2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing

Task Optimization Based on CPU Pipeline Technique in Multicore System

Bo Wang,Yongwei Wu, Weimin Zheng
Tsinghua National Laboratory for Information Science and Technology
Department of Computer Science and Technology, Tsinghua University
Beijing, China
Email: gawain102000@163.com, {wuyw, zwm-dcs}@tsinghua.edu.cn

***Abstract*—Currently, multi-core system is prevalent in desktop, laptop or servers. The web proxy as a server provider can save network traffic overhead and shorten communication time. Especially with the fast development of wireless Internet accessing, the web proxy will take more important role. In this case, we research the web proxy behavior and exploit parallel subtasks deeply. Based on this, we propose CP (CPU Pipeline) technique to build parallel tasks in multi-core system. The result shows that our scheme can efficiently improve throughput for handling the incoming requests per second and take full advantage of the computing capacity provided by multicore system.**

***Keywords*-multicore system; task optimization; pipeline;**

I. INTRODUCTION

People have put forward the multi-core concept since early 90's. Even at that time, the single-core processor still occupies the most of that business. The design for Linux system adopts time-sharing mechanism to run much application software concurrently. Each application is allocated one given time-sliced by the process management which duties on putting one process into running queue or idle queue. Due that the time slice is calculated on the level of millisecond the users often can smoothly run several applications not conscious of any delay resulting from the short stop of an application. Linux timer is triggered at the internals of several milliseconds which can result in the execution of series of services which include process management. The parameter for time slice takes very important role in the whole concurrent design. If this value is set too long to switch to another application, users often trend to loss their patients.

Therefore, parallel execution must take full account of many factors which can affect the whole rational design. From early 90's, Due that the constant development for the hardware manufacturing process such as wafer incision, people can improve computing capacity and processor frequency to regulate time slice at a wide scope. Currently, the frequency for single-processor nearly closes its limitation. In this case, multi-core system becomes prevalent which avoids that limitation and begins toward parallel direction. In multi-core system, the process management and process queue management at best guarantee the fair balance load among different processors. Graph below shows how to migrate one process from one processor to another. The important parameter for this is the time-out value for a timer which can trigger process queue management to decide if the process migration is happened.

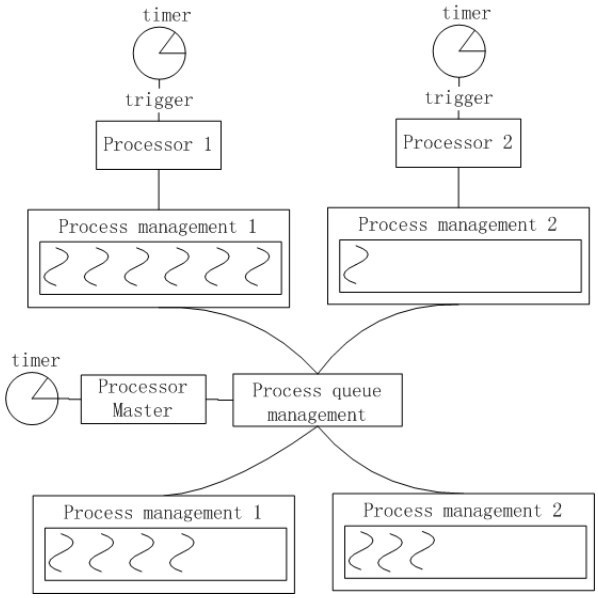

Figure 1. Linux scheduling in multi-core system.

The system performance shows some differences when we make the process or thread as our execution unit. In this paper, in order to focus on our research, we neglect their difference and use both of them during our experiments.

From above, we give enough description about low layer unit supporting parallel task execution, including hardware such as multi-core architecture and software such as process scheduler. Our parallel task developing is based on this and our application scenario is the web proxy service. The web proxy provides the service which intercepts the HTTP request from a client and after complex handling, returns the reply to this client. Its behavior is similar to the Apache web server to some extent, except that it does not produce the page content what the client really wants.

