Computers and Mathematics with Applications 63 (2012) 536–543

Contents lists available at SciVerse ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa

Task optimization based on CPU pipeline technique in a 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

ARTICLE INFO

Keywords:
Pipeline
Multicore system
Task optimization

ABSTRACT

Currently, the multicore system is prevalent in desktops, laptops or servers. The web proxy can save network traffic overhead and shorten communication cost. Especially with the fast development of wireless Internet accessing, the web proxy will take a more important role in the future. To obtain the fast response and high hit rate from the proxy, we study the processing of web proxy and deeply exploit parallel features which exist in kinds of proxy work flow. We propose the CP technique to build parallel tasks in a proxy system. The result shows that our scheme can efficiently improve the data throughput and fully utilize the computing resources provided by the multicore system.

© 2011 Elsevier Ltd. All rights reserved.

1. Introduction

The multi-core concept was proposed in the early 90's. Even at that time, the single-core processor still occupies the most of that business. The design for Linux system adopts a time-sharing mechanism to run many applications concurrently. Each application is allocated one given time-slice by the process management which allocates one process into a running queue or an idle queue. The time slice is calculated on the level of milliseconds and the users can often smoothly run several applications oblivious to any delay resulting from the short stop of an application. The Linux timer is triggered at intervals of several milliseconds which can result in the execution of a series of services which include process management. The parameter for the time slice takes a very important role in the whole concurrent design. If this value is set too long to switch to another application, users often tend to lose their patience.

Therefore, parallel execution must take full account of many factors which can affect the whole rational design. From the early 90's, due to the constant development of hardware manufacturing processes such as wafer incision, people have improved their computing capacity and processor frequency to regulate time slices on a wide scope. Currently, the frequency for the single-processor has nearly reached its limitation. In this case, the multi-core system has become popular, avoiding that limitation and moving towards a parallel direction. In the multi-core system, the process management and process queue management at best guarantees the fair balance load among different processors. The 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 occurring.

The system performance shows some differences when we make the process or thread 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 the above, we give a sufficient description about low layer unit supporting parallel task execution, including hardware such as a multi-core architecture and software such as a process scheduler. Our parallel task developing is based

* Corresponding author. Tel.: +86 106 279 4950.
E-mail addresses: bo-wang06@mails.tsinghua.edu.cn, gawain102000@163.com (B. Wang), wuyw@tsinghua.edu.cn (Y. Wu), zwm-dcs@tsinghua.edu.cn (W. Zheng).

0898-1221/$ – see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.camwa.2011.09.024

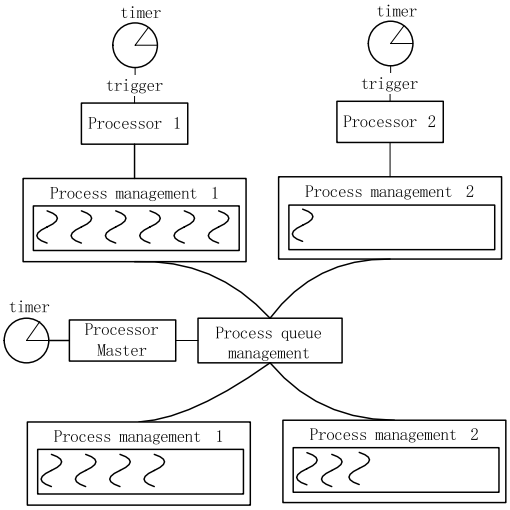

Fig. 1. Linux scheduling in a multi-core system.

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 that the client really wants.

Why do we put forward one concrete application scenario with a web proxy server? First, parallel computing has no general solution for each application. We can not smoothly write parallel programs like sequential programs because we must reasonably partition one big task into several parts which can keep the multi-core load balanced and running (Fig. 1). 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 a two-level loop statement or one subtask, it may only contribute little to the whole application. Second, the web proxy server can stand for types of applications including Apache, Firefox or some network routing applications. These applications play an improving role in our social progress. Third, our research focuses on how to analyze the relation among different modules, including logging, accessing and filtering mechanisms, 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 many applications to guarantee 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 and header information parsing. In future, due to the fact that wireless surfing on mobile phones, 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 a powerful computing platform like a web service provider or web proxy.

Many methods have been presented to support 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 it prevalent in multi-core systems.

