rIOMMU: Efficient IOMMU for I/O Devices that Employ Ring Buffers

Moshe Malka Nadav Amit Muli Ben-Yehuda Dan Tsafrir
Technion – Israel Institute of Technology
{moshemal,namit,muli,dan}@cs.technion.ac.il

The IOMMU allows the OS to encapsulate I/O devices in their own virtual memory spaces, thus restricting their DMAs to specific memory pages. The OS uses the IOMMU to protect itself against buggy drivers and malicious/errant devices. But the added protection comes at a cost, degrading the throughput of I/O-intensive workloads by up to an order of magnitude. This cost has motivated system designers to trade off some safety for performance, e.g., by leaving stale information in the IOTLB for a while so as to amortize costly invalidations.

We observe that high-bandwidth devices—like network and PCIe SSD controllers—interact with the OS via circular ring buffers that induce a sequential, predictable workload. We design a ring IOMMU (rIOMMU) that leverages this characteristic by replacing the virtual memory page table hierarchy with a circular, flat table. A flat table is adequately supported by exactly one IOTLB entry, making every new translation an implicit invalidation of the former and thus requiring explicit invalidations only at the end of I/O bursts. Using standard networking benchmarks, we show that rIOMMU provides up to 7.56x higher throughput relative to the baseline IOMMU, and that it is within 0.77–1.00x the throughput of a system without IOMMU protection.

Categories and Subject Descriptors B.3.2 [*memory structures*]: design styles—virtual memory; B.4.2 [*I/O and data communications*]: I/O devices—channels and controllers; D.4.2 [*operating systems*]: storage management—virtual memory, allocation/deallocation strategies

General Terms design, experimentation, performance

I/O memory management unit

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. ing with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey.
Copyright c 2015 ACM 978-1-4503-2835-7/15/03…$15.00.
http://dx.doi.org/10.1145/ .

1. Introduction

I/O device drivers initiate direct memory accesses (DMAs) to asynchronously move data from their devices into memory and vice versa. In the past, DMAs used physical memory addresses. But such unmediated access made systems vulnerable to (1) rogue devices that might perform errant or malicious DMAs [9, 12, 28, 32, 52], and to (2) buggy drivers that account for most operating system (OS) failures and might wrongfully trigger DMAs to arbitrary memory locations [8, 13, 23, 36, 47, 50]. Subsequently, all major chip vendors introduced I/O memory management units (IOMMUs) [3, 7, 25, 28], which allow DMAs to execute with I/O virtual addresses (IOVAs). The IOMMU translates the IOVAs into physical addresses according to I/O page tables that are setup by the OS. The OS thus protects itself by adding a translation just before the corresponding DMA, and by removing the translation right after [11, 16, 51]. We explain in detail how the IOMMU is implemented and used in §2.

DMA protection comes at a cost that can be substantial in terms of performance [4, 10, 51], for newer, high-throughput I/O devices like 10/40 Gbps network controllers (NICs), which can deliver millions of packets per second. Our measurements indicate that using DMA protection with such devices can reduce the throughput by up to 10x. This penalty has motivated OS developers to trade off some protection for performance. For example, when employing the "deferred" IOMMU mode, the Linux kernel defers IOTLB invalidations for a short while instead of performing them immediately, because invalidations are slow. Later, the kernel processes the accumulated invalidations en masse by flushing the entire IOTLB, thus amortizing the overhead at the risk of allowing devices to erroneously utilize stale IOTLB entries. While this tradeoff can double the performance relative to the stricter IOMMU mode, the throughput is still 5x lower than when the IOMMU is disabled. We analyze and model the overheads associated with using the IOMMU in §3.

We argue that the degraded performance is largely due to the IOMMU needlessly replicating the design of the regular MMU, which is based on hierarchical page tables. Our claim pertains high-bandwidth I/O devices, such as NICs and PCIe

SSD drives, which utilize circular "ring" buffers to interact with the OS. A ring is an array of descriptors that the OS driver sets when initiating DMAs. Descriptors encapsulate the DMA details, including the associated IOVAs. Importantly, ring semantics dictate that (1) the driver work through the ring in order, one descriptor after the other, and that (2) the I/O device process these descriptors in the same order. Thus, IOVAs are short-lived and the sequence in which they are used is linearly predictable: each IOVA is allocated, placed in the ring, used in turn, and deallocated.

