View metadata, citation and similar papers at core.ac.uk brought to you by CORE provided by CiteSeerX

Characteristics of Workloads Using the Pipeline Programming Model

Christian Bienia and Kai Li
Department of Computer Science, Princeton University
{cbienia, li}@cs.princeton.edu

Abstract

Pipeline parallel programming is a frequently used model to program applications on multiprocessors. Despite its popularity, there is a lack of studies of the characteristics of such workloads. This paper gives an overview of the pipeline model and its typical implementations for multiprocessors. We present implementation choices and analyze their impact on the program. We furthermore show that workloads that use the pipeline model have their own unique characteristics that should be considered when selecting a set of benchmarks. Such information can be beneficial for program developers as well as for computer architects who want to understand the behavior of applications.

1 Introduction

Modern parallel workloads are becoming increasingly diverse and use a wide range of techniques and methods to take advantage of multiprocessors. The pipeline parallelization model is one such method that is particularly noteworthy due to its prevalence in certain application areas such as server software.

Despite its frequent use, the characterizations of workloads using the pipeline model have not been studied much. One reason for this might be the traditional focus of the research community on scientific workloads, which typically do not exploit pipeline parallelism. Until recently few benchmark programs that implement pipelines have been available.

This issue is further exacerbated by the fact that pipeline parallelism is emerging as a key method to take advantage of the large number of cores that we can expect from future multiprocessors. Methods such as the stream programming model [10, 12], assisted parallelization [16] and even automatic parallelization [13, 15] can be used to parallelize programs by expressing the computational steps of a serial workload as a parallel pipeline. These trends might lead to an explosive increase of pipelined programs on multiprocessors.

This paper makes two main contributions. First, we present a brief survey of how the pipeline model is used in practice. Our overview can help other researchers to determine what part of the design space of pipelined programs is covered by their applications. Second, we demonstrate that the programs using the pipeline model have different characteristics compared to other workloads. The differences are significant and systematic in nature, which justifies the existence of pipelined programs in the PARSEC benchmark suite [2]. This suggests that pipelined workloads should be considered for the inclusion in future benchmark programs for computer architecture studies.

The remainder of the paper is structured as follows: Section 2 presents a survey of the pipeline parallelization model. In Section 3 we discuss how we studied the impact of the pipeline model on the workload characteristics, and we present our experimental results in Section 4. Related work is discussed in Section 5 before we conclude in Section 6.

2 Pipeline Programming Model

Pipelining is a parallelization method that allows a program or system to execute in a decomposed fashion. A pipelined workload for multiprocessors breaks its work steps into units or *pipeline stages* and executes them concurrently on multiprocessors or multiple CPU cores. Each pipeline stage typically takes input from its input queue, which is the output queue of the previous stage, computes and then outputs to its output queue, which is the input queue of the next stage. Each stage can have one or more threads depending on specific designs. Figure 1 shows this relationship between stages and queues of the pipeline model.

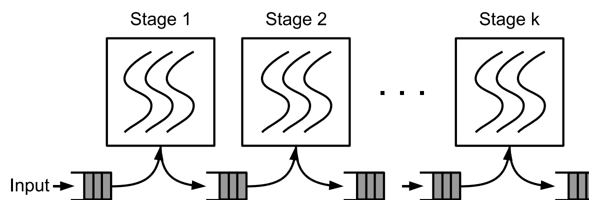

Figure 1: A typical linear pipeline with multiple concurrent stages. Pipeline stages have a producer - consumer relationship to each other and exchange data with queues.

2.1 Motivation for Pipelining

In practice there are three reasons why workloads are pipelined. First, pipelining can be used to simplify program engineering, especially for large-scale software development. Pipelining decomposes a problem into smaller, well-defined stages or pieces so that different design teams can develop different pipeline stages efficiently. As long as the interfaces between the stages are properly defined, little coordination is needed between the different development teams so that they can work independently from each other in practice. This typically results in improved software quality and lowered development cost due to simplification of the problem and specialization of the developers. This makes the pipeline model well suited for the development of large-scale software projects.

Second, the pipeline programming model can be used to take advantage of specialized hardware. Pipelined programs have clearly defined boundaries between stages, which make it easy to map them to different hardware and even different computer systems to achieve better hardware utilization.

Third, pipelining increases program throughput due to a higher degree of parallelism that can be exploited. The different pipeline stages of a workload can operate concurrently from each other, as long as enough input data is available. It can even result in fewer locks than alternative parallelization models [11] due to the serialization of data. By keeping data in memory and transferring it directly between the relevant processing elements, the pipeline model distributes the load and reduces the chance for bottlenecks. This has been a key motivation for the development of the stream programming model [8], which can be thought of as a fine-grained form of the pipeline programming model.

