DPDK中的memcpy性能优化及思考

原创 DPDK开源社区 2017-04-01

作者 王志宏

内存拷贝（memcpy）这个操作看似简单，但长期以来存在很多关于其优化的讨论，各种编程语言库也都有对应实现，而对于memcpy性能评估测试的讨论就更多了。

那么如下的memcpy实现到底有什么问题？

```
void * simple_memcpy(void *dst, const void *src, size_t n)
{
	const uint8_t *_src = src;
	uint8_t *_dst = dst;
	size_t i;

	for (i = 0; i < n; ++i)
		_dst[i] = _src[i];

	return dst;
}
```

很简单，首先，这看起来太简单，不够高端，气势上就先输了；同时，代码没有使用Vector指令，没有指令级并行，没有做地址对齐处理，最终性能完全依赖于编译器的优化——然而这些并没有什么问题，在某些应用场景中这个函数的性能甚至会比glibc的memcpy性能更高——当然，这的确完全得益于编译器的优化。

本文的观点是：不存在一个“最优”的适用于任何场景（硬件+软件+数据）的memcpy实现。这也是DPDK中rte_memcpy存在的原因：不是glibc中的memcpy不够优秀，而是它和DPDK中的核心应用场景之间不合适，有没有觉得这种说法很耳熟？本文将着重探讨如何针对具体应用进行memcpy（或其他任何程序）的性能优化。

常见的优化方法

关于memcpy的优化存在大量各种官方和民间资料，因此不再赘述，在这里只简单地进行总结。

通常memcpy的性能开销包含：

1. 数据的Load/Store
2. 附加计算任务（例如地址对齐处理）
3. 分支预测

通用的memcpy优化方向：

1. 最大限度使用memory/cache带宽（Vector指令、指令级并行）
2. Load/Store地址对齐
3. 集中顺序访问
4. 适当使用non-temporal访存执令
5. 适当使用String指令来加速较大的拷贝

最后，所有的指令都经过CPU的流水线执行，因此**对流水线效率的分析至关重要**，需要优化指令顺序以避免造成流水线阻塞。

如何进行优化

从2015年初开始DPDK对rte_memcpy做了数次优化，优化方向是加速DPDK中memcpy的应用场景，例如Vhost收发包，所有的分析和代码修改都可以通过git log进行查看。

关于如何着手进行优化这个问题，并没有唯一的答案。最简单直接的方法就是暴力破解法：仅凭经验通过想象和猜测进行各种改进尝试，并一一在目标场景中进行验证，然后通过一组评价标准来选择其中较优的一个。这是比较接近人工智能的一种方法，也就是说其实这并不怎么智能，但是往往能够得到一些效果。

另一种常见的方法是通过对现有代码进行运行时采样分析，得出理论最佳性能，并分析现有代码的缺陷，思考是否存在改进方法。当然，这需要大量的经验和分析，即是用人工代替人工智能。一个DPDK 16.11中的真实案例：花了四周时间进行采样和分析，最后将三行代码的位置进行了移动，得到了1.7倍的性能。这是暴力破解法很难完成的，因为搜索空间实在太大。不过这种看似高端的方法也存在较大的风险，极有可能分析了四周然后不了了之。

至于如何进行采样分析，如perf、vTune等都是非常有效的性能分析工具，其**重点不在于工具多样，而在于知道需要什么数据**。

最终用数据说话

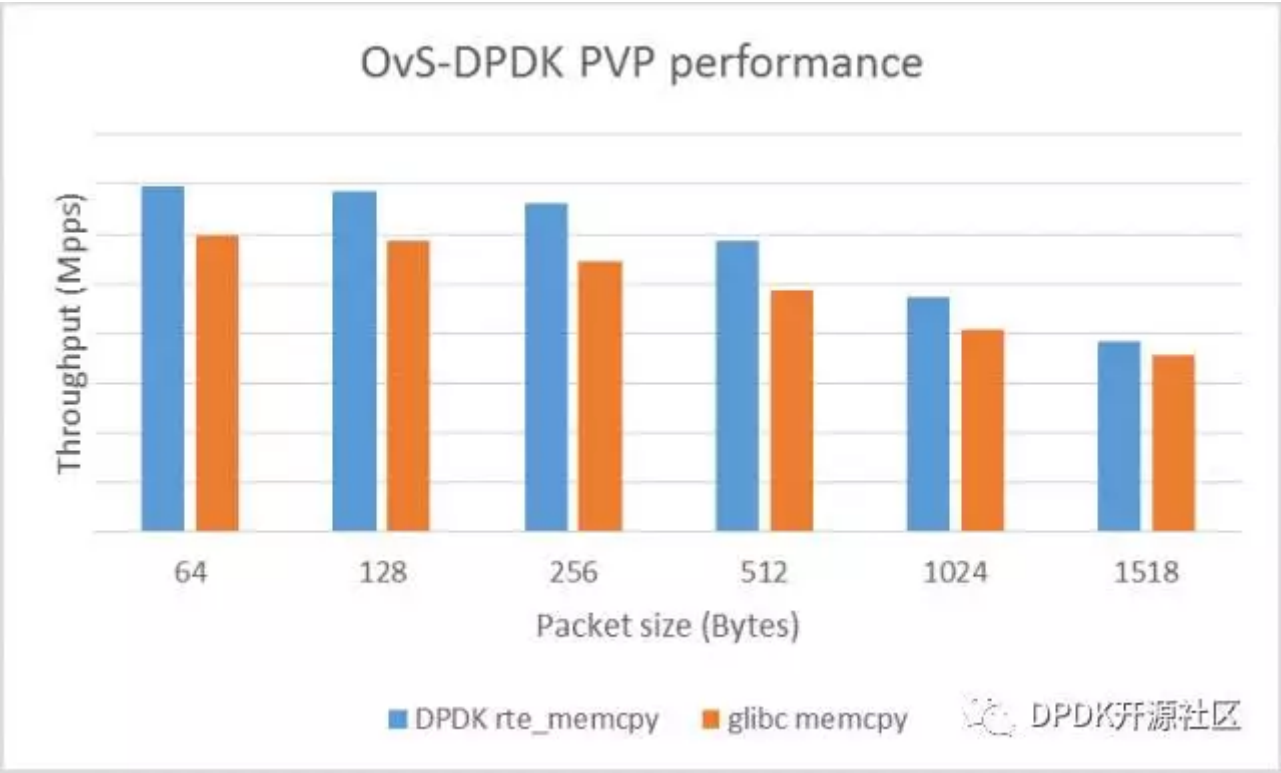
优化的最终目的是为了加速应用程序的性能，这就需要结合代码、数据特征以及实际硬件平台来进行评估，至于评估的方法则是五花八门。

