

Using Uncacheable Memory to Improve Unity Linux Performance

Ning Qu, Xiaogang Gou, Xu Cheng

Microprocessor Research and Development Center, Peking University

{quning, gxg, chengxu}@mprc.pku.edu.cn

Abstract—Although cache is popular, it is expensive to assure cache coherence and it is not advantageous with a poor locality program. This paper points out that uncacheable memory, to a memory area with cache disabled, will have better performance in some situations. We implement uncacheable page table and uncacheable buffer for ethernet driver in the Unity system. The results indicate that uncacheable memory will make 5%-29% performance improvement in some important aspects. Meanwhile it does not hurt other performance metrics. We believe this approach could be applied in similar situations to improve other embedded systems' performance.

Index Terms—uncacheable memory, cache, operating system, network, embedded system

I. INTRODUCTION

Main memory is the center of the memory system in modern computers. As processors become relatively faster than main memory accesses [1], all modern general-purpose computer systems need cache [2], [3], [4] to reduce the average latency of memory accesses. But cache has its own limitations. First, cache introduces cache coherency problem to both uni-processor and multi-processor computer systems, which has great impacts on the system performance. Second, cache is based on property of locality [5], in other words, poor program locality will cause cache miss more frequently. For example, recent research [6] indicates that the majority of the data in TCP/IP processing shows no temporal locality, hence the existing cache policies do not help.

Uncacheable memory refers to the main memory which is accessed bypassing the cache. Accessing uncacheable memory is much slower than cacheable memory. Uncacheable memory has no coherency problem and does not rely on the program locality. We propose that uncacheable memory has advantage on data transfer in the Unity system in the following two aspects. First uncacheable memory helps eliminating most of the cache coherency operations in Unity Linux. Second it helps avoiding cache pollution during TCP/IP processing. We apply uncacheable memory in two aspects, the uncacheable page table and the uncacheable socket buffer (skb) in our operating system implementation. The performance of some basic system functions improves from 5% to 29% compared with the cacheable memory method. The results indicate that the benefit of uncacheable memory may have been underestimated in the past years.

The rest of this paper is organized as follows. Section II gives an overview of the related work. Section III introduces the experiment platform. In Section IV we present analysis and implementation on how to utilize uncacheable memory to improve performance. Section V evaluates the performance of this utilization of uncacheable memory. Finally, we summarize our work in Section VI.

II. RELATED WORKS

The design of Cache and TLB are always hotspots in computer architecture area [7], [5], [8], [9] due to their great impacts on the system performance. But cache introduces cache coherency problem [2], [10] into both uni-processor and multi-processor computer systems in that frequent coherency operations may hurt the system performance.

Cache performs well only when a program has good locality. Some work done in the past [11], [12] shows that TCP/IP process is memory intensive. And a recent research [6] has found that the majority of TCP/IP data processing shows no temporal locality, hence the existing cache policies do not help. This proves that in modern computer system, cacheable memory is not always the best choice even in some important areas where it is widely used .

Many architectures provide support for uncacheable memory in multiple ways. One way is to reserve certain address space for uncacheable usage by default. Another way is to provide a bit in page table entry to dynamically decide whether a page is cacheable or not. Intel has provided the WC(write-combining) memory since P6 family, a kind of uncacheable memory which could do speculative reads and write combining. The WC enables the P6 family processor to achieve great improvements on the graphics I/O performance [13]. The WC memory is widely used in order to optimize the I/O throughput performance from memory to I/O device [14].

Uncacheable memory is also widely adopted in the embedded system. For example, uncacheable memory is used to reduce power consumption because enabling cache will cost more on-chip energy [15], [16], [17]. Also uncacheable memory has the predictable execution time which could satisfy the requirement of the real time systems. We propose uncacheable memory for improving performance in one example of embedded system, the Unity system.

III. EXPERIMENT PLATFORM

In the following sections, our analysis is based on a real system, Unity network computer. This architecture has some simple and representative hardware designs in both the low end PC and the high end embedded systems.

The Unity SoC[18] is developed by Peking University Microprocessor R&D Center. In the Unity SoC, there is a 32-bit RISC processor Unicore32, which is a Harvard architecture implemented using a five-stage pipeline. And the Unity SoC provides separate level-one instruction and data cache which is virtual-index physical-tag addressed with write-back policy. The MMU provides full memory management to support general operating system. The TLB is designed completely by hardware translation table walking, which can fetch and update page table entries directly in main memory.

