Utilizing Multiple Hardware Threads with Pipeline Parallelism

M. Anton Ertl*, TU Wien

Abstract. In recent years general-purpose CPUs have aquired multiple hardware threads by providing multiple cores per CPU (multi-core) and multiple hardware threads per core (simultaneous multi-threading). Making good use of such resources has been a challenge for several decades that has been successfully attacked for scientific applications, but not to a significant extent for general-purpose applications. The main current paradigm for this programming problem is to have explicit threads that share memory and are synchronized by a variety of synchronzation constructs. Unfortunately, it seems to be too hard to program profitably in this paradigm for general-purpose applications. Pipeline parallelism is a programming paradigm that has proved so easy to understand that shell programmers use it even on machines that have only one hardware thread. In this work we present the case for better support for pipeline parallelism in programming languages, and present ideas on how to improve the scalability of the implementation of pipeline parallelism.

1 Introduction

PCs have gained more than one hardware thread in the last few years. This poses the challenge of utilizing all the threads that the hardware makes available. While multi-processor computers have existed for decades, and there has also been a huge amount of research in parallel computing, this research has focused on scientific applications, the main use of multi-processors in earlier times.

However, PCs are usually not used for scientific applications, and the characteristics of general-purpose applications differ from scientific applications in ways that affect parallelisation; e.g., loop trip counts are typically high for scientific applications, but low for general-purpose applications [Lar91].

2 Background

2.1 Threads and tasks

The hardware supplies thread contexts (e.g., one core on a machine with multiple cores but without simultaneous multi-threading (SMT), or one thread on a core

* Correspondence to: M. Anton Ertl, Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; Email: anton@mips.complang.tuwien.ac.at

that supports SMT). The operating system (OS) supplies OS threads and it allocates hardware threads dynamically to OS threads as it sees fit. Programming languages often also provide something that is often called threads or tasks, and the implementation of the programming language can map these programming language features to OS threads in a 1:1 fashion, or try to avoid system call overhead by using a more complicated scheme.

In this paper, when we discuss the implementation, in particular, the mapping of programming language constructs to OS constructs, we use *threads* to mean OS or hardware threads, and *tasks* to mean programming language tasks.

2.2 How much parallelism is there?

Is there actually parallelism in non-numeric applications? Some of the limits studies that looked at the dynamic data flow dependencies of application runs found high levels of parallelism in non-numeric applications [LW92]. We only have to find out how to organize the data flow into threads (for thread-level parallelism).

2.3 Why is parallel programming hard?

Parallel programming introduces two new classes of errors: When a *race conditions* occurs, the outcome of the program depends on the specific order in which certain actions in different threads are executed. To avoid race conditions, programmers can use synchronization constructs (e.g., locks or semaphores); that can lead to the other class of error: *deadlocks*, where several threads wait for each other. These errors (especially race conditions) are often not deterministically reproducible, which makes it much harder to reveal them in testing than most bugs in sequential programs, and, even after they have shown up, they are still harder to debug.

If the threads access shared memory (a common feature of multi-threaded environments), another problem is that the hardware does not implement an intuitive memory model like sequential consistency. The programmer can use memory barriers to work around that problem, but memory barriers are expensive, so programmers want to avoid them; but again, if they fail to put in a memory barrier that's necessary, the result will be a race condition, but the problem will be even harder to find and fix, because understanding the bug requires understanding the counter-intuitive behaviour of the hardware.

Common synchronization constructs interact in ways that undermine the modularity of the program. So, the main weapon we use to keep programs manageable loses its edge once we decide to parallelize programs using such synchronization constructs.

Parallel programming is also hard is because the goal is to increase the performance. So the goal is not just to have a program that exposes more parallelism, the goal is that the result should run in less real time than the original sequential program. The problems here are that, on one hand, the speedups are limited by

the number of CPUs (so it really only pays off after you have squeezed all the available sequential performance from the program, which does its own damage).

But even worse, parallelizing a program introduces overheads: thread creation and destruction, synchronization, and barriers. So you cannot just parallelize whatever parts look parallelizable, because then the overheads will tend to eat up the potential speedup and more. Instead, parallelization has to be performed looking at the whole program, and finding out where the large parallelization opportunities are while avoiding overheads. Again, this undermines modularization, and it is hard to achieve in a typical general-purpose program where most of the run-time is spent in a number of places, not just one or a few inner loops.

Another problem is that, while some patterns of parallelism (such as independent loop iterations, aka data or DOALL parallelism) are easy to scale across a wide range of thread counts, these patterns do not tend to dominate in general-purpose applications. So the result of parallelizing will often not be very scalable to varying thread counts. The result will be that either the hardware threads will be underutilized, or that there will be more synchronization overhead than necessary.

Overall, the common ways of supporting thread-level parallelism are either too hard to use for widespread use in application programming (conventional synchronization constructs), or they are often not applicable for general-purpose applications (data parallelism).

