Energy-Efficient Synchronization through Program Patterns

Yu David Liu
Department of Computer Science
SUNY Binghamton
Binghamton, New York 13902, USA
davidl@cs.binghamton.edu

***Abstract*—This paper addresses energy consumption in multi-threaded programs. In particular, it demonstrates why *synchronizations* – a fundamental fabric of multi-core software – may lead to unnecessary energy consumption, and proposes a pattern-based compilation technique to improve energy efficiency. The key insight is that energy efficiency may be improved by adjusting the relative speed of individual threads participating in a synchronization, and different synchronization patterns can offer clues on how adjustments should be made.**

***Keywords*- energy efficiency; multi-core software; synchronization**

I. Introduction

In the recent decade, the impact of multi-core architectures has clearly been felt by computer users from all walks of life. To adapt to the new hardware platform, the software community has been very active in developing novel software techniques to address a wide range of properties of multithreaded programs, such as *correctness*, *programmability*, and *performance*. One grand challenge that has so far received much less attention is *energy consumption*. As the number of cores increases, so is energy consumption. If a multi-threaded program receives a 5x speed-up through a multi-core execution, but at the same time yields a 10x increase of energy consumption (as compared with a single core execution), energy efficiency – energy consumption with regard to performance improvement – degrades as the user embraces multi-core CPUs. This challenge, if unaddressed, may have a severe negative impact on the future of multi-core technologies.

Existing solutions to achieve energy efficiency are mostly from digital circuit, architecture, and OS research communities. What remains much less explored is the *software-centered route*. After all, it is software that drives the hardware that ultimately leads to energy consumption. Energy consumption is the combined effect of software and hardware. This paper is aimed at offering insights over one fundamental research question: How can we save energy of multi-core software *by tapping the knowledge of program structures*?

Specifically, this paper addresses how to make *synchronization* green. Synchronization is a ubiquitous operation in multi-threaded programs running on multi-core platforms. Some synchronization points are inserted for guaranteeing program correctness (*e.g.*, Java synchronized blocks are often used to avoid data races), while others are intended for increased programmability (*e.g.*, in a fork-join framework, the join point helps the main thread ensure the completion of the forked thread and/or query its state). 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 either spinning or blocking [12] of the first thread.

Unfortunately, neither spinning nor blocking is energy-efficient. Spinning consumes energy with no execution throughput directly related to program code, whereas blocking increases overall CPU utilization but comes with the cost of context switch. Context switch usually leads to significant loss in CPU affinity, and the resulting cache misses are known to be one of the most expensive operations in terms of energy consumption [19].

In a recent paper [16], we argued that the energy-efficiency of one key parallelism feature, futures [11], may be improved by allowing the parent and future threads execute at different "paces," so that parent thread – which is dependent on the result of the future thread – "saunters" to the synchronization point whereas the future thread "sprints." In this paper, we extend the narrower focus of that work to consider the general case of synchronization. In particular, we identify 5 common synchronization patterns existing in real-world multithreaded code, and propose a distinct strategy for each category. As it turns out, futures are a particular instance in one of our categories.

The rest of the paper is structured as follows. Sec.II describes our general principles and insights. Sec. III describes each pattern and their impact on energy efficiency. Sec. IV describe several technical issues related to the design choices and implementation.

II. The General Approach

Spinning or blocking consumes energy without contributing to program progress. As energy E is the accumulated effect of power (P) over time (t), minimizing spinning or blocking time helps reduce energy consumption. To see how this can be achieved, consider an idealized case where two threads, say T_1 and T_2, are running toward a synchronization

Pattern	Figure	Strategy	Example Scenarios
(A) Dependent Join		downscale "parent" thread	futures (MultiLisp, Java, C++0x, C#, *etc*) promise pipelining (Argus, E) fork-join async-finish (X10)
(B) Counted Sync		upscale late comers based on counter	Clock (X10) CountDownLatch (Java 1.5) N Semaphores
(C) Declarative Sync		upscale late comers based on program structure	Chord (C#) Exchange (Java 1.5) synchronized SEND/RECV (MPI)
(D) Critical Path		upscale the critical execution; downscale late comers	monitors synchronized blocks (Java, C#) CTRITICAL directive (OpenMP)
(E) Symmetric Join		scheduling dependent	MapReduce loop parallelization (OpenMP, X10, Fortress, *etc*)