We propose a ring IOMMU (rIOMMU) that supports this pervasive sequential model using flat (1D) page tables that directly correspond to the nature of rings. rIOMMU has three advantages over the baseline IOMMU that significantly reduce the overhead of DMA protection. First, building/destroying an IOVA translation in a flat table is quicker than in a hierarchical structure. Second, (de)allocation of IOVAs—the actual integers serving as virtual addresses—is faster, as IOVAs are indices of flat tables in our design. Finally, the frequency of IOTLB invalidations is substantially reduced, because the rIOMMU designates only one IOTLB entry per ring. One is enough because IOVAs are used sequentially, one after the other. Consequently, every translation inserted to the IOTLB removes the previous translation, eliminating the need to explicitly invalidate the latter. And since the OS handles high-throughput I/O in bursts, explicit invalidations become rare. We describe rIOMMU in §4.

We evaluate the performance of rIOMMU using networking benchmarks and find that it improves throughput by 1.00—7.56x, shortens latency by 0.80–0.99x, and reduces CPU consumption by 0.36–1.00x relative to the existing IOMMU. Our fastest rIOMMU variant is within 0.77–1.00x the throughput, 1.00–1.04x the latency, and 1.00–1.22x the CPU consumption of a system that disables the IOMMU entirely. We describe our experimental evaluation in §5.

2. Background

2.1 Operating System DMA Protection

The role the IOMMU plays for I/O devices is similar to the role the regular MMU plays for processes, as illustrated in Figure 1. Processes typically access the memory using virtual addresses, which are translated to physical addresses by the MMU. Analogously, I/O devices commonly access the memory via DMAs associated with IOVAs. The IOVAs are translated to physical addresses by the IOMMU.

The IOMMU provides *inter*- and *intra-OS protection* [4, 49, 51, 53]. Inter-OS protection is applicable in virtual setups. It allows for "direct I/O", where the host assigns a device directly to a guest virtual machine (VM) for its exclusive use, largely removing itself from the guest's I/O path and thus improving its performance [22, 36]. In this mode of operation, the VM directly programs device DMAs

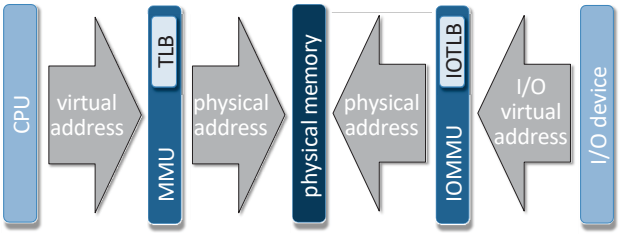

Figure 1. The IOMMU is for devices what the MMU is for processes.

using its notion of (guest) "physical" addresses. The host uses the IOMMU to redirect these accesses to where the VM memory truly resides, thus protecting its own memory and the memory of the other VMs. With inter-OS protection, IOVAs are mapped to physical memory locations infrequently, typically only upon such events as VM creation and migration. Such mappings are therefore denoted *static* or *persistent* [51]; they are not the focus of this paper.

Intra-OS protection allows the OS to defend against the DMAs of errant/malicious devices [9, 12, 17, 28, 32, 52] and of buggy drivers, which account for most OS failures [8, 13, 23, 36, 47, 50]. Drivers and their I/O devices can perform DMAs to arbitrary memory addresses, and IOMMUs allow OSes to protect themselves (and their processes) against such accesses, by restricting them to specific physical locations. In this mode of work, *map* operations (of IOVAs to physical addresses) and *unmap* operations (invalidations of previous maps) are frequent and occur within the I/O critical path, such that each DMA is preceded and followed by the mapping and unmapping of the corresponding IOVA [32, 40]. Due to their short lifespan, these mappings are denoted *dynamic* [11], *streaming* [16] or *single-use* [51]. This strategy of IOMMU-based intra-OS protection is the focus of this paper. It is recommended by hardware vendors [24, 28, 32] and employed by operating systems [6, 11, 16, 26, 38, 51].[1] It is applicable in non-virtual setups where the OS has direct control over the IOMMU. It is likewise applicable in virtual setups where IOMMU functionality is exposed to VMs via paravirtualization [10, 36, 45, 51], full emulation [4], and, more recently, hardware support for nested IOMMU translation [3, 28].

