Green Thieves in Work Stealing

Yu David Liu
SUNY Binghamton
Binghamton NY 13902, USA
davidL@cs.binghamton.edu

Abstract

This paper proposes an energy-efficient approach for programming languages that support work stealing. The key insight is that thieves and victims in the work stealing algorithm can coordinate their execution paces for more energy efficiency, through dynamic adjustment of CPU frequencies.

1. Introduction

Popularized by Cilk [2], work stealing is a classic approach for achieving load balancing in parallel executions. It is particularly interesting from a programming language perspective because scheduling – a traditionally OS concept – is meshed with program constructs and their compilations. Work stealing has been implemented in several industry-strength languages (*e.g.* Java [5], X10 [1], Intel TBB [3]), and remains active in research in the era of multi-core CPUs.

1.1 Work Stealing Overview

The following Cilk program prepares a CRC-enabled packet given a piece of data:

```
1 cilk char* packet(char* data)
2 {   char* crc, body;
3       crc = spawn computeCRC(data);
4       body = format(data);
5       sync;
6       return strcat(body, crc);
7 }
```

From a programmer's perspective, **spawn/sync** is similar to standard fork-join: **spawn** logically creates a thread, whereas **sync** is a local barrier, waiting for all threads spawned in the same scope to complete. In this example, `computeCRC` and `format` may execute in parallel.

What makes Cilk interesting are some highly stylistic features of its compilation and runtime. First, Cilk adopts Lazy Task Creation [6]: when function `packet` is invoked and the execution reaches L. 3, the newly created thread is *not* going to execute `computeCRC`, but the continuation of L. 3, *i.e.* L. 4-6. This allows the current thread to immediately execute the most "imminent" code the that would have been encountered if the execution were serial, *i.e.* invoking `computeCRC`. The continuation thread is lazily created: the evaluation of the **spawn** expression merely adds a frame to the tail of a *deque* – a per-CPU-core queue-like structure – and a physical thread is not created until another CPU core becomes available. An available CPU core is a potential *thief* and may steal a frame associated with the deques of other CPU cores (a *victim*). The frame being stolen in any deque must start from the head element.

Second, Cilk compiles each "stealable" unit – *e.g.* the `packet` function – into two binaries, a *fast clone* and a *slow clone*. The fast clone can be conceptually viewed as serial code with all **spawn/sync** elided. The only difference is that checkpoints are inserted before each **spawn**, and its execution terminates immediately if the corresponding continuation frame is already stolen, implying the rest of the code is being executed by other CPU cores. A fast clone bears its name because every **sync** is a no-op: when there is no parallelism, there is no need for synchronization. The slow clone on the other hand is full "concurrent" code with proper synchronizations. As predicated, it starts with program counter restoration, jumping to the appropriate program point based on the stolen frame. A Cilk function always starts its execution as a fast clone, but a thief always executes a slow clone.

1.2 Energy Efficiency in Work Stealing

One key difference between fast clones and slow clones – whether synchronization exists – also has consequences on their different energy behaviors. When two threads synchronize, the first thread arriving at the synchronization point needs to wait for the arrival of the second. Operationally, the intuitive notion of "wait" translates to spin locks in Cilk. Also known as busy waiting, spinning is rather inefficient from the perspective of energy: there is no execution throughput directly related to program progress. An alternative solution would be to implement blocking semantics for synchronization: context-switching the first thread and schedule the core with other threads. This route – even though possible – would complicate work stealing, which happens to be about scheduling as well. To make things worse, blocking itself is energy-inefficient too: context switch comes with a cost, and CPU affinity loss may lead to significant cache misses, an indicator strongly correlated with high energy consumption.

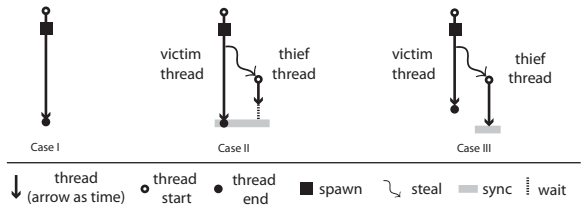
2. Green Thieves

We propose executing the thief thread and the victim thread at different *execution paces*. This is made possible by dynamically adjusting CPU frequencies, a feature supported by virtually all CPUs used today (*a.k.a.* Dynamic Voltage and Frequency Scaling, or DVFS). Multi-core CPUs (*e.g.* Power 7 and AMD Operton), together with semi-customized CMPs, are increasingly equipped with the ability of performing DVFS on a per-core basis.

