Virtual Machine Support for Parallel Language Runtimes

Jedidiah McClurg Kaicheng Zhang Yixi Zhang
{jrmcclurg, kaichengzhang2016, yixizhang2012}@u.northwestern.edu
Dept. of EECS
Northwestern University
Evanston, IL 60202

ABSTRACT

Palacios is an open-source virtual machine monitor (VMM) which targets the x86 and x86_64 architectures. Kitten is a lightweight operating system based on Linux which is designed to operate on supercomputer nodes. Palacios can be embedded into the Kitten OS to provide a virtual machine environment on these nodes. This offers a very flexible and extensible environment for running parallel applications. One family of parallel applications with increasing importance is that of programming language runtimes such as Racket. Racket offers two forms of parallelism called futures and places, and we seek to test the operation of these on top of a virtualized minimal OS such as Kitten. To accomplish this goal, we produced a modified Kitten OS with support for the system calls involved in these parallel constructs. We also customized Racket to run in the virtual filesystem (VFS) environment of Kitten. Finally, we successfully ran some parallel Racket examples on top of a Kitten OS virtualized in Palacios. In this paper, we present some of the techniques and difficulties involved in providing this support for parallel language runtimes at the OS-level, as well as some ideas for future customizations at the VMM-level which could offer better performance.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Experimentation

Keywords

Palacios, Racket, Kitten OS, Virtual Machine Monitor

1. INTRODUCTION

As we are entering the era of parallel computing, support to parallelism in both OS and programming languages are increasingly required. From supercomputers to personal machines, multi-core architectures are taking the place. Programs are expected to exploits this parallelism when possible, and accordingly, programming languages are beginning to offer built-in constructs which allow programmers to do this more naturally. Especially the functional programming languages, whose immutable style is more suitable to speed up parallel applications, are receiving more requests to support parallelism and make it easy to harness the power on supercomputers.

In this report, we talk about our work in this year's Virtualization course, which was intended to obtain the concrete answers to the question: what exactly is needed at the OS and (virtual) machine level to support the runtime system for such a language, by doing the following:

- Identify a minimalist environment which could conceivably support a parallel language runtime.
- Port the Racket language runtime to this minimal OS.
- Add necessary OS-level functionality to support Racket's parallelism constructs.
- Test the setup running on a virtual machine.

This provided us with an understanding of the features that are unique to parallel language runtimes system, and point us towards ways of improving the cooperation between the OS/hardware and the runtime system.

2. BACKGROUND

In this section, we will introduce the background information of the Kitten OS, Virtual Machines, Palacios virtual machine monitor, and Racket language with its parallelism models and implementations. Besides, we will talk about the current situation of running parallel applications written and run in Racket on top of Kitten. These are related to our work and also our future researches.

2.1 Kitten

Lightweight kernel (LWK) is developed to run on supercomputers and enhanced to provide necessary support for parallel applications and consumes the minimum amount of resources it needs. Kitten OS is one of the custom lightweight

kernel compute node operating system in this category, invented and actively updated by Sandia Corporation. Actually, it's not the first LWK developed by Sandia. Like its prior LWKs (SUNMOS, Puma, Cougar, and Catamount), Kitten is designed to target the extremely parallel and scalable distributed-memory machines within the tightly-coupled network, and especially focuses on the fast message passing and execution. However, what makes Kitten outstanding from its prior LWKs, is that it provides a more Linux-compatible user environment, with a smaller and more manageable code base, which also provides the foundation for any further functional extension[11].

We choose Kitten as the target OS to port Racket's runtime system, not only because Kitten has a more user friendly Linux-compatible environment, but also because Kitten supports parallelism naturally by linking its user-applications with the standard GNU C library, which includes the Pthreads implementation. More than that, since compilers usually build the OpenMP support on top of Pthreads, Kitten also has the functionality to run applications employ OpenMP. Unfortunately, Kitten lacks many essential supports to port Racket easily, such as the support to shared libraries, and the functionality of console read, etc. However, as a custom lightweight kernel, Kitten can be either run as the host OS with the embedded virtual machine monitor Palacios, or as the guest OS on top of typical Linux distributions[10].

2.2 VM

Supercomputers usually run thousands or millions of Linux kernels, all of which runs as a virtual machine on its distributed nodes. The virtual machine here is a shorten phrase of the system virtual machine, which could provides several advantages, such as, multiple OS environments can co-exist inter-independently on the same computer; virtual machine can provides an instruction set architecture which is different from that of the host machine; and because it's isolated from the host OS and other virtual machines, maintenance and recovery can be much easier and portable[8].

However, since a virtual machine does not access the hardware directly, its efficiency is typically lower than the host OS directly runs on the hardware. Either the multiple virtual machines concurrently running on the same hardware could expect stable performances, because its performance highly depends on the workload from the other virtual machines[8]. In these circumstances, certain virtual machine monitors need to come in the place to use proper techniques to solve these problems.