Why do we put forward one concrete application scenario with web proxy server? First, parallel computing has no general solution for each application. We can not smoothly

978-0-7695-4372-7/11 $26.00 © 2011 IEEE
DOI 10.1109/IMIS.2011.55

write parallel program like sequential program due that we must reasonably partition one big task into several parts which can keep multi-core load-balanced running. We must take account of parallelism between the partial parallel execution and the whole system. Even if you can keep favorable partial parallel execution for two-level loop statement or one subtask, maybe it only contributes few to the whole application. Second, web proxy server can stand for types of applications including Apache, Firefox or some network routing applications. These applications play an improving role to our social progress. Third, our research focuses on how to analyze the relation among different modules, including logging, accessing and filtering mechanism, cache management, header parsing, translation and transcoding. Traditional programming style emphasizes the internal structure clarity and whole logical correction, therefore, modular design is imported by much application software to guarantee the successful running. Meanwhile, the disadvantage of this style pays little attention to the subtask parallelism. Fourth, many compute-bound tasks exist in this application, including picture data transcoding, javascript file translation, header information parsing. In future, due that the wireless surfing on mobile phone which is considered as a weak computing platform, becomes more prevalent, we can predict that more computing tasks including request analyzing and CSS parsing will be transferred to the powerful computing platform like web service provider or web proxy.

Many methods have been presented to support the parallel computing on the multi-core system. OpenMP[1] can excellently decompose a complex loop statement which exists in a function into many subtasks which can be simultaneously executed on different processors. TBB[2] belongs to one kind of task-oriented programming language which builds parallel tasks to improve application efficiency. Users can focus on how to rationally partition complex relations into many parallel execution units without manually managing subtask switching. Erlang[3] language highlights concurrency, real-time, robustness, distribution and portability during application design which makes erlang prevalent in multi-core system.

From above description, we introduce CPU pipeline, give some parallel execution existing in web proxy and compare some similar places between pipeline technique and our proposed scheme. Our contribution in this paper can be summarized as below:

1) This scheme is based on one practical application and aims to acquire the favorable parallel execution through analyzing the whole architecture.
2) This scheme provides great guides for some similar application including some web services.
3) Each request from client or reply from server can be considered as one pipeline, and some intermediate states can be written to ”memory” which can keep next session valid.
4) Comparing with other methods, our proposed method (CP) shows favorable scalability.

II. Related Work

Recently, more work about the scalar architecture has been researched in order to improve web server performance including FTP transferring, network routing and network monitoring etc. they commonly aim to make the application parallel execution in order to acquire the performance boost no matter what the hardware or software is.

Mahdi proposes a high-performance network monitoring software architecture. His application background is network package monitoring. He partitions the network package workflow into three stages: accepting, flow reassembly and transmission. And he builds the parallel task on the multi-core system, which shows that great improvement has been acquired[4]. Danhua Guo proposes a high scalable parallelized L7-filter system architecture with affinity-based scheduling on a multi-core system. He treated the whole processing as this: accepting the incoming packages, pre-processing, scheduling, matching and transmitting[5]. Even if more parallel schemes are given, it lacks full analysis for the task complexity. Katerina Argyraki studies the soft router scalability with two challenges: first, the per-package processing capability of each server must scale with O(R); second, the aggregate switching capability of the server cluster must scale with O (NR). Based on this, he proposes a solution: a cluster-based architecture that uses an interconnect commodity server platforms to build software routers that are both incrementaly scalable and fully programmable[6]. Furthermore, when we take account of the order of the incoming data stream for some network applications, how to handle them one by one in multi-core system can affect the system performance to a great extent.

Mauricio Marin designs high-performance priority queues for the parallel crawlers. The processing of network crawler can operate on one same URL which wastes more computing and storage resources. His team solves this through the synchronization management for the URLs queues. They propose efficient and scalable strategies which consider intra-node multi-core multithreading on an inter-nodes distributed memory environment, including efficient use of secondary environment[7]. Based on this, more multi-core synchronization algorithms have been put forward over these years in support of the parallel application. Some research work focuses on the system bottleneck and thinks of all the ways to overcome this and make this part execute parallel[8][9][10][11][12].

