Operating System Support for Pipeline Parallelism on Multicore Architectures

John Giacomoni and Manish Vachharajani
University of Colorado at Boulder

Abstract.

The industry wide shift to multicore architectures presents the software development community with an opportunity to revisit fundamental programming models and resource management strategies. Continuing to track the historical performance gains enabled by Moore's law with multicores may be difficult as many applications are fundamentally sequential and not amenable to data- or task-parallel organizations. Fortunately, an important subset of these applications stream data (e.g., video processing, network frame processing, and scientific computing) and can be decomposed into pipeline-parallel structures, delivering increases proportional to the pipeline depth (2x, 3x, etc.).

To realize the potential of pipeline-parallel software organizations requires reexamining some basic historical assumptions in OS design, including the purpose of time-sharing and the nature of applications. The key architectural change is that multicore architectures make it possible to fully dedicate resources as needed without compromising existing OS services. This paper describes the minimal OS extensions necessary to support efficient pipeline-parallel applications on multicore systems with supporting evidence from the domain of network frame processing.

1 Introduction

The industry wide shift to multicore[1] architectures presents the software development community with a rare opportunity to revisit fundamental programming models and resources management strategies. Multicore systems are now present in every class of system including embedded systems, workstations, and laptops. The question that must be addressed by the systems community is how to utilize the additional computational resources and what minimum OS changes are needed to maximize their potential. The obvious use is improve overall system throughput by increasing task and data parallelism. However, there exists an important set of applications that are sequential and thus cannot utilize task or data parallelism to achieve performance improvements.

For sequential and other applications, an appealing option is to utilize the resources for important novel programming tasks such as shadow profiling [16] and transient fault tolerance [21]. Shadow Profiling works by running a snapshot of a process on a separate core to perform deep instrumentation while the fault tolerance work runs process clones in parallel to detect and correct transient soft errors without additional hardware support. By using multiple cores, both systems extend the system's functionality without impacting performance.

The assumption in the above scenarios is that multicore augment the historical per-core performance increases. The reality is that limitations arising from power consumption, design complexity, and wire delays limit our ability to increase the computational capabilities of a single core. Fortunately, Moore's law continues to hold and it is possible to continue increasing resources by doubling cores according to the historic exponential growth in transistor density. Therefore it is possible to continue receiving dividends from existing data- and task-parallel strategies. Sequential applications have traditionally relied upon ever increasing processor performance to improve their performance.

Fortunately, many sequential applications of interest such as video decoding, network frame processing, scientific computing while sequential in nature may be restructured either by hand or optimizing compiler to manifest innate pipeline parallelism. Pipelines are instantiated in software by bind-

[1] We are using the the term "multicore" to refer to systems with 4 to 100 processing cores [4].

ing pipeline stages to different threads of execution and feeding data serially through the different stages. For optimal performance these stages will be simultaneously bound to different processors.

In this work we focus on efficiently supporting those applications that are streaming in nature and can be restructured with a pipeline-parallel structure. Pipeline-parallel structures are of interest as they can deliver performance increases proportional to the pipeline depth; a basic three stage networking application (Section 2.2) can increase either its available computation time or increase its throughput by approximately *three times*. We know of no other technique, short of a high-level redesign, that can deliver equivalent increases on sequential applications.

These sequential applications exhibit a critical property that makes them particularly suited to a pipeline-parallel decomposition, data streams sequentially through a well defined code path from input to output. This sequential flow is relatively easy to analyze and decompose into a pipeline-parallel components. For example, video processing algorithms (e.g., mpeg) and the basic TCP/IP stack have very well defined boundaries that can be used to recover pipeline stages without much effort.

In situations where applications appear to be fundamentally sequential, such as the SPECint benchmarks, recovering parallelism may not be possible without detailed knowledge of the machine and a thorough code analysis. Compiler techniques such as Decoupled Software Pipelining [17–19] may extract some pipeline-parallelism yielding on average 10% performance improvements by performing fine-grain parallelization with dedicated resources.