2.3 Palacios

If we want to run Kitten as a guest on top of Palacios, we need to first compile Palacios as an embedded kernel module into a host OS. However, the host OS needs only minimal functionality for Palacios to work[9], so it's also feasible to compile Kitten, or other certain custom lightweight kernel with Palacios as the host OS.

Even though Palacios does not naturally support to run with another VMM concurrently on the same machine, it can be configured to run solely on any certain physical core(s)[9], which makes it feasible to have Palacios run with other VMM at the same time. However, Palacios currently have some issues regarding hardware supports, so it raised some difficulties when we were testing our custom Kitten ISO as a guest on Palacios.

In our work, we compiled Palacios with Fedora 15 and tried running Kitten as guest on top of Palacios. As Palacios is designed as highly configurable, the embedding process requires no source code change to the host kernel, and only small part of changes to the configuration[9]. The detailed steps is included in the appendix and has been pushed to the project repo.

2.4 Racket

The power of Racket[5] is known for its extensive macro system[3], using which, programmers can write highly concise embedded or domain-specific languages. The pattern matching function[4] and the `define-type` macro[1] in the built-in `plai` programming language can support very complicated parsing work and user-defined data structures. These features simplify the writing of state machine dispatching and other many more expensive work processes. As we have experienced, using the Racket system to write a compiler with the possible employment of `typed racket` language[7] is a very straightforward work. The Lisp/scheme read function and the first contract system that works for higher-order values such as first-class functions would save many of the programmers' effort[6].

Because of the immutation feature, functional programming languages are very suitable for parallel applications. Therefore, it's very exciting if we can add the necessary functionalities for Racket's parallelism models to run on the minimal OS like Kitten. So that scientists will have the chance to write scientific-purpose parallel applications, with the powerful macro system and many other functionalities that have been mentioned above.

Racket uses a bytecode compiler[6], which means it first compiles the program into bytecode which runs on its own Racket virtual machine. And the bytecode could be further compiled using a Just-in-time compiler at the runtime. The Racket system also has an interpreter. The racket command accepts either console input via the interactive read-eval-print loop, or input from source files. This is the point we will focus on to add the necessary support for Kitten to run the racket interpreter and its parallel applications on top of it.

2.5 Futures

Racket, like many other scripting and dynamic languages, its runtime system was originally designed to run as a single thread. Even though Racket supports constructs for concurrency, it's implemented through co-routines. So namely, Racket used to have no parallelism support on hardwares with multiple processors[12].

The recently added feature, Futures, exploits the chance that to run the safe portion of the racket programs in parallel, with the minimal changes to Racket's highly enhanced runtime system[12]. Basically, you can use the `future` function to start a parallel computation and use the `touch` function to receive its result. The detailed format and description could be found on the Rackets documentation page.

Here we have the sample function shown in the paper of James Swaine, et al.[12].

```
(define (f x y)
  (let ((s (future (lambda () (+ x y))))
       ((d (future (lambda () (- x y))))
    (* (touch s) (touch d))))
```

Because the main function can proceed in parallel to a future, the function above could also be written as[12]:

```
(define (f x y)
  (let ((d (future (lambda () (- x y)))))
    (* (+ x y) (touch d))))
```

The later one is the template of the test programs that we use in our work. Our goal is to add the necessary functionalities that Kitten lacks to run this program in the interactive console environment.

The Futures implementation divides all kinds of operations, with consideration of its arguments as well, into different categories: safe, unsafe, and synchronized. It's based on the assumption that programmers, if not all, mostly share the same observation, that the portion could benefit the most from parallelism, has few side effects in the language implementation's internal state[12].

Futures builds its support for parallelism on Pthreads. Because Kitten has included the Pthreads implementation via the standard GNU C library, we don't have to add any additional functionality, except to make sure that threads run correctly on Kitten, in order to have Futures run as expected. Futures has done more work on the safety categorization, handling different categories of operations, and adjustment on memory management, but we are not going to include them in our report, since these detailed implementation theory is not related to our work to make it start to run[12].

2.6 Places

Last year, Racket added a new feature called Places to support message-passing parallelism. A sample fibonacci parallel function from the paper of Kevin Tew, et al.[13]. could be used to demonstrate the typical pattern to use Places.

```
(define (fib n) ....)

(define (start-fib n)
  (define p
     (place ch
        (define n (place-channel-get ch))
        (place-channel-put ch (fib n))))
  (place-channel-put p n)
  p)
```

The `place` function is used to create a place that run its body expressions, and the *ch* is a descriptor bound to a place channel.

Now, as we have the start-fib function, we can use it to start two fibonacci computationion in parallel[13]:

```
(define p1 (start-fib n1))
(define p2 (start-fib n2))
(values (place-channel-get p1)
        (place-channel-get p2))
```

This sample function is a demonstration of the `place` and `place-channel-get/put` function that we used in our test program. A more detailed description of Places could be found on the Racket documentation page.