2.4 Transactions

In recent years there has been a huge amount of research into transactional memory, especially software transactional memory [LR06]. This provides transactions to the programmers, i.e., a sequence of statements that is (logically) executed atomically. This concept is easier to program than traditional synchronization. However, in this work I look in a different direction, for the following reasons:

- There is already lots of research into transactions, and most of the gold in that area probably has already been mined, whereas other directions have received comparatively little attention.
- Transactions are used in a shared-memory setting, which may not be the best approach to modularity.
- Software transactional memory has a significant implementation cost, in complexity and in performance.

3 Pipeline parallelism

Unix pipelines provide a form of parallelism that avoids many of the problems discussed above:

- It is so easy to use pipelines that shell programmers use them even when parallelism is not the goal.

– Unix pipelines are constructed from reusable modules (called filters). Pipeline parallelism is not just compatible with modularisation, it provides an additional form of modularisation.
– Pipelines can be used in general-purpose applications. E.g., pipelines in Unix are often used for text processing and other non-numerical applications. Compilers used to be organized in multiple filter-like passes. General-purpose applications often contain complex control flow (e.g., a recursive graph walk), which makes it hard to divide work for data parallelism, but pipeline stages can contain such complex control flow without problems.

Therefore, we believe that adding pipeline parallelism to general-purpose programming languages would be a good idea, both for parallelism and for modularization. One other benefit that we will focus on in this paper is that one can implement pipeline parallelism in a way that scales from many threads down to few or one.

3.1 Stream languages

Stream languages were designed for implementing digital signal processing (DSP) algoritms on highly parallel hardware [ADK+04,TLM+04]; pipeline (or stream) parallelism is a central feature of these programming languages. Later, such languages were also implemented on conventional CPUs [GR05,ZLRA07]. These implementations were designed for huge data sets and implemented the filters as working on large (256KB and more) blocks, with the following filter only scheduled on a block after the previous filter has finished it, i.e., no direct communication.

3.2 XJava

The general-purpose language XJava [OPT09] is an extension of Java that supports pipeline and data parallelism. The current implementation uses an approach similar to the implementations of stream languages on general-purpose processors, i.e., with filters working on blocks and then returning to a scheduler which then assigns another ready task to the thread.

The advantages of this implementation approach are: It allows a uniform handling of data parallelism and pipeline parallelism. And, if the pipeline has enough data to process, it scales nicely across a wide number of threads (as long as the program provides enough parallelism); short-running and long-running pipeline stages are balanced automatically.

The downside is that there is a ramp-up in parallelism at the start of a pipeline and a ramp-down in the end; with a large block size, these ramps can take a while, and if there are only few blocks, the parallelism will be severely limited. If all the data fits in one block between stages, all the stages are run sequentially and no parallel execution occurs. While large amounts of data may be taken for granted in DSP applications, that is not necessarily the case for general-purpose applications (e.g., compilers).

4 Fine-grained pipeline implementation

This section presents some ideas on how to implement pipelines in programming languages in a way that does not suffer from these problems.

In order to reduce low parallelism from ramp-up and ramp-down effects, we could reduce the block size. However, this increases the overhead, because the scheduler is invoked once for every block.

Instead, we propose using a single-writer single-reader ring buffer fifo for communicating between pipeline stages. This allows communication between the pipeline stages at a fine granularity: cache line granularity would be good for efficiency in the usual case, but even finer granularity is possible. At the same time, large amounts of data can be transferred without involving a scheduler.

Of course, avoiding the scheduler works only if all pipeline stages run at the same time in different threads, and if each pipeline stage produces data at the same rate as the next stage consumes it. Otherwise, some of the ring buffers will sooner or later become full or empty, and either the producer (if full) or the consumer (if empty) will block and won't fully utilize the thread it has available.

How can this scenario be avoided? And what can we do if we don't have or get enough threads to run all the filters at the same time?

There is another efficient and fine-grained implementation of pipelines that does not need multiple threads or frequent scheduler invocations: coroutining. In coroutining one pipeline stage passes control of the thread directly to the next one (on writing to the next stage) or to the previous one (when reading from the previous stage); this just requires saving a few registers of one task and restoring a few registers of the other task (most notably, the instruction and stack pointers).

Now we can reduce the number of threads needed by our pipeline stages by putting a number of consecutive stages into one thread and letting them communicate with each other through coroutining; the communication with stages in other threads is still through ring buffers.

This arrangement does not do anything for balancing the production and consumption rates of the different sub-pipelines. However, this can be achieved by moving pipeline stages between threads when the ring buffer approaches the full or the empty state.

E.g., consider a ring buffer that approaches emptyness: Obviously the writer to this ring buffer is too slow and the reader too fast. This can be remedied by moving the last pipeline stage (B) of the writing thread T1 to the reading thread T2.

