Dual-Layered File Cache On cc-NUMA System

ZHOU Yingchao[1,2], MENG Dan[1], MA Jie[1]
[1]National Research Center for Intelligent Computing Systems, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080, P.R. China
[2]Graduate School of the Chinese Academy of Sciences
{yc_zhou, md, majie}@ncic.ac.cn

Abstract

CC-NUMA is a widely adopted and deployed architecture of high performance computers. These machines are attractive for their transparent access to local and remote memory. However, the prohibitive latency gap between local and remote access deteriorates applications' performance seriously due to memory access stalls. File system cache, especially, being shared by all processes, inevitably triggers many remote accesses. To address this problem, we suggest and implement a mechanism that uses local memory to cache remote file cache, of which the main purpose is to improve data locality. Using realistic workload on a two-node cc-NUMA machine, we show that the cost of such a mechanism is as low as 0.5%, the performance can be increased 14.3% at most, and the local hit ratio can be improved as much as 40%.

1. Introduction

Cache Coherent Non-Uniform Memory Access (cc-NUMA) multiprocessors provide transparent access to local and remote memory. However, the access latency gap between them is very high. For example, benchmark on AMD *Opteron* 246 model shows a local access latency of 70ns and a one-hop remote access latency of 104ns, the gap exceeds 48% [1].

The prohibitive remote access latency stalls execution streams, and eventually deteriorates applications' performance. Popularity of data-intensive applications, including science computing applications, multimedia applications and database applications, makes things even worse. To counteract the negative effects, data locality is definitely a very important issue on such architectures [4].

As a deep insight, remote accesses can be broadly classified into two groups. The first group is anonymous data. Heap and stack are candidates of this kind. The second group is buffer cache of block device. To make things easier, we consider memory mapped accesses of files to be in the first group. The access of group one is issued directly. Operating systems have no idea of access frequency / recency and exact time of dirtiness of the data. Early work on page migration and replication [2,3,4] introduced new hardware components, including registers and monitor circuits, to record page access patterns, which in turn is used to help for determination of page migration or replication. However, the cost of such a design is too expensive, especially when taking into account the uncertainty of performance advantage. On the other hand, the access of group two is issued through file system interface such as *read / write / sendfile*, and the operating system can acquire full knowledge of access patterns. It is suggested that a low-cost software mechanism can be advanced to increase locality of this kind of accesses.

In this paper, we pursue the idea of using dual-layered file cache to improve locality of accesses to files. In the work, normal cache is abbreviated as NC, and cache of normal-cache, which cache remote NC on the local node, is abbreviated as NCC. When an access to a file block is issued by an application, the operating system firstly finds the NC entry of the block. If the NC entry is found on a remote node, it activates the NCC-generator, in which a NCC entry will be generated on the local node conditionally. Any later accesses to the block on the same node can be satisfied by the generated NCC entry. In such a way, the advantage of page replication and migration can be obtained without any hardware modifications. Furthermore, the dual-layered cache mechanism makes it possible to implement independent replacement and prefetching algorithms of the NCC and the NC. That is an important property for releasing the indirect cost of memory pressure resulted from page replication when memory is tight.

The idea had been implemented on the linux kernel. Using realistic workloads on the implementation, we reveal that data locality can be improved substantially. Experimental results on a two-node cc-NUMA machine show that the local hit ratio can be improved as much as

1-4244-0054-6/06/$20.00

40%. For good candidates, the maximum speedup of 14.3% can be obtained over the default first-touch policy of the untouched linux kernel, and for the worst candidates, the slowdown is no more than 1%.

The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 describes the detail of the design and implementation, and analyzes the costs and advantage introduced by the NCC. Section 4 shows workloads and experimental environment. Section 5 illustrates results of experiments. And finally, section 6 concludes the paper and depicts our near future plan of optimizations.