2.2 IOMMU Design and Implementation

Given a *target memory buffer* of a DMA, the OS associates the physical address (PA) of the buffer with an IOVA. The OS maps the IOVA to the PA by inserting the IOVA⇒PA

[1] For example, the DMA API of Linux notes that "DMA addresses should be mapped only for the time they are actually used and unmapped after the DMA transfer" [40]. In particular, "once a buffer has been mapped, it belongs to the device, not the processor. Until the buffer has been unmapped, the [OS] driver should not touch its contents in any way. Only after [the unmap of the buffer] has been called is it safe for the driver to access the contents of the buffer" [16].

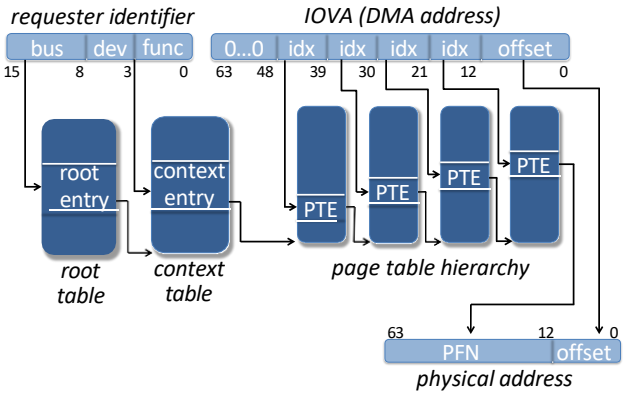

Figure 2. IOVA translation with the Intel IOMMU.

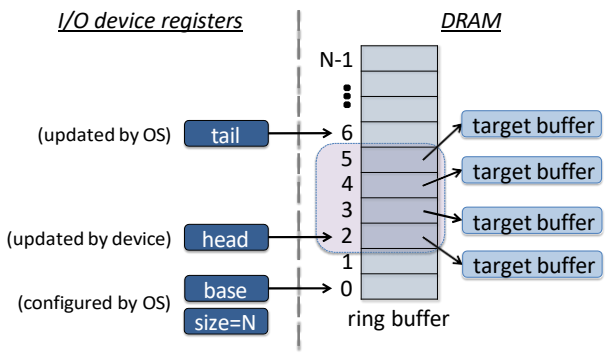

Figure 3. The driver drives its device via a ring. With an IOMMU, register/target pointers are IOVAs.

translation to the IOMMU data structures. Figure 2 depicts these structures as implemented by Intel x86-64 [28]. The PCI protocol dictates that each DMA operation is associated with a 16-bit *request identifier* comprised of a *bus-device-function* triplet that uniquely identifies the corresponding I/O device. The IOMMU uses the 8-bit bus number to index the *root table* in order to retrieve the physical address of the *context table*. It then indexes the context table using the 8-bit concatenation of the device and function numbers. The result is the physical location of the root of the page table hierarchy that houses all of the IOVA⇒PA translations of that I/O device.

The purpose of the IOMMU page table hierarchy is similar to that of the MMU hierarchy: recording the mapping from virtual to physical addresses by utilizing a 4-level radix tree. Each 48-bit (I/O) virtual address is divided into two: the 36 high-order bits, which constitute the *virtual page number*, and the 12 low-order bits, which are the *offset* within the page. The translation procedure applies to the virtual page number only, converting it into a *physical frame number* (PFN) that corresponds to the physical memory location being addressed. The offset is the same for both physical and virtual pages.

Let T_j denote a page table in the j-th radix tree level for $j = 1, 2, 3, 4$, such that T_1 is the root of the tree. Each T_j is a 4KB page containing up to $2^9 = 512$ pointers to physical locations of next-level T_{j+1} tables. Last-level—T_4—tables contain PFNs of target buffer locations. Correspondingly, the 36-bit virtual page number is split into a sequence of four 9-bit indices i_1, i_2, i_3 and i_4, such that i_j is used to index T_j in order to find the physical address of the next T_{j+1} along the radix tree path. Logically, in C pointer notation, $T_1[i_1][i_2][i_3][i_4]$ is the PFN of the target memory location.

