Harnessing Chip-Multiprocessors with Concurrent Threaded Pipelines

John Giacomoni and Manish Vachharajani
University of Colorado at Boulder

University of Colorado at Boulder
Technical Report CU-CS-1024-07
January 2007

Dept. of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309-0430

Dept. of Electrical and Computer Engineering
Campus Box 425
University of Colorado
Boulder, Colorado 80309-0425

Harnessing Chip-Multiprocessors with Concurrent Threaded Pipelines

John Giacomoni and Manish Vachharajani
University of Colorado at Boulder

Abstract

Single-core performance increases have stalled. To increase available cycles, microprocessor designers have shifted to chip-multiprocessor (CMP) designs. Unfortunately, the additional processors provided by CMPs may remain idle because most applications lack data-parallelism and task-parallelism is unlikely to saturate future CMP designs. The systems community needs to rethink how systems are structured to fully utilize CMPs.

We propose that operating systems be adapted to harness CMP resources by leveraging recent results in Concurrent Threaded Pipeline (pipeline-parallel) organizations. This paper discusses potential performance improvements of CTPs and the necessary OS support.

1 Introduction

Traditionally, increases in transistors and fabrication technology have led to increased performance. However, these techniques are showing diminishing returns due to limitations arising from power consumption, design complexity, and wire delays. In response, designers have turned to chip-multiprocessors (CMPs) that incorporate multiple cores on a single die. While CMPs are a boon to throughput driven applications such as web servers, single-threaded applications' performance remains stagnant. This is because the typical approach to parallelizing software (data-parallel or task-parallel) has been to find, extract, and run nearly independent code regions on separate processors [24]; a difficult task for general purpose applications [2].

An alternative and more promising approach is to use a pipeline-parallel organization. This is accomplished by decomposing a task into a series of sequential stages connected by a data-forwarding mechanism. Data-dependencies are easily handled, provided each datum only references previous data. Further, throughput may increase proportionally to the depth of the pipeline with a short completion interval. For these reasons, modern hardware systems, from microprocessors to routers, are built on a pipeline design. While software-based pipelines have been proposed in the past, only today's CMPs deliver the resources to capture the performance benefits of software pipeline-parallel organizations.

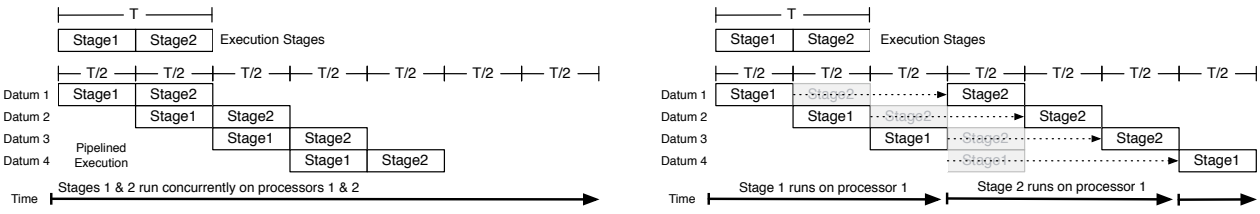
Superficially, CMP systems may appear equivalent to traditional SMP systems. However, CMPs exhibit sufficient differences in the details to warrant a closer examination, similar to the CISC→RISC transition discussed by Anderson [3]. First, cores on a CMP die may be heterogeneous in function and performance. General purpose cores may be packaged together with graphics and other specialized processing units [1]. Asymmetric cores that support the same instruction set with different performance characteristics may also be included [15]. Second, communication latencies between cores will be asymmetric if two cores are communicating on-die or off-die, presently up to 10x. Operating systems will need to be updated to efficiently support the heterogeneous and asymmetric systems of the near future.

The third difference is the most fundamental change for general purpose operating systems. Systems with eighty sophisticated general purpose cores per system are expected within five years [7]. No longer will the operating system be required to time-share every processor in the system. It will be possible to run applications to termination on bound processors without impacting the overall responsiveness of the system.

Leveraging CMP systems, recent work in Concurrent Threaded Pipelining (CTP) has shown that it is now possible to capture the performance benefits of software pipeline-parallel organizations [11, 20, 21, 24]. Additionally, this recent work shows that sequential applications that do not exhibit high-level pipeline-parallelism can still benefit by introducing fine-grain CTP optimizations operating at time-scales comparable to main-memory latency. These pipelines will be flexible analogs of hardware pipelines implemented on general purpose commodity systems.

Therefore, we propose that CTP organizations be embraced as a means to capture the performance potential of CMP systems by parallelizing most sequential applications. However, supporting CTPs in a general purpose OS is not straightforward.