2. Related Work

Most commercial operating systems nowadays have integrated the static page placement mechanisms [5,6,7], which attempt to initially locate each physical memory page in the memory providing the highest percentage of local memory accesses [8]. The static mechanisms consist of first-touch and round-robin. They are the most basic optimization for cc-NUMA architectures. However, the complicated mixture of process movement, data sharing between processes and application behaviors changing over execution time makes static mechanisms hard to work effectively. For example, Kenneth M. Wilson reported in [9] that local hit ratio is only 34% with first-touch when running *TPC-C* on a four-node cc-NUMA multiprocessor machine, and in [8] they found it decreased to 25% with round-robin.

Another work has focused on dynamic page placement mechanisms [2,3,4,10,11,12]. With such mechanisms, after the initial placement, pages are replicated or migrated due to changes of the application's behavior. Page replication is the copying of physical pages on multi-nodes so that two or more processors have a local copy. Page migration is the moving of pages closer to the processor accessing them. Dynamic mechanisms are more complicated and effective than static ones. In [8], Kenneth M. Wilson discovered that by using dynamic mechanism provided in [4], local hit ratio can be increased to 73% for *TPC-C* on the same configuration. Although being closely related, our work differs from these studies in three aspects: (i) the focus of our work is the file system cache rather than whole physical pages, and we manage the replicated pages independently in a cache way rather than just treat them as a copy of other pages; (ii) our work is purely based on software rather than any hardware modification, while most of the studies above involve hardware support; (iii) our work is carried out on real platform rather than in simulation environment, which makes our work more realistic and relevant to the truth.

The above studies of dynamic page placement mechanisms use history access information as the criterion of migration or replication. However, history information cannot always imply future behaviors nicely. For example, the past reference history of application's stack is obsolete because it is not correlated with the actual status of the computation which is reflected by the new mapping of threads to processors. To address this problem, the work in [13,14,15] suggests to take into account scheduling information when implementing page migration and replication policy.

3. Policy Framework

This section firstly provides a detailed analysis of the problem we are trying to resolve, in terms of benefits and costs of the dual-layered file cache. And then, we describe the detailed issues related to the implementation.

3.1 Problem Statement

File cache is accessed through filesystem interface such as *read / write / sendfile* by applications. The idea of dual-layered cache is that each node maintains an independent NCC, which consists of copies of normal file cache pages on remote nodes. Once an entry enters the NCC, the following operations issued by filesystem interface will select the local kernel page in the NCC. In this way, the probability of remote accesses can be decreased. Figure 1 shows the detailed information of effects of NCC on *read* interface, which involves a

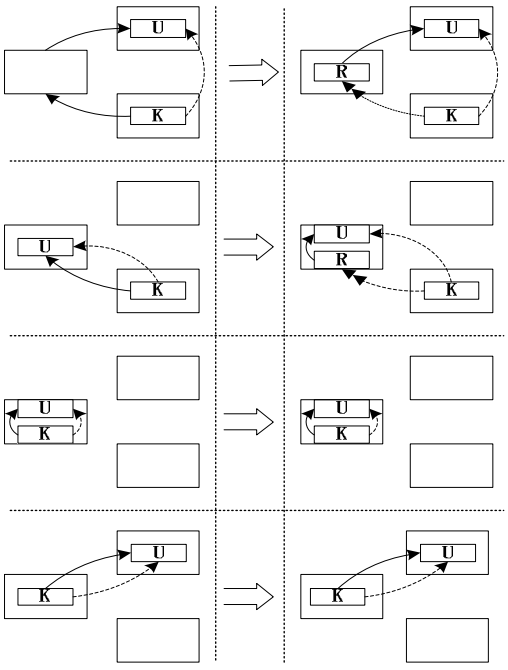

Figure 1 detailed information of NCC impaction on read

memory copying from a kernel-space page to an user-space page. In figure 1, the left side stands for four distributions of pages without NCC, and the right side stands for the correspond distributions of pages with NCC. In each distribution, we suppose the process running on the cpu of the left node. In the figure, each bigger rectangle represents a node, each smaller rectangle represents a page, K denotes a NC entry, U denotes an user-space page, R denotes a NCC entry, the solid arrow represents really accessing paths, the dashed arrow represents desired accessing paths, and the dashed double-arrow represents a NCC entry generator. From the figure, it can be seen that with the NCC, the cost of reading on the first distribution is the same with the fourth, and the second is the same with the third. Each of the two situations involves one less time remote access. The reduction to remote memory access can eventually speed up applications. However, the speedup will be eliminated by the costs introduced.