2.2 Uses of the Pipeline Model

These properties of the pipeline model typically result in three uses in practice:

1. Pipelining as a hybrid model with data-parallel pipeline stages to increase concurrency
2. Pipelining to allow asynchronous I/O
3. Pipelining to model algorithmic dependencies

The first common use of the pipeline model is as a hybrid model that also exploits data parallelism. In that case the top-level structure of the program is a pipeline, but each pipeline stage is further parallelized so that it can process multiple work units concurrently. This program structure increases the overall concurrency and typically results in higher speedups.

The second use also aims to increase program performance by increasing concurrency, but it exploits parallelism between the CPUs and the I/O subsystem. This is done either by using special non-blocking system calls for I/O, which effectively moves that pipeline stage into the operating system, or by creating a dedicated pipeline stage that will handle blocking system calls so that the remainder of the program can continue to operate while the I/O thread waits for the operation to complete.

Lastly, pipelining is a method to decompose a complex program into simpler execution steps with clearly defined interfaces. This makes it popular to model algorithmic dependencies which are difficult to analyze and might even change dynamically at runtime. In that scenario the developer only needs to keep track of the dependencies and expose them to the operating system scheduler, which will pick and execute a job as soon as all its prerequisites are satisfied. The pipelines modeled in such a fashion can be complex graphs with multiple entry and exit points that have little in common with the linear pipeline structure that is typically used for pipelining.

2.3 Implementations

There are two ways to implement the pipeline model: fixed data and fixed code. The fixed data approach has a static mapping of data to threads. With this approach each thread applies all the pipeline stages to the work unit in the predefined sequence until the work unit has been completely processed. Each thread of a fixed data pipeline would typically take on a work unit from the program input and carry it through the entire program until no more work needs to be done for it, which means threads can potentially execute all of the parallelized program code but they will typically only see a small subset of the input data. Programs that implement fixed data pipelines are therefore also inherently data-parallel because it can easily happen that more than one thread is executing a function at any time.

The fixed code approach statically maps the program code of the pipeline stages to threads. Each thread executes only one stage throughout the program execution. Data is passed between threads in the order determined by the pipeline structure. For this reason each thread of a fixed code pipeline can typically only execute a small subset of the program code, but it can potentially see all work units throughout its lifetime. Pipeline stages do not have to be parallelized if no more than one thread is active per pipeline stage at any time, which makes this a straightforward approach to parallelize serial code.

2.3.1 Fixed Data Approach

The fixed data approach uses a static assignment of data to threads, each of which applies all pipeline stages to the data until completion of all tasks. The fixed data approach can be best thought of as a full replication of the original program, several instances of which are now executed concurrently

and largely independently from each other. Programs that use the fixed data approach are highly concurrent and also implicitly exploit data parallelism. Due to this flexibility they are usually inherently load-balanced.

The key advantage of the fixed data approach is that it exploits data locality well. Because data does not have to be transferred between threads, the program can take full advantage of data locality once a work unit has been loaded into a cache. This assumes that threads do not migrate between CPUs, a property that is usually enforced by manually pinning threads to cores.

The key disadvantage is that it does not separate software modules to achieve a better division of labor for teamwork, simple asynchronous I/Os, or mapping to special hardware. The program will have to be debugged as a single unit. Asynchronous I/Os will need to be handled with concurrent threads. Typically, no fine-grained mapping to hardware is considered.

Another disadvantage of this approach is that the working set of the entire execution is proportional to the number of concurrent threads, since there is little data sharing among threads. If the working set exceeds the size of the low-level cache such as the level-two cache, this approach may cause many DRAM accesses due to cache misses. For the case that each thread contributes a relatively large working set, this approach may not be scalable to a large number of CPU cores.

2.3.2 Fixed Code Approach

The fixed code approach assigns a pipeline stage to each thread, which then exchange data as defined by the pipeline structure. This approach is very common because it allows the mapping of threads to different types of computational resources and even different systems.

The key advantage of this approach is its flexibility, which overcomes the disadvantages of the fixed data approach. As mentioned earlier, it allows fine-grained partitioning of software projects into well-defined and well-interfaced modules. It can limit the scope of asynchronous I/Os to one or a small number of software modules and yet achieves good performance. It allows engineers to consider fine-grained processing steps to fully take advantage of hardware. It can also reduce the aggregate working set size by taking advantage of efficient data sharing in a shared cache in a multiprocessor or a multicore CPU.