The Unity Network Computer is based on the Unity SoC and its operating system is the Unity Linux ported from Linux 2.4. Nowadays, the Unity NCs are widely used as Windows-based, X-based or Browser-based Terminals. It is also used as embedded devices in many special areas.

A. Cache Coherence Problem of the Unity System

The Unity SoC doesn't have any hardware support to automatically solve the cache coherence problem. In the SoC, There are five components which can directly access main memory: CPU directly read/write (uncacheable memory), data cache read/write, TLB read/write, instruction cache read, and DMA read/write. So coherence problem exists between instruction cache and data cache, TLB and data cache, DMA and data cache, uncacheable memory access and data cache, etc.

IV. IMPROVE UNITY LINUX PERFORMANCE USING UNCACHEABLE MEMORY

In Unity Linux we solve the cache coherence problem mainly by using hardware coherence operations when necessary. In that implementation there are three scenario to perform cache coherence operations. One is during DMA transferring, one is during synchronizing between instruction cache and data cache and the last one is during synchronizing between data cache and TLB. We conclude two key points which have great impacts on the system performance based on the system characteristics. First, cache coherence operations are expensive. In the Unity system, cache only supports operations for the whole instruction or data cache. It maintains the system coherence by sacrificing its system performance. Second, data cache could utilize burst transfer on system bus while uncacheable memory access can not.

The network computer's performance severely depends on TCP/IP processing which requires expensive cache coherence operations. There are two alternative ways to eliminate the expensive cache operations in the Unity system. One is software simulation method, which is by accessing enough

garbage data to make useful data in the cache to be replaced. To be effective, using this method must know the exact value of the critical parameters in cache design. The other is to use uncacheable memory. Uncacheable memory has no coherency problem because the data in the main memory is always the latest. Next subsection we study these different methods by carefully analyzing the cost of all related and important factors.

A. Different Coherency Methods

There are three kinds of coherency methods: *CH*(Cache Hardware method), *CS*(Cache Software simulation method) or *NC*(uncacheable method). To decide which method is the best, we choose the ethernet card driver as an example to analyze. First we calculate the cost and make clear the side effect of each method as Table I shows. *U* and *K* are buffer names, (*N*) following them means that the buffer is uncacheable. Next we describe the processing details of the network sending and receiving.

During sending data by DMA, in the *CH* method the kernel will copy the user data from a cacheable buffer *U* to a cacheable buffer *K* in the TCP/IP stack, and then pass the buffer *K* to the ethernet card driver. Finally the driver needs to clean data cache before starting the DMA transfer. In *CS* method, the behavior is the same as described above except that the data cache clean operation is implemented by accessing enough garbage memory. In *NC* method, the kernel will copy the user data from a cacheable buffer *U* to an uncacheable buffer *K*, then pass the buffer *K* to the ethernet card driver. The driver directly starts the DMA transfer without any cache coherence operations. During the whole process, the cost of the *CH* and *CS* methods contains two parts: copying the data from a cacheable buffer to a cacheable buffer and cleaning the data cache, while the cost of *NC* method includes only copying data from a cacheable buffer to an uncacheable buffer. Looking into more details, *CH* and *CS* indeed can make use of locality to accelerate the copy process from the buffer *U* to the buffer *K*, but data cache will be polluted because the buffer *K* is useless after sending packets which means that the buffer *K* is one-off.

During receiving data by DMA, in the *CH* method kernel will pre-allocate a cacheable buffer *K* and flush the corresponding part of the data cache for this allocated buffer. Then kernel starts DMA transfer and waits until it finishes. After receiving a packet, the driver passes it to the TCP/IP stack and the payload of the packet is copied from the buffer *K* to the cacheable buffer *U* which is used by the application. In *CS* method, the behavior is the same except that the data cache flush operations are implemented by accessing enough garbage memory. In *NC* method, the kernel will pre-allocate an uncacheable buffer *K*, start DMA and wait until it finishes. After receiving a packet, the driver passes the buffer *K* to the TCP/IP stack, and the stack will read the packet header several times to do processing, for example getting IP address. Finally the payload of the packet is copied from

		send	receive
<i>CH</i>	process	1 copy from U to K 2 clean data cache	1 clean&invalidate data cache 2 copy from K to U
<i>CS</i>	process	the same as <i>CH</i>	the same as <i>CH</i>
	side effect	1 more data cache pollution	1 more data cache pollution
<i>NC</i>	process	1 copy from U to $K(N)$	1 copy from $K(N)$ to U
	side effect	1 accessing uncacheable memory is slower 2 no data cache pollution 3 no cache clean operation	1 Accessing uncacheable memory is slower 2 no data cache pollution 3 no cache invalidate operation