In our work on exploiting pipeline-parallelsim for network frame processing and scientific computing we found that widely deployed general purpose operating systems (e.g., Unix variants, Windows, and MacOS X) are not prepared to efficiently support pipeline-parallel applications. This is because these general purpose OSes are designed to optimize overall throughput in a resource constrained (i.e., oversubscribed) environment while maintaining acceptable interactive behavior. This behavior historically made sense in the era of the Computer Utility, first proposed by John McCarthy, and later with personal computers where the number of tasks to be serviced dwarfed the available computational resources.

1.1 Claims

Multicore architectures alter the landscape by providing sufficient resources to handle background tasks while dedicating resources for performance-critical tasks. This is the key observation upon which our work is based.

We suggest that when dealing with multicore architectures, where the computing power of individual cores is stagnant but where cores are plentiful, the resource management strategy of OSes be extended to permit full resource dedication instead of optimizing the resource sharing as done by SEDA [22] and Synthesis [15]. Building efficient pipelines as used by network frame processing and scientific computing requires that: (1) interstage communication be proportionally small; (2) zero-stalls enter the pipeline; (3) services are pipelineable. Additionally, we argue that (4) managing efficient pipeline applications will require a new labeling abstration that is orthogonal to the existing application abstractions. With these features it is possible to build a pipeline in software that is optimally efficient without necessitating substantial changes to the OS.

The rest of this paper is organized as follows. Section 2 reviews task, data, and pipeline parallelism as well as presenting a case study from our work on high-rate network frame processing and addresses our first claim, the need for low-cost communication. Section 3 addresses our second claim, the need for selective disabling of timesharing, based on the need for a zero-stall guarantee. Section 4 addresses our third claim, the value of pipelineable services. Section 5 addresses our fourth claim and argues that a new labeling abstraction is needed for pipeline applications that is orthogonal to existing abstactions. Section 6 concludes.

2 Background

This section reviews the three basic parallel structures (i.e., task, data and pipeline) and discusses our overarching motivating example (i.e., network frame processing) from which we recover our three claims. Included in the example is a discussion of our first claim on the value of very low-cost stage-to-stage communication (35-40 ns).

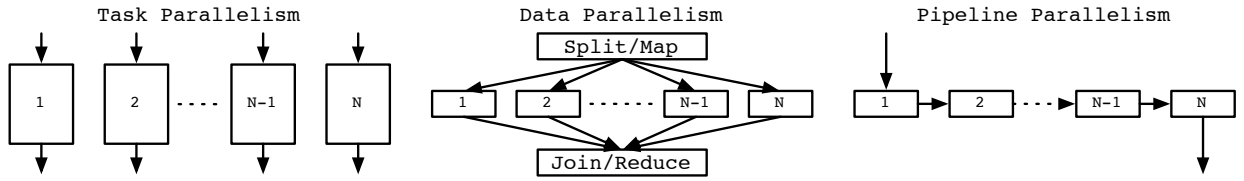


Fig. 1. Parallel Structures

2.1 Parallel Structures

Pipeline parallelism's ability to parallelize applications that are sequential due to internal data dependencies and therefore not amenable to other forms of parallelism is the key point of interest. Examples of well suited applications are network frame processing (Section 2.2), video and audio decoding, and scientific computing. Notice how each of these examples is both of interest and poorly suited to task- and data-parallel approaches as there exists a partial or total ordering in each data stream. Further, these examples span the systems space from embedded systems to workstation and supercomputer class systems. For clarity, we describe the three forms of parallelism below and depict them in Figure 1.

Task Parallelism consists of running multiple independent tasks in parallel and is limited by the availability of independent tasks.

Data Parallelism consists of simultaneously processing multiple independent data elements in a single task. This technique scales well from a few processing cores to an entire cluster [6]. The flexibility of the technique relies upon stateless processing routines (filters) implying that the data elements must be fully independent.