In more detail, this works as follows: the writing stage B (still in T1) tells the reader C that B switches to the thread T2 (which contains C), then it switch the output of the previous stage A to use a ring buffer instead of coroutining, and transfers control of its current thread T1 to A. When C reads from the B-C ring buffer, it finds the switching message from B. First it completely consumes all the data in the B-C ring buffer. When that is empty, it switches its reader to use coroutining from B instead of reading from the B-C ring buffer; so when it needs to read after draining the B-C ring buffer, it will do a coroutine call to

B; at some point B will need to read something, and will read it from the A-B ring buffer.

Since T2 now needs more processing time than before and T1 needs less, the ring buffer between A and B should drain slower than the one between B and C used to, or it might even fill up. If it still drains overall, then A could move to T2, too; if the buffer fills up and approaches fullness, then B can be transferred back to T1. By transfering B back and forth between the threads now and then, both threads can be fully employed. This can all happen without any central scheduler being involved.

This means that the overheads of pipelining are relatively small, so an application can be divided into many pipeline stages without incurring much overhead, even if only a few threads are available. And if even less overhead is desired (at the cost of lower flexibility), several pipeline stages could be merged into one fused stage at compile time.

This approach puts certain demands on the operating system scheduler for good performance: all the threads of the application should run at the same time. Otherwise, i.e., if the OS preempts one thread, that thread will be stuck in some state unknown to the other threads, so other threads cannot pick up pipeline stages from that thread. So the other threads will soon fill or drain all the ring buffers involved, and they will eventually block themselves waiting for the descheduled thread. So, if the OS needs to deal with more runnable threads than the hardware provides, our approach will not lead to good performance.

The good news is that in that scenario the hardware is obviously utilized well anyway; conversely, on a typical PC that is mostly idle, that scenario will not happen frequently, and once the load drops, everything will run smoothly again.

Also, with cooperation from the OS, this scenario can be avoided: The OS should either provide gang scheduling, or should have a way of communicating with the application that it needs a thread for other purposes. Then pipeline stages can migrate away from one thread, and the thread can be returned to the OS while the other threads keep running.

5 Conclusion

Increasing the performance of general-purpose programs by parallelizing them is hard for several reasons: Simple data parallelism is often not present due to dependences, or cannot be exploited profitably due to low loop trip counts and short iteration running times. Using the usual synchronization constructs is complex and undermines the modularity. And a balance has to be found between splitting the application into threads and the overheads of thread creation and synchronization, otherwise performance will decrease rather than increase; and finding that balance again undermines the modularity.

Pipeline parallelism can be used for general-purpose programs, and it enhances modularity by splitting programs into reusable filters. Stream languages and XJava implement pipeline parallelism by letting each filter run on a block of data and then calling the scheduler.

We propose an implementation that is based on using coroutining and single-reader single-writer ring buffers. The ring buffers are used for communicating between threads, and coroutining is used for having several pipeline stages in one thread, to reduce the number of threads (for less capable hardware), or to balance the work between different threads. Pipeline stages can move between threads (and the communication switch between coroutining and ring buffer) to dynamically balance the pipeline so that all threads are always employed.

References

[ADK+04] Jung Ho Ahn, William J. Dally, Brucek Khailany, Ujval J. Kapasi, and Abhishek Das. Evaluating the imagine stream architecture. In *Proc. 31th Ann. Intl Symp. on Computer Architecture (31th ISCA'04)*, pages 14–25, Munich, Germany, 2004.

[GR05] Jayanth Gummaraju and Mendel Rosenblum. Stream programming on general-purpose processors. In *38th Annual International Symposium on Microarchitecture (MICRO-38)*, 2005.

[Lar91] J. R. Larus. Parallelism in numeric and symbolic programs. In Alexandru Nicolau, David Gelernter, Thomas Gross, and David Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, Research Monographs in Parallel and Distributed Programming, pages 331–349. Pitman, London, 1991.

[LR06] James Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[LW92] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *The* 19^{th} *Annual International Symposium on Computer Architecture (ISCA)*, pages 46–57, 1992.

[OPT09] Frank Otto, Victor Pankratius, and Walter F. Tichy. XJava: Exploiting parallelism with object-oriented stream programming. In *Euro-Par '09*, pages 875–886. Springer LNCS 5704, 2009.

[TLM+04] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matthew Frank, Saman P. Amarasinghe, and Anant Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proc. 31th Ann. Intl Symp. on Computer Architecture (31th ISCA'04)*, pages 2–13, 2004.

[ZLRA07] David Zhang, Qiuyuan J. Li, Rodric Rabbah, and Saman Amarasinghe. A lightweight streaming layer for multi-core execution. In *2007 Workshop on Design, Architecture and Simulation of Chip Multi-Processors*, 2007.