From the above description, we introduce CPU pipeline, give some parallel executions existing in a web proxy and compare some similar places between pipeline techniques 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 effective guidance for some similar applications including some web services.
3. Each request from a client or reply from a server can be considered as one pipeline, and some intermediate states can be written to "memory" which can keep next session valid.
4. Compared with other methods, our proposed method (CP) shows favorable scalability.

2. Related work

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

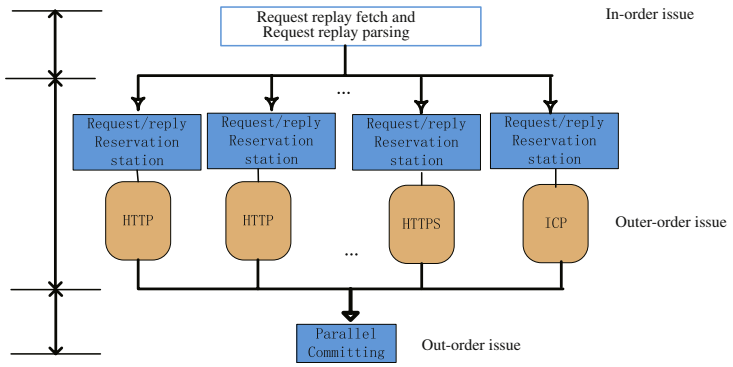

Fig. 2. CP architecture.

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 a great improvement has been made [4]. Danhua Guo proposes a highly scalable parallelized L7-filter system architecture with affinity-based scheduling on a multi-core system. He treated the whole processing as the following: accepting the incoming packages, preprocessing, scheduling, matching and transmitting [5]. Even if more parallel schemes are given, it lacks a full analysis for the task's 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 interconnect commodity server platforms to build software routers that are both incrementally scalable and fully programmable [6]. Furthermore, when we take into account the order of the incoming data stream for some network applications, how to handle them one by one in a 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 a network crawler can operate on the same URL which wastes more computing and storage resources. His team solves this through the synchronization management for the URL's queues. They propose efficient and scalable strategies which consider intra-node multi-core multithreading on an inter-node distributed memory environment, including efficient use of the secondary environment [7]. Based on this, more multi-core synchronization algorithms have been put forward over the years in support of parallel applications. Some research work focuses on the system bottleneck and provides all the ways to overcome this and make this part execute in a parallel manner [8–12].

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

1. We assume that the low layer such as the operating system scheduler, TCP/IP stack or communication architecture (PCIE) or transferring pattern (EDMA) works well. Some studies modify these to acquire better performance.
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 and transfer-bound. We take full account of the possible details and decompose these into many subtasks which support our proposed CP technique.

3. CP technique

As far as the 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 Fig. 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 the user, reply from the 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, the web proxy generates the corresponding web content, returns it to the user and deletes the socket connection [13–15]. The following figure gives the whole processing:

In Fig. 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. The proxy also possesses memory space like a program instruction. However 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.

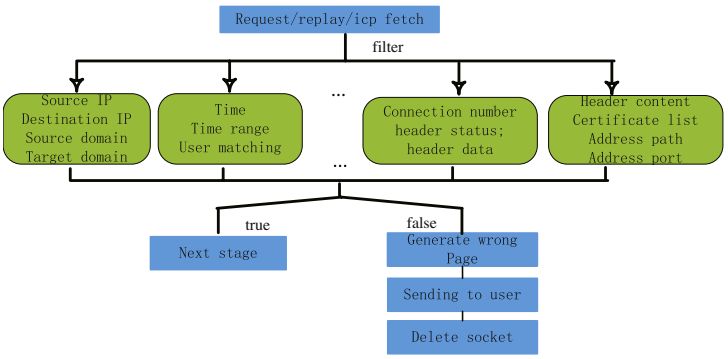

Fig. 3. Request or reply fetches processing.