Figure 1. Synchronization Patterns and DVFS

point, and they are arriving in 2 seconds and 10 seconds respectively. This means, T_1 eventually would have to wait for T_2 for 8 seconds, either spin its lock repeatedly, or risk being scheduled off (the longer the duration, the higher the risk). Knowing the unfortunate fate, wouldn't it be better if T_1 slows itself down and "saunters" to the synchronization point, or T_2 speeds itself up and "sprints" to the synchronization point? This example leads to a key insight guiding our approach:

Insight 1: Spinning/blocking time – and the resulting energy consumption – can be reduced if participating threads run "cooperatively" in speed.

The "speed" of threads can be dynamically set by scaling up/down the frequency of the host CPU cores. The general technique, dynamic voltage and frequency scaling (DVFS) [4], is widely supported by modern CPUs, especially those on energy-conscious devices such as laptops and mobile phones. Interestingly, DVFS is long known to be an effective energy management strategy itself. Roughly speaking, the majority of CPU's power dissipation P – the rate of energy consumption – can be computed via $P = C * V^2 * F$, where V is the voltage, F is the switching frequency, and C is the capacitance. Due to the innate nature of CPU VLSI design, voltage and frequency are often scaled together. In a multi-core context, it is known that power has a somewhat cubic relation to DVFS scaling [13]. When Insight 1 advises a thread to run at a lower speed, the energy savings not only come from minimized spinning or blocking cost, but also from the slowed-down execution. On the flip side, when a thread is advised to speed up, the energy consumption does increase, but such consumption promotes program progress, and offsets the energy that would have been consumed by thread(s) waiting for it at synchronization point. Overall, it can be observed that

Insight 2: DVFS can be used for thread speed control, and can further contribute in energy management.

The more challenging problem is *which* thread should be scaled up and *which* should be scaled down. Designing principled, precise, and widely applicable procedures – no matter through static analysis, dynamic analysis, or profiling – to predict "T_1 is going to arrive at synchronization point in 2 seconds and T_2 in 10 seconds" is notoriously hard in the general setting. As we shall see, synchronization is in fact supported by different language designs in very different fashions, and used by programmers in diverse ways as well — it might be too hard to design a "master" approach effective for all sychronizations in all programs of all languages. Interestingly, we argue that the vast majority of multi-threaded programs do use synchronizations through a small number of *programming patterns*, and

Insight 3: Different synchronization patterns offer distinct clues on how threads can "cooperate" in speed.

III. Synchronization-Pattern-Based DVFS

We now summarize a number of synchronization patterns recurring in multi-threaded software, and sketch how each pattern can be designed to be more energy-friendly via energy saving solutions such as DVFS. The patterns are summarized in Fig. 1.

Pattern (A): "Dependent Join": In this common pattern, there is a "parent-child" relationship between the synchronizing threads: the parent thread creates the child thread, and the synchronization point often signifies the termination of the child. The traditional Unix-style fork-join idiom falls into this category, so are its numerous variants, such as the async-finish construct in X10 [5] and futures [11]. We propose to execute the "parent" thread at a lower voltage/frequency level than the "child" thread. This design decision takes advantage of one fundamental trait of this style of synchronizations: the child thread dies upon reaching the synchronization point. Since there is no/little spinning or blocking if it arrives first, the algorithm encourages this scenario to happen.

Pattern (B): "Counted Sync": In this pattern, threads wait for each other until an (implicit or explicit) counter indicates all have arrived; from that point, all threads continue. Unlike Pattern (A), none of the synchronizing threads die at the synchronization points (hence no "join"). The often used `CountDownLatch` in Java 1.5 follows this pattern, so is the more generalized scenario where a semaphore initialized with a value N is used. In this scenario, the thread(s) that reaches the synchronization point first must either spin or block. To shorten the spinning and blocking cycle, we believe a counter-based voltage/frequency scaling strategy for incrementally speeding up the synchronizing threads. All threads first start the execution at a low voltage/frequency level, and the level is scaled up every time (or every few times) the counter is decremented. The rationale behind is that the more threads are in the mode for spinning or blocking, the more energy waste there is, and the more urgent it becomes for the late comers to reach the synchronization point.

Pattern (C): "Declarative Sync": In this pattern, synchronization does not happen based on a counting of participating threads, but on the "behavioral matches" of participating threads, usually reflected by the declarative structure of the program itself. As one example, two MPI processes synchronize when one issues `MPI_Send` and the other issues `MPI_Recv` with matching source/destination addresses. As a second example, Java 1.5 introduces an `Exchange` mechanism, so that two threads blocks until each has provided the information the other needs. As a third example, consider C#'s Chord mechanism [2]:

```
1 public class ConditionalPut {
2     public void Put(object o)
3                 & async Good() {
4             ...
5 }}
```

In this example, if a thread sends a synchronous message (*i.e.*, standard C# message) `Put` to the `ConditionalPut` object, the method body at L. 4 cannot be executed until the thread asynchronously receives a message `Good` (likely from a different thread). In C# terms, `Put` and `Good` forms a *chord* for synchronization.

The chord-like mechanism shares some resemblance to Patterns (A) and (B). Clearly, to reduce the energy cost of spinning and blocking for the `Put` thread, the thread sending the `Good` message should speed up. The unique situation here is that, unlike Pattern (A) where the child thread is created by the parent, the thread sending the `Good` message can very well be created before the thread executing `ConditionalPut` is. We propose a "lazy" form of scaling: it is only when the `ConditionalPut` object has received the `Put` message and realized the `Good` message hasn't arrived yet, the `Good` message sender thread(s) will be scaled up.

Pattern (D): "Critical Path": To guarantee correct behaviors of multi-threaded applications, there is often a need for threads to execute a block of code serially. Critical sections in procedural languages and monitors such as those in the form of Java/C# synchronized blocks are often used to achieve this form of synchronization. We propose to scale up the voltage/frequency for executions inside the "critical path." This allows the executing thread to leave the critical path as fast as possible, and in turn minimizes the energy consumption of threads waiting to enter. The strategy can be further refined, such as scaling up proportionally to the number of threads waiting to enter the "critical path."

Pattern (E): "Symmetric Join": This pattern represents the commonly used N-way barrier at the end of a divide-and-conquer algorithm. For example, the MapReduce [6] framework processes a large data set in parallel by dividing the data to multiple mapper threads that are running in parallel, and the mappers are eventually synchronized to aggregate results computed by the mappers. Other common examples include how OpenMP performs loop parallelization, and how X10 [5] and Fortress [1] provide support for parallel processing of array elements.

This synchronization pattern overlaps with Pattern (A) in the sense that the potentially many threads created for divide-and-conquer also finish at the synchronization point, so it is favored that the main thread forking them running at a lower speed. In this scenario however, a strategy like in Pattern (A) is only able to play a non-essential part in energy savings, because the largest part of the energy consumption results from the many divide-and-conquer threads. The more

essential question is how these more "symmetric" threads should be executed to save energy. This happens to overlap with prolific operating system research on energy-efficient scheduling [20]. In this context, one of the difficult questions is to understand the nature of the work load, especially the relative difference between different tasks and load balacing. Pattern (E) offers a very direct answer: all threads are likely to have similar loads.

IV. Discussion

In this section, we discuss a number of design and implementation issues encountered through our initial exploration, an ongoing research project.

A. Incompleteness

The idea described here is inspired by design patterns [8]. Any pattern-based approach is inherently incomplete: "why are there only 5 patterns, not 50?" The answer to this question brings in several unique opportunities and challenges to our project.

First of all, the pattern-based approach here can be viewed as incremental optimization. In software methodologies, what "incompleteness" entails differs drastically in different scenarios – for instance, in language design, an incomplete type system implies some useful programs cannot be programed. Fortunately in our scenario, each pattern is an energy optimization strategy and each optimization can be performed independently. In other words, 50 patterns represent a larger optimization space (and more desirable), but implementing 5 patterns – or even 1 pattern alone – may still present benefits for optimizing energy consumption of multi-threaded software.

Second, an underlying theme of our project is to identify new synchronization patterns and design new ways of exploiting synchronization patterns to achieve energy efficiency. The patterns listed in Fig. 1 are what we find interesting so far, and the exploration is open-ended. The by-product of the project is that it *de facto* offers a categorization of synchronization patterns themselves. With the rise of multi-core architectures and the flourishing of programming models for these platforms, the theme of "design patterns for synchronizations" might be an independent route interesting to the software engineering community.

Third, just as design pattern research has inspired research on anti-patterns [3], it would be interesting to study what "anti-patterns" are for energy efficiency, *i.e.* a group of synchronization code patterns rather energy inefficient by design. The identification of such patterns may help programs identify "power bugs."

B. Pattern Identification

As a first step, we rely on syntactical elements to identify patterns. Each of the patterns introduced in this paper is represented by distinct syntactical elements in languages such as Java, significantly simplifying the identification process. For example, Pattern (A) can be identified by the use of `join` method of the Java `Thread` classes, as well as explicit use of the Java 1.5 `Future` class. Java classes `CountDownLatch` and `Exchange` are direct manifestations of Pattern (B) and Pattern (C) respectively. Synchronized blocks can be used to identify Pattern (D). The use of MapReduce libraries or the `CyclicBarrier` class can be used for identifying Pattern (E). Popular source code search engines [9][14] reveal that the uses of these classes are frequent.

Syntax-based identification has the drawback of being only applicable to programs that use specific classes or syntactical constructs. The open question is whether inference algorithms can be designed to automatically identify patterns in programs where only the most basic multi-threading language constructs are used. For instance, a Java program can obviously exhibit Pattern (E) even though neither MapReduce libraries nor `CyclicBarrier` is used. We speculate at least some patterns can be identified with automated procedures with reasonable effectiveness. For instance, it is common in Java programs that a thread creation point (the instantiation of `Thread` objects or objects of `Runnable` interface, followed by invoking the `start` method) is nested in a loop and the initial values set to the constructor of each thread object is a divide-and-conquer of a data array. Program analyses can be designed to identify this code pattern in Java programs, a candidate for Pattern (E). As another example, what epitomizes Pattern (C) is the data dependency between coordinating threads. On the very high level, this code pattern resembles that of race condition – except that the former is correctly synchronized yet the latter is morbid. Program analyses for detecting race condition are numerous (*e.g.* [18] [7]). Some of their techniques can serve as the basis towards suggesting candidate programs of Pattern (C).

C. Implementation Issues

To demonstrate high-level ideas, the description in Sec. III makes the oversimplified assumption that the synchronization point explicitly demonstrated in those patterns is the first one to encounter for participating threads. An obvious counterexample is that, two threads might implicitly synchronize on some shared memory locations (through locks for instance). The implicit synchronization points need to be identified by static analyses [21]. As of now, we take the simple approach that the pattern-based DVFS approach is only applied when liveness (*i.e.* deadlock freedom) and non-blocking (*i.e.* implicit synchronization freedom) is guaranteed for code fragment whose execution will be dynamically scaled.

DVFS is known to come with a small cost on performance and energy. The conventional wisdom is that the time overhead is within tens of micro-seconds in existing

architectures and the energy overhead is generally ignored. Our experiments show that JNI-enabled DVFS has a higher overhead (one magnitude higher), but it still has a negligible effect considering the length of threads. Indeed, the very use of multi-threading indicates threads as units of logical tasks have a duration magnitudes longer; otherwise, the cost of thread management (creation, scheduling, *etc*) – which could very well go beyond micro-seconds – would have defeated the very purpose of multi-threading.

D. Related Work

To the best of our knowledge, this is the first work illustrating the importance of identifying and taking advantage of synchronization patterns to address energy efficiency of multi-threaded programs. The general problem of minimizing energy consumption in a multiple-process/thread environment through adjusting CPU speed was first addressed on the operating system level ([20], [22]). In their approach, each task is independent from each other, with statically known deadlines, and scheduled in a single-CPU setting. Since then, there has been a large body of work applying DVFS to process/thread scheduling. For instance, scheduling dependent tasks with known task dependency graphs has been thoroughly studied (*e.g.*, [10], [23], [24]). The general philosophy of finding critical paths of task execution and taking advantage of slacks on non-critical paths is a shared theme. Hardware optimizations [17] exist for synchronization-intensive applications. Putting CPU into sleep state [15] has also been proposed for reducing energy consumption of synchronization.

V. Conclusion

In this paper, we demonstrