Towards RTOS: A Preemptive Kernel Basing on Barrelfish*

Jingwei Yang, Xiang Long, Xukun Shen, Lei Wang,
Shuaitao Feng, and Siyao Zheng

Beihang University, Beijing 100191, PRC
{yangjingwei,fengshuaitao,zhengsiyao}@les.buaa.edu.cn,
{long,xkshen,wanglei}@buaa.edu.cn

Abstract. Multi-core or even many-core systems are becoming common. New operating system architectures are needed to meet the ever increasing performance requirements of multi-/many-core based embedded systems. Barrelfish is a multi-/many-core oriented open-source operating system built by ETH Zurich and Microsoft Research in a multi-kernel style. Every kernel runs on a dedicated core with local interrupts disabled, which may result in a failure in real-time systems. To make it preemptive, new kernel state and private kernel stack are added. Capability is the central mechanism of Barrelfish, lots of analyses and modifications have been done to make the duration interrupts are disabled as short as possible when kernel objects in capability space are accessed. As a result, meaningful real-time performance have been obtained. Combined with the inherent scalability of multi-kernel design in Barrelfish, the work in this paper could be a useful reference for multi-/many-core based embedded systems.

Keywords: embedded system, real-time, multi-/many-core, multi-kernel, preemption.

1 Introduction

Multi-/Many-core based embedded systems are widely used in many fields such as network routers, automobile electronics, large scale complex control systems and so on. It's believed that processors with hundreds or even thousands of cores(hard-threads) will be used in the near future[1][2][3][4]. As a result, operating system could become a bottle-neck for computers with large number of cores because of the poor scalability[5][6][8][19]. Many-core based embedded systems may suffer more from operating system[13][18]. Predictability is often one of the most important issues faced by embedded systems, especially for hard real-time systems[9][10], multi-/many-core processors make it a great challenge for the system to be predictable. Under shared memory programming paradigm, multi-/many-core based embedded systems often perform poorly with respect to scalability and predictability because of reasons as following:

* The work is supported by the National High Technology Research and Development Program(No. 2011AA01A204).

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 47–61, 2013.

1. scalability:

   - ping-ponging and false-sharing effects of cache-coherence increase system overhead as the number of cores increases.
   - the increase of communication speed among cores and memory modules doesn't quite follow the increase of cores.
   - accesses to shared variables protected by locks must follow a serial pattern although lots of cores exist in the system.

2. predictability:

   - optimized processor designs such as super-scaler architecture, pipelines, and branch prediction make CPI(Cycles Per Instruction) varies.
   - the complex cache unit and poor associated control capability make it hard to predict the latency of cache access.
   - it's difficult to control or predict the precise time when each core gets access to the shared bus and memory modules.
   - the time it takes to access a memory unit is greatly affected by when and where the transaction occurs.
   - prediction of the time needed to finish access to shared memory space in a multi-/many-core environment is also a challenge.

2 Multi-/Many-core Oriented Operating System

To take up the challenges mentioned above, solutions of both hardware and software are provided. In this paper, a software solution is discussed. Barrelfish[5] is a multi-/many-core oriented open-source operating system built by ETH Zurich and Microsoft Research. The multi-kernel philosophy of Barrelfish mainly focus on the scalability of operating system on multi-/many-core platforms, while it also provides excellent predictability for real-time systems. Each kernel in Barrelfish runs on a dedicated core(hard-thread) and shares almost nothing with others except for some necessary global variables. Lots of factors that result in poor scalability and predictability can be avoided. The kernel schedules and runs dispatchers, which are implementation of scheduler activations. It follows a message passing programming paradigm in Barrelfish kernel, inter-dispatcher communication is realized by sending messages[7].

The kernel is also called CPU driver[5], and it provides only a limited number of facilities such as scheduling, interrupt management, capability[16], memory mapping, inter-dispatcher communication and so on. Barrelfish uses the microkernel model, most of the architecture independent modules(memory server, monitor, device driver etc.) are moved up to user space, and the whole system can run on a heterogeneous multi-/many-core system with the support of CPU divers[6]. This is good for embedded systems using both CPU and DSP. Explicit message-passing, user space drivers and system services make it easier to model the timing characteristic of Barrelfish, which is meaningful for real-time

applications[12]. What's more, every domain can run an application specific run-time systems or even operating systems in user space, to better satisfy the particular design constraint of application.

Despite so many characteristics suitable for multi-/many-core real-time system, Barrelfish is not meant for it particularly. Preemption[11], which is an important feature in most real-time operating system, is not supported in Barrelfish. Functions in kernel control path such as system calls, and interrupt handlers run in a single core environment with interrupts disabled until it returns to user mode, which may prevent dispatchers and interrupt handlers with higher priority from preempting current kernel control path. What's more, every kernel function shares the same kernel stack in a time-division manner, context switch for dispatcher in kernel mode will arise an error.

3 Analysis on Interrupt Latency

Interrupt latency is one of the most important performance indexes in real-time system. The time it consumes before handling an interrupt is affected not only by the hardware unit in processor, but also by the duration when interrupt is disabled. The WCET(Worst Case Execution Time)[12] of interrupt latency depends not on the duration interrupt is allowed, but on the duration interrupt is not allowed. As all of the kernel control path of Barrelfish execute with interrupt disabled, there is lot of space to improve the interrupt latency.

Interrupt latency consists of hardware latency and kernel latency. L_H stands for hardware latency and L_K stands for kernel latency, then the whole interrupt latency can be represents as:

$$L = L_H + L_K \tag{1}$$

Usually, hardware latency depends on the interrupt logic of processor, which is supposed to be constant. Kernel latency is the difference between the time interrupt is detected by processor and the time kernel begins to handle it. In real-time system, instruction steam executed by every processor core can be divided into N sections(see Figure 1), where $1 \leq N < \infty$, every section consists of two subsections: kernel subsection and user subsection, which mean instructions run in kernel mode and instructions run in user mode respectively. We assume that the time it consumes to execute N sections of instructions is T, and the time needed to execute the $i_{th}(1 \leq i \leq N)$ section is Ts_i, which equals $Tk_i + Tu_i$, where Tk_i stands for kernel subsection execution time, and Tu_i stands for user subsection execution time.

When interrupt occurs, the probability it disturbs the i_{th} user subsection and kernel subsection are Tu_i/T and Tk_i/T respectively, so the probability it disturbs the i_{th} section is $(Tk_i + Tu_i)/T$. In user subsection, interrupt can be handled immediately if interrupt is enabled, no matter the kernel is preemptive or not. In this situation, the kernel latency is 0 in theory.

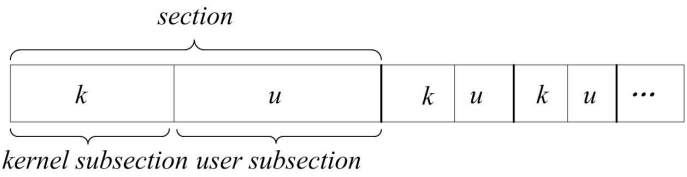

Fig. 1. Sections of instruction stream

3.1 Non-preemptive Barrelfish

In kernel mode of Barrelfish, an interrupt which happens in the i_{th} kernel subsection won't be handled until the system returns to user mode, so the kernel latency is the time it takes to finish the kernel subsection after interrupt has been detected. In every section, the instant at which interrupt occurs is supposed to conform to uniform distribution, and handling of every interrupt happened in kernel mode experiences an unique latency while handling of interrupt happened in user mode experiences no latency, then the value of kernel latency can be described by the following cumulative distribution function:

$$Fs_i(x) = \begin{cases} 0, & x < 0 \\ x/Ts_i + Tu_i/Ts_i, & 0 \le x \le Tk_i \\ 1, & x > Tk_i \end{cases} \tag{2}$$

During the whole execution time of T, which consists of N sections, the kernel latency can be described as:

$$F(x) = Ps_1 * Fs_1(x) + Ps_2 * Fs_2(x) + ... + Ps_N * Fs_N(x) = \sum_{i=1}^{N} Ps_i * Fs_i(x) \tag{3}$$

Where Ps_i stands for the probability interrupt detected in the i_{th} section, it can be calculated by Ts_i/T. The probability density function of kernel latency in the i_{th} section is $fs_i(t)$ and:

$$Fs_i(x) = \int_{-\infty}^{x} fs_i(t)dt \tag{4}$$

So we can get that:

$$F(x) = \sum_{i=1}^{N} Ps_i * Fs_i(x) = \sum_{i=1}^{N} Ps_i * \int_{-\infty}^{x} fs_i(t)dt = \int_{-\infty}^{x} \sum_{i=1}^{N} Ps_i * fs_i(t)dt \tag{5}$$

Then the density function of kernel latency during T is:

$$f(t) = \sum_{i=1}^{N} Ps_i * fs_i(t) \tag{6}$$

The expected value of kernel latency during Ts_i is:

$$Es_i(x) = \int_{-\infty}^{+\infty} xfs_i(x)dx \tag{7}$$

And expected value of kernel latency during T can be obtained:

$$E(x) = \int_{-\infty}^{+\infty} xf(x)dx = \sum_{i=1}^{N} Ps_i * \int_{-\infty}^{+\infty} xfs_i(x)dx = \sum_{i=1}^{N} Ps_i * Es_i(x) \tag{8}$$

3.2 Preemptive Barrelfish

In preemptive kernel, every kernel subsection can be divided into several regions, every of which consists of two parts: critical subregion and non-critical subregion(see Figure 2). Interrupts detected in kernel subsection can be handled in non-critical subregion, but not in critical subregion in case race condition happens. Provided that the number of regions in every kernel subsection is M, where $1 \leq M < \infty$, the time it takes to execute the $j_{th}(1 \leq j \leq M)$ critical subregion and non-critical sub region are Tc_j, Tnc_j respectively, the total time needed to run the j_{th} region in a kernel subsection is $Tc_j + Tnc_j$, which is represented by Tr_j, the distribution of kernel latency in every region is described as:

$$Fr_j(x) = \begin{cases} 0, & x < 0 \\ x/Tr_j + Tnc_j/Tr_j, & 0 \leq x \leq Tc_j \\ 1, & x > Tc_j \end{cases} \tag{9}$$

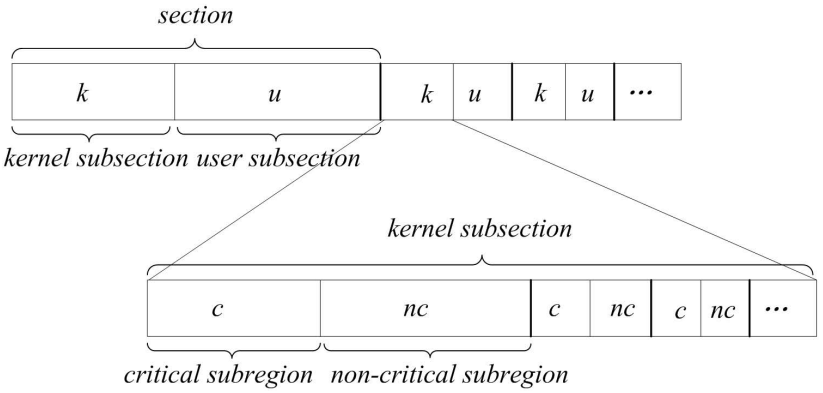

Fig. 2. Regions in kernel subsection

During the execution of the i_{th} kernel subsection, which includes M regions, the kernel latency can be represented as:

$$Fk_i(x) = Pr_1 * Fr_1(x) + Pr_2 * Fr_2(x) + ... + Pr_M * Fr_M(x) = \sum_{j=1}^{M} Pr_j * Fr_j(x) \tag{10}$$

Where Pr_j stands for the conditional probability that interrupts are detected in the j_{th} region of a kernel subsection, which can be calculated by Tr_j/Tk_i. The probability density function of kernel latency in the j_{th} region of a kernel subsection is $fr_j(t)$ and:

$$Fr_j(x) = \int_{-\infty}^{x} fr_j(t)dt \tag{11}$$

Then:

$$Fk_i(x) = \sum_{j=1}^{M} Pr_j * Fr_j(x) = \sum_{j=1}^{M} Pr_j * \int_{-\infty}^{x} fr_j(t)dt = \int_{-\infty}^{x} \sum_{j=1}^{M} Pr_j * fr_j(t)dt \tag{12}$$

So the probability density of the i_{th} kernel subsection is:

$$fk_i(t) = \sum_{j=1}^{M} Pr_j * fr_j(t) \tag{13}$$

The expected kernel latency in j_{th} region is:

$$Er_j(x) = \int_{-\infty}^{+\infty} xfr_j(x)dx \tag{14}$$

So the expected kernel latency in i_{th} kernel subsection is:

$$Ek_i(x) = \int_{-\infty}^{+\infty} xfk_i(x)dx = \sum_{j=1}^{M} Pr_j * \int_{-\infty}^{+\infty} xfr_j(x)dx = \sum_{j=1}^{M} Pr_j * Er_j(x) \tag{15}$$

3.3 Preemptive Barrelfish vs. Non-preemptive Barrelfish

Provided the i_{th} instruction section that locates in a time zone $[a, b]$, there exists an instant c, where $a < c \leq b$, system runs in kernel mode before c and returns to user mode after c in the zone.

In non-preemptive Barrelfish, according to equation (2), the distribution function of kernel latency in zone $[a, b]$ is:

$$Fs_i(x) = \begin{cases} 0, & x < 0 \\ Fk_i(x) * (c-a)/(b-a) + (b-c)/(b-a), & 0 \leq x \leq (c-a) \\ 1, & x > (c-a) \end{cases} \tag{16}$$

$Fk_i(x)$ is the distribution function of kernel latency in kernel subsection:

$$Fk_i(x) = \begin{cases} 0, & x < 0 \\ x/(c-a), & 0 \le x \le (c-a) \\ 1, & x > (c-a) \end{cases} \tag{17}$$

And expected value of kernel latency in kernel subsection is:

$$Ek_i(x) = (c-a)/2 \tag{18}$$

The density function of kernel latency in the i_{th} section is:

$$fs_i(t) = \begin{cases} 1/(b-a), & 0 < t < (c-a) \\ 0, & t \le 0, t \ge (c-a) \end{cases} \tag{19}$$

So the expected kernel latency is:

$$Es_i(x) = \int_0^{c-a} x/(b-a)dx = (c-a)^2/(2*(b-a)) = (c-a)/(b-a)*Ek_i(x) \tag{20}$$

In non-preemptive Barrelfish, the expected kernel latency in a section is just represented by equation (20). In preemptive Barrelfish, the expected kernel latency in a section can be represented by equation (20) too, but the kernel subsection $[a, c]$ can be divided to M regions, so $Es_i(x)$ can be transformed according to equation (15):

$$Es_i(x) = (c-a)/(b-a) * Ek_i(x) = (c-a)/(b-a) * \sum_{j=1}^{M} Pr_j * Er_j(x) \tag{21}$$

Assuming that the j_{th} region in kernel subsection $[a, c]$ starts at a_j and stops at b_j, in every region, kernel runs in critical subregions before c_j and non-critical regions after c_j, where $a_j < c_j \le b_j$, $a_1 = a$, $a_{j+1} = b_j$, and $b_M = c$.

The expected kernel latency in a region is calculated similarly with that in a section:

$$Er_j(x) = \int_0^{c_j-a_j} x/(b_j-a_j)dx = (c_j-a_j)^2/(2*(b_j-a_j)) = (c_j-a_j)/(b_j-a_j)*Ec_j(x) \tag{22}$$

Where $Ec_j(x) = (c_j - a_j)/2$.

So the expected kernel latency in a section is:

$$Es_i(x) = (c-a)(b-a) * \sum_{j=1}^{M} Pr_j * (c_j - a_j)/(b_j - a_j) * Ec_j(x) \tag{23}$$

Compare equation (23) with equation (20) we can get that preemptive Barrelfish outperforms non-preemptive Barrelfish in aspect of kernel latency because:

$$\begin{aligned}
&(c-a)/(b-a) * \textstyle\sum_{j=1}^{M} Pr_j * (c_j - a_j)/(b_j - a_j) * (c_j - a_j)/2 \\
&\leq (c-a)/(b-a) * \textstyle\sum_{j=1}^{M} Pr_j * (c_j - a_j)/(b_j - a_j) * (b_j - a_j)/2 \\
&\leq (c-a)/(b-a) * \textstyle\sum_{j=1}^{M} (b_j - a_j)/2 \\
&= (c-a)/(b-a) * (b_M - a_1)/2 \\
&= (c-a)/(b-a) * (c-a)/2 \\
&= (c-a)^2/(2 * (b-a))
\end{aligned} \tag{24}$$

Which means that $Es_i(x)$ in preemptive Barrelfish is less than or equal to $Es_i(x)$ in non-preemptive Barrelfish, then during T, kernel latency in preemptive Barrelfish is less than or equal to that in non-preemptive Barrelfish.

All of the equations above only describe the average case of kernel latency, in real-time system, the worst case matters too. In non-preemptive Barrelfish, kernel latency in section $[a, b]$ will reach the largest possible value, which is $b - a$, when $c = b$, while in preemptive Barrelfish, kernel latency in section $[a, b]$ depends the largest critical subregion $[a_j, b_j]$, which is at most $b_j - a_j$ when $c_j = b_j$. It's known that $(b_j - a_j) \leq (b - a)$ because region $[a_j, b_j]$ locates in section $[a, b]$, which means $a_j \geq a$ and $b_j \leq b$. So, in worst case, preemptive Barrelfish still outperforms non-preemptive Barrelfish in aspect of kernel latency.

4 Preemption Support of Barrelfish

4.1 Analysis and Design for Preemption

When interrupt is handled in user mode, kernel of Barrelfish saves the register snapshot of current dispatcher in area shared between kernel and user space. There are two kinds of register saving area: enabled saving area and disabled saving area, it't up to the state of current dispatcher that kernel chooses which one to save register snapshot in. A boolean type variable *disabled* in DCB(Dispatcher Control Block) indicates the state of the dispatcher. If *disabled* == **false**, register snapshot will be saved in enabled saving area, while *disabled* == **true**, register snapshot will be saved in disabled saving area. When a dispatcher is chosen to run after scheduling or interrupt handling, the kernel will choose one of the snapshots in enabled saving area or disabled saving area to reset/restore the dispatcher's registers.

The time-division shared kernel stack in Barrelfish is not used to save context in user mode, but that in kernel mode. To be preemptive, each dispatcher should have a private kernel stack to save context of kernel functions executing in its address space in case it's switched out. What's more, a kernel saving area is added to DCB, which is used to save the register snapshot the first time dispatcher executing in kernel mode is preempted by an interrupt handler. Just like the variable of *disabled*, a new boolean type variable *kernel* is cited to indicate if the current dispatcher executing in kernel mode when preempted. In this new design, a dispatcher could be in one of the three states such as **kernel**, **disabled**,

enabled. The state of **kernel** has a higher priority than the other two, when *kernel* == **true**, dispatcher is definitely in kernel mode, no matter what *disabled* equals, otherwise, the dispatcher is either in **disabled** or **enabled** state, which are mutual exclusive to each other, according to the variable *disabled*.

In preemptive kernel, interrupt can be allowed when kernel control path such as system calls, exception handlers, and interrupt handlers are executing, context switch is not allowed when handling interrupts. An integer type variable *preempt_count* is used to count the number of interrupts under handling, it increases by one before enter interrupt handlers while decreases by one after leave interrupt handlers. If dispatcher runs in user mode, the register snapshot is saved in enabled or disabled saving area by the common IRQ handler at first, meanwhile, stack pointer is set to the top of private kernel stack of current dispatcher. The same operations on stack pointer will be carried out when system calls and exceptions occurs in user mode. If dispatcher runs in kernel mode, which may result from system calls or exceptions, the register snapshot is saved in kernel saving area the first time interrupt occurs, and private kernel stack will still be used to save context of kernel functions. If interrupt arrives when other interrupts are being handled, the register snapshot will be saved in the current kernel stack instead of kernel saving area, current kernel stack will still be used too. Notice that there exists a special kernel state, in which no dispatcher is runnable and the system stops to execute **hlt** instruction in kernel mode, once interrupt arrives, the dedicated handler will run. In this situation, no private kernel stack but the time-division shared kernel stack in original design is used to save context of kernel functions.

Table 1. Rules related with kernel stack, context switch and snapshot area

	user mode		kernel mode			
	enabled	disabled	syscall	exception	interrupt	waiting
kernel stack	private	private	private	private	private/shared	shared
saving area	enabled	disabled	kernel	kernel	private/shared	N/A
context switch	allowed	allowed	allowed	allowed	not allowed	allowed
restoring area	enabled	disabled	kernel	kernel	private/shared	N/A

1.private: private kernel stack 2.shared: shared kernel stack
3.enabled: enabled saving area 4.disabled: disabled saving area 5.kernel: kernel saving area

Scheduling and context switch is only allowed in the context of systems calls, exception handlers or at the end of interrupt handling. An interrupt handler must check if it's nested in other interrupt handlers before scheduling or switching to another dispatcher, only when all the interrupt handlers have been finished(*preempt_count* == *0*) should scheduling or context switching be carried out. Every dispatcher gets to run again by means of *dispatch*, a function carrying out context switch. *dispatch* chooses the proper register saving area to reset/restore target dispatcher registers. When interrupt arrives, first the register snapshot of current dispatcher should be saved into dedicated saving area,

and then the kernel stack should be chosen. The rules for setting kernel stack and register saving area are shown in table 1. When interrupt handler finished, either another interrupt handler or dispatcher get to run. The rules for context switching and choosing the right target saving area to restore is shown in table 1 too.

4.2 Protection in Critical Region

Preemption only makes it possible to handle interrupt in kernel mode, it's sure that interrupt is not allowed all along the kernel control path. In Barrelfish, every kernel(CPU driver) run in a single core environment, critical region should be avoided when interrupt is allowed. The mostly invoked kernel code is related with capability operation, capability is a central security mechanism and most of the kernel functions such as system calls, message passing, and interrupt handling are invoked by means of capability reference. Call graph about capability is fully analyzed, race conditions should never occur in any section of code. Interrupt is disabled before entering critical region and restored when leaving. Capability functions such as *insert_after*, *insert_before*, scheduling list operation functions *dispatch*, *scheduling*, *make_runnale* operating on shared data structure, and operations on hardware which may be disturbed should all be called in a context where interrupt is not allowed.

5 Experiment

To evaluate the preemptive kernel basing on Barrelfish, we have measured the kernel latency and kernel overhead[13][14]. According to the result obtained in section 3, kernel latency depends heavily on the portion of instructions executed in kernel mode. If only user mode code is executed all the time, there will almost be no improvement in kernel latency because nearly all of the interrupts can be handled immediately in both preemptive and non-preemptive kernel. If code executed is full of I/O operations or system calls, then preemptive kernel would outperform non-preemptive kernel. In general, both of the kernel overhead and kernel latency should be measured to evaluate the performance of preemptive kernel in this paper.