As we have stated in the section of Racket, its define-type macro is useful to write very concise user defined data structures. So it is expected that Racket programmers will leverage this feature very frequently. If so, the traditional way to adding message-passing parallelism to a language by exploring the unix `fork()` primitive is this scenario, since that will limit the communication, and make the abstraction more difficult[13]. So the implementation of Places is directly inside the runtime system.

The architecture of Places' implementation, is that Racket starts with a single place, and with Places function, it can generate more than one places, and each of them has a local garbage collector. Besides transforming most global variables into thread-local variables, more work needs to be done in implementations of place channels and OS page-table locks[13].

As the work of Kevin Tew, et al.[13]. has shown, the Racket's garbage collector uses the OS-implemented `mprotect()` function to implement write barriers. However, as the latest Kitten code base shows, the `mprotect()` implementation in Kitten is not fully functional, which is simply stubbed out yet does not do any real work. So if we want to make Racket's Places function, or more generally, any function that will invoke Racket's garbage collector (parallel functions are supposed to be in this category, since we intend to run programs that are complicated and also memory consuming in parallel, as performance increasing is the goal), run as expected on top of Kitten, a more functional implementation of `mprotect()` is required[2].

2.7 Racket on Kitten

Racket does not run on Kitten out of box. As we'll give more detailed explanation in the following sections, porting Racket's runtime system has many difficulties according to the minimalism of Kitten inherited from all its prior LWKs.

Basically, Kitten does not support shared libraries, access to physical disks, console read, pipes, and many other system call implementations. All these are necessary for the running of Racket and its parallelism models on top of Kitten. As we present in the following sections, by increasingly adding the missing functionalities to Kitten OS, we have achieved a better understanding of parallel language runtime system, and its cooperation with the OS/hardware.

3. CONTRIBUTIONS

Our project succeeded in laying the groundwork for future study in the area of virtual machine support for parallel languages. The specific ways that we have contributed include the following.

- Compiled Racket as a static binary and connected it with the Kitten ISO image
- Added support for console input in Kitten
- Added support for instantiating a directory tree in the Kitten VFS (Racket collections)
- Implemented needed system calls in Kitten
- Worked on debugging Racket memory issues
- Implemented pipes in Kitten
- Tested Racket running in a Kitten guest on top of Palacios

The next section of this paper discusses these items in detail.

4. PORTING RACKET TO THE KITTEN OS

Before we can do experimentation regarding parallel language runtimes in the context of a virtual machine, it is necessary to port such a runtime to a guest OS which is suitable for testing. Thus, we focus on porting Racket to run on the Kitten OS. Overall, this is a relatively straightforward process, but there are many subtleties/difficulties along the way, so we present these here.

4.1 Static Racket Binary

The Kitten OS does not have a straightforward method for supporting shared libraries. Thus, it was necessary to build Racket from source and link statically. This involved some changes to the standard configuration, so we describe our build procedure here.

First, we clone the PLT Git repository

```
git clone git://git.racket-lang.org/plt.git
```

Then, entering the `plt` directory, we configure for a static build

```
./configure --disable-gracket CFLAGS="-g -O2 -static"
   LDFLAGS="-static -lpthread -lc"
   --enable-cgcdefault
```

(adding the "–export-dynamic" option to LDFLAGS is purported to make static builds work with dlopen(), which could allow the Racket `dynamic-require` to work, but so far that does not appear to be the case).

Now, entering the `src/racket` directory and executing `make` and `make install` will build the static binary `racket` in the `plt/bin` folder.

Kitten allows a binary having the name `init_task` to be packaged into the ISO image during the Kitten build. This binary will then be started by Kitten upon OS boot. Thus, we need to copy the racket binary from the `plt/bin` folder into the `kitten` folder and rename it to `init_task`. We edit the Kitten Makefile to do this automatically before each build.

4.2 Kitten Console Input

Kitten previously had console *write*, but no *read*. Since we wanted to use Racket interactively from a VM console, we found it useful to implement support for console input. The following diagram shows an overview of how this functionality is structured.

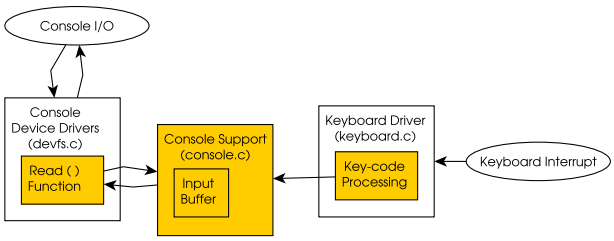

Figure 1: Console Input for Kitten

First, Keyboard interrupts are intercepted by the keyboard driver in `keyboard.c`. Previously, this driver simply printed an alert regarding the interrupt, so we had to add the key handler here.

The first step in the key handler is to process the key-code in order to determine the actual key character. We adapted this key-code processing functionality from the open-source GeekOS. After the key's corresponding character is determined, the character is sent to the console support module via our function `console_inbuf_add`.