We can compare our work with the above research from several sides as follows:

1) We assume that the low layer such as operating system scheduler, TCP/IP stack or communication architecture (PCIE) or transferring pattern (EDMA) works

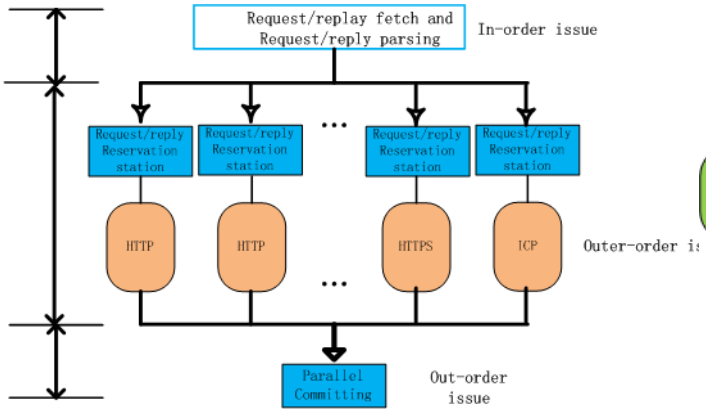

Figure 2. CP architecture

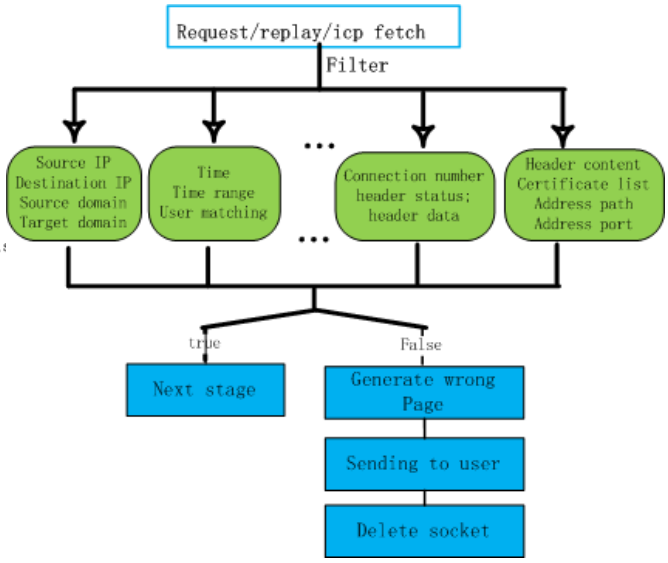

Figure 3. request or reply fetches processing.

well. Some studies modify these to acquire better performance improving.

2) Some research only partitions the whole application into several simple subtasks, in fact, one common application involves many subtasks to handle including compute-bound, transfer-bound. We take full account of the possible details and decompose these into many subtasks which support our proposed CP technique.

III. CP Technique

As far as web proxy system is concerned, it handles the incoming requests. These requests can be categorized into several types: HTTP, HTTPS, FTP, ICP, and HTCP. Each type of request will flow along different paths: some paths go through long stages and some short.

In figure 2, we partition the whole processing into three stages: request/reply fetch and parsing; package processing; parallel committing. In the first stage, the incoming package may belong to one kind of request from user, reply from server or inquiry from cache peers. The web proxy will filter some contents according to the filter tables. The corresponding filter items include source IP, destination IP, maximal connection number, accessing port, user information, HTTP status and accessing time etc. Based on this, it can determine the next action. If satisfying one of filter items, web proxy generates the corresponding web content, returns it to the user and deletes the socket connection[13][14][15]. The following figure gives the whole processing:

In figure 3, if the request or reply passes the filter strategy, it can be forwarded to the next stage; otherwise the system will generate the wrong page content to the user which notices that some bad things have happened. Proxy also possesses memory space like program instruction. But it only records some status information which can be referenced by the next handling. In our design, we partition the whole processing from the incoming request or reply to the sending-out package into several independent parts. In order to reach this, some intermediate information must be written into the memory or file to be accessed by the next request or reply.