Pipeline Parallelism allows for parallelization of a single task when there exists a partial or total order in the data set implying the need for state and therefore preventing the use of data parallelism. Parallelism is achieved by running each stage simultaneously on sequential elements of the data flow. This form of parallelism is limited only by the sequential decomposability of the task and the length of the longest stage.

2.2 Case Study: Network Frame Processing

Network frame processing provides an interesting case study for pipeline parallelism as such systems

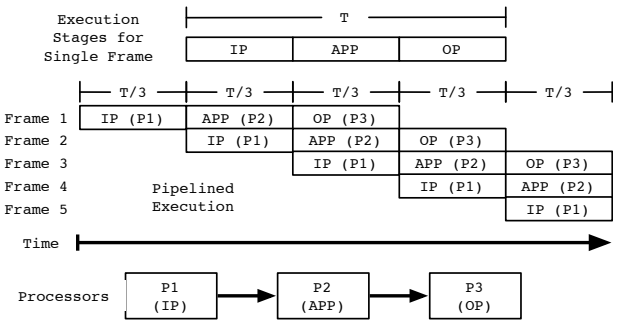


Fig. 2. Frame Shared Memory Pipeline

are both useful (intrusion detection, firewalls, and routers) and may exhibit high data rates (672 ns per datum/frame) which challenges previous implementation techniques. Using our Frame Shared Memory (FShm) architecture [9] we are able to forward at a rate of 1.428 million frames per second, the limit of our hardware, while achieving a 3x or larger increase in frame processing time without dropping any frames. The lessons learned from this domain can be generalized to any domain with ordered data and thus are candidates for OS level support.

In the past and present network frame processing system such as this one have been built with custom or specialized hardware (e.g., network processors). However, to build such a system on a general purpose system requires that the system at a minumum provides a guarantee of zero-stalls and a very low-cost communication mechanism [10]. Below we describe the scenario and recover our first claim, the need for low-cost communication. Our second claim, the zero-stall guarantee and its OS implications, will be discussed in detail in Section 3.

The network frame processing domain is challenging if one wants to handle the two degenerate cases, maximum bandwidth and maximum frame rate. Handling the maximum bandwidth requires only high bandwidth hardware as internal communi-

cation mechanisms have low overhead compared to the frame arrival rate. Handling the maximum frame rate case is more challenging as the arrival rate on modern networks stresses both the hardware (e.g, bus arbitration) and software (e.g., locking methods).

Considering gigabit Ethernet, at maximum bandwidth there are only 81,274 frames per second while at the maximum frame rate case there are 1,488,095 frames per second. This means that a new frame can arrive every 672 ns. The complication for software is that once the frame arrival is signaled, there might only be 672 ns to remove the frame from the data structures shared with the network card, processes the frame, and if the frame is being forwarded insert it into the output network interface's data structures. In networking, increasing the throughput beyond the arrival rate is meaningless; The goal is to increase the available per frame processing time.

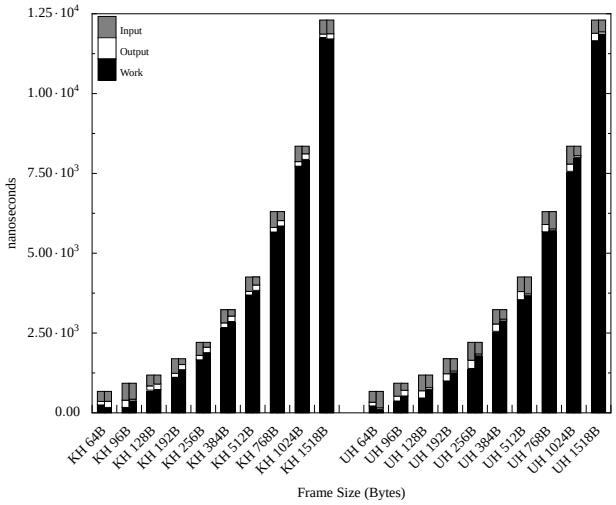


Fig. 3. Frame Shared Memory Capture Results. This figure shows the time available for the input and application stages for both pinned kernel and user-space stages.

