFP is 2,372,022, and that with soft-fp is 2,488,242 in bytes. The code size of mibench with VFP is about 100 K bytes smaller than that with soft-fp. The difference is about 5%, which includes megabytes of C library.

In addition, we compare the energy consumption of mibench with virtual VFP and soft-fp. To measure the energy consumption, we run large-FFT in Section 5.1. The measured energy consumption with virtual VFP is 0.98 Joules, and the energy consumption with soft-fp is 3.58 Joules during the entire execution. That is, the total energy consumption with virtual VFP is only about 27% of the energy consumption with soft-fp because it significantly reduces execution time. Concisely, virtual VFP is more energy efficient than soft-fp.
Furthermore, the power efficiency is also measured by the energy consumption divided by the entire execution times. The power consumption with virtual VFP is 65.83 milliWatt (mW), and that with soft-fp is 73.06 mW. That is, virtual VFP consumes 10% less power than soft-fp.

5.5. Multimedia application performance

To present viability of virtual VFP with more practical and widely used application, we additionally run multimedia encoding application, x264. We compiled x264 with and without VFP options, and measure the encoding time of YUV raw video data\(^5\) into mkv video format.\(^6\) The two-pass encoding time with VFP is 15.39 s, and the encoding time with soft-fp is 40.03 s. The result shows video encoding time with virtual VFP presents 2.6 times faster than that with soft-fp. Namely, virtual VFP can enhance the performance of applications in practice as well as benchmarks such as Imbench and mibench.

6. Discussion

Our initial result from mibench-FFT was not satisfactory. Performance gain was only about 5–10%. The low performance in initial result was due to the following two reasons. First, mibench-FFT is compiled with math library that does not use VFP. The main routine of mibench-FFT consists of two parts: (1) calculating sin/cos values, and (2) invoking fft_float. Our profiling result shows that sin/cos calculation takes 85% of the total execution time, and only about 15% of execution time is spent in fft_float. The routine that calculates sin/cos values intensively uses math library. However, it is not compiled with VFP option; thus, the performance gain is obtained only from fft_float routine that does not uses math library. To fully utilize VFP, we need to recompiled math library (in the glibc) with VFP option.

Second, Xen-Linux for Xen-ARM was not sufficiently optimized for performance for cortex-a8. At current, Xen-ARM supports Nvidiia’s Tegra250 that uses dual core cortex-a9-MP. To support cortex-a8, we re-implemented TLB, cache functions, which have considerable impact on performance.

Cortex-a8 processor supports advanced SIMD (Single Instruction Multiple Data), named neon extension. With VFP, neon supports faster vectorized operations for arithmetic calculations. Neon also uses VFP registers, that is enabled by FPEXC.en bit. Thus, our implementation accordingly supports neon as well as VFP, and presents improved SIMD operation performance although performance results are not presented in this paper due to the focused presentation.

Recently, ARM has announced a new processor, Cortex-A15 that supports hardware virtualization extension. The processor generates traps for virtualization-sensitive instructions, which was not possible in previous ARM processors. With the virtualization extension support, ARM-based mobile devices can run virtualization with smallest code modification. Xen, KVM and L4 are actively working on supporting virtualization extension to accommodate virtual machines on ARM-based computing platforms. With the processor, our PV modification effort can be further reduced because all VFP operations at guest OS generates trap, natively.

7. Conclusion

In this paper, we present a new virtualization of VFP with Xen-ARM. VFP provides enhanced performance for floating-point operations with the help of hardware extension. To fully utilize the advantage of VFP with virtualization, this paper presents a virtual VFP. Our virtual VFP provides up to eight times faster unit floating-point latency, more than three times higher throughput for floating-point FFT calculation, and improved power-efficiency, compared with the software emulation version.

Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MEST) (No. 2011-0029848).

References


Seehwan Yoo received the B.S. and the M.S. degrees in computer science from Korea University, Seoul, Korea, in 2002 and 2004, respectively. He is currently a Ph.D. candidate at Korea University, Seoul, Korea. His current research interests are in embedded and mobile systems virtualization and real-time scheduling.
SungBae Yoo received the B.S. degree in computer science from Konkuk University, Seoul, Korea, in 2012. He is currently a M.S student at Korea University, Seoul, Korea. His current research interests are in embedded and mobile systems virtualization and system architecture.

Chuck Yoo received the B.S. degree from Seoul National University, Seoul, Korea and M.S. and Ph.D. in computer science from University of Michigan, Ann Arbor. He worked as a researcher in Sun Microsystems Laboratory, from 1990 to 1995. He is now vice dean of graduate school of convergence IT, Korea University, Seoul, Korea. His research interests include virtualization of mobile devices and virtualization of networked systems. He served as a member of the organizing committee for NOSSDAV2001 and technical committee for EMSOFT 2008.