5.1 Experiment Environments

In this experiment, preemptive barrelfish executes on a platform basing on 2 quad-core Xeon-5606 processors, which run at 2.13GHz, the capacity of main memory is 8GB. In order to simulate an external interrupt which triggers in a predictable interval, local APIC(Advanced Programmable Interrupt Controller) timer is set to expire at a dedicated instant in one-shot mode[20][21], then the timer interrupt handler will get invoked in a few time, which is treated as kernel latency here, after the expiring. The experiment is carried out in environments of CPU-bound and I/O-bound workloads separately, which are the two extreme of a real-world scenario.

- CPU-bound workloads: characterized by pure arithmetic logical computing tasks that keep processor core busy, there is almost no critical region during execution, except for some interrupt handling and process scheduling.
- I/O-bound workloads: characterized by a large number of I/O operations such as ethernet transfer, communication port access, where interrupts are disabled to maintain the consistency of system.

Kernel overhead equals the difference between real time needed to execute a CPU-bound workload and the wall time consumed to execute it. The real execution time can be measured when the program runs in kernel mode with interrupt disabled, and the wall time consumed is measured when it runs in user mode with interrupt enabled.

5.2 Evaluating Kernel Latency

In Figures 3(a)-(d), x-axis illustrates the number of times every experiment is carried out. Figure 3(a) and Figure 3(b) represent the kernel latency of non-preemptive Barrelfish and preemptive Barrelfish respectively, each sample represents a single measurement. In I/O-bound workload, the system executes in kernel mode with a higher probability, then preemptive kernel has a lower kernel latency in both average case and worst case, which matches the result in section 3. As shown in Figure 3(b), the worst case kernel latency in preemptive kernel is less than 2500 cycles while that in non-preemptive kernel is as much as 3500 cycles in preemptive kernel as Figure 3(a), it's a encouraging improvement for hard real-time system. In addition, the average case kernel latency is also improved approximately from 2400 cycles to 2200 cycles, which is meaningful for soft real-time system.

In the CPU-bound workload represented in Figure 3(c) and Figure 3(d), system executes in user mode with higher probability, preemptive kernel doesn't outperform non-preemptive too much, interrupt can both be handled immediately after it occurs because CPU-bound workloads run in kernel mode seldom. The average kernel latency are almost the same in preemptive and non-preemptive kernel. Only in few case that non-preemptive kernel has larger kernel latency, just as shown in 3(c) and 3(d), the worst kernel latency is more than 2500 cycles in non-preemptive kernel and less than 2400 cycles in preemptive kernel. (e)-(h) in Figure 3 also represent the different kernel latency between non-preemptive Barrelfish and preemptive Barrelfish, they are the cumulative distribution function of kernel latency. In all of the four figures, probability reaches to 1 from 0 rapidly during a narrow domain in x-axis, which represents the kernel latency. The steeper the curve is, which means less standard deviation, the better the kernel performs with respect to kernel latency. It's clear that preemptive kernel has a better kernel latency when running I/O-bound workload, as shown in (e) (f) of Figure 3. Figure 3(g) and Figure 3(h) show the equivalent result to Figure 3(c) and Figure 3(d).

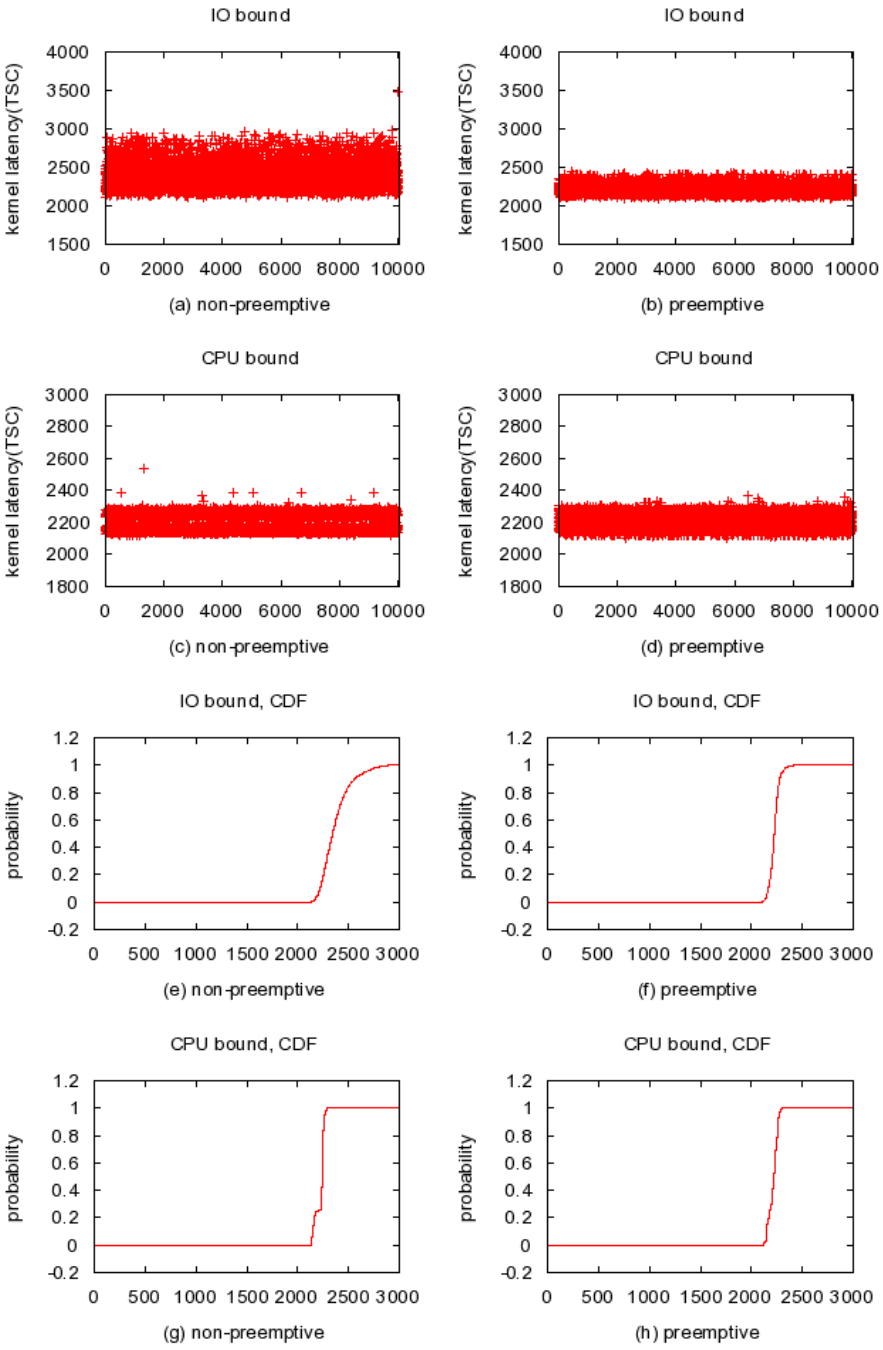

Fig. 3. Experiment data of kernel latency

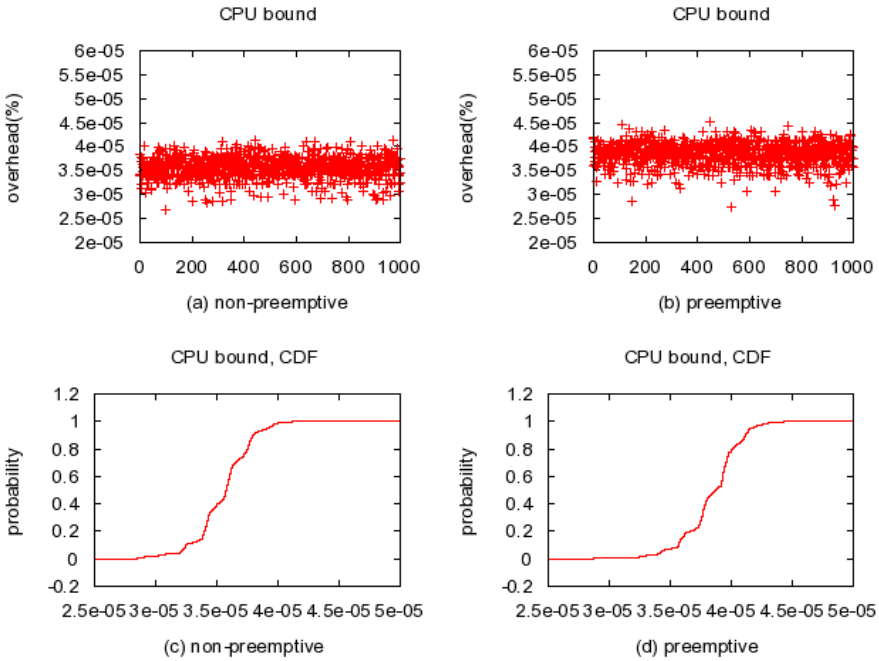

Fig. 4. Experiment data of kernel overhead

5.3 Evaluating Kernel Overhead

The kernel overhead of Barrelfish is also measured in this paper(see Figure 4). x-axis indicates number of experiments and overhead(cycles) respectively in Figure 4(a)-(b) and Figure 4(c)-(d). When running CPU-bound workload, preemptive kernel in this paper may even performs a little worse than non-preemptive kernel in aspect of average kernel latency(see Figure 3(c)-(d)), the main reason is that preemptive kernel may execute more code related with preemption prior to interrupt handling. When it comes to kernel overhead, preemptive kernel is also a little worse than non-preemptive kernel because of the same reason. Figure 4(a) and Figure 4(b) show the result. But Figure 4(c) and Figure 4(d) give a clear representation that the worse overhead in preemptive kernel is so negligible that it's completely acceptable.

6 Conclusions and Future Work

In this paper we have introduced a multi-/many-core oriented operating system: Barrelfish. The multi-kernel architecture of Barrelfish makes it perform well in both aspects of scalability and predictability, it could be a valuable choice for embedded systems basing on multi-/many-core platforms if preemption is supported.

We first analysis the architecture of Barrelfish, and gives a model of the kernel control path. Then, we prove that preemption support in Barrelfish kernel will give rise to a better kernel latency. According the model, we redesign some component of Barrelfish and realize a preemptive kernel. Finally, experiments show that the preemptive kernel in this paper can improved the performance of kernel latency as much as 20% or even more(Figure 3(a)(b)), which is an encouraging result.

Next, a deep optimization and research will be carried out, we are also planning to port Barrelfish to the MIC(Many Integrated Core) platform from Intel, which consists of as many as 60 cores(240 hard-threads). Basing on the architecture of MIC, topology aware task mapping, real-time scheduling, power management and optimized message passing are all meaningful topics to engage in.

References

1. Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S., Yelick, K.: The Landscape of Parallel Computing Research: A View from Berkeley. Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, 18(2006-183):19 (December 2006)
2. Vajda, A., Brorsson, M.: Programing Many-Core Chips. Springer (2011) ISBN 144 1997385, 9781441997388
3. Held, J.: "Single-chip cloud computer", an IA tera-scale research processor. In: Proceedings of the 2010 Conference on Parallel Processing (May 2011)
4. Liu, B.-W., Chen, S.-M., Wang, D.: Survey on Advance Microprocessor Architecture and Its Development Trends. Application Research of Computers (2007)
5. Baumann, A., et al.: The Multi-kernel: A new OS architecture for scalable multicore systems. In: Proceedings of the 22nd ACM Symposium on OS Principles, Big Sky, MT, USA (October 2009)
6. Schüpbach, A., Peter, S., Baumann, A., Roscoe, T., Barham, P., Harris, T., Isaacs, R.: Embracing diversity in the Barrelfish manycore operating system. In: Proceedings of the 1st Workshop on Managed Multi-Core Systems (2008)
7. Baumann, A., et al.: Your computer is already a distributed system. Why isn't your OS? In: Proceedings of the 12th Workshop on Hot Topics in Operating Systems, Monte Verità, Switzerland (May 2009)
8. Bryant, R., Hawkes, J.: Linux scalability for large NUMA systems. In: Ottawa Linux Symp., Ottawa, Canada (July 2003)
9. Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. J. ACM 20(1), 46–61 (1973)
10. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv. 43(4), Article 35, 44 pages (2011)
11. Burns, A.: Scheduling hard real-time systems: a review. Software Engineering Journal (May 1991)
12. Wilhelm, R., et al.: The Worst-Case Execution Time Problem: Overview of Methods and Survey of Tools. ACM Transactions on Embedded Computing Systems, 1–53 (April 2008)
13. Betti, E., Bovet, D.P., Cesati, M., Gioiosa, R.: Hard real-time performances in multiprocessor-embedded systems using asmp-linux. EURASIP Journal on Embedded System, 10:1–10:16 (April 2008)

14. Sacha, K.M.: Measuring the Real-Time Operating System Performance. In: Proceedings of Seventh Euromicro Workshop on Real-Time Systems, pp. 34–40 (1995)
15. Bovet, D.P., Cesati, M.: Understanding the Linux Kernel, 3rd edn. O'Reilly (November 2005) ISBN 0-596-00565-2
16. Klein, G., Elphinstone, K., Heiser, G., et al.: seL4: Formal verification of an OS kernel. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles (October 2009)
17. Sauermann, J., Thelen, M.: Realtime Operating Systems: Concepts and Implementation of Microkernels for Embedded Systems (1997)
18. Ramamritham, K., Stankovic, J.A.: Scheduling Algorithms and Operating Systems Support for Real-Time Systems. Proceedings of the IEEE 82(1), 55–67 (1994)
19. Franke, H., Nagar, S., Kravetz, M., Ravindran, R.: PMQS: scalable Linux Scheduling for high end servers. In: Proceedings of the 5th Annual Linux Showcase Conference (November 2001)
20. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation (May 2007)
21. AMD: AMD64 Architecture Programmer's Manual. AMD Corporation (September 2006)