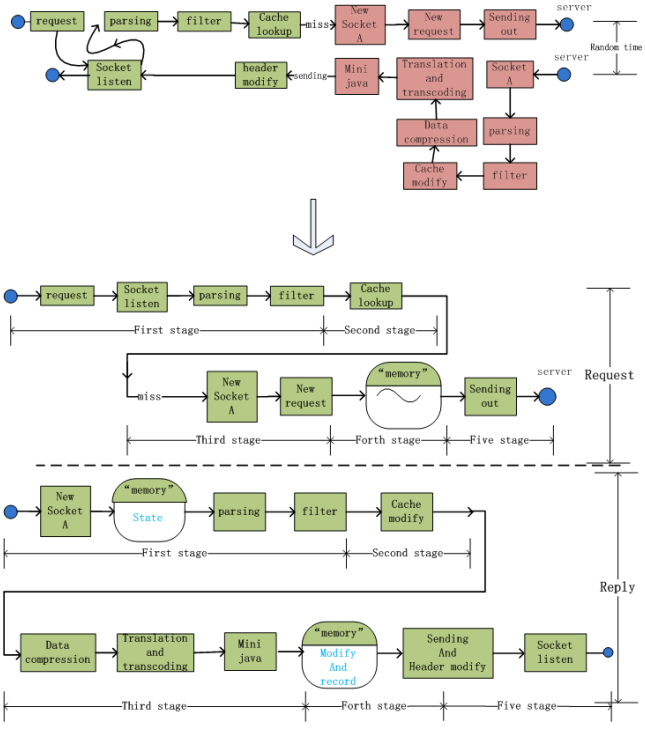

Figure 4. comparisons between traditional method and CP.

In figure 4, we compare the traditional processing with our CP method. We take the request or reply as one instruction which keeps considerable independency. Our execution includes five steps: (1) accepting request or reply, parsing the data package and filtering some things; (2) cache

modifying; (3) creating new socket or processing the data compression, etc; (4) memories recording; (5) sending out data. Comparing with the traditional method, we have moved the information record function into the memory modify stage. Therefore, the function for each subtask keeps distinct and is more prone to be partitioned into parallel execution units. Figure 5 describes the memory stage when request instruction and reply instruction execute, respectively.

When one instruction has been handled, some intermediate states must be written to the memory which can be referenced by the next request or reply. Take an example, when proxy accepts one reply through its listening socket, after series of checking, it misses the cache, therefore, it creates another child process, creating one socket through which it keeps connections with the server. The socket can not be deleted until completing the data transferring from the server.

Time-complexity and space-complexity are two factors to judge the algorithm efficiency. Due to our analyzing based on multi-core system whose memory is commonly more than 8GB.This figure is commonly appropriate for most application including web proxy, web server and browser. We will give the concrete time-bound analysis for subtasks in following subsections.

After this, we can focus on the subtask decomposition which aims to keep them equal execution. According to the compute-bound feature of the subtask, we produce a certain number of processes in one subtask which can be executed simultaneously. Rather than fully analyzing every task, we can focus on the major tasks which can affect the whole system performance to a great extent.

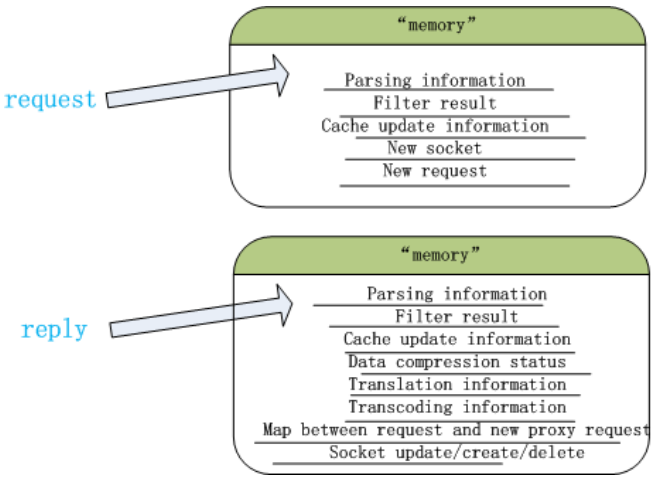

Figure 5. Memory operations for request and reply instruction.

From above description, we limit each subtask to a single function which may be executed simultaneously. Next, we will discuss some details from the whole task partition and each independent subtask partition. Here we assume the minimal execution cost is 1 for all the subtasks.

IV. Experiment

Based on section III, we will go on series of experiments to illustrate CP method and build each subtask according to its characteristic. The past research shows that the better performance improvement can be reached through building more parallel tasks than more threads. According to this, on multi-core system, we focus on the parallel task building rather than thread building.

For each subtask, we will build one pipeline[16]including several stages with different functions and each stage possesses certain execution time. As far as the compute-bound pipeline is concerned, the pipeline acceleration is limited by the number of stages and the longest execution time among all the stages. Some other details need be careful too during building pipeline and parallel tasks. For an example, when a task scale is very small, it is not worth building parallel tasks. The difficulty is how to find the correct boundary point from which we can begin to build the parallel tasks. Furthermore, when many subtasks including different compute-bound stages concurrently exist on the multi-core system, the tough problem is to adopt the suitable partition which can make the performance reach the maximum. We design different types of experiments to present these problems and propose their corresponding suggestions. Here again, we partition tasks into two parts: the parallel and not. For the latter, if it needs long time to execute, we can consider other schemes such as putting this on the powerful computing platform.

In order to acquire the satisfied speedup, we need analyze each subtask in detail. Some subtasks such as socketbuild or writing back frequently access the hardware device through corresponding drivers. From our experiments, parallel execution for these instructions can not improve the speedup due to some latency on these low speed devices.

The traditional Moore's law can not accurately reflect the speedup of the multi-core application program due to many complex factors including thread overhead, context switching[17][18]. One of proposed speedup function by Erlin Yao[19], is:

$$\text{Speedup}_{\text{symmetric}}(f,n,r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{fr}{perf(f)n}} \quad (1)$$

In this formula, the word symmetric points out that the multi-core processor is symmetric. The word f is one fraction of parallel execution time without any scheduling overhead. n is the number of processor. They assume that architects can expand the resources of r base core equivalents (BCE) to create a powerful core with sequential performance $perf(r)(1 < perf(f) < r)$. According to this cost model, they give three types of architecture of multi-core chips: symmetric, asymmetric and dynamic. In our experiment, in order to simplify the problem complexity, we only consider the symmetric platform and give the symmetric formula, correspondingly. Our experiment platform is 8-core processor. Some parameters are below:

Operating system: Linux el5xen
CPU: SMP Intel(R) Xeon(R) 8 CPU E5310 @ 1.60GHz
Cache size on each core: 4096KB
Memory size: 8GB

A. *String comparison cost under certain number of pipeline.*

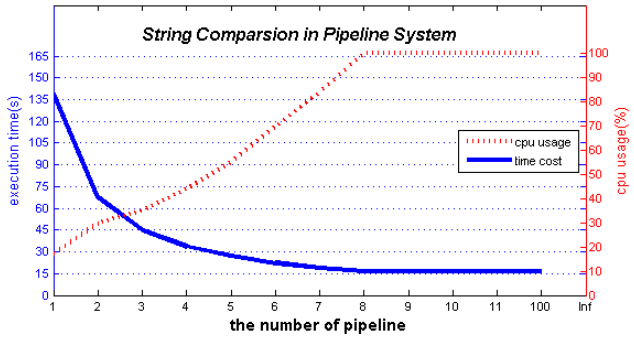

Figure 6. Comparison between execution time and CPU usage.

CPU usage is one important index for testing soft developing[20][21][22][23]. In figure 6, we compare string matching time and CPU usage when the number of pipeline varies from 1 to 11,100. From the CPU usage curve, when the number of pipeline reaches more than 8, the CPU usage keeps 100% and the speedup stays for 800%. When the number of pipeline is 1, the multi-core system keeps low CPU usage due to few subtasks existing in the scheduling queue. With the improvement of pipeline number, CPU usage boosts, correspondingly. When the pipeline number is super than 8, the speedup keeps level. The main reason is that no matter when the task scheduler acquires one task from the queue, the fast task production can always satisfy the request for task consuming.

This experiment result also reflects that it does not gain for multi-core system if we constantly produce short and vast tasks.

B. *When we allot different workload for each subtask, the execution time varies, correspondingly.*

In figure 7, we allot each subtask certain workload according to our proposed scheme in figure 7. This workload does not involve any system call. In this case, we can better view the relation between execution efficiency and subtask workload. The variable value for x axis decides each subtask's workload. In this figure, variable i is varied from 30000 to 3000000 which decides the whole computing cost. In order to test our series of experiments, we have developed several scripts written by Perl to produce our required data. With the increment of i, the curve gradient also boosts, correspondingly. This reflects that when the workload linearly improves the execution time linearly increments, correspondingly. Furthermore, the improvement for