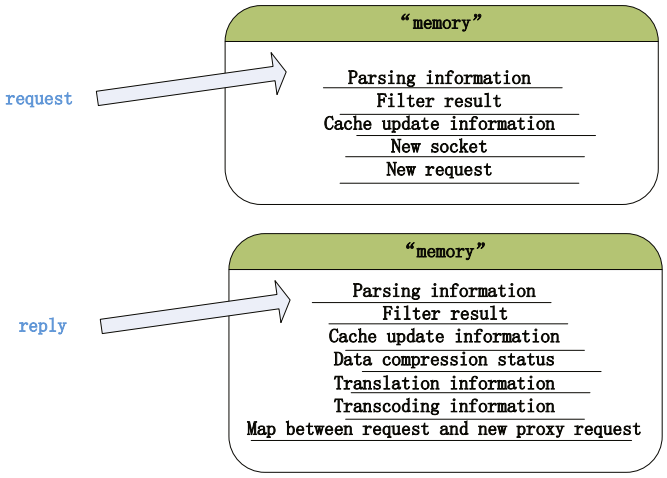

Fig. 4. Memory operations for request and reply instruction.

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) memory recording; (5) sending out data. Compared with the traditional method, we have moved the information record function into the memory modify stage. Therefore, the function for each subtask stays distinct and is more prone to be partitioned into parallel execution units. Fig. 4 describes the memory stage when the 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. For example, when the proxy accepts one reply through its listening socket, after a series of checks 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 transfer from the server.

Time-complexity and space-complexity are two factors to judge the algorithm's efficiency. Due to our analyzing based on a multi-core system whose memory is commonly more than 8 GB, this figure is commonly appropriate for most applications including web proxies, web servers and browsers. We will give the concrete time-bound analysis for subtasks in the following subsections.

After this, we can focus on the subtask decomposition which aims to keep them equal in 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's performance to a great extent.

We partition the proxy work flow into several different subtasks. By optimizing these subtasks, we can greatly improve the speedup over the multicore system. Before optimization, we analyze the dependency, execution cost and characteristics of each subtask in detail. During optimization, we use the parallel and pipeline methods to handle different subtasks.

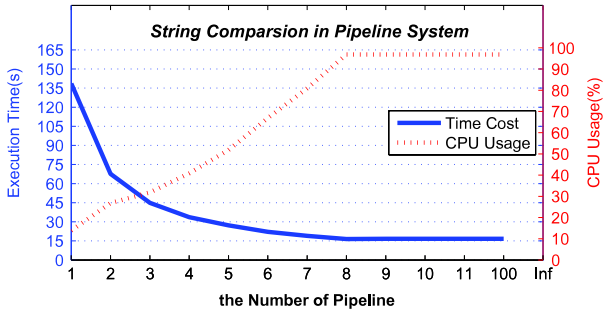

Fig. 5. Comparison between execution time and CPU usage.

4. Experiment

Based on Section 3, we will go on a series of experiments to illustrate the CP method and build each subtask according to its characteristic. The past research shows that a better performance improvement can be reached through building more parallel tasks than more threads. According to this, in the 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 a 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 taken care of too during building pipeline and parallel tasks. For 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 in the multi-core system, a tough problem is to adopt the suitable partition which can make the performance maximal. 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 a long time to execute, we can consider other schemes such as putting this on the powerful computing platform.

In order to acquire the required speedup, we need to 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 the proposed speedup functions by Yao et al. [19], is:

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

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 processors. 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's complexity, we only consider the symmetric platform and give the symmetric formula correspondingly. Our experiment platform is an 8-core processor. Some parameters are below:

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

4.1. String comparison cost under a certain number of pipelines.

CPU usage is one important index for testing soft developing [20–23]. In Fig. 5, we compare string matching time and CPU usage when the number of pipelines varies from 1 to 11,100. From the CPU usage curve, when the number of pipelines reaches more than 8, the CPU usage stays at 100% and the speedup stays at 800%. When the number of pipelines is 1, the multi-core system keeps a low CPU usage due to the few subtasks existing in the scheduling queue. With the improvement of pipeline number, CPU usage boosts correspondingly. When the pipeline number is more than 8, the speedup stays the same. 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 consumption.

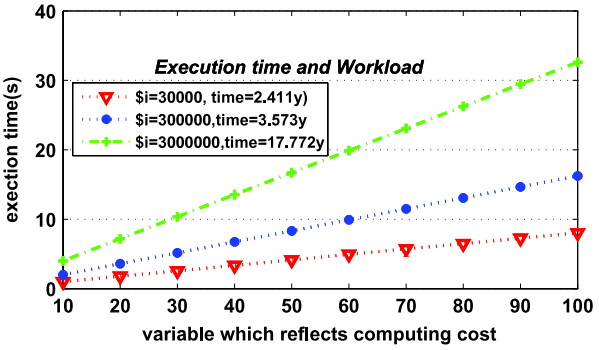

Fig. 6. Execution time and workload.