The `console_inbuf_add` function in `console.c` allows a character to be added to the input buffer. At this point, special care must be taken to avoid problems when multiple threads are accessing this buffer. We do this by invoking the Kitten `spin_lock_irqsave` function before accessing the buffer, and then calling `spin_lock_irqrestore` after we have finished modifying the buffer. This ensures that other threads will wait for the buffer resource appropriately.

The actual character buffer is defined in `console.c`, and is implemented as a fixed-length character array FIFO, with pushed characters appended onto the end, and popped characters removed from the front. There are some important things to note regarding adding characters to the buffer. If a backspace character is detected, instead of adding it to the buffer, we actually wish to delete the previous character in the buffer. If the buffer has no characters, a detected backspace is simply discarded. If a newline character is detected, a flag is set so that waiting `read` requests can receive their data (this corresponds to the semantics of the UNIX `read` command). After a character is added to the buffer, it can be echoed to the console by using the Kitten `printk` function. Finally, if the character caused the buffer to expand, the add function calls `waitq_wakeup` to inform any waiting `read` operations that data has arrived.

Similarly, the `console_inbuf_read` function (also in `console.c`) first does a spin lock to obtain access to the character buffer. Once it acquires the lock, if it finds an empty buffer, the function releases the lock and adds the current task to a wait queue, to be woken up by the `console_inbuf_add` function when data is available, specifically when a newline character is detected. Once a full buffer is detected (i.e. a newline character is seen), this function returns up to n characters from the buffer, where n is the length argument to the `read` function. These n characters are deleted from the buffer by shifting the final $(length - n)$ characters to the left by n in the buffer. The `console_inbuf_read` function releases the buffer lock, and returns the requested characters.

To connect this buffer write/read functionality with the actual Kitten console, we modified the `devfs_console_read` function in `devfs.c` to grab data from the buffer using `console_inbuf_read`. This causes calls to the `read` system call to be satisfied from the buffer as expected.

4.3 Racket Collections and the Kitten VFS

Normally we would like to add a virtual disk to our Kitten VM setup, to prevent us from losing saved state each time the OS boots and runs Racket. Unfortunately, Kitten does not support physical disk drives, meaning that all filesystem access is via the memory-resident virtual file system (VFS).

This presents a problem in regards to the Racket startup, since Racket requires a large set of pre-compiled libraries in order to start up. We pruned these libraries down to a minimal set of about 1200 compiled racket files, in about 100 (sub)directories.

By specifying command-line options in the Kitten ISO configuration, we can cause Racket to start up on Kitten using the command "`racket -collects /tmp/racket`". This causes the program to search for these libraries at the path `/tmp/racket` on the system. In order to install the racket collections to this path, we first encode the entire racket collections directory structure into the Racket binary itself.

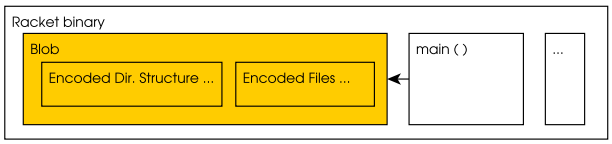

Figure 2: Racket collections embedded into the Racket binary

The format of this encoding is as follows:

```
numFolders(4byte)
numFiles(4byte)
rootFolderPath NULL
folderPath NULL
...
fileSize(4byte) filename NULL data
...
```

That is, the number of folder and files respectively occupy the first and second words of the blob. This is followed by a null-terminated string corresponding to the root folder path (in our case `/tmp/racket`). After this is a list of null-terminated folders corresponding to the collections directory tree, with each folder relative to the root folder. Finally, the file data is listed, with each file being encodes as a one-word file size, a null-terminated filename, and the binary file data.

The creation of this encoded blob is performed by a C program `makeblob.c` which we have added into the Racket build. The Racket Makefile is then modified in the following way.

```
racketcgc: blob.o libracket.a ...
$(CC) -o racketcgc blob.o main.o libracket.a ...

blob.o: racketblob
        objcopy -I binary -O elf64-x86-64 -B \
        i386:x86-64 racketblob blob.o

racketblob: makeblob
           ./makeblob ../../../kitten/collects \
           /tmp/racket racketblob

makeblob: makeblob.c
          gcc -o makeblob makeblob.c
```

This causes the Racket binary to be built and linked with the blob, which is now accessible within the Racket `main` function via the symbols `_binary_racketblob_start`, `_binary_racketblob_end`, and `_binary_racketblob_size`.

At the beginning of Racket's `main` function, we have added a check for the "`-expandblob`" command-line argument. If this exists, we delete it from the list of arguments (to prevent it from being passed to the subsequent Racket command-line processing), and call the `expand_blob` function. This function accesses the blob using the aforementioned symbols, and first iterates through the directories, creating them using the `mkdir` system command. On Kitten, this causes entries to be created in the virtual file system. After this, it iterates through the files, and uses the `fopen` and `fwrite` system commands to create and initialize the files with their original data. As an integrity check, we then use the `fseek` and `ftell` system commands to check that the newly-created file's size matches its size as specified in the blob.