The costs can be classified into five categories. The first category is the costs of recording information to help determining whether a NCC entry should be generated on a certain node, which involves the costs of some counting and comparing operations, so they should be very small. The second category is the overhead of NCC entries generation, which includes allocating of new pages and additional memory copyingw. The third category is the management overhead of the NCC, which includes the replacement of entries and shrinking of the NCC. In the future, it probably includes entries prefetching. The fourth category is the cost of keeping consistency between the NCC and the NC. It can be copying of dirty data to each NCC or discarding entries in NCC depends on implementation. The fifth category is the indirect memory pressure resulted from NCC. The optimization we considered is to shrink NCC much more rapidly than NC. With such optimization, when memory is exhausted in the worst situation, the NCC will be empty, and it seems there is no NCC.

Considering the potential benefits and costs, we can determine good candidates and bad candidates. There are three kinds: (i) applications have little sense to the NCC; (ii) applications will be harmed by the NCC; (iii) applications will benefit from the NCC. We review them in details.

Obviously, applications only accessing few quantities of files will have little sense to the existence of NCC. They are neither good nor bad candidates. Applications accessing some files only once or very few times will be harmed by the NCC. For this kind of applications, the additional costs introduced by the NCC include the first kind and perhaps the others due to miscarriage of justice at the process of NCC entries generation. But compared to the long elapse of accessing disk, the above costs are relevant insignificant. Thereby we also consider this kind of applications neither good nor bad candidates. Applications accessing many files will exhaust the memory. With optimization mentioned above, such applications will also be not sensitive to the NCC. Applications accessing some files repeatedly are the most complicated category. The potential good candidates and bad candidates are all belong to this category.

Single thread applications accessing some files repeatedly on system with frequent thread migrations are potential bad candidates. Consider the following situation: thread migrating to the first node produces many NCC entries on that node, and then the thread migrates to the second node, NCC entries generation requesting on node two probably results in memory tight and leads to recycle of NCC entries on node one. While later, the thread will probably migrate to the first node again. Such a process ends with NCC thrashing. If the memory is abundant, frequently migrations will make each NC entry produce many NCC entries on other nodes other than the node holding it, and which will probably speed up the application.

Multi-threaded applications sharing some files are potential good candidates. The system will satisfy accessing of each thread by the local NCC entry or the NC entry if thread is running on the node holding NC entry.

Although potential benefits and costs varied with applications' behaviors, the throughput of the whole system is not so uncertain. If the memory pressure is light, files' blocks accessed frequently tend to exist in many nodes' NCC, the throughput as a whole will be increased. And when memory pressure becoming tighter and tighter, more rapidly shrinking of the NCC will soon make it empty, the system will work as the NCC does not exist at all.

3.2 Implementation details

This sub-section provides implementation details for our dual-layered file cache mechanism. As shown in figure 2, the NC as a whole distributes on nodes of the

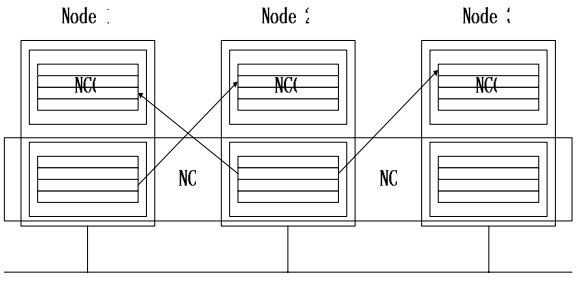

Figure 2 example of distributions of NC and NCC

system. And each node owns an independent NCC. On a n node cc-NUMA system, there exists at most one NC entry, at most $n - 1$ NCC entries for each file block. A NC entry may correspond to $n - 1$ NCC entries on other nodes. On a node, the NC entry and the NCC entry related to the same block of the same file cannot co-exist.