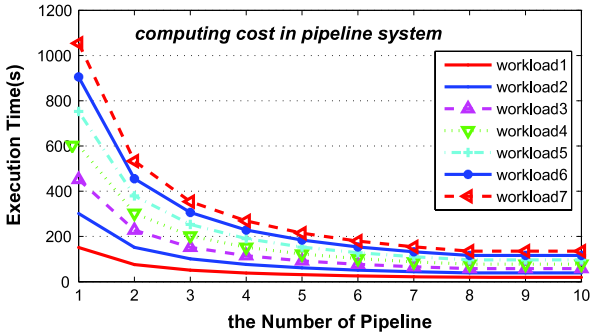

Fig. 7. Computing cost in pipeline system.

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

4.2. When we allocate different workloads for each subtask, the execution time varies correspondingly

In Fig. 6, we allocate each subtask certain workload according to our proposed scheme in Fig. 7. This workload does not involve any system call. In this case, we can better view the relationship between execution efficiency and subtask workload. The variable value for the *x* axis decides each subtask's workload. In this figure, variable *i* is varied from 30,000 to 3000,000 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 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 stay idle for short intervals which can reduce the parallel execution efficiency.

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

4.3. i varies from 100,000 to 700,000, and the number of pipelines vary from 1 to 10

Fig. 7 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 the computing cost step by step. From this figure, when the number of pipelines is more than 8, the speedup begins to stay level. This reflects that in the 8-core system, 8-way parallel execution can reach a fair 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 needs to traverse the whole queue to pick out the appropriate task to be scheduled in time. This will result in a slight 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.

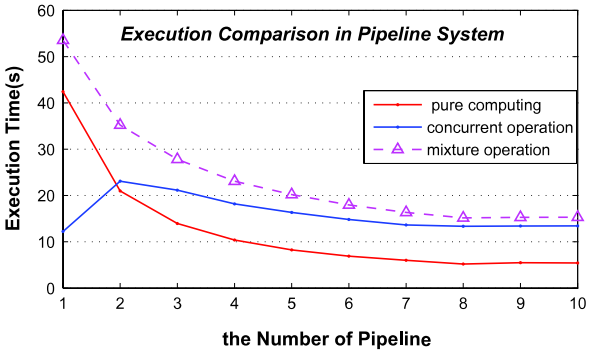

Fig. 8. Comparison among pure computing, concurrent accessing and mixture operation.

In Fig. 7, we can also see that when the number of pipelines 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. In particular, when the number changes from 1 to 2, the execution time nearly drops to half. 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 resulting 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 a 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.

4.4. 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 methods 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 Fig. 8, 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 less CPU usage than the pure computing without any concurrent operation. When the pure computing mixes with the concurrent operation, the speedup is not as high as the pure computing. From the real application, the mixture operation is very common including Apache and Squid. Even if the concurrent operations can reduce the pipeline speedup, we can select some alternative methods to improve concurrent efficiency. First, one task includes more pure computing and fewer concurrent operations. Second, the pipeline should contain fewer subtasks. These methods can keep the resource competition the lowest probability.

4.5. Summary

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

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

5. Future work

We have processed the current work for almost a year and more challenges await us, including the pipeline execution minimal model, switching/pipeline cost/multi-core number analysis model [24–28]. Even if the web proxy design can stand for some common applications, we still need to consider some slight differences among them in order to give the perfect concurrent execution in multi-core systems.

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

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

6. Conclusion

In this paper we extensively research the web proxy architecture, and based on this we partition the whole application into several different tasks according to their features. Each task includes several subtasks from 2 to 6. These subtasks constitute one complete pipeline. In the multi-core system, the scheduling strategy can concurrently execute several tasks which come from different pipelines. According to the 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 features. The result shows that our CP shows good performance no matter how the pipeline changes its relative parameters.