4.4 Kitten Pipes

Kitten previously had no support for pipes. This is problematic, since Racket uses pipes to implement the message passing functionality of Places. Thus, we have implemented pipes, using an approach which ties in closely with the current Kitten support for regular files. This allows the file-specific system commands to work as expected for pipes as well.

The above diagram shows an overview of our added pipes functionality. A call to the system `pipe` command causes two regular files to be created in the VFS. These files have paths in `/proc` which correspond to the process ID of their

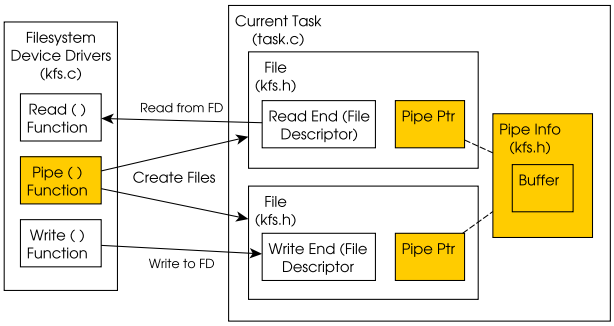

Figure 3: Our Implementation of Pipes for Kitten

parent process, and the index of the file descriptors. These files also differ from regular files in that their associated OS file data structure contains a pointer to a pipe character buffer. When `read` or `write` system calls refer to an file descriptor, the corresponding file data structure is checked to see whether it is actually a pipe. If so, the `read`/`write` are satisfied by reading/writing from/to the pipe buffer.

The implementation involved first modifying the `file` data structure in kfs.h, and adding a new structure for pipes.

```
struct pipe {
   char * buffer;
   int amount;        // num chars in buffer
   unsigned char eof; // whether an EOF is contained
   waitq_t buffer_wait;
   spinlock_t buffer_lock;
   int ref_count;
};

struct file
{
   struct inode * inode;
   ...
   unsigned char pipe_end_type;
   struct pipe * pipe;
   // the "pipe" ptr is NULL, it's not a pipe
};
```

We then modified the `kfs_init_stdio` function in kfs.c to create a directory corresponding to the current process in `/proc` on startup.

Also, if kfs.c, we modified the `sys_write` to first check whether the specified file descriptor is actually a pipe. If it is, the request will be satisfied from the pipe character buffer. This functionality is implemented using an approach similar to the approach we used in regards to accessing the console input buffer. First, we grab the lock for the pipe buffer. Once we have obtained the lock, the function enters a loop, writing as much as possible into the buffer and then waiting (using the `waitq` functions) until the buffer has been emptied enough to write more, until the entire request has been written.

Similarly, the `sys_read` function has been modified to check whether the requested FD is actually associated with a pipe. If so, the function grabs the pipe buffer lock, and then enters a loop, reading as much as possible from the buffer, copying into the result, and subsequently waiting for the buffer to fill, and then repeating these steps until the requested number of bytes has been read.

The `pipe_end_type` can be one of `PIPE_END_READ` or `PIPE_END_WRITE`, identifying the file as the read/write end of a pipe respectively.

4.5 Racket Memory Issues

Even with all the preceding functionality in place, Racket was segfaulting (and failing to start) on Kitten. We verified that the segfault occured on our host OS as well (using `gdb`), but it did not prevent Racket from starting on that machine. The following is a portion of the `gdb` backtrace.

```
#0  scheme_gmp_tls_unload (s=0x7ffff5e50 ...
#1  0x00000000005ef4b3 in done_with_GC ( ...
#2  0x0000000000062bee2 in garbage_collec ...
    switching_master=0, lmi=0x0) at ./ne ...
#3  0x00000000006318fe in allocate_slowp ...
    allocate_size=<optimized out>, gc=<o ...
#4  allocate (type=0, request_size=<opti ...
#5  allocate (type=0, request_size=<opti ...
#6  GC_malloc_one_tagged (s=<optimized o ...
#7  0x00000000005d7813 in scheme_make_st ...
    span=9, src=0xa17be8, props=0xa17be8 ...
#8  0x0000000000582eed in read_number_or ...
    port=0x7ffff5e51428, stxsrc=0xa17be8 ...
    radix=10, radix_set=0, is_symbol=1,  ...
    indentation=0x7ffff5e51578, params=0 ...
...
#48 0x0000000000058c6c3 in scheme_interna ...
#49 0x0000000000044ac22 in scheme_top_lev ...
    eb=<optimized out>, new_thread=0, dy ...
#50 0x0000000000058c49c in scheme_interna ...
    cantfail=<optimized out>, recur=0, e ...
    magic_sym=0x0, magic_val=0x0, delay_ ...
#51 0x0000000000058c5a1 in scheme_read_sy ...
    at ./../src/read.c:2361
#52 0x0000000000432888 in do_eval_string ...
    env=0x7ffff5e51bf8, cont=-2, w_promp ...
#53 0x00000000006375f4 in scheme_add_emb ...
    at ./../src/startup.inc:134
#54 0x0000000000416b4b in place_instance ...
    initial_main_os_thread=<optimized ou ...
...
```