The main challenge of this approach is that each pipeline stage must use the right number of threads to create a load-balanced pipeline that takes full advantage of the target hardware because the throughput of the whole pipeline is determined by the rate of its slowest pipeline stage. In particular, pipeline stages can make progress at different rates on different systems, which makes it hard to find a fixed assignment of resources to stages for different hardware. A typical solution to this problem on shared-memory multiprocessor systems is to over-provision threads for pipeline stages so that it is guaranteed that enough cores can be assigned to each pipeline stage at any time. This solution delegates the task of finding the optimal assignment of cores to pipeline stages to the OS scheduler at runtime. However, this approach introduces additional scheduling overhead for the system.

Fixed code pipelines usually implement mechanisms to tolerate fluctuations of the progress rates of the pipeline stages, typically by adding a small amount of buffer space between stages that can hold a limited number of work units if the next stage is currently busy. This is done with synchronized queues on shared-memory machines or network buffers if two connected pipeline stages are on different systems. It is important to point out that this is only a mechanism to tolerate variations in the progress rates of the pipeline stages, buffer space does not increase the maximum possible throughput of a pipeline.

3 Methodology

We studied the impact of the pipelining model with the PARSEC benchmark suite [2]. To analyze the behavior of the programs we chose a set of characteristics and measured them for the PARSEC `simlarge` input set on a particular architecture. We then processed the data with Principal Component Analysis (PCA) to automatically eliminate highly correlated data. The result is a description of the program behavior that is free of redundancy. The results are visualized using scatter plots.

This methodology to analyze program characteristics is the common method for similarity analysis. Measuring characteristics on an ideal architecture is frequently used to focus on program properties that are inherent to the algorithm implementation and not the architecture [1, 2, 19]. PCA has been in use for years as an objective way to quantify similarity [4, 5, 7, 14, 18].

3.1 Workloads

We used PARSEC 2.1 to study the impact of the pipeline model. The suite contains workloads implementing all the usage scenarios discussed in Section 2.2. Table 1 gives an overview of the four PARSEC workloads that use the pipeline model.

`Dedup` and `ferret` are server workloads which implement a typical linear pipeline with the fixed code approach (see Section 2.3.2). `X264` uses the pipeline model to model dependencies between frames. It constructs a complex pipeline at runtime based on its encoding decision in which each frame corresponds to a pipeline stage. The pipeline has the form of a directed, acyclical graph with multiple root nodes formed by the pipeline stages corresponding to the I frames. These frames can be encoded independently

Workload	Parallelism			Dependency Modeling
	Pipeline	Data	I/O	
bodytrack	N	Y	Y	N
dedup	Y	Y	Y	N
ferret	Y	Y	Y	N
x264	Y	N	N	Y

Table 1: The four workloads of PARSEC 2.1 which use the pipeline model. 'Pipeline parallelism' in the table refers only to the decomposition of the computationally intensive parts of the program into separate stages and is different from the pipeline model as a form to structure the whole program (which includes stages to handle I/O).

from other frames and thus do not depend on any input from other pipeline stages.

The bodytrack workload only uses pipelining to perform I/O asynchronously. It will be treated as a data-parallel program for the purposes of this study because it does not take advantage of pipeline parallelism in the computationally intensive parts. The remaining three pipelined workloads will be compared to the data-parallel programs in the PARSEC suite to determine whether the pipeline model has any influence on the characteristics.

3.2 Program Characteristics

For our analysis of the program behavior we chose a total of 73 characteristics that were measured for each of the 13 PARSEC workloads, yielding a total of 949 sample values that were considered. Our study focuses on the parallel behavior of the multithreaded programs relevant for studies of CMPs. The characteristics we chose encode information about the instruction mix, working sets and sharing behavior of each program as follows:

Instruction Mix 25 characteristics that describe the breakdown of instruction types relative to the total amount of instructions executed by the program

Working Sets 8 characteristics encoding the working set sizes of the program by giving the miss rate for different cache sizes

Sharing 40 characteristics describing how many lines of the total cache are shared and how intensely the program reads or writes shared data

The working set and sharing characteristics were measured for a total of 8 different cache sizes ranging from 1 MBytes to 128 MBytes to include information about a range of possible cache architectures. This approach guarantees that unusual changes in the data reuse behavior due to varying cache sizes are captured by the data. The range of cache sizes that we considered has been limited to realistic sizes to make sure that the results of our analysis will not be skewed towards unrealistic architectures.