Acknowledgments

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] Y. He, C.H.Q. Ding, Proceedings of the OpenMP Applications and Tools 2003 International Conference on OpenMP Shared Memory Parallel Programming, WOMPAT'03, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 195–210.
[2] T. Willhalm, N. Popovici, Proceedings of the 1st International Workshop on Multicore Software Engineering, IWMSE'08, ACM, New York, NY, USA, 2008, pp. 3–4.
[3] K. Claessen, H. Svensson, Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang, ERLANG'05, ACM, New York, NY, USA, 2005, pp. 78–87.
[4] M. Dashtbozorgi, M. Abdollahi Azgomi, Proceedings of the 2nd International Conference on Security of Information and Networks, SIN'09, ACM, New York, NY, USA, 2009, pp. 117–122.
[5] D. Guo, G. Liao, L.N. Bhuyan, B. Liu, Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS'09, ACM, New York, NY, USA, 2009, pp. 50–59.
[6] K. Argyraki, S. Baset, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, E. Kohler, M. Manesh, S. Nedevschi, S. Ratnasamy, Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow, PRESTO'08, ACM, New York, NY, USA, 2008, pp. 21–26.
[7] M. Marin, R. Paredes, C. Bonacic, Proceeding of the 10th ACM Workshop on Web Information and Data Management, WIDM'08, ACM, New York, NY, USA, 2008, pp. 47–54.
[8] B. Veal, A. Foong, Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS'07, ACM, New York, NY, USA, 2007, pp. 57–66.
[9] A. Gotsman, B. Cook, M. Parkinson, V. Vafeiadis, SIGPLAN Not. 44 (2009) 16–28.
[10] W.N. Scherer, III, D. Lea, M.L. Scott, Commun. ACM 52 (2009) 100–111.
[11] A. Iyengar, D. Rosu, Sci. Program. 10 (2002) 75–89.
[12] V. Beltran, J. Torres, E. Ayguadé, Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems, IEEE Computer Society, Washington, DC, USA, 2008, pp. 303–310.
[13] D.E. Taylor, ACM Comput. Surv. 37 (2005) 238–275.
[14] E. Cohen, H. Kaplan, Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM'01, ACM, New York, NY, USA, 2001, pp. 41–53.
[15] A. Yates, S. Schoenmackers, O. Etzioni, Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing, EMNLP'06, Association for Computational Linguistics, Stroudsburg, PA, USA, 2006, pp. 27–34.
[16] P. Garcia, H.F. Korth, Proceedings of the 3rd International Workshop on Data Management on New Hardware, DaMoN'07, ACM, New York, NY, USA, 2007, pp. 1:1–1:8.
[17] E.P. DeBenedictis, Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, SC'04, IEEE Computer Society, Washington, DC, USA, 2004, p. 45.
[18] J. Cong, N.S. Nagaraj, R. Puri, W. Joyner, J. Burns, M. Gavrielov, R. Radojcic, P. Rickert, H. Stork, Proceedings of the 46th Annual Design Automation Conference, DAC'09, ACM, New York, NY, USA, 2009, pp. 202–203.
[19] E. Yao, Y. Bao, G. Tan, M. Chen, SIGMETRICS Perform. Eval. Rev. 37 (2009) 24–26.
[20] I. Park, Implicitly-multithreaded processors, Ph.D. Thesis, West Lafayette, IN, USA, 2003. AAI3113856.
[21] T. Tabata, S. Hakomori, K. Yokoyama, H. Taniguchi, Proceedings of the 2007 International Conference on Multimedia and Ubiquitous Engineering, MUE'07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 141–152.
[22] J. Lipowski, MG&V 15 (2006) 493–503.
[23] C. Dumitrescu, I. Foster, Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, GRID'04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 53–60.
[24] R.C. Bunescu, Proceedings of the Conference on Empirical Methods in Natural Language Processing, EMNLP'08, Association for Computational Linguistics, Stroudsburg, PA, USA, 2008, pp. 670–679.
[25] W.-K. Liao, A. Choudhary, D. Weiner, P. Varshney, J. Supercomput. 31 (2005) 137–160.
[26] J. Kuntraruk, W.M. Pottenger, A.M. Ross, IEEE Trans. Parallel Distrib. Syst. 16 (2005) 1154–1165.
[27] D. Roth, K. Small, Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2, AAAI Press, 2008, pp. 683–688.
[28] Y. Tian, B. Liu, Z. He, Front. Comput. Sci China 4 (2010) 500–515.