We choose Linux as the platform due to its' popular deployment and open source. The implementa-tion consists of four aspects: (i) the modification to the linux kernel; (ii) the NCC generation; (iii) the manage-ment of NCC entries; (iv) the coherence between NC entry and NCC entries. The following describes them in details.

Modification to the kernel Our work is based on Linux kernel 2.6.7. In this kernel, pages of file cache are managed through two data structures. The first is the radix tree, a search tree used to find a file block's cache page through offset quickly. The second is the doubly linked lists. For *cc-NUMA*, each node covers some zones, and each zone consists of two lists: one is the active page list and the other is the inactive page list. The active list links pages that are accessed recently, and the inactive list links pages that are not accessed for a long time. When memory is tight, the kernel transfers pages from active list to inactive list and reclaims pages from inactive list in LRU way. To make dual-layered cache mechanism cooperate with the original file cache management system, two changes has been made to the linux kernel:

A pointer has been added to the page structure. The pointer is used to record corresponded NCC entries if the page is an entry of the NC. When the kernel uses the radix tree to find page cache for a file block, the NC entry will be found firstly, and then it detects whether a local NCC entry exist. If so, the local NCC entry will be returned as the cached page.

Hooks for reclaiming NCC entries have been added to the kernel path for trying to free pages. In order to make the system works the same as the original one when memory is tight, the NCC shrinks much more rapidly than the NC. In our implementation, we reclaimed twice from the NCC than from the NC. For instance, when three pages are asked to free, we will try to free two from the NCC and one from the inactive list (the NC).

Generation of NCC entries When pages in the NC are accessed by a remote node, the counter of access by that node is increased. If the counter reaches a pre-defined threshold, a NCC entry on that node will be generated. In our implementation, the threshold is set to two. In order to avoid potential thrash of NCC generation, a timer for memory tight checking has been added. The timer checks usage condition of the memory every second, and enable or disable genera-tion of NCC entries for the following access due to the proportion of free memory.

Management of NCC entries There are three kinds of cache management algorithms. The first kind is frequency / recency based algorithms, including LRU [16], LRU-2 [17, 18], 2Q [19], LIRS [20, 21], LRFU [22], MQ [23], ARC [24], and so on. The second kind is pattern based algorithms, including SEQ [25], EELRU [26], DEAR [27], UBM [28], PPC[29], and so on. The third kind is hint-based algorithms, including application-controlled cache management [30, 31] and compiler hinted cache management [32]. For the NCC, the algorithm is expected to make frequently accessed pages except those burst accessed ones existing as long as possible, and it should be easy to implement. Therefore, a variation of MQ seems to be a good choice. The algorithm uses eight LRU queues, $Q_0, Q_1, \ldots, Q_7$, where Q_i contains pages that have been accessed at least 2^i times but no more that $2^{i+1} - 1$ times recently. Within a given queue, pages are ranked by recency of access according to LRU. On a NCC entry hit, the page frequency is increased and the page is placed at the MRU position of the appropriate queue. Each time the kernel refills inactive list from active list, the NCC queues are also scanned, and each page's frequency is decreased until the frequency reaches one. At the same time, the queues are adjusted according to the new values.

When pages are asked to free from the NCC, pages selected from Q_0 to Q_7 orderly. In order to shorten lives of burst accessed pages, all NCC entries are discarded when the corresponded NC entry is selected to be reclaimed.

Cache Coherence To make the system work correctly, the coherence between the NC and the NCC must be maintained. Operations that break coherence include *write* and *memory mapped write*. Two obvious methods can be used to keep the coherence. The first is to invalidate related NCC entries when a NC entry is written. The second is to write each related NCC entries along with writing to the NC entry. In our implementation, a page in NC is forbidden to generate NCC entries when it is to be memory mapped, and the existing related NCC entries are to be discarded. In this way, *the memory mapped write* cannot break coherence any more. For *write* operation to a NC entry, the existing related NCC entries are discarded, and count of access time for each node to the NC entry is reset to zero. The method adopted here is paranoia somewhat. For future optimizations, the restriction of *memory mapped read / write* operations can be released. Another potential optimization is to switch role of the NCC and the NC when NC entries are to be discarded, i.e. change a NCC entry to be a NC entry and change the original NC entry to be a NCC entry.