3.3 Experimental Setup

To collect the characteristics of the workloads we simulate an ideal machine that can complete all instructions within one cycle using Simics. We chose an ideal machine architecture because we are interested in properties inherent to the program, not in characteristics of the underlying architecture. The binaries which we used are the official precompiled PARSEC 2.1 binaries that are publicly available on the PARSEC website. The compiler used to generate the precompiled binaries was gcc 4.4.0.

We simulated an 8-way CMP with a single cache hierarchy level that is shared between all threads. The cache is 4-way associative with 64 byte lines. The capacity of the cache was varied from 1 MB to 128 MB to obtain information about the working set sizes with the corresponding sharing behavior. Only the Region-of-Interest (ROI) of the workloads was characterized.

3.4 Principal Component Analysis

Principal Component Analysis (PCA) is a mathematical method to transform a number of possibly correlated input vectors into a smaller number of uncorrelated vectors. These uncorrelated vectors are called the principal components (PC). We employ PCA in our analysis because PCA is considered the simplest way to reveal the variance of high-dimensional data in a low dimensional form.

To compute the principal components of the program characteristics, the data is first mean-centered and normalized so it is comparable with each other. PCA is then used to reduce the number of dimensions of the data. The resulting principal components have decreasing variance, with the first PC containing the most amount of information and the last one containing the least amount. We use the Kaiser's Criterion to eliminate PCs which do not contain any significant amount of information in an objective way. Only the top PCs with eigenvalues greater than one are kept, which means that the resulting data is guaranteed to be uncorrelated but to still contain most of the original information.

4 Experimental Results

In this section we will discuss how the use of the pipeline programming model has affected the characteristics of the PARSEC workloads. Our analysis shows that there are substantial, systematic differences, which suggests that researchers can improve the diversity of their benchmark selection by including pipelined programs.

Figure 2 shows the first three principal components derived from all studied characteristics. As can be seen the three workloads which employ the pipelining model (represented by blue dots) occupy a different area of the PCA

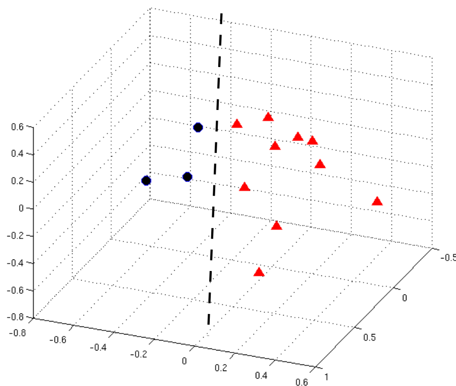

Figure 2: Comparison of the first three principal components of all characteristics of the PARSEC workloads. Pipeline workloads are represented by blue dots, all other workloads by red triangles. The data shows significant systematic differences between the two types of programs.

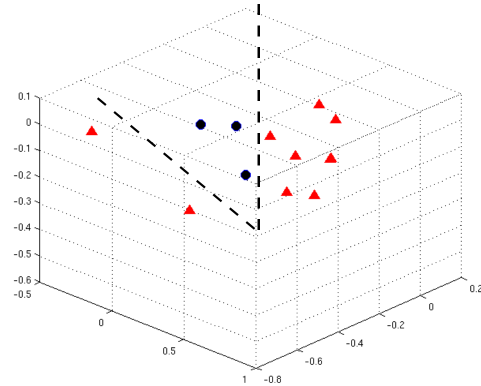

Figure 3: Comparison of the first three principal components of the sharing characteristics of the PARSEC workloads. Pipeline workloads are represented by blue dots, all other workloads by red triangles. Systematic differences in sharing are a major source for the different behavior of pipelined workloads.

space as the rest of the PARSEC programs (represented by red triangles). The PCA space can be separated so that the different clusters become visible, as is indicated by the dashed line which we have added as a visual aid.

A further investigation of the individual characteristics reveals the sharing behavior of the workloads as a major source for the differences. In Figure 3 we present a scatter plot that was obtained with just the sharing characteristics. As can be seen the PCA space of the sharing characteristics can also be separated so that the two types of workloads occupy different areas. However, the difference seems to be less pronounced than in the previous case which considered all characteristics.

The remaining characteristics which encode the instruction mix and working sets of the workloads also exhibit a small tendency to group according to the parallelization model of the workloads. However, the differences are much smaller in scope and separation. The aggregate of these differences appears to be the reason for the clearer separation seen in Figure 2 compared to Figure 3.