TABLE I
DMA SEND AND RECEIVE COST ANALYSIS

the buffer K to the cacheable buffer U . During the whole process, the cost of the *CH* and *CS* methods contains two parts: flush operations and the data copying from a cacheable buffer to another cacheable buffer. While the cost of *NC* method includes the TCP/IP processing on an uncacheable buffer and copying payload from an uncacheable buffer to a cacheable buffer. Differing from the sending scenario, the *CH* and *CS* methods can make use of locality to accelerate not only the copy processing but also the TCP/IP processing on the buffer K , while the *NC* method will make the TCP/IP processing much slower because the buffer K is uncacheable. However, the disadvantage of *CH* and *CS* methods is that the cache invalidate operation will pollute data cache, which is a heavy cost in Unity system.

1) *Cost calculation of different methods:* In Unity system, cache line size is 8 words (32 bytes) which can be read or written in one burst transfer. So for simplicity, all the following costs are defined in unit of 8 words transfer. We define $C_{br} = 22$ cycles as data cache read cost, $C_{bw} = 12$ cycles as data cache write cost, $C_{cr} = 105$ cycles as uncacheable memory read cost, $C_{cw} = 48$ cycles as uncacheable memory write cost, CL_{total} as total data cache lines, $L_n = \lceil S/8 \rceil + 1$ as data size, R_d as data cache dirty ratio, $L_{max} = \lceil 1536/32 \rceil + 1 = 49$ as max size of ethernet packet, $C_{way} = 4$ as data cache set-associate size. We assume the cost of reading and writing buffer U , buffer K as BC_{ur} , BC_{uw} , BC_{kr} and BC_{kw} . To simplify the equation, we omit the cost of data cache written back.

First we calculate the cost of the three methods for sending data by DMA. In *CH* method:

$$\begin{aligned} C_{ch} &= BC_{ur} + BC_{kr} + C_{clean} \\ &= BC_{ur} + L_n * C_{br} + CL_{total} * R_d * C_{bw} \end{aligned} \quad (1)$$

In *CS* method:

$$\begin{aligned} C_{cs} &= BC_{ur} + BC_{kr} + C_{softclean} \\ &= BC_{ur} + L_n * C_{br} + L_n * C_{way} * C_{br} \\ &= BC_{ur} + L_n * C_{br} * (1 + C_{way}) \end{aligned} \quad (2)$$

In *NC* method:

$$\begin{aligned} C_{nc} &= BC_{ur} + BC_{kw} \\ &= BC_{ur} + L_n * C_{cw} \end{aligned} \quad (3)$$

From the equations (2) and (3) we know that *NC* method is definitely better than *CS* method because *CS* reads much more data. Now we only need to compare *CH* and *NC* methods. According to the equations (1) and (3) the differences between *CH* and *NC* are positively linear with cache dirty ratio and negatively linear with L_n . *NC* will have more benefits on the high dirty ratio and the small size packets. Considering the data cache pollution, *NC* is expected to work better than *CH*. Now we move to the scenario of receiving a packet.

In *CH* method:

$$\begin{aligned} C_{ch} &= C_{clean} + BC_{kr} + BC_{ur} + BC_{uw} \\ &= L_n * C_{br} + CL_{total} * R_d * C_{bw} + BC_{ur} + BC_{uw} \end{aligned} \quad (4)$$

In *CS* method:

$$\begin{aligned} C_{cs} &= C_{softclean} + BC_{kr} + BC_{ur} + BC_{uw} \\ &= L_n * C_{br} + C_{way} * L_{max} * C_{br} + BC_{ur} + BC_{uw} \\ &= (L_n + C_{way} * L_{max}) * C_{br} + BC_{ur} + BC_{uw} \end{aligned} \quad (5)$$

In *NC* method:

$$\begin{aligned} C_{nc} &= BC_{kr} + BC_{ur} + BC_{uw} \\ &= L_n * C_{cr} + BC_{ur} + BC_{uw} \end{aligned} \quad (6)$$

From the equations (5) and (6) we know that *CS* is worse than *NC* because it reads much more data than *NC*. The difference between *CH* and *NC* is similar to the above situation. *CH*'s cost is linear with cache dirty ratio: it costs more under high dirty ratio. When considering which is better, we observe on one hand *CH* has to do data cache invalidation, which hurts performance in Unity system. However, on the other hand, in *NC* method the cost of the TCP/IP processing on uncacheable memory is also high. Hence we cannot conclude directly which one is better.