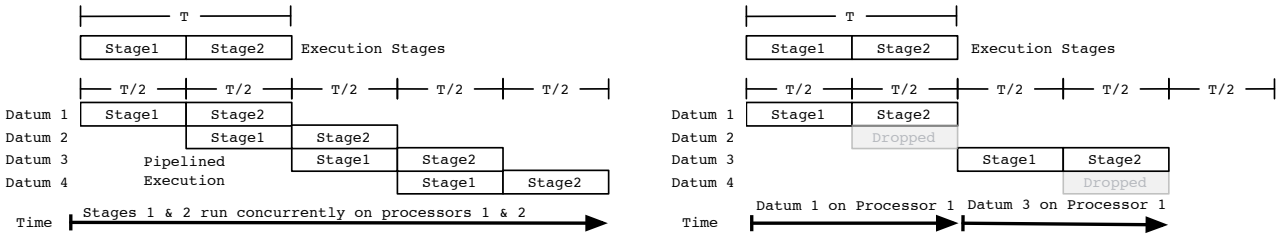
In FShm we increase the per-frame processing time by using pipelining to overlap different stages of frame processing. In a basic forwarding application we are able to decompose the task into three stages, with each being alloted the full frame computation time period and therefore tripling the total time available to the software to manipulate the frame. Figure 2 shows the structure of FShm for this basic forwarding application. The task is decomposed into input, application, and output stages. The output and input stages handle transferring the frame to and from the network interfaces. The application stage performs the actual application related frame processing without worrying about the cost of the input and output work. By simultaneously binding the three stages to different processors we can build a pipeline with full overlap potential in every time step. Notice also that the per frame processing time can be extended to 4x and beyond if the application stage can be further decomposed.

The difficulty in achieving this performance is the stage-to-stage communication cost. If the communication cost is too high there may be a negative return in decomposing an application into a pipeline. Initially, we built a standard queue with mutexes on a 2 GHz Operton 270 based system and found the per-operation (get or put) cost to be at least 600 ns. This cost was found to increase dramatically as the contention increased by shortening the period between operations. To address this, FShm provides a suitable streaming queue implementation costing only 35-40ns per operation [10]. The performance of a two stage capture application is shown in Figure 3.

3 Zero-Stall vs. Oversubscription

Maintaining a smoothly flowing pipeline, that is a pipeline where a datum is never waiting for processor time, requires the system to provide a zero-stall guarantee. Pipelines implemented in hardware are based on this guarantee and ensure it by having every stage operate in lockstep with a uniform stage length of 1 cycle. OSes in general do not make this guarantee as they have been built on the principle of timesharing resources, dating back to the Computer Utility, on oversubscribed systems 3.1. Multicore systems are different in that they may provide abundant processing resources permitting a system to use "selective timesharing" (Section 3.2) and fully dedicate resources to an application for an extended period of time. With dedicated resources it is possible to achieve the zero-stall guarantee.

More precisely, a zero-stall guarantee ensures that allocated resources are never unexpectedly made unavailable by the system for any reason (e.g., oversubscription). Failing to meet this requirement intro-



(a) Ideal (Concurrent)

(b) Oversubscribed (Serial)

Fig. 4. Pipeline Timing in Ideal and Oversubscribed Conditions

duces stalls into the pipeline which may not be easily predicted or compensated for. Consider a pipeline, show in Figure 4.a, where every stage is running efficiently (i.e., little to no built-in slack time). If a stall is introduced into this pipeline, no amount of queueing can compensate for an indefinite period of time. This is definitely a problem in the network frame processing example described in Section 2.2 or any other system operating with similar data rates. Note that in security related systems, dropping a single datum can be considered a system failure as an attack could have been contained in the datum and would be lost forever.

Meeting a zero-stall guarantee may be impossible for software as the underlying architecture may make some stalls inevitable. Therefore, the goal for an OS is to eliminate itself as the cause of stalls. This means eliminating all non-application related work on the processors used by the application. For example, we have found that even tiny OS related stalls, such as the hardclock background tick (to maintain clock accuracy) have a noticeable effect on performance.

3.1 Oversubscription

The zero-stall guarantee as described above is impossible to make on an oversubscribed general purpose system that is timesharing all of its resources. We are only making an overarching performance goal for a given pipeline application that processes an unbounded stream of data. We are not making claims about overall system throughput. Section 3.2 discusses our rationale in more detail.

Pipelining as a software architecture has been extensively studied for its performance and software engineering benefits. Examples of use include Unix pipes [20], the Staged Event Driven Architecture (SEDA) [22], and Synthesis [15].

We claim that despite the existence of efficient pipelining systems, they are not used for performance reasons due to the previously limited number of available processing resources (1-4). Prior work on optimizing the scheduling of limited resources has certainly had success and improved the efficiency of pipeline applications and the host systems [14, 23]. However, these systems are all incapable of meeting the zero-stall guarantee as they were designed to optimize in an oversubscribed environment. Further, just-in-time scheduling is also less than optimal for streaming applications as data will always be available, making the cost of such scheduling infrastructures pure overhead.

In contrast with the ideal pipeline depicted in Figure 4.a, Figure 4.b depicts an example of pipeline scheduling in oversubscribed conditions. In the ideal case, a task is broken into a number of equal length stages (for simplicity, we assume two) with each stage simultaneously bound to a processor. The throughput of such a system will be one per time step, just as in a hardware pipeline. However in the oversubscribed scenario it could happen that it is only possible to schedule the two stages in series due to other system resource demands. Throughput in this situation would be one every two time steps, or half the ideal situation. The system's inability to concurrently schedule the two stages is shown to cause a datum drop every other time step. While the depicted drop rate may be extreme, it does highlight that not every datum can be handled in situations where concurrent scheduling could handle the load. Further, the oversubscribed scenario must account for the overhead of scheduling and context swapping.

3.2 Selective Timesharing

Meeting the zero-stall guarantee for all scenarios, as we argued above, requires that the OS *not* share its resources, particularly its computational resources. In practice this means that the OS needs to be capable of partitioning its resources into two groups, those that are dedicated and those that timeshared in a traditional manner. This partitioning is different in motivation from existing work on binding execution contexts to processors. We question whether the resource sharing strategies first explored under the Computer Utility banner in the 1960s are still valid on multicore systems with limited numbers of tasks. Our conclusion is that OSes need to be altered to include user directed "selective timesharing."

Traditionally OSes have focused on different strategies to "fairly" share limited resources among many users and ensure that every task made forward progress. This strategy is the only acceptable strategy when scheduling many tasks on systems with limited resources. The notion of a Computer Utility, first formulated by John McCarthy, proposed that since computers were expensive, their resources should be distributed from a central location in a fashion similar to other utilities (e.g., electric utilities). Therefore, OSes were design to share the resources while simultaneously optimizing for both interactive response and overall throughput. This strategy was the basis of the CTSS [5] and has influenced all subsequent general purpose OSes.

The emergence of the personal computer depreciated the notion of a Computer Utility. However, its legacy continues to form the basis of our current OSes. This is no surprise as the average user, on their PC, concurrently executes multiple tasks on a limited number of processors (1-2) just as the Computer Utility did with multiple tasks from different users.

The emergence of the Internet and multicore systems are completing the conceptual shift begun by the personal computer. Internet companies such as Google, Yahoo!, and Microsoft are creating large centralized services on the Internet that, for practical purposes, are Computer Utilities. Further, multicore architectures are dramatically increasing the number of processing cores per system to the point where many systems may not have enough task or data parallelism to fully saturate the available resources.