对于memcpy来说，使用micro benchmark可以很方便地得出一个一目了然的数据，不过这缺乏实际参考价值。因为memcpy算法本身没有太大的改进空间，相关优化都是针对具体平台和应用场景所做的编程语言级和指令级的优化，而这个层面的优化都是针对特定的代码和硬件平台而言的，不同的场景需要不同的优化方法。因此，一套memcpy的代码在某个micro benchmark中性能出色只能证明其指令流适应这个场景。

此外，通过计算某段memcpy的CPU周期数来评估性能也是不可取的。例如，DPDK中Vhost enqueuer/dequeue函数，不能简单地通过rdtsc来标记memcpy占用了多少个时钟周期，从而判断memcpy的性能。因为目前的CPU都有着非常复杂的流水线，支持预取和乱序执行，这就造成rdtsc测量小粒度时间间隔时误差非常大，虽然加入序列化指令可以进行强制同步，但是这改变了指令流执行顺序并降低了程序性能，违背了性能测试的初衷。同时，经过编译器高度优化的代码，其指令相对于编程语言来说也是乱序的，如果强制顺序编译则会影响性能，其结果也不具有参考价值。此外，一段指令流的执行时间除流水线的理论执行时间外，还包括数据访问带来的延时，通过改变程序的行为很容易让一段代码看起来执行时间更短，而实际上只是将某些数据访问延时转提前或延后而已。这些错综复杂的因素让看似简单的memcpy性能评估变得似乎毫无头绪。

因此，**任何优化工作都应该用最终的性能指标来作为评估依据**。例如针对OvS在Cloud中的应用，其中的Vhost收发函数大量使用了memcpy，那么针对这个特定的场景，应当以最终的包转发速率作为性能评估标准。

下图展示了一个接近实际应用场景的例子：首先测试基于DPDK 17.02的OvS-DPDK的包转发性能，再通过使用glibc所提供的memcpy替换DPDK Vhost中rte_memcpy得到对比数据。测试结果显示，仅通过使用rte_memcpy加速整个程序中的Vhost收发部分就能提供最大约22%的总带宽提升。



如果你在使用DPDK时发现有性能不够优化的地方，欢迎到dev@dpdk.org或DPDK微信群进行讨论。

最后，对于优化来说，达到最优或许并不是最重要的，因为根本达不到，就算达到也不能说（参见《广告法》）——这些都不重要，优化的意义是离最优又更近了一步，并为目标业务带来了实际的收益。

附录

测试环境：

- PVP流：使用IXIA发包到物理网卡，OvS-DPDK将物理网卡收到的包转发到虚拟机，虚拟机则将包处理后通过OvS-DPDK发送回物理网卡，最后回到IXIA
- 虚拟机中使用DPDK testpmd的MAC-forwarding
- OvS-DPDK版本： Commit f56f0b73b67226a18f97be2198c0952dad534f1c
- DPDK版本：17.02
- GCC/GLIBC版本：6.2.1/2.23
- Linux：4.7.5-200.fc24.x86_64
- CPU：Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz

OvS-DPDK编译和启动命令如下：

```
make 'CFLAGS=-g -Ofast -march=native'
./ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-init=true
./ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-socket-mem="1024,1024"
./ovs-vsctl add-br ovsbr0 -- set bridge ovsbr0 datapath_type=netdev
./ovs-vsctl add-port ovsbr0 vhost-user1 -- set Interface vhost-user1 type=dpdkvhostuser
./ovs-vsctl add-port ovsbr0 dpdk0 -- set Interface dpdk0 type=dpdk options:dpdk-devargs=0000:06:00.0
./ovs-vsctl set Open_vSwitch . other_config:pmd-cpu-mask=0x10000
./ovs-ofctl del-flows ovsbr0
./ovs-ofctl add-flow ovsbr0 in_port=1,action=output:2
./ovs-ofctl add-flow ovsbr0 in_port=2,action=output:1
```

虚拟机中使用DPDK testpmd进行转发，命令如下：

```
set fwd mac
start
```

作者简介

王志宏，英特尔软件工程师，主要方向为网络虚拟化，系统性能优化。