Our analysis suggests that pipelined programs form their own type of workload with unique characteristics. Their behavior is different enough to warrant their consideration for inclusion in a mix of benchmarks for computer architecture studies.

5 Related Work

Kuck published a survey about parallel architectures and programming models [9] over thirty years ago. He covers various early methods to parallelize programs but does not include the pipeline model.

For main memory transaction processing on multiprocessors, Li and Naughton demonstrate that pipelined programs can achieve higher throughput and less locking overhead [11].

Subhlok et al. study how the stages of a pipeline can be mapped optimally to processors [16]. They developed a new algorithm to compute a mapping that optimizes the latency with respect to constraint throughput and vice versa. The algorithm addresses the general mapping problem, which includes processor assignment, clustering and replication.

Thies et al. present a systematic technique to parallelize streaming applications written in C with the pipeline parallelization model [17]. They suggest a set of annotations that programmers can use to parallelize legacy C programs so they can take advantage of shared-memory multiprocessors. The programmer is assisted by a dynamic analysis that traces the communication of memory locations at runtime.

The *stream programming model* is a parallelization approach that decomposes a program into a parallel network of specialized kernels which are then mapped to processing elements [3, 6, 8]. Data is organized as streams, which is a sequence of similar elements. A kernel in the stream programming model consumes streams, performs a computation, and produces a set of output streams. It corresponds to a pipeline stage of the pipeline programming model. Stream programs are suitable for execution on general-purpose multiprocessors [10, 12].

Decoupled Software Pipelining (DSWP) is an automatic parallelization method which uses the pipeline model [13, 15]. It exploits the fine-grained pipeline parallelism inherent in most applications to create a multithreaded version of the program that implements a parallel pipeline. Low-

overhead synchronization between the pipeline stages can be implemented with a special synchronization array [15].

6 Conclusions

This paper gives an overview of the pipeline programming model, its implementation alternatives on multiprocessors and the challenges faced by developers.

To analyze how pipeline parallelization affects the characteristics of a workload we studied the programs of the PARSEC benchmark suite. The suite contains several programs that implement the pipeline model in different ways. Our results show that workloads that use the pipeline model have systematically different characteristics. A major reason for the changed characteristics are differences in the sharing behavior. Our results suggest that researchers should consider adding pipelined workloads to their mix of benchmark programs for computer architecture studies.

References

[1] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In *Proceedings of the 2008 International Symposium on Workload Characterization*, September 2008.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *International Conference on Computer Graphics and Interactive Techniques 2004*, pages 777–786, New York, NY, USA, 2004. ACM.

[4] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Quantifying the Impact of Input Data Sets on Program Behavior and its Applications. *Journal of Instruction-Level Parallelism*, 5:1–33, 2003.

[5] R. Giladi and N. Ahituv. SPEC as a Performance Evaluation Measure. *Computer*, 28(8):33–42, 1995.

[6] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[7] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring Benchmark Similarity Using Inherent Program Characteristics. *IEEE Transactions on Computers*, 28(8):33–42, 1995.

[8] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media Processing with Streams. *IEEE Micro*, 21(2):35–46, 2001.

[9] D. J. Kuck. A Survey of Parallel Machine Organization and Programming. *ACM Computing Surveys*, 9(1):29–59, 1977.

[10] M. Kudlur and S. Mahlke. Orchestrating the Execution of Stream Programs on Multicore Platforms. *SIGPLAN Notices*, 43(6):114–124, 2008.

[11] K. Li and J. F. Naughton. Multiprocessor Main Memory Transaction Processing. In *Proceedings of the First International Symposium on Databases in Parallel and Sistributed Systems*, pages 177–187, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[12] S.-w. Liao, Z. Du, G. Wu, and G.-Y. Lueh. Data and Computation Transformations for Brook Streaming Applications on Multiprocessors. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 196–207, Washington, DC, USA, 2006. IEEE Computer Society.

[13] G. Ottoni, R. Rangan, A. Stoler, and D. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the 38th Annual International Symposium on Microarchitecture*, page 12, 2005.

[14] A. Phansalkar, A. Joshi, and L. K. John. Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 412–423, New York, NY, USA, 2007. ACM.

[15] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August. Decoupled Software Pipelining with the Synchronization Array. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 177–188, 2004.

[16] J. Subhlok and G. Vondran. Optimal Latency-Throughput Tradeoffs for Data Parallel Pipelines. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 62–71, New York, NY, USA, 1996. ACM.

[17] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[18] Vandierendonck, H. and De Bosschere, K. Many Benchmarks Stress the Same Bottlenecks. In *Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 57–64, 2 2004.

[19] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.