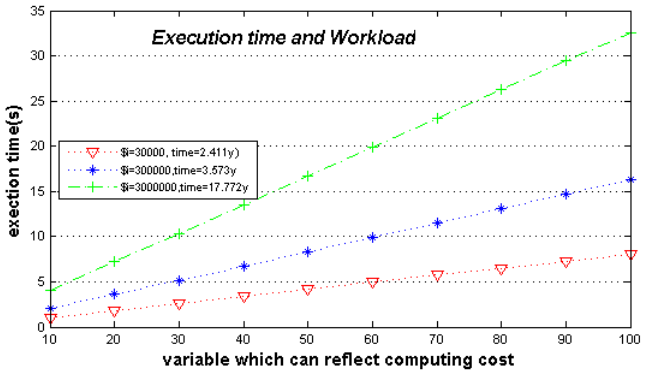

Figure 7. execution time and workload.

the gradient value means that the more the whole execution cost is, the less the speedup of the whole pipeline is.

This result comes from the unbalance of the whole pipeline. When the longest subtask spends long time computing, other parts of subtasks have finished their work in advance. In this case, the multi-core system can keep idle for short interval which can reduce the parallel execution efficiency.

In spite of the low efficiency for unbalance pipeline, some designs must adopt this scheme which can keep each stage executing concrete task. In this case, from our experiment and analysis, several ways can be selectable in order to keep system high efficient.

1) Producing more subtasks which can be scheduled when some time-consuming tasks terminate.
2) Trying to reduce the gap between the longest subtask and shortest subtask.

C. *i varies from 100000 to 700000, and the number of pipeline varies from 1 to 10.*

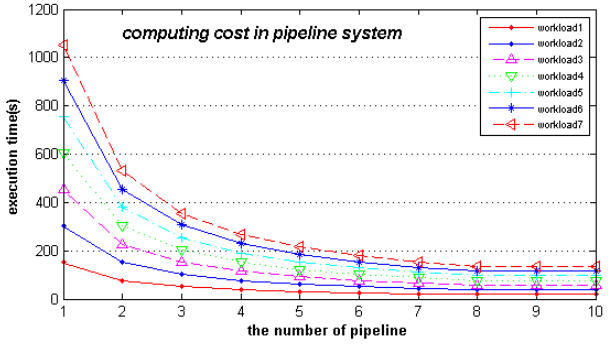

Figure 8. computing cost in pipeline system.

Figure 8 mainly tests the point where the speedup can reach the maximum. This figure mainly focuses on the point where the whole irregular pipeline can reach the maximal

speedup when we add up to the computing cost step by step. From this figure, when the number of pipeline is more than 8, the speedup begins to keep level. This reflects that in 8-core system, 8-way parallel execution can reach fairly speedup. When we build more than 8 parallel threads, due to the limitation on the number of the total processors, many threads are waiting in the scheduling queue. Furthermore, if vast tasks exist in the waiting queue, no matter which task scheduling algorithm is applied in this system, it need traverse the whole queue to pick out the appropriate task to be scheduled in ready time. This will result in slightly overhead due to the long task queue. So the velocity for producing the tasks should be equal or slightly greater than the velocity for task termination.

In figure 8,we can also see that when the number of pipeline improves from 1 to 10, the execution curve becomes more and more flat. This reflects that when the total number of produced tasks is small, the real processor can execute them at a high throughput. Especially when the number changes from 1 to 2, the execution time nearly descends to the half time. In this case, only about 4 real processors can finish all the produced tasks. With the increase of the produced tasks, the real processor must execute more tasks simultaneously and this can also result in the decrease of the speedup.

Generally speaking, four parameters take the important role in the whole execution: the task producing velocity, task execution cost, the processor basic frequency and the context switching. When the task execution is compute-bound, the overhead resulted from the context switching can be neglected. When the basic frequency is high, in the limited time, more tasks can be finished and the high speedup is acquired. When the execution cost is high, the tasks in the ready queue often need long time to be scheduled. During this interval, if other tasks have been frequently scheduled many times, this means that this time-consuming task should be degraded into several parallel slices in order to improve the speedup of the whole application.

D. concurrent operation and mixture operation

For web proxy, Apache server or other application software, some operations involve concurrent actions including same file writing, information debugging, data accessing. Generally speaking, two ways can reach this: first, merging all the concurrent operations into one task; second, scattering them to different tasks which can reduce the concurrent accessing probability.