4. Experimental environment

This section describes the experimental environm-ent and workloads.

4.1 Platform

We have implemented our design in the linux kernel 2.6.7. The implementation consists of around 2000 lines of *C* code.

We used a small NUMA server with the following configuration:

- Two AMD 242 processors of 1.6 GHZ;
- 2G total RAM, each node owns 1G;
- NetXtreme BCM5704 Gigabit Ethernet;
- Fedora Core 3 for x86-64 is installed, and the kernel is compiled by hand.

4.2 Workloads

We used the following applications and benchmar-ks to evaluate the system:

Grep *Grep* is a frequently-used tool in shells and configurations. It is a single thread application, and uses *read* interface to copy the whole file to its' own buffer in user-space. Here, to simulate a configuration procedure, we ran *grep* to find a string in the /etc directory 120 times in several parallel shells.

Tar + Gcc + Cscope The combination consists of a previous execution of *tar* and following concurrent executions of *gcc* and *cscope*. It represents the typical workload in a code development environment. The input files of these three applications are linux kernel source code of 2.6.12. We used *tar* to make a *zipped* tar package of the source codes; used *gcc* to compile a kernel image for our platform, and used *cscope* to build indexed files for fast search of symbols. All these applications access files through *read* interface.

Iozone *Iozone* is a file-system benchmark. It generates and measures a variety of file operations, include *write, read, re-read, re-write, backward read, stride read, fread, fwrite, random read / write* and so on. We carried out the evaluation in posix multi-thread mode.

Web The *Apache httpd* is a widely deployed web server. In the experiment, we used the *apache benchmark* released with the package to evaluate the impact of the NCC to the *httpd* server.

5. Evaluation of results

In this section, we examine how the NCC affects each of the applications and benchmarks.

Grep

We ran *grep* totally 120 times to find strings in */etc* directory in parallel shells. The concurrent number of shells consists of 2, 4, 6, 8 and 10. The size of the */etc* directory is 40M. In the test, we recorded three kinds of results: (i) average run-time of each grep; (ii) average run-time of the first grep in each shell; (iii) totally local page hit number.

Table 1 gives the results of the average run time and the average run time of first *grep* in each shell. The degree of slowdown of the average run time of the first *grep* in shells means that the cost of our system is less than 0.6%. And when more *greps* ran in parallel, the more speedup could be obtained. The reason for such phenomena probably resides in the processors' cache. When more *greps* run in parallel, the disorder access pattern should invalidate the processors' cache frequently, which in turn will reduce the hit ratio. When hit misses in the processors' cache, data will be accessed from the memory, and the existing of the NCC can reduce time spent in the memory access.

t	Original		NCC		Speedup	
	First	Avg	First	Avg	First	Avg
2	11.89	3.36	11.92	3.36	-0.2%	0%
4	13.84	6.98	13.92	6.90	-0.6%	1.1%
6	15.95	10.38	15.99	9.98	-0.3%	3.9%
8	18.53	13.78	18.64	12.93	-0.6%	6.2%
10	20.79	17.12	20.92	15.75	-0.6%	8.0%

Table 1 results of *grep* (unit: *s*)

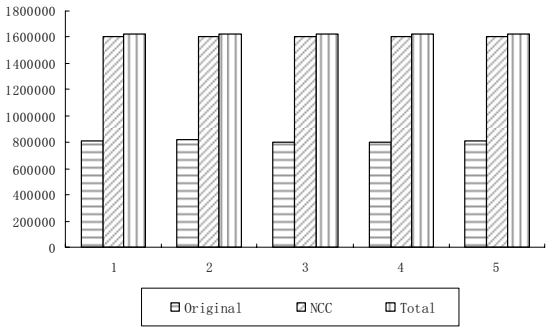

Figure 3 local hit number of *grep*

Figure 3 gives the totally local hit number. It shows that local page hit ratio is about 50% for the original system, and about 98.8% for the NCC. In our option, on an untouched kernel, the local hit ratio is *1/n* for a *n-node* system. It is clear the more the number of nodes, the greater the gap between the local hit ratios.