Considering the overall performance of both sending and receiving, we believe that uncacheable memory has a great chance to outperform cacheable memory in sending packets, because it can remove the heavy coherency cost and avoid much data cache pollution. In order to verify this argument, we conduct several experiments in next section to analyze *CH* and *NC* methods.

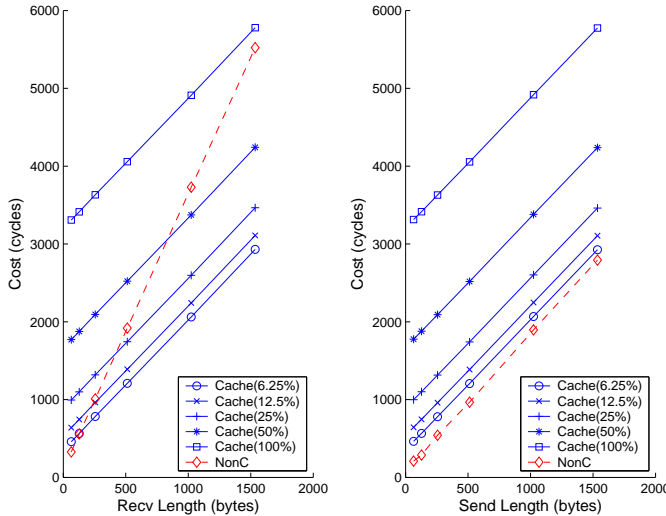


Fig. 1. Recv and Send Performance CH vs NC

2) *CH vs NC by experiments*: To evaluate the cost and the TCP/IP processing performance improvement by uncacheable method compared with hardware cacheable method, we design a simplified experiment according to the TCP/IP processing described in Table I. There are two factors in the experiment, one is the data length which determines the cost of data copy operations and the other is the data cache dirty ratio which determines the cost of data cache’s clean&invalidate operations. We implement a kernel module to measure the cost of sending and receiving packets by these two methods. The results are separately shown in Fig. 1. In the diagrams Cache(6.25%) means cacheable method with 6.25% data cache dirty ratio and NonC means uncacheable.

Fig. 1 shows that uncacheable method has distinct advantage over cacheable method when sending packets with any size and receiving packets with small size. The results convince us that it’s likely to gain benefits from uncacheable method when sending packets. But for receiving packets, uncacheable method costs too much because of copy and TCP/IP processing. As a conclusion, we observe it’s much harder to directly apply uncacheable memory to receive.

B. Uncacheable Memory Method Implementation

By using uncacheable memory in Unity system, two of the three coherency operations can be eliminated. Meanwhile, we avoid much cache pollution during TCP/IP processing.

1) *Uncacheable Page Table*: In Unity system TLB can directly access the page table entry in main memory, thus there will be no coherency problem by using uncacheable page tables. In Unity Linux, we allocate all page tables including both page table directory and page table entries in uncacheable memory.

2) *Uncacheable skb*: The socket buffer (skb) is used to represent a packet in the TCP/IP stack of Linux. In general

the skb data field is allocated in cacheable memory. We add an option in skb allocation interface to decide whether to allocate skb’s data in cacheable memory or in uncacheable memory. For sending, by default we allocate all skbs data in uncacheable memory. This optimization removes the cost of cache coherency operations during sending and avoids data cache pollution and improves the network performance, which will be shown in next section. .

V. SYSTEM PERFORMANCE EVALUATION

In this section we evaluate how the uncacheable memory improves the system performance with Netperf, Lmbench and Modified Andrew benchmark.

Based on these benchmarks, our results show that uncacheable memory: (1)improves substantially the sending performance. (2)reduces the program’s startup cost by speeding up application’s initialization and file’s initial access. (3)will not hurt the overall performance of the operating system. As a conclusion, almost all applications can benefit from these optimizations. Next we will show each experiment in detail.