In figure 9, we can see that when the pipeline number increases, the concurrent accessing efficiency becomes low. The main reason is that more threads compete for the same data resources. The thread which successfully enters the data resource can block the other threads. This can also result in the less CPU usage than the pure computing without any concurrent operation. When the pure computing mixes

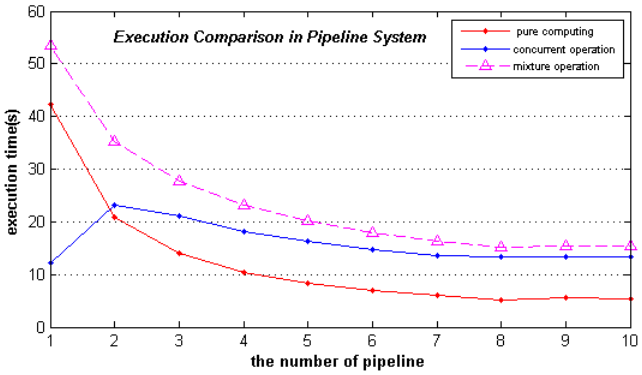

Figure 9. comparison among pure computing, concurrent accessing and mixture operation.

with the concurrent operation, the speedup becomes not as high as the pure computing. From the real application, the mixture operation is very common including Apache, Squid. Even if the concurrent operation can reduce the pipeline speedup, we can select some alternative methods to improve concurrent efficiency. First, one task includes more pure computing and less concurrent operations. Second, pipeline should contain fewer subtasks. These ways can keep the resource competition the lowest probability.

E. Summary

Series of experiments have processed to illustrate our proposed CP pipeline. From our analysis, CP can show high efficient no matter how the total number of request changes. We accurately test the pipeline characteristic based on different parameters including pipeline stage, irregular/regular pipeline, minimal subtask cost, concurrent operation, instruction feature. We regulate our parameter value according to our experiment result.

Our design is based on web proxy which represents some types of applications including web server, browser. We believe in future web proxy technique can express more important role, especially in wireless scope. Parallel design based on multi-core platform can greatly improve the processing efficiency.

V. Future Work

We have processed current work almost for one year, more challenging is waiting in front of us including pipeline execution minimal model, switching/pipeline cost/multi-core number analysis model[24][25][26][27]. Even if the web proxy design can stand for some common applications, we still need consider some slight difference among them in order to give the perfect concurrent execution in multi-core system.

Resource competition is another important factor which can greatly affect the concurrent execution in multi-core system. Through execution migration, we can reduce this bad affect for parallel execution to some extent. However,

the better way for solving this is optimizing the multi-core concurrent algorithm. Currently, even if people have proposed different multi-core algorithm for parallel list, vector, hash, these algorithms often shows bad efficient. Through enough tests, we find that some are even worse than corresponding sequential version. In future work, we should connect the concrete application with the multi-core system, further focus on the reduction for the resource competition.

VI. Conclusion

In this paper, we deeply research the web proxy architecture and based on this, we partition the whole application into several different tasks according to their feature. Each task includes several subtasks from 2 to 6. These subtasks constitute one complete pipeline. In multi-core system, the scheduling strategy can concurrently execute several tasks which come from different pipeline. According to pipeline function feature, each subtask includes certain operations including resource competition, system call, and instruction characteristic. In each case, we process the corresponding experiment to show its performance. Meanwhile, we also compare CP with the other pipeline which possesses different feature. The result shows that our CP show good performance no matter how the pipeline changes its relative parameter.

VII. Acknowledgement

This Work is supported by Natural Science Foundation of China (60803121, 60773145, 60911130371, 90812001, 60963005), National High-Tech R&D (863) Program of China (2009AA01A130, 2006AA01A101, 2006AA01A108, 2006AA01A111, 2006AA01A117) and MOE-Intel Foundation.

References

[1] http://openmp.org/wp/

[2] http://www.threadingbuildingblocks.org/

[3] http://www.erlang.org/doc/

[4] Dashtbozorgi, M. and M.A. Azgomi. A scalable multi-core aware software architecture for high-performance network monitoring. in Proceedings of the 2nd international conference on Security of information and networks. 2009. North Cyprus, Turkey : ACM.