The rest of this paper is organized as follows. Section 2 explains why CMP could revive pipeline-parallel organizations. Section 3 discusses CTPs on CMP hardware and provides a summary of initial results. Section 4 discusses how a CTP application differs from normal applications and thus requires a different application model. Section 5 discusses the additional research needed to optimally support CTP applications. Section 6 concludes.



(a) Concurrent

(b) Sequential

Figure 1: Pipeline Timing

2 Classic Pipelining

Pipelining as a software architecture has been extensively studied and is visible in some systems, including Unix pipes [25] and the Click Modular Router [14]. Synthesis, a direct ancestor of CTPs, used concurrent lock-free queues to implement efficient data passing [18, 19].

Despite the existence of efficient pipelining systems, pipelines have not made significant inroads to production systems beyond Unix pipes. This may be because previous systems optimized for obsolete architectural limitations or focused on software engineering advantages. The limited number of processors previously available per system (1-4) created an oversubscription scenario necessitating work on improved scheduling [17, 29]. However, no matter how efficient one makes the scheduling and communication components [4, 8, 13, 18, 22], they can never be faster than direct procedure calls.

To understand why additional computational resources may revive pipeline-parallel designs, contrast a two stage pipeline where both stages are scheduled on *2 processors for simultaneous execution* (scheduled concurrently) to a scenario where they are not. In the concurrent scheduling scenario, throughput is one datum per timestep (Figure 1.a). However, in an oversubscribed scenario (Figure 1.b), where both stages cannot be scheduled concurrently, the application degenerates into a sequential application with half the throughput. Therefore, if the all stages in a pipeline-parallel application are run sequentially, the communication and scheduling overhead may cause the pipeline application to perform worse than the sequential version. Further, in longer pipelines, bubbles introduced by stalled stages will temporally move through the remainder of the pipeline, resulting in further delays. Therefore until now, pipeline-parallel architectures have provided only software engineering benefits to system architects.

3 Concurrent Threaded Pipelining

Concurrent Threaded Pipelining is a generalized performance oriented extension of prior work on pipelined software architectures that relies on concurrency. The performance focus of CTPs permits the harnessing of previously inaccessible thread-level parallelism in common applications by software architects or optimizing compilers. However prior work has shown that exploiting CTPs for a wide range of common applications requires the underlying OS and communication primitives to provide for very fine-grain stage-to-stage interactions. In many cases, observed processor-to-processor communication latencies must be less than half the latency of a single main-memory access [20, 26].

One example of such prior work is the Decoupled Software Pipelining (DSWP) work by the Liberty group at Princeton. They show that it is possible to automatically find and extract CTP parallelism from sequential applications, gaining 9.2% mean performance improvements [20, 21, 24]. Note, however, that to achieve these performance gains, the Princeton work requires very low-cost communication and synchronization which they implement in hardware.

Fortunately, our recent work has shown that by carefully managing threads it is possible to achieve and maintain the requisite performance in software [11]. This result is critical because it demonstrates that it is possible to maintain the flexibility of software without relying upon custom hardware. Further, since our communication library has an inexpensive polymorphic interface, pipeline stages can be interconnected using the most appropriate software or hardware primitive. This flexibility allows one to link in any system component as a pipeline stage, be it a user-space thread, an OS service, software on a specialized core, a hardware accelerator, or a programmable gate array.

Note that the above software communication system cannot be realized by a straightforward application of prior classic pipeline communication approaches to modern systems. For example, Synthesis style queues [18] result in cache-thrashing yielding unacceptable performance[1] [11]. Our solution eliminates the cache-line thrashing by enforcing a temporal slip between the producer and consumer to ensure that their enqueues and

[1] The Synthesis quamachine was cacheless with zero-wait memory.

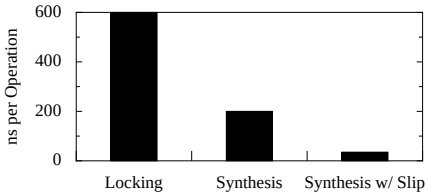


Figure 2: Performance of Software Communication

dequeues are separated by at least a cacheline (see Figure 2 for performance comparisons). Therefore, the performance requirements of CTPs can be met while only mildly increasing latency. Furthermore, the resulting queues are compatible with any memory consistency model[2] as they rely only upon cache-coherence [10, 11].

While the aforementioned work addresses most of the organizational problems in CTPs, Section 5 describes many OS challenges, some of which stem from the nature of CTP applications that are described in the following section.