We run the experiments on a 160 MHz Unity network computer with 256 MB DRAM and a SoC build-in 10M/100M ethernet card. We use a Dell 4600 as the server during the network benchmark test, which has two Intel Xeon PIII 700 MHz processors with 4 GB DRAM and 1000M/100M ethernet card. The server runs Red Hat Enterprise Linux AS 3, and the Unity network computer connects with the server by a 100M switch. We ran each of the benchmark programs five times. Most of the benchmark programs themselves also loop several times over their respective routines. We report the average result for the total number of iterations. All benchmarks are executed in single-user mode.

A. Netperf Benchmark

In this section we use netperf-2.3 [19], which is to measure various aspects of networking performance. It mainly focuses on bulk data transfer and request/response performance using TCP or UDP and the Berkeley Sockets interface. We select two tests to evaluate the throughput of TCP send and TCP Request/Response. Both two tests are based on TCP connection which is mostly used in Unity network computer.

The TCP sending test results are shown in Fig. 2. X axis is sent message size and Y axis is the throughput normalized by ‘ori 64’ (64 bytes cacheable memory). Sending performs very well in uncacheable memory. The improvement is from 9% to 15%, and the average improvement is 12.4%. Even though the packet grows larger, the benefits are still very high, which means that send throughput can get benefits regardless of the packet size.

The TCP request/response test results are shown in Fig. 3. X axis is request message size. For each request message size, we list four kinds of response message size to compare, label ‘ori 128’ and ‘opt 128’ mean response message size is 128 for cacheable memory and uncacheable memory, the rest may