Tar + Gcc + Cscope

We ran the three applications separately firstly, and then we ran them in combination mode. In the combination mode, a previous execution of *tar* is followed by concurrent executions of *gcc* and *cscope*. In

all tests, the execution time and local page hit number is recorded.

Table 2 gives the results of execution time and local page hit number. From the table, we can see that the slowdowns for separately execution are 0.48%, 0.16% and 0.59%. The *gcc* shows a particular small slowdown. The possible reason is that *gcc* access *C* header files many times, and the NCC can counteract costs of theses accesses. It can be verified from the increment of local page hit number. The slowdowns of separate execution are due to the costs introduced by the NCC. However, when the applications were executed in combination mode, the advantage of the NCC could overwhelm the costs. As a result, the speedup of the execution time is 6.7% and the local page hit number is improved 49.6%.

App	Original		NCC		improved	
	time	local	time	local	time	local
tar	103.1	76602	103.6	76657	-0.5	0
gcc	429.6	386070	430.3	421631	-0.2	9.2
csope	67.5	88025	67.9	88031	-0.6	0
multi	526.5	439517	491.2	657510	6.7	49.6

Table 2 results of *tar+gcc+cscope* (unit: *s* and *percentage*)

Iozone

In our experiments, the file size for *iozone* is 128 megabytes, the record size is 4 kilobytes, and number of threads includes 2, 4, 6, 8 and 10. The threads of *iozone* ran independently. In each thread, it initially creates and writes a file, and then uses other interfaces to access the file. In our configuration, the tested interfaces included *re-write, read, re-read, backward read* and *stride read*. In the tests, the size of memory is larger than total size of files, so the results are mainly determined by the performance of page cache. Table 3 gives the results of the untouched kernel, table 4 gives the results of the NCC, and table 5 gives the percentage of speedups.

For *iozone*, the remote page access is produced by thread migration among processors. The tables show that cost of NCC depressed performance of *write, re-write and read*, but the extent is very feeble. For *re-read*, the cost and advantage of the NCC counteracted, and finally resulted in a bit performance improvement. For *backward read* and *stride read*, the advantage overwhelms the cost, and obvious performance increasing can be obtained. When number of threads increased, the memory became tight, which resulted in rapidly shrinking of the NCC, and finally ended with less performance improvement. The results are consistent with our deduction.

Figure 4 gives the comparison of totally local hit number. Again we can see much higher local hit ratio for all tests.

t	write	re-Write	read	re-Read	back read	stride read
2	328	731	2087	2155	1618	1542
4	306	868	1903	2093	2179	1555
6	304	818	1938	1912	2136	1672
8	249	937	1932	2153	2077	1663
10	235	870	2191	2198	2153	1953

Table 3 results of untouched kernel (unit *MB/s*)

t	write	re-Write	read	re-Read	back read	stride read
2	328	731	2071	2146	1675	1764
4	305	866	1895	2143	2324	1752
6	304	818	1921	1915	2205	1751
8	248	934	1917	2168	2325	1807
10	235	871	2195	2204	2168	1996

Table 4 results of NCC kernel (unit *MB/s*)

T	write	re-write	read	re-Read	back read	stride read
2	0	0	-0.8	-0.4	3.5	14.3
4	-0.3	-0.2	-0.4	2.4	6.7	12.7
6	0	0	-0.9	0.2	3.2	4.7
8	-0.4	-0.4	-0.8	0.7	11.9	8.7
10	0	0.1	0.2	0.3	0.7	2.2

Table 5 results of percentage of improvement

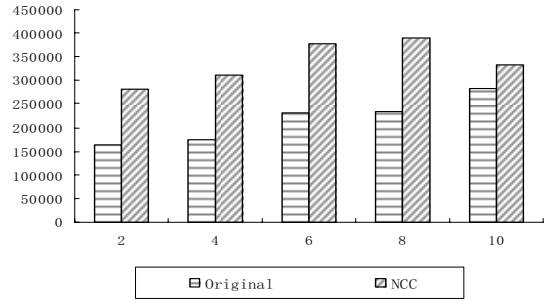

Figure 4 local hit number of *iozone*

Web

We ran *httpd server* of *APACHE* as the web server, and ran *apache benchmark* released with the *APACHE* package on the client. In our configuration, the *apache benchmark* generated 100 requests of 100 concurrent accesses to the server for a 40 megabytes file.