4 Multi-domain CTP Applications

Concurrent Threaded Pipelines, and to a lesser extent classic pipelines, fundamentally challenge the OS concept of an application. Section 2 argued that full concurrency is necessary to realize the potential of any pipelined application. To properly manage a pipeline it must be treated as a single object and not merely a collection of stages (i.e., plain threads). Handling a pipeline that is fully contained within an application, as in DSWP, is relatively straightforward. Recall, however, that a pipeline may require services from another application or from the OS (e.g., input or output). In the CTP model, these service stages are not independent of other application pipeline stages. Thus, with this organization the existing OS single-domain application model breaks down. The model breaks down further if pipeline stages are implemented as a hardware resource that is not a general purpose execution unit.

To address this, the OS definition of an application should be extended so that the application is defined as the collection of its pipeline stages. Figure 3 depicts a multi-domain application (gray) that is composed of operating system services and two different applications connected by shared memory. Observe that each application keeps its own protected memory space separate from the shared pipeline application's memory.

This new multi-domain model is fundamentally different from previous models. Previous work focused on either bringing everything into a single address space [5]

[2]The Synthesis quamachine had sequential consistency, modern architectures support weaker models such as weak or release consistency.

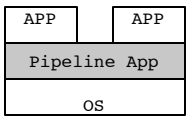


Figure 3: Multi-domain Pipelining

or allowed data to flow between domains under very controlled situations [4, 8, 9, 13]. Synthesis demonstrated that significant performance improvements were possible with careful tuning while maintaining the single domain application model, though the model is blurred by threads crossing the application/kernel boundary to handle tasks on behalf of the operating system [16]. The multi-domain application model respects the private data model implicit in single-domain applications while providing first-class naming for multi-domain pipelines.

Motivating Example: To understand the multi-domain nature of pipeline applications consider the following example based on a CTP network application we constructed [11]. The performance goal was to support the processing of all frames at all frame sizes on a 1,000 Mbps Ethernet network in user-space on commodity hardware. The initial implementation used libpcap and libdnet to move network frames in and out of the application. Unfortunately, the system call overhead of these interfaces was too large to support the smallest sized frames. The solution was to manually segment the application into three sequential pipeline stages: input, user-space application processing, and output. The input and output stages were implemented inside the kernel and attached directly to the network devices. The three stages were then connected by a shared-memory region and our software communication mechanism. In this system, the new kernel I/O services do not belong entirely to the OS or to the application in the classic sense. This CTP application is the sum of its pipeline stages but not the union of the two processes that contain the stages. Other multi-domain pipeline applications are no different.

5 Research Opportunities

Reorganizing operating systems to optimally support Concurrent Threaded Pipelines will not be straightforward given the performance requirements and the special nature of pipeline applications. The non-exhaustive list below may serve as a starting point for discussion on this reorganization.

Pipelineable OS Services are the crux of this line of research. Without them one is relegated to basic CTPs. A

set of basic OS services is needed to permit a CTP application to directly link kernel resources as a pipeline stage, similarly to the prior motivating example. These services must provide a virtualized interface for shared resources without impacting the overall performance of pipeline or traditional applications.

Resource Allocation in a heterogeneous environment is difficult as the demands of competing long running applications must be balanced. Asynchronous memory access latencies (on- and off-die) suggest that pipelines should be allocated on as few dies as possible. However, on systems with asymmetric access to I/O devices (HyperTransport) it might be beneficial to schedule the input and output stages closer to the I/O devices, even if this requires the use of more dies. Additionally, some CMP organizations have shared cache hierarchies between cores on a single die, therefore scheduling needs to minimize cache conflicts [27]. Thus, algorithms for proper allocation and transparent migration of stages among heterogeneous resources are required.

Timesharing may unfortunately be necessary if a system becomes oversubscribed. However, classic priority, proportional [28], and fine-grain [17] scheduling techniques are insufficient as stages with no available work may be accidentally scheduled (since sleep-based blocking may lead to overly long latency). Scheduling decisions need to be made with full awareness of each pipeline and the status of the associated communication primitives. If a single stage is idle, starvation will cascade through the remainder of the pipeline.

Stage Fusion is related to the above scheduling problems and provides an alternative recourse to timesharing in oversubscribed scenarios. It would be useful to be able to shrink the pipeline depth by transparently fusing stages [23] into a single unit, thereby freeing computational resources. Thus, algorithms to find and fuse appropriate stages may be fruitful. These algorithms must decide if it is preferable to fuse stages so that they run round-robin (ABAB) or in stage batches (AABB).

Memory Management is made harder by the tight performance constraints of CTPs. Since communication buffers are shared across domains in multi-domain CTP applications, the system must decide upon a static or dynamic allocation strategy for buffers. Furthermore, buffer tracking, sharing, migration, and reclamation must be considered in light of performance constraints.

Memory Consistency is a critical consideration when building CTPs using shared-memory. Our previous work shows that Synthesis style queues [11, 18] can be used with any consistency model [10], provided each datum fits in a cacheline. However, when each datum is larger than a cache line (ex., network frames) attention must be paid to consistency. On sequential or processor consistency models (x86), consistency is ensured by queuing after all writes are completed. On weak or release consistency models (Itanium), this is not the case and appropriate ordering instructions need to be introduced. Maintaining correct consistency semantics without impacting overall performance is an open problem.

Safety is trivial in the simple case where bounds checking on pointers suffices. However, there are entire classes of problems that can be avoided by using sandbox techniques and typesafe languages to enforce correct operation [12]. Enforcing enhanced safety requirements without impacting performance is also an open problem.

CTP libraries will be critical to assisting programmers to construct CTP pipelines in the most efficient manner possible. Reasoning about parallel applications is difficult; therefore, providing a robust set of library routines that provide pipeline-parallelism from a sequential interface would be of significant value. Examples include mathematical operations and routines to traverse recursive data structures (e.g., searching and iteration).

Pipeline Extraction is generally beyond the scope of the operating system. However, extracting pipelines from sequential applications or algorithms is non-trivial and warrants further investigation. Promising avenues of research include compiler extraction [20], special purpose languages, language extensions [6], and runtime decomposition of sequential applications.

6 Conclusion

Concurrent Threaded Pipelines have shown positive results in harnessing the computation power of chip-multiprocessors in ways not previously possible with symmetric-multiprocessors. This is possible because the increasing number of cores per system will permit operating systems to selectively disable time-sharing to concurrently run all stages of a pipeline-parallel application. The close coupling of cores on a single die provides a fast core-to-core path enabling the fine-grain optimizations required for CTP conversion of many sequential applications. Finally, the heterogeneity of CMP cores will permit pipeline stages to be run on the task optimal core.

To optimally harness these new resources we proposed that operating systems be adapted to treat CTP applications as first-class citizens. We argued that the definition of an application needs to be extended so that a CTP application is the collection of its pipeline stages, whether the stages are from a single or multiple domains. Finally, we presented a list of future research topics that will be important for the success of CTPs: (1) pipelineable

OS services; (2) resource allocation; (3) timesharing; (4) stage fusion; (5) memory management; (6) memory consistency; (7) safety; (8) CTP libraries; (9) pipeline extraction. Finally, we believe that it is possible to architect a system that simultaneously supports CTP and traditional applications without sacrificing performance.

Acknowledgements

We thank Christoph Reichenbach, Anmol Sheth, Brian Shucker, and Jeremy Siek for their help on this work.

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] S. Amarasinghe. Multicores from the compiler's perspective: A blessing or a curse? In *2005 International Symposium on Code Generation and Optimization (CGO)*.

[3] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The interaction of architecture and operating system design. In *ASPLOS, International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, Santa Clara, CA (USA), 1991.

[4] 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.

[5] 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.

[6] G. Chrysanthakopoulos and S. Singh. An asynchronous messaging library for c#. In *Proceedings of the 2005 Symposium on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, October 2005.

[7] CNET News.com. Intel pledges 80 cores in five years. `http://news.com.com/2100-1006_3-6119618.html`, September 2006.

[8] 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.

[9] 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.

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years ISCA: Retrospectives and Reprints*, pages 376–387, 1998.

[11] 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, Univerity of Colorado at Boulder, 2006.

[12] G. C. Hunt, J. R. Larus, D. Tarditi, and T. Wobber. Broad new OS research: Challenges and opportunities. In *Proceedings of the 2006 HotOS Workshop on Hot Operating Systems*, 2006.

[13] Y. A. Khalidi and M. N. Thadani. An efficient zero-copy i/o framework for unix. Technical report, Mountain View, CA, USA, 1995.

[14] 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.

[15] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 64, Washington, DC, USA, 2004. IEEE Computer Society.

[16] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, 1992.

[17] H. Massalin and C. Pu. Fine-grain adaptive scheduling using feedback. *Computing Systems*, 3(1):139–173, 1989.

[18] 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.

[19] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared — memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[20] 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.

[21] 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.

[22] V. S. Pai, P. Druschel, and W. Zwaenepoel. Io-lite: a unified i/o buffering and caching system. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 15–28, Berkeley, CA, USA, 1999. USENIX Association.

[23] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[24] 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.

[25] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6 (part 2)):1905+, 1978.

[26] J. E. Smith. Decoupled access/execute computer architectures. *ACM Trans. Comput. Syst.*, 2(4):289–308, 1984.

[27] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[28] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Operating Systems Design and Implementation*, pages 1–11, 1994.

[29] 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.