It appears that this problem arises during garbage collection, but with only limited exposure to the Racket garbage collector, we were not able to pinpoint the problem. We did notice that the default Kitten page fault handler generated an unrecoverable error, so in an attempt to fix the problem, we modified the Kitten page fault handler to send the SIGSEGV signal to the offending task. Racket has a SIGSEGV handler, which should take care of the problem, but although this allowed Racket to initialize further, a subsequent segfault caused the task to hang. Finally, we switched from the Racket 3M garbage collector to the conservative garbage collector (CGC), and this solved the problem.

4.6 Other System Calls

Some other unimplemented system calls caused the Racket startup to fail, so we implemented these in Kitten:

- `getrlimit`. Specifically, the `RLIMIT_AS` flag was not supported, which corresponds to a request for the process address space size. We added basic support by causing this option to return `RLIM_INFINITY`, informing Racket that it has an arbitrary amount of memory space.
- `wait4`. Specifically, the process identifier (PID) "0" was not supported, which caused Racket's places initialization to fail. We fixed this by simply returning the ID of the current process.

4.7 Kitten / Racket Memory Limits

The Kitten VFS functionality causes created files to occupy a fixed amount of space. Thus, we found it necessary to increase the specified size in `in_mem_fs.c` from $512 * 8$ to $512 * 800$, in order to accomodate the largest Racket collection file. On a related note, we needed to increase the total kernel memory pool, since the VFS is stored in kernel memory. We did this by changing the kernel memory pool limit from 64 MB to 1024 MB in `bootmem.c`. We also needed to increase the initial task heap size from 32 MB to 200 MB by modifying `init_task.c`, and the initial task stack size from 256 KB to 1 MB.

5. TEST SETUP

In order to test the proper functionality of the Racket runtime operating within Kitten, we embedded Kitten as a virtual machine guest. Our host machine is running Fedora 15 (2.6.38 kernel), and has 4GB of RAM and an Intel Core i3 processor at 4x1.3GHz.

5.1 Kitten as a VirtualBox Guest

VirtualBox setup was fairly straightforward with respect to Kitten. We created a new virtual machine, and pointed it to the Kitten ISO image. We gave the machine 2048MB of memory (IO APIC enabled) and 1 virtual core (PAE/NX enabled). We also enabled the VT-x and nested paging options. For other settings, we gave the machine 6MB of video memory, and selected PIIX4 for the IDE controller type. We disabled networking and pointed the serial port at a local file.

Kitten can then be started by pressing "start", and output from Kitten is sent to either the console or the local file, depending on the output mode specified during the Kitten build

5.2 Kitten as a Palacios Guest

Kitten can also be set up to function as a guest within Palacios. The setup is similar to the above, except that Palacios uses an XML file instead of a graphical interface for specifying machine configurations. We created a file `kitten.xml` based on the `guest_os.xml` configuration that comes with Palacios. We used this to specify 1536MB memory and 2 cores. We also specified the location of the Kitten ISO for the boot disk image. Finally, we enabled IO APIC and the SERIAL and CHAR_STREAM virtual devices. Then, we built the Palacios VM image and executed the following command in the `utils/guest_creator` directory:

```
./build_vm kitten.xml -o kitten.img
```

At this point, Palacios can be started with the following commands:

```
sudo insmod v3vee.ko
sudo v3_mem 2048
sudo v3_create build/guest_creator/kitten.img test
sudo v3_launch /dev/v3-vm0
```

Currently, VGA output does not seem to work properly in Palacios, but if Kitten uses the serial port for output, we can use the following command to monitor the port:

```
sudo v3_stream /dev/v3-vm0 stream1
```

6. RESULTS

We succeeded in running Racket interactively and non-interactively on top of a Kitten VirtualBox (or Palacios) guest. We confirmed that our modifications allow futures and places operate as expected by running the following simple Racket demo programs.

This is the file `test.rkt`, which demonstrates some basic Racket output functionality.

```
(let ([x 12345])
   (begin (print x)
   (begin (printf "\nHello from Racket\n")
   "This is the return value\n")))
```

This is the file `test2.rkt`, which demonstrates some simple usages of futures/places

```
#lang racket
(let ([f (future (lambda () (+ 1 2)))])
    (print (list (+ 3 4) (touch f))))

(let ([pls (for/list ([i (in-range 2)])
   (dynamic-place "tmp/racket/place-worker.rkt"
      'place-main))])
   (for ([i (in-range 2)]
         [p pls])
      (place-channel-put p i)
      (printf "$>>>$Place message: ~a\n"
         (place-channel-get p)))
   (map place-wait pls))
```

This is the file `place-worker.rkt` used by the dynamic place example

```
#lang racket
```

```
(provide place-main)

(define (place-main pch)
  (place-channel-put pch
      (format "Hello from place ~a"
         (place-channel-get pch))))
```

7. FUTURE WORK

7.1 Memory sharing among multiple VMs

It is very common to deploy multiple identical virtual machines on one host executing similar jobs. Our stack of kitten and racket gives a good example. When deployed, we can create a virtual machine for each user needs a racket enviroment. In such configurations, there are many data are identical in the virtual machines. Typical cases are the guest OS's code and loaded library. So it can save some memory space if we share these portion of data among multiple virtual machines. Also, sharing memory can also improve cache ultiliaztion and enhance the overall performance.