The *httpd server* uses *sendfile* interface to send files to the client. In the linux kernel, the *sendfile* interface firstly reads pages of the file and then writes them to the network socket, no data be copied to user-space. The *httpd server* ran a several pre-forked processes to deal with clients' requests. In such a way, when there are

concurrent requests to the same file, the NCC can be used to reduce inter-node memory accesses. In our test, the results of network bandwidth and local hit number are recorded. Table 6 shows them.

Original		NCC		Improved	
band	localhit	band	localhit	band	localhit
104.8	1041	112.6	1354	7.4%	30.1%

Table 6 results of web (unit: band *MB/s* localhit *1000*)

From the table we can see that the bandwidth is improved 7.4% and the local hit numbers are improved 30.1%.

6. Conclusions

This paper provides a dual-layered file cache solution for *cc-NUMA* architecture. The dual-layered file cache mechanism can reduce remote memory access times and in turn improve performance of applications. We evaluated the system on a range of real workloads and benchmarks, observed performance benefits of 2.2% to 14.3% in several cases, and showed that the cost was low, as much as 0.5%.

There are potential optimizations to the solution. Our near plan of optimizations involves three aspects: (i) Delayed generation of NCC entries. In current implementation, the NCC entries are generated immediately after it is verified to do so. The immediate generation could fully warm-up the processor cache, but the cost of an abundant copying probably impact performance of the running application. An alternative is to delay the generation of NCC entries until the system is idle. Using such mechanism, the cost of additional copying probably can be released. (ii) Conversion of NC entries and NCC entries. In current implementation, when a NC entry is selected to be reclaimed, all related NCC entries are also reclaimed. And the NC entry is fixed. If we can convert the actor of the NC entry and the NCC entry due to access pattern changes, potential advantage of more flexible cache management policy for the NC and the NCC can be obtained. (iii) Making memory mapped accesses of files to use NCC entries. In current implementation, when memory mapped accesses to file pages are issued, the NCC is disabled on such pages. This restriction makes file accesses through memory map unable to use NCC entries. However, such accesses are very popular on real systems, especially for executable files. So, it is necessary to release the restriction.

References

[1] AMD 64 Technology, AMD Opteron™ ProcessorBenchmarking for Clustered Systems. *http://www.amd.com/usen/assets/content_type/DownloadableAssets/dwamd_39497A_HPC_WhitePaper_FINAL.pdf*

[2] D.Black, A. Gupta, and W.D. Weber. Competit-ive management of distributed shared memory. In *Proceedings of COMPCON*, pages 184-190, March 1989.

[3] R. Chandra, S Devine, B Verghese, A Gupta, and Mendel Rosenblum. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Proceedings of conference on Architectural Support for Programming Languages and Operating Systems*, 12-24, October 1994.

[4] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 279-289. ACM Press, 1996.

[5] L. N. Bhuyan, R. Iyer, H. Wang, and A. Kumar. Impact of CC-NUMA Memory Management Policies on the Application Performance of Multistage Network. *IEEE Transactions on Parallel and Distributed Systems*, 11(3):230-251, March 2000.

[6] M. Marchetti, L.Kontothanassis, R.Bianchini and M.Scott. Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems. *Proceedings of the 9th International Parallel Processing Symposium*, pp.480-485. Santa Barbara, CA, April 1995.

[7] C.Holt, J.Pal Singh and J.Henessy. Application and Architectural Bottlenecks in Large Scale Shared Memory Machines. *Proc. of the 23rd Annual International Symposium on Computer Architecture*, pp.134-145. Philadelphia, PA, June 1996.