Specifically, our design is guided by the following general principle:

> **Green Thief Principle**: the thief thread is set to execute at a slower pace than the victim thread.

Let us now demonstrate why this **Principle** may lead to improved energy efficiency. We first consider the case where a thread **spawn**s a frame for continuation thread, which includes a **sync**. Observe that there can only be three cases:



In Case I, a frame is pushed onto the deque upon **spawn**. Before any thief could steal the frame and create a continuation thread however, the thread itself has reached the continuation point. This case is equivalent to a serial execution; no pace change happens.

In Case II, stealing indeed happens. Despite having the thief thread executing at a slower pace, it still reaches the **sync** point before the victim. The thief thread – a slow clone – does need to wait, but the duration of wait – hence useless energy consumption – is reduced compared with the scenario where the thief had chosen to run faster and reached the **sync** point even earlier. In addition, a slower pace also implies that the frequency of the CPU core is lower. In a multi-core context, it is known that energy has a somewhat cubic relation to DVFS scaling [4]. Overall, Case II is energy-efficient both due to shorter wait and DVFS downscaling. Note that the overall execution time of the program is determined by the victim thread, so there is no performance degradation.

In Case III, stealing happens as well and the victim thread successfully completes itself *before* the thief synchronizes over it – perhaps thanks to the slower pace of the thief thread. The termination of the victim implies no wait at its end. The thief does not need to wait as well since the value it needs has been computed by the victim. Had the thief not slowed down, there is a likelihood that it would have reached **sync** before the victim, which would have been less energy efficient due to waiting.

For the general case where thief 1 steals from a victim, thief 2 steals from thief 1, and so on, until thief n finally reaches a **sync** point, note that thieves $1, 2, \ldots, n$ execute on decreasing paces before reaching **sync**. The key fact used by the previous case analysis – the thief with **sync** runs on a slower pace – transitively hold.

2.1 Design Issues

Thanks to the compilation strategy adopted by Cilk, the **Principle** can be directly implemented through compiler-time DVFS instrumentations over fast/slow clones. An ongoing project is implementing a Cilk variant with this **Principle**. We now discuss two design issues.

First, the **Principle** only specifies the *relative* execution paces, with several implementation choices: 1) slowing down the thief; 2) speeding up the victim; 3) a combination of 1 and 2. Scaling factor selection – how much the frequencies should increase/decrease – further enriches the design space. In addition, observe that a naive implementation that simply reduces the relative thief/victim pace has the problem of *pace irrevivability*: the continuations of the program would execute at a lower and lower pace which never comes back up again. In our current design, the execution pace after **sync** should be able to revive to the level before the first **spawn** happens. Overall, the design space is an exploration of the well-known trade-off between performance and energy consumption, a path both analytical and experimental.

Second, the thief and the victim in Cilk conforms to *implicit atomicity*: if the two access shared memory areas, the observable behavior is the same as the two executes serially. Cilk uses locks to enforce this, which are implicit synchronization points and may lead to energy inefficiency. We are addressing this problem by designing: 1) a static analysis that infers the smallest zone demarcated by the first lock and the last unlock; 2) a DVFS instrumentation strategy that allows for the execution the inferred zone *at an elevated execution pace*. The philosophy here is to minimize the possibility of implicit synchronization by quickly running through the zone of contention.

References

[1] CONG, G., KODALI, S., KRISHNAMOORTHY, S., LEA, D., SARASWAT, V., AND WEN, T. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP'08* (2008), pp. 536–545.

[2] FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. The implementation of the cilk-5 multithreaded language. In *PLDI'98* (1998), pp. 212–223.

[3] INTEL. Threading Building Blocks, `http://threadingbuildingblocks.org/`.

[4] ISCI, C., BUYUKTOSUNOGLU, A., CHER, C.-Y., BOSE, P., AND MARTONOSI, M. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *MICRO'39* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 347–358.

[5] LEA, D. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande* (New York, NY, USA, 2000), JAVA '00, ACM, pp. 36–43.

[6] MOHR, E., KRANZ, D. A., AND HALSTEAD, JR., R. H. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM conference on LISP and functional programming* (New York, NY, USA, 1990), LFP '90, ACM, pp. 185–197.