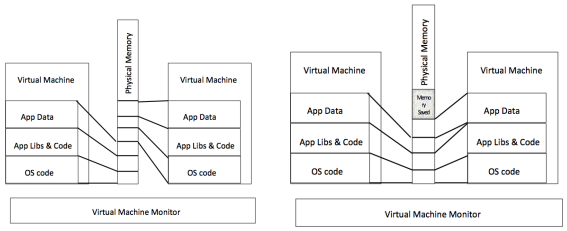

Figure 4: How shared memory helped system

7.1.1 Identifing sharable memory

The challenge of sharing memory among VMs is acutally its first step. Actually it is very hard to dynamicly identify sharable memory from the starting of VM boots up. Because a given memory will be written several times before it become stable. For example a memory page which turn out to be used as a code page for a common library, will be first initialized by the guest OS, and loaded library data on it, and rewrited several times to link the library. If the identification is implemented dynamically, it will first identify such page private and then shared some time after the page become stable.

To avoid this challenge, we propose to start VMs from a checkpoint. We have already implemented start racket environment automatically after kitten boot up. We could make a checkpoint after the system is boot up and start other virtual machines from this checkpoint. Thus from the starting point, the guest system has load a bunch of necessary library for sharing.

When we have a started VM with libraries loaded, we need to identify the memory for share. There are several different appoarches. The first one is simply mark all the guest memory space as shared and then do a copy-on-write to make the page private when guest system trying to write that memory space. However, the majority of the memory is either private or unused. So it will add much overheads with the introdution of copy-on-write mechanism. the second one is by human analysis. We could manually mark up the memory pages capable for sharing. This appoarch can be very accurate and high efficiency when executing. However, it will take a lot of human work and not practical when we are doing on other type of guest systems. The third appoarch is mark it with the knowledge of guest systems memory mapping. Since the VMM has a complete knowledge of the guest's page table. It can walk through guest's page table and make decisions about which page can be shared and which can not. There are few problems need to be addressed here as well. The first one is that one guest system may have multiple page tables, the other is that guest OSs may map the whole physical memory writable for its own use. However, just adding the pages that mapped read-only to user space or kernel in at least one page table entry should be accuracy enough.

7.1.2 Sharing the memory among multiple Virtual Machines

We propose to share the memory using Copy-on-write mechanism. Copy-on-write is commonly used on operating systems to fork new processes and share common data. Here we use it similarly in VMMs to share data among multiple virtual machines.

When VMM is creating a new VM from a checkpoint, if a memory page it is going to allocate is identified as sharable, it will first copy the data to it and then map it read-only to guest system and mark it as a copy-on-write page. If a new vM from the same checkpoint is created, the VMM will simply map the page to the same physical page with read-only privilege and marked as copy-on-write. Thus the data in this physical page is shared among the VMs.

In most cases, the shared pages will not be written by the VMs. However, if one of the shared pages is going to be written by one of the VMs, the control will be passed to the VMM. The VMM can then copy the data of this shared page to a new page, and map it to the VM with write previlege to replace the shared page. Note that the shared page is still shared among other VMs except the VM trying to write it. Since the shared pages is supposed not to be written by guest systems, this data copying should not introduce a significant overhead.

7.2 Ballooning

Racket's runtime environment will do garbage collection, which means it will run with a tight memory usage. Therefore, most of its runtime it will be far less memory usage than the total memory size given by the VMM. Such gap will event be greater if there are multiple VMs running simutanusly. Ballooning [14] is designed to solve this problem by notifying VMM unused memory.

The idea of ballooning is let guests talks with host about the actual memory it is using. However, in order to let ballooning portable to variaty of operating systems and easy to implement, ballooning is implemented as an kernel process that require physical pages(seen by the guest) using guest OS's interface and never use the physical pages for them. Thus can reduce the amount of work when implementing ballooning in new operating systems and leave the problem simple. The ballooning process will require more physical

pages when the guest system runs at a low memory usage and release them if the guest sytem goes high. The ballooing process will talk with the VMM about the address of the pages it get from the guest OS and the VMM thus can mark those pages as free and use them in other places.