[8] Kenneth M. Wilson, Bob B. Aglietti: Dynamic page placement to improve locality in CC-NUMA multiprocessors for TPC-C. *SC 2001*

[9] Kenneth M. Wilson. Static and Dynamic Page Placement to Improve Locality in CC-NUMA Multiprocessors. *HP Labs Tech Rept*. HPL-98-150, 1998.

[10] Silicon Graphics. Cellular IRIX 6.4 for Origin-2000 Technical Report. *http://www.sgi.com/Technology-/Irix6.4/cellular-irix6.4tr.html*

[11] William Bolosky, Michael Scott, Robert Fitzger-ald, Robert Fowler, and Alan Cox. NUMA Poli-ces and Thiner Relation to Memory Architectur-e. *ASPLOS IV*, Santa Clara, CA, pp 212-221, April 1991.

[12] Richard LaRowe Jr., Carla Ellis, and Laurence Kaplan. The Robustness of NUMA Memory Managem-

ent. *Thirteenth ACM Symposium on Ope-rating System Principles*, pp 137-151, October 1991.

[13] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou Constantine D. Polychronopoulos, Jesus Labarta, and Eduard Ayguadé. User-Level Dynamic Page Migration for Multiprogrammed Shared Memory Multiprocessors. *International Co-nference on Parallel Processing.*

[14] D.Nikolopoulos et.al. A Case for User-Level Dynamic Page Migration. *Proc. Of the 14th ACM International Conference on Supercomputing*, pp. 119-130, Santa Fe, NM, May 2000.

[15] D.Nikolopoulos et.al. UPMlib: A Runtime System for Tuning the Memory Performance of OpenMP Programs on Cache-Coherent NUMA Multiprocessors. *Proc. of the 5th ACM Workshop on Languages, Compilers and Runtime Systems for Scalable Computers*, Rochester, NY, May 2000.

[16] R.W.Carr and J.L.Hennessy. WSCLOCK – a simple and effective algorithm for virtual memory management. In *Proc. ACM SOSP-08*, Dec 1981.

[17] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. ACM SIGMOD Conference*, May 1993.

[18] E. J. O'Neil, P. E. O'Neil, and G.Weikum. An optimality proof of the LRU-K page replacement algorithm. *J. ACM*, 46(1):92-112, 1999.

[19] T. Johnson and D. Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In *Proc. 20th VLDB*, Jan 1994.

[20] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proc. USENIX ATC*, Apr. 2005.

[21] S. Jiang and X. Zhang. LIRS: an efficient low interreference set replacement policy to improve buffer cache performance. In *Proc. ACM SIG-METRICS*, June 2002.

[22] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 50(12):1352-1360, 2001.

[23] Y. Zhou, P. M .Chen, and K. Li. The MultiQueue Replacement Algorithm for Second-Level Buffer-Caches. In *Proc. USENIX ATC*, June 2001.

[24] N. Megiddo and D. S. Modha. ARC: A Self-tuning, Low Overhead Replacement Cache. In *Proc. 2nd USENIX FAST*, Mar 2003.

[25] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. In *Proc. ACM SIGMETRICS*, June 1997.

[26] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: simple and effective adaptive page replacement. In *Proc. ACM SIGMETRICS*, May 1999.

[27] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. Towards application/file-level characterization of block references: a case for fine-grained buffer management. In *Proc. ACM SIGMETRICS*, June 2000.

[28] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. A Low-Overhead, High Performance Unified Buffer Management Sche-me that Exploits Sequential and Looping Refere-nces. In *Proc. 4th USENIX OSDI*, Oct 2000.

[29] C. Gniady, A. R. Butt, and Y. C. Hu. Program counter based pattern classification in buffer caching. In *Proc. 6th USENIX OSDI*, Dec.2004.

[30] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application controlled file caching, prefetching, a-nd disk scheduling. *ACM TOCS*, 14(4):311-343, 1996.

[31] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. ACM SOSP*-15, Dec. 1995.

[32] A. D. Brown, T. C. Mowry, and O. Krieger. Compiler based I/O prefetching for out-of-core applications. *ACM TOCS*, 19(2):111-170, 2001.