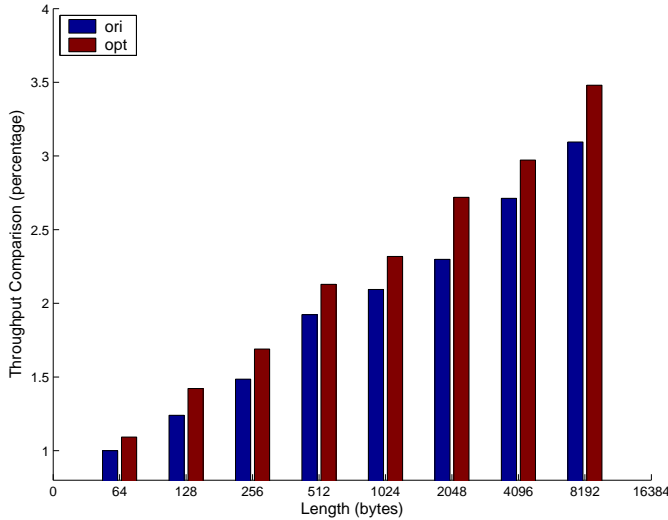


Fig. 2. Netperf TCP_STREAM Send Performance

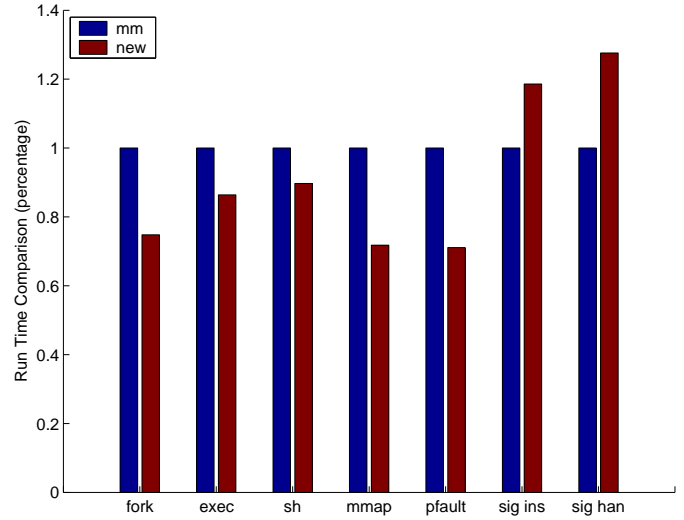


Fig. 4. Lmbench Performance

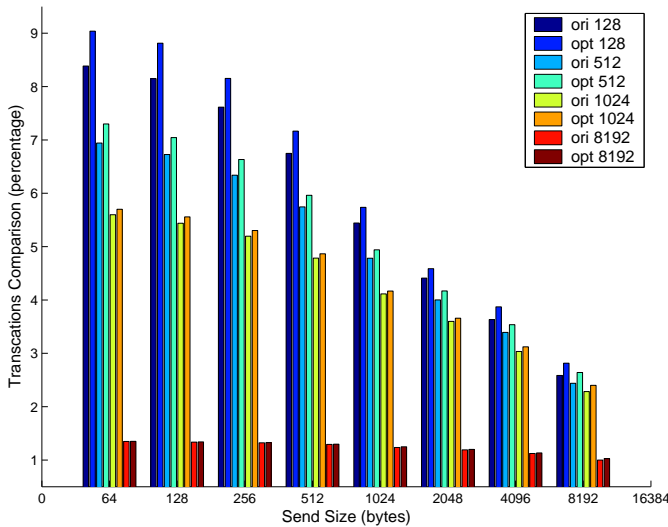


Fig. 3. Netperf TCP_RR Performance

be deduced by analogy. Y axis is the transaction throughput normalized by 'ori 8192'. The transaction throughput is also improved much from 4% to 9% regardless of the send message size. This proves that the network performance benefits much by uncacheable memory.

Our network benchmark shows that network throughput is improved in both tests. Sending benefits from the uncacheable memory. And in the TCP request/response benchmark which is close to the real application's behavior, the transaction throughput is also improved much. The results mean that uncacheable memory always improves the network throughput performance in Unity system.

B. Lmbench Benchmark

Let's see how the uncacheable page table affects the operating system performance. We use Lmbench [20] which

provides a suite of benchmarks to measure system latency and bandwidth. Each benchmark captures some unique performance aspects of the system. Lmbench is widely used by many system designers. We use almost all the OS related benchmarks of Lmbench to evaluate Unity Linux. Only network benchmarks are not included because we use Netperf for that purpose. The bandwidth benchmarks we use cover TCP(using localhost), pipe, cached file read, memory copy(bcopy), memory read and memory write. Latency benchmarks cover context switching, networking(connection establishment, pipe, TCP, UDP), file system creates and deletes, signal handling, system call overhead and memory read latency. Most results show it causes performance varies less than 2%, which can be omitted. It means that uncacheable page table will not hurt the system performance in both latency and throughput.

But still a few aspects change significantly, which are shown in Fig. 4. The label 'new' means the uncacheable page table optimization and label 'mm' means no optimization. All results of the optimization version are normalized by the corresponding non-optimization one. We can see that the run time of fork, mmap and page fault improves most, from 25% to 29%, also exec and sh improve 15%-18%. These operations are related to a new process creation(fork, exec), initial running(page fault), initial accessing the file(mmap, page fault), so much page table processing are involved in, such as clearing the page directory table, modifying the page table entry permission. Using uncacheable page table will eliminate all the cache coherency requirements and improve the performance most. We are sure that applications would spend much less time starting up when using uncacheable page table, and access file more quickly at the time of initial using.

A strange problem is that the signal processing performance is reduced about 20%-25%. Since there is no enough evidences to prove that signal handling has any additional

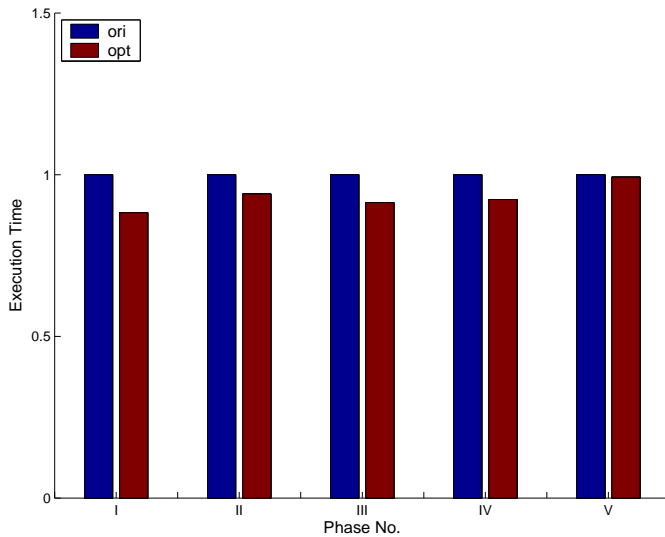


Fig. 5. Andrew Benchmark

cache coherency requirement or any page table processing, we can't explain this exception well so far.

C. Modified Andrew Benchmark

The Modified Andrew Benchmark, developed from the Andrew Benchmark at CMU [21], has been used at the University of Berkeley to benchmark BSD FFS and LFS. Nowadays it is widely used as a standard for comparing Unix file system performance.