Similarly to the MMU translation lookaside buffer (TLB), the IOMMU caches translations using an IOTLB, which it fills on-the-fly as follows. Upon an IOTLB miss, the IOMMU hardware hierarchically *walks* the page table as described above, and it inserts the IOVA⇒PA translation to the IOTLB. IOTLB entries are invalidated explicitly by the OS as part of the corresponding unmap operation.

An IOMMU table walk fails if a matching translation was not previously established by the OS, a situation which is logically similar to encountering a null pointer value during the walk. A walk additionally fails if the DMA being processed conflicts with the read/write permission bits found within the page table entries along the traversed radix tree path. We note in passing that, at present, in contrast to MMU memory accesses, DMAs are typically not restartable. Namely, existing systems usually do not support "I/O page faults", and hence the OS cannot populate the IOMMU page table hierarchy on demand. Instead, IOVA translations of valid DMAs are expected to be successful, and the corresponding pages must be pinned to memory. (Albeit I/O page fault standardization does exist [42].)

2.3 I/O Devices Employing Ring Buffers

Many I/O devices—notably NICs and disk drives—deliver their I/O through one or more producer/consumer *ring* buffers. A ring is an array shared between the OS device driver and the associated device, as illustrated in Figure 3. The ring is circular in that the device and driver wrap around to the beginning of the array when they reach its end. The entries in the ring are called DMA *descriptors*. Their exact format and content vary between devices, but they specify at least the address and size of the corresponding target buffers. Additionally, the descriptors commonly contain status bits that help the driver and the device to synchronize.

Devices must also know the *direction* of each requested DMA, namely, whether the data should be transmitted from memory (into the device) or received (from the device) into memory. The direction can be specified in the descriptor, as is typical for disk controllers. Or the device can employ different rings for receive and transmit activity, in which case the direction is implied by the ring. The receive and transmit rings are denoted *Rx* and *Tx*, respectively. NICs employ at least one Rx and one Tx per port. They may employ multiple Rx/Tx rings per port to promote scalability, as different rings can be handled concurrently by different cores.

Upon initialization, the OS device driver allocates the rings and configures the I/O device with the ring sizes and base locations. For each ring, the device and driver utilize a *head* and a *tail* pointers to delimit the ring content that can be used by the device: [*head*, *tail*). The device iteratively consumes (removes) descriptors from the head, and it increments the head to point to the next descriptor to be used next. Similarly, the driver adds descriptors to the tail, incrementing the tail to point to the entry it will use subsequently.

A device asynchronously informs its OS driver that data was transmitted or received by triggering an interrupt. The device coalesces interrupts when their rate is high. Upon receiving an interrupt, the driver of a high-throughput device handles the entire I/O burst. Namely, it sequentially iterates through and processes all the descriptors whose corresponding DMAs have completed,

3. Cost of Safety

This section enumerates the overhead components involved in using the IOMMU in the Linux/Intel kernel (§3.1). It experimentally quantifies the overhead of each component (§3.2). And it provides and validates a simple performance model that allows us to understand how the overhead affects performance and to assess the benefits of reducing it (§3.3).

3.1 Overhead Components

Suppose that a device driver that employs a ring wants to transmit or receive data from/to a target buffer. Figure 4 lists the actions it carries out. First, it allocates the target buffer, whose physical address is denoted p (1). (For simplicity, let us assume that p is page aligned.) It pins p to memory and then asks the IOMMU driver to map the buffer to some IOVA, such that the I/O device would be able to access p (2). The IOMMU driver invokes its IOVA allocator, which returns a new IOVA v—an integer that is not associated with any other page currently accessible to the I/O device (3). The IOMMU driver then inserts the $v \Rightarrow p$ translation to the page table hierarchy of the I/O device (4), and it returns v to the device driver (5). Finally, when updating the corresponding ring descriptor, the device driver uses v as the address for the target buffer of the associated DMA operation (6).