Our proposition is to harness these resources by recognizing that many systems are focused on running a few important tasks with many support tasks. This ordering of tasks is true for many classes of systems including embedded systems and some clusters as well. Our proposed solution is to acknowledge this ordering and to dedicate resources to important tasks to optimize their throughput and let the support tasks use the remaining timeshared resources. Notice that this organizing is not only necessary for many pipeline applications, it is also of use to any task, be it sequential or data-parallel.

4 Pipelineable Services

Returning to the network frame example from Section 2.2 we found it useful to create pipelinable OS services (thus supporting our third claim), that is services which could be included directly into a pipeline application. Being able to directly incorporate services into a pipeline means that those services can be overlapped with application stages and therefore increase the amount of parallelism available in the pipeline. These services differ from traditional OS services in that they are active and not passive. An active service is simply a service that proceeds without waiting for a service request. Careful consideration suggests that these pipelineable services need not be restricted to the OS, but could be provided by other applications or by special purpose hardware (e.g., a cryptography accelerator).

In our networking example we discussed having three stages (i.e., input, application, and output) in a basic forwarding pipeline. However, we did not discuss where these stages were instantiated. Each stage can be instantiated either in a single application context or in multiple. One approach is to instantiate all three stages in the application and bind the network interfaces to the application. However, this implies that not only are the necessary processors dedicated to the application, but the network interfaces as well. This dedication of hardware devices works for certain devices (e.g., network interfaces) but is problematic for other devices that are usually shared (e.g., storage and network attached storage). Therefore, to maintain both resource sharing and efficiency in the

pipeline, some stages might need to be inside the OS while others in a user-space application.

The closest design paradigm to pipelineable OS services are message passing based microkernel systems. While microkernels could use their message passing interface to implement pipelineable services, they are often used in a synchronous manner similar to the system call interface (i.e., send message with service request and wait for completion). This mode of use makes sense in an oversubscribed environment where it impossible to overlap the producer and consumer stages due to a lack of computational resources. Click [13] and Synthesis [15] go so far as to support call through semantics, where the producer may switch to the consumer context to complete the request. In contrast, a pipeline stage is an active element that operates independently, though usually in lockstep, with the other stages in a pipeline.

In other situations it is worth considering the integration of active services from either another application or a special purpose hardware element. One example of a service provided by another application is a database system for managing log files. Consider two applications: (1) a web server feeding entries into the database (2) a reporting application stage streaming requests through the database to another stage in the reporting application. In both cases it is possible to overlap the database stages with the application stages maintaining the pipeline even though the producer and consumer are not in the same execution domain. Finally, integrating hardware components is straightforward given a producer/consumer communication mechanism understood by both the software and hardware.

5 Nature of Applications

Supporting the zero-stall guarantee, especially with pipelineable services, is difficult for general purpose OSes. A new abstraction is needed to mange the shared execution context of a pipeline application (fourth claim). Existing abstractions are designed to manage resources as individual entities and not in terms of collections of partially overlapping resources. Further, OSes are not designed to manage resources that simultaneously affect multiple execution contexts.

We suggest that a new abstraction is needed to efficiently manage pipeline applications that may exist in multiple execution contexts simultaneously. With multicore systems, these resources may consist of general purpose cores and specialized cores such as graphics processing units [1] that do not share memory or the OS image. Mapping the stages to general purpose computational resources could be done with a modified version of gang scheduling [11] that is biased towards efficient long term resource allocation. However, as multicore systems become heterogeneous, this new abstraction would need to act as a new form of application label defining the full execution context, hardware requirements, and memory address space for a single pipeline application. This label needs to be orthogonal to existing application labels as a pipeline stage may be a subset of an application and therefore it would be inappropriate to label an entire application as being part of a pipeline.

The key insight is that the pipeline application is its own application-like entity that is overlaid on both the application and the OS. Recall from Section 2.1 that pipeline stages are permitted to maintain their own local state. Therefore, there are in fact multiple distinct memory storage locations. First, the pipeline has its own state that is distinct from other applications. Second, stages have state that may be shared with individual applications. Finally, the applications that contain stages may have their own local private state that should not be shared with any pipeline stages. Efficiently maintaining the privacy of these different regions requires correct labeling.