The results are shown in Fig. 5. The label 'opt' means optimization with both uncacheable page table and uncacheable skb, and the label 'ori' means no optimization. All results of the optimization one are normalized by the corresponding non-optimization one. As expected, execution time reduces from 6%-12% for the first four phases. For Phase V, the time reduction is less than 1%. This is reasonable because only Phase V heavily depends on computation instead of I/O.

D. Summary

From these benchmarks, we can conclude that uncacheable memory can really improve the system performance much. Network sending and applications startup gain great benefits about 4%-29%. Based on these results, We believe by using uncacheable memory with careful design, overall performance of Unity system will outperform the implementation by simply using cacheable memory and cache hardware operations.

VI. CONCLUSION

This paper focuses on the uncacheable memory usage in modern computer systems. It improves the system performance by eliminating coherency operations and avoiding much cache pollution. To use uncacheable memory, we must

consider the uncacheable memory cost [22], the cache coherency operation cost together with the locality of program. In our experiments we implement uncacheable page table and uncacheable skb for sending packets. The results show that it does improve Unity system performance about 5% to 29% compared with the cacheable memory method.

It's time to consider the fields the uncacheable memory can be applied in. Many embedded architectures are widely used nowadays. They have a lot of their own design specialties due to the limitation on energy cost, area size and design complexity. In the future we will study how to use uncacheable memory to accelerate network receiving operation and find more areas to utilize uncacheable memory well.

REFERENCES

- [1] J. L. Hennessy and D. A. Patterson, *Computer Architecture-A Quantitative Approach*. Los Altos, CA: Morgan Kaufmann Publishers, second ed., 1996.
- [2] M. Cekleov and M. Dubois, "Virtual-address caches part 1: Problems and solutions in uniprocessors," *IEEE Micro*, 1997.
- [3] P. J. Denning, "Virtual memory," *ACM Computing Surveys*, 1970.
- [4] P. J. Denning, "Virtual memory," *ACM Computing Surveys*, 1996.
- [5] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, 1982.
- [6] L. Zhao, S. Mäkinen, R. Illikkal, R. Iyer, and L. Bhuyan*, "Efficient caching techniques for server network acceleration," 2004 Advanced Networking and Communications Hardware Workshop, 2004.
- [7] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," (Stockholm Sweden), Proceedings of the Tenth Annual International Symposium on Computer Architecture, Jun 1983.
- [8] A. J. Smith, "Bibliography and readings on cache memories," *Computer Architecture News*, Jan 1986.
- [9] A. J. Smith, "Second bibliography on cache memories," *Computer Architecture News*, Jun 1991.
- [10] M. Cekleov and M. Dubois, "Virtual-address caches, part 2: Multiprocessor issues," *IEEE Micro*, 1997.
- [11] D. Clark and et. al., "An analysis of tcp processing overhead," *IEEE Communications*, Jun 1989.
- [12] A. Foong and et al., "Tcp performance analysis re-visited," Proceedings of the International Symposium on Performance Analysis of Systems and Software, Mar 2003.
- [13] Intel, "Write combining memory implementation guidelines," Tech. Rep. 244422-001, Intel Corporation, 1998.
- [14] D. Tong, P. Lo, K. Lee, and P. Leong, "A system level implementation of rijndael on a memory-slot based fpga card," Proceedings of the 2002 IEEE International Conference on Field Programmable Technology (FPT), 2002.
- [15] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: a first step towards software power minimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1994.
- [16] A. Sama, J. F. M. Theeuwens, and M. Balakrishnan, "Speeding up power estimation of embedded software," in *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, ACM Press, 2000.
- [17] W. Tang, R. Gupta, and A. Nicolau, "Power savings in embedded processors through decode filter cache," Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition, 2002.
- [18] MPRC, "Unity soc architecture manual handbook (version 1.3)," tech. rep., Peking University Microprocessor Research Center, 2002.
- [19] I. N. Division, "Netperf: A network performance benchmark, revision 2.1," tech. rep., Hewlett-Packard Company, 1996.
- [20] L. W. McVoy and C. Staelin, "Imbench: Portable tools for performance analysis," in *USENIX Annual Technical Conference*, 1996.
- [21] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Trans. Comput. Syst.*, 1988.
- [22] I. Goddard, "Division of labor in embedded systems," *Queue*, 2003.