Assume that the latter is a receive DMA. Figure 5 details the activity taking place when the I/O device gets the data. The device reads the DMA descriptor through its head register. The address held by the head is an IOVA, so it is intercepted by the IOMMU (1). The IOMMU consults its IOTLB to find a translation for the head IOVA. If the translation is missing, the IOMMU walks the page table hierarchy of the device to resolve the miss (2). Equipped with the head's physical address, the IOMMU translates the head descriptor for of the device (3). The head descriptor specifies that v (IOVA defined above) is the address of the target buffer (4), so the device writes the received data to v (5). The IOMMU intercepts v,

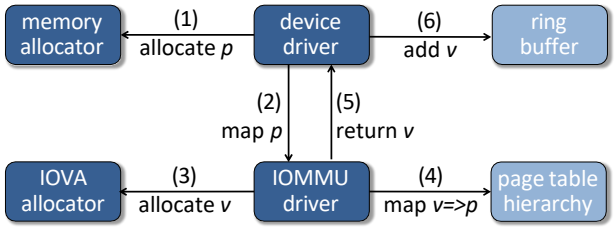

Figure 4. The I/O device driver maps an IOVA v to a physical target buffer p. It then assigns v to the DMA descriptor.

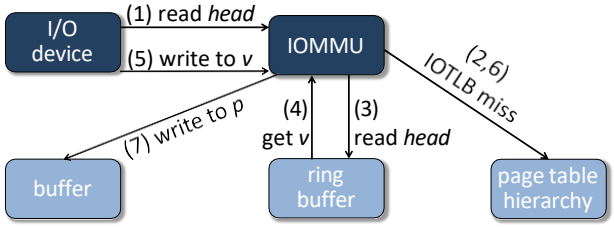

Figure 5. The I/O device writes the packet it receives to the target buffer through v, which the IOMMU translates to p.

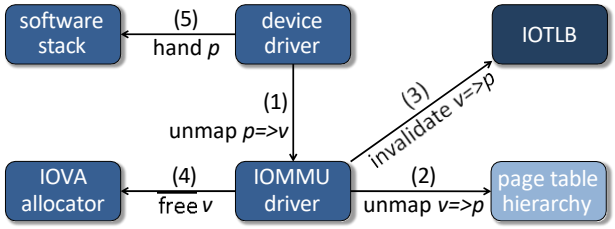

Figure 6. After the DMA completes, the I/O device driver unmaps v and passes p to a higher-level software layer.

walks the page table if the $v \Rightarrow p$ translation is missing (6), and redirects the received data to p (7).

Figure 6 shows the actions the device driver carries out after the DMA operation is completed. The device driver asks the IOMMU driver to unmap the IOVA v (1). In response, the IOMMU driver removes the $v \Rightarrow p$ mapping from the page table hierarchy (2), purges the mapping from the IOTLB (3), and deallocates v (4). (The order of these actions is important.) Once the I/O device can no longer access p, it is safe for the device driver to hand the buffer to higher levels in the software stack for further processing (5).

3.2 Protection Modes and Measured Overhead

We experimentally quantify the overhead components of the map and unmap functions—outlined in Figures 4 and 6—of the IOMMU driver. To this end, we execute the standard Netperf TCP stream benchmark, which attempts to maximize network throughput between two machines over a TCP connection. (The experimental setup is detailed in §5.)

function	*component*	*strict*	*strict+*	*defer*	*defer+*
map	iova alloc	3986	92	1674	108
	page table	588	590	533	577
	other	44	45	44	42
	sum	4618	727	2251	727
unmap	iova find	249	418	263	454
	iova free	159	62	189	57
	page table	438	427	471	504
	iotlb inv	2127	2135	9	9
	other	26	25	205	216
	sum	2999	3067	1137	1240

Table 1. Average cycles breakdown of the (un)map functions of the IOMMU driver for different protection modes.

Strict Protection We begin by profiling the Linux kernel in its safer IOMMU mode, denoted *strict*, which strictly follows the map/unmap procedures described in §3.1. Table 1 shows the average duration of the components of these procedures in cycles. The strict/map breakdown indicates that its most costly component is, surprisingly, IOVA allocation (Step 3 in Figure 4). Upon further investigation, we found that the reason for this high cost is a nontrivial pathology in the Linux IOVA allocator that regularly causes some allocations to be linear in the number of currently allocated IOVAs. We were able to come up with a more efficient IOVA allocator, which consistently allocates/frees in constant time [37]. We denote this optimized IOMMU mode—which is quicker than strict but equivalent to it in terms of safety—as *strict+*. Table 1 shows that strict+ indeed reduces the allocation time from nearly 4,000 cycles to less than 100.