This new multi-domain application abstraction is fundamentally different from previous models. Previous related work has focused on either bringing everything into a single address space [3] or allowed data to flow between domains under very controlled situations [2, 7, 8, 12]. The important difference is that this multi-domain application model respects the private data model implicit in single-domain applications while providing first-class naming for multi-domain pipelines.

6 Conclusion

This paper argued that multicore architectures will make it possible to achieve the optimal performance

potential of pipelines and presented the system extensions necessary for implementation. A supporting example from the domain of network frame processing was presented showing that performance improvements may be proportional to the depth of the pipeline (three in the example). It was argued that realizing the improvements requires the operating system to provide a zero-stall guarantee that cannot be realized in an environment where the computational resources are oversubscribed. Meeting the zero-stall guarantee for any pipeline requires that the system: (1) fully dedicates (i.e., by selectively disabling time-sharing) sufficient computational resources to the application and (2) provides a set of pipelineable services. Finally, supporting a pipeline that spans multiple execution contexts (e.g., uses pipelineable services) requires a new abstraction to label the pipeline as single entity for resource allocation and security.

References

1. Advanced Micro Devices. AMD completes ATI acquisition and creates processing powerhouse. `http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~113741,00.html`, October 2006.
2. B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. In *12th Symposium on Operating Systems Principles*, pages 102–113, December 1989.
3. B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, 1995.
4. CNET News.com. Intel pledges 80 cores in five years. `http://news.com.com/2100-1006_3-6119618.html`, September 2006.
5. F. J. Corbató, M. M. Daggett, R. C. Daley, R. J. Creasy, J. D. Hellwig, R. H. Orenstein, and L. K. Korn. *The Compatible Time-Sharing System: A Programmer's Guide*. Cambridge, Massachusetts, 1963.
6. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters, December 2004.
7. P. Druschel and L. L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 189–202, New York, NY, USA, 1993. ACM Press.
8. D. R. Engler, M. F. Kaashoek, and J. J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM Press.
9. J. Giacomoni, J. K. Bennett, A. Carzaniga, M. Vachharajani, and A. L. Wolf. FShm: High-rate frame manipulation in kernel and user-space. Technical Report CU-CS-1015-07, Univerity of Colorado at Boulder, 2006.
10. J. Giacomoni, M. Vachharajani, and T. Moseley. FastForward for concurrent threaded pipelines. Technical Report CU-CS-1023-07, Univerity of Colorado at Boulder, 2007.
11. M. A. Jette. Performance characteristics of gang scheduling in multiprogrammed environments. 1997.
12. Y. A. Khalidi and M. N. Thadani. An efficient zero-copy I/O framework for UNIX. Technical report, Mountain View, CA, USA, 1995.
13. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.
14. H. Massalin and C. Pu. Fine-grain adaptive scheduling using feedback. *Computing Systems*, 3(1):139–173, 1989.
15. H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)*, volume 23, pages 191–201, 1989.
16. T. Moseley, A. Shye, V. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization (CGO)*, March 2007.
17. G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 105–118, Los Alamitos, CA, USA, 2005. IEEE Computer Society.
18. G. Ottoni, R. Rangan, A. Stoler, M. Bridges, and D. August. From sequential programs to concurrent threads. *Computer Architecture Letters, IEEE*, 5:6–9, 2006.
19. R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *13th International Conference on Parallel Architecture and Compilation Techniques (PACT'04)*, pages 177–188, Los Alamitos, CA, USA, 2004. IEEE Computer Society.
20. D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6 (part 2)):1905+, 1978.
21. A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2007.
22. M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM Press.
23. H. Zheng and J. Nieh. SWAP: A scheduler with automatic process dependency detection. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI-2004)*, pages 145–158, San Francisco, CA, Mar. 29–31 2004. Usenix Association.