[5] Guo, D., et al. A scalable multithreaded L7-filter design for multi-core servers. in Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems. 2008. San Jose, California : ACM.

[6] Argyraki, K., et al. Can software routers scale?. in Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow. 2008. Seattle, WA, USA: ACM.

[7] Marin, M., R. Paredes and C. Bonacic. High-performance priority queues for parallel crawlers. in Proceeding of the 10th ACM workshop on Web information and data management. 2008. Napa Valley, California, USA : ACM.

[8] Veal, B. and A. Foong. Performance scalability of a multi-core web server. in Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems. 2007. Orlando, Florida, USA : ACM.Fastforward for efficient pipeline parallelism.

[9] Gotsman, A., et al. Proving that non-blocking algorithms don't block. in Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2009. Savannah, GA, USA : ACM.Adaptive work stealing with parallelism feedback.

[10] William N. Scherer, I., D. Lea and M.L. Scott, Scalable synchronous queues. Commun. ACM , 2009. 52 (5): p. 100-111.

[11] Souders, S., High Performance Web Sites. Queue , 2008. 6 (6): p. 30-37.

[12] Vicen, et al. Improving Web Server Performance Through Main Memory Compression. in Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems. 2008: IEEE Computer Society.

[13] Taylor, D.E. and J.S. Turner, ClassBench: a packet classification benchmark. IEEE/ACM Trans. Netw. , 2007. 15 (3): p. 499-511.

[14] Cohen, E., B. Krishnamurthy and J. Rexford. Improving end-to-end performance of the Web using server volumes and proxy filters. in Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication. 1998. Vancouver, British Columbia, Canada: ACM.

[15] Yates, A., S. Schoenmackers and O. Etzioni. Detecting parser errors using web-based semantic filters. in Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing. 2006. Sydney, Australia : Association for Computational Linguistics.

[16] Garcia, P. and H.F. Korth. Pipelined hash-join on multithreaded architectures. in Proceedings of the 3rd international workshop on Data management on new hardware. 2007. Beijing, China : ACM.

[17] DeBenedictis, E.P. Will Moore's Law Be Sufficient. in Proceedings of the 2004 ACM/IEEE conference on Supercomputing. 2004: IEEE Computer Society.

[18] Cong, J., et al. Moore's Law: another casualty of the financial meltdown?. in Proceedings of the 46th Annual Design Automation Conference. 2009. San Francisco, California: ACM.

[19] Yao, E., et al. , Extending Amdahl's law in the multicore era. SIGMETRICS Perform. Eval. Rev. , 2009. 37 (2): p. 24-26.

[20] Park, I., B. Falsafi and T.N. Vijaykumar. Implicitly-multithreaded processors. in Proceedings of the 30th annual international symposium on Computer architecture. 2003. San Diego, California : ACM.

[21] Tabata, T., et al. Controlling CPU Usage for Processes with Execution Resource for Mitigating CPU DoS Attack. in Proceedings of the 2007 International Conference on Multimedia and Ubiquitous Engineering. 2007: IEEE Computer Society.

[22] Jaros and A. Lipowski, Minimizing CPU usage in soft shadow volumes algorithm. MG;V, 2006. 15 (3): p. 493-503.

[23] Dumitrescu, C. and I. Foster. Usage Policy-Based CPU Sharing in Virtual Organizations. in Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing. 2004: IEEE Computer Society.

[24] Bunescu, R.C. Learning with probabilistic features for improved pipeline models. in Proceedings of the Conference on Empirical Methods in Natural Language Processing. 2008. Honolulu, Hawaii : Association for Computational Linguistics.

[25] Liao, W., et al. , Performance Evaluation of a Parallel Pipeline Computational Model for Space-Time Adaptive Processing. J. Supercomput. , 5. 31 (2): p. 137-160.

[26] Kuntraruk, J., W.M. Pottenger and A.M. Ross, Application Resource Requirement Estimation in a Parallel-Pipeline Model of Execution. IEEE Trans. Parallel Distrib. Syst.

[27] Roth, D. and K. Small. Active learning for pipeline models. in Proceedings of the 23rd national conference on Artificial intelligence - Volume 2. 2008. Chicago, Illinois : AAAI Press.