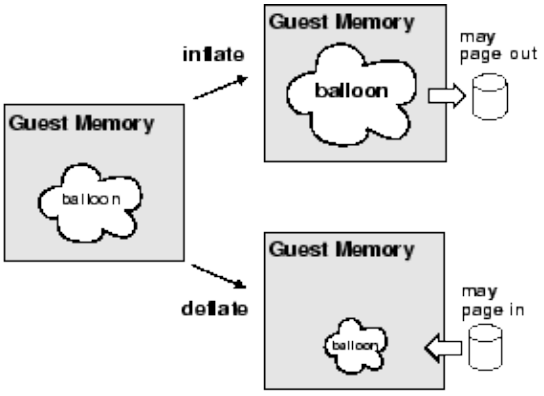

Figure 5: How ballooning adjust the memory usage (http://static.usenix.org/events/osdi02/tech/full_papers/waldspurger/waldspurger_html/node6.html)

Note that it is still possible that the ballooned physical page is touched by guest system. Therefore VMM should handle this kind of cases. When the guest system try to touch the ballooned physical page, the host machine will generate a fault that could be handled by VMM. Thus VMM can now map a new physical page for the guest system and return to the guest.

8. CONCLUSIONS

We have demonstrated that it is possible to port a parallel language runtime to a minimal OS. More generally, we have demonstrated that parallel language runtimes can be supported with only a minimal set of OS functionalities, namely threads and pipe, and etc. We have also shown that a parallel language runtime can run successfully in a virtual machine environment like Palacios. The details we have presented are a useful (and necessary) first step in investigating the interaction between such a language runtime and the (virtual) machine.

9. ACKNOWLEDGMENTS

We would like to thank the EECS 441 teaching assistants for their help regarding configuration of the tools and development environment during the project. Secondly, we would like to thank Kevin Pedretti for his helpful advice regarding the Kitten OS. Last but not least, we especially wish to thank Professor Peter Dinda for developing/teaching this useful class, for being readily available to answer all of our questions, and for offering us the opportunity to work on this project. It has been a very educational experience, and allowed us to develop some skill in programming at the OS- and VMM-level.

10. REFERENCES

[1] define-type, 2012. http://docs.racket-lang.org/ts-reference/special-forms.html?q=define-type&q=macro&q=places#(form._((lib._typed/racket/base..rkt)._def type)).

[2] Kitten release, 2012. http://www.cs.sandia.gov/web1400/1400_download.html.

[3] macro, 2012. http://docs.racket-lang.org/guide/macros.html?q=macro&q=places#(tech._macro).

[4] match, 2012. http://docs.racket-lang.org/reference/match.html?q=macro&q=places#(form._((lib._ra

[5] Racket, 2012. http://racket-lang.org.

[6] Racket (programming language), 2012. http://en.wikipedia.org/wiki/Racket_(programming_language).

[7] typed racket, 2012. http://docs.racket-lang.org/ts-guide/index.html?q=define-type&q=macro&q=places.

[8] Virtual machine, 2012. http://en.wikipedia.org/wiki/Virtual_machine.

[9] J. Lange, P. Dinda, K. Hale, and L. Xia. An introduction to the palacios virtual machine monitor–version 1.3. Technical report, Northwestern University, Nov. 2011. http://www.v3vee.org/palacios/palacios-1.3-tr.pdf.

[10] K. Pedretti. Kitten: A lightweight operating system for ultrascale supercomputers, 2011. https://software.sandia.gov/ ktpedre/kitten_overview.pdf.

[11] Sandia Corporation. Kitten lightweight kernel, 2011. https://software.sandia.gov/trac/kitten.

[12] J. Swaine, K. Tew, P. Dinda, R. B. Findler, and M. Flatt. Back to the futures: incremental parallelization of existing sequential runtime systems. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2010.

[13] K. Tew, J. Swaine, M. Flatt, R. Findler, and P. Dinda. Places: adding message-passing parallelism to racket. In *Proceedings of the 7th symposium on Dynamic languages*, pages 85–96. ACM, 2011.

[14] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, Dec. 2002.

APPENDIX

A. DELIVERABLES

All of our code has been pushed to the repository. We have maintained patches containing the work we did in the Kitten and Racket trees. For Kitten, the following Mercurial patches can be applied sequentially starting from the 1059:c0b5e0d310e0 changeset:

1. 01_console_read.patch
2. 02_syscalls.patch
3. 03_pipes.patch
4. 04_limits.patch
5. 05_more_limits.patch

For Racket, the following Git patches can be applied sequentially starting from the 98e06248b50ad35420970120fe70bb8ea7423f9c commit:

1. 0001-Some-changes-for-static-compilation.patch
2. 0002-Tools-for-packaging-extracting-a-filesystem-blob-for.patch
3. 0003-Bug-fixes-and-documentation-for-the-Kitten-blob-func.patch
4. 0004-Added-support-for-packaging-extracting-collections-f.patch

Other items pushed to the repository include the following:

- Racket test files
  1. place-worker.rkt
  2. test.rkt
  3. test2.rkt
  4. test3.rkt
- Detailed setup instructions
  1. kitten_setup.txt
  2. racket_setup.txt
  3. palacios_fedora_setup.txt