The remaining dominant strict(+)/map overhead is the insertion of the IOVA to the IOMMU page table (Step 4 in Figure 4). The 500+ cycles of the insertion are due to explicit memory barriers and cacheline flushes that the driver performs when updating the hierarchy. Flushes are required, as the I/O page walk is incoherent with the CPU caches on our system. (This is common nowadays; Intel started shipping servers with coherent I/O page walks only recently.)

Focusing on the unmap components of strict/strict+, we see that finding the unmapped IOVA in the allocator's data structure is costlier in strict+ mode. The reason: like the baseline strict, strict+ utilizes a red-black tree to hold the IOVAs. But the strict+ tree is fuller, so the logarithmic search is longer. Conversely strict+/free (Step 4 in Figure 6) is done in constant time, rather than logarithmic, so it is quicker. The other unmap components are: removing the IOVA from the page tables (Step 2 in Figure 6) and the IOTLB (Step 3). The removal takes 400+ cycles, which is comparable to the duration of insertion. IOTLB invalidation is by far the slowest unmap component at around 2,000 cycles; this result is consistent with previous work [4, 53].

Deferred Protection In order to reduce the high cost of invalidating IOTLB entries, the Linux *deferred* protection mode relaxes strictness somewhat, trading off some safety for performance. Instead of invalidating entries right away, the IOMMU driver queues the invalidations until 250 freed IOVAs accumulate. It then processes all of them in bulk by invalidating the entire IOTLB. This approach affects the cost of (un)mapping in two ways, as shown in Table 1 in the defer and defer+ columns. (Defer+ is to defer what strict+ is to strict.) First, as intended, it eliminates the cost of invalidating individual IOTLB entries. And second, it reduces the cost of IOVA allocation in the baseline deferred mode as compared to strict (1,674 vs. 3,986), because deallocating IOVAs in bulk reduces somewhat the aforementioned linear pathology. The drawback of deferred protection is that the I/O device might erroneously access target buffers through stale IOTLB entries after the buffers have already been handed back to higher software stack levels (Step 5 in Figure 6). Notably, at this point, the buffers could be (re)used for other purposes.

3.3 Performance Model

Let C denote the average number of CPU cycles required to process one packet. Figure 7 shows C for each of the aforementioned IOMMU modes in our experimental setup. The bottommost horizontal grid line shows C_{none}, which is C when the IOMMU is turned off. We can see, for example, that C_{strict} is nearly 10x higher than C_{none}.

Our experimental setup employs a NIC that uses two target buffers per packet: one for the header and one for the data. Each packet thus requires two map and two unmap invocations. So the processing of the packet includes: two IOVA (de)allocations; two page table insertions and deletions; and two invalidations of IOTLB entries. The corresponding aggregated cycles are respectively depicted as the three top stacked sub-bars in the figure. The bottom, "other" sub-bar embodies all the rest of the packet processing activity, notably TCP/IP and interrupt processing. As noted, the deferred modes eliminate the IOTLB invalidation overhead, and the "+" modes reduce the overhead of IOVA (de)allocation. But even C_{defer+} (the most performant mode, which introduces a vulnerability window) is still over 3.3x higher than C_{none}.

We find that the way the value of C affects the overall throughput of Netperf is simple and intuitive. Specifically, if S denotes the cycles-per-second clock speed of the core, then S/C is the number of packets the core can handle per second. And since every Ethernet packet carries 1,500 bytes, the throughput of the system in Gbps should be $Gbps(C) = 1500 \text{ byte} \times 8 \text{ bit} \times \frac{S}{C}$, assuming S is given in GHz. Figure 8 shows that this simple model (thick line) is accurate. It coincides with the throughput obtained when systematically lengthening C_{none} using a carefully controlled busy-wait loop (thin line). It also coincides with the throughput measured under the different IOMMU modes (cross points).

Consequences As our model is accurate, we conclude that the translation activity carried out by the IOMMU (as depicted in Figure 5) does not affect the performance of the system, even when servicing demanding benchmarks like

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：https://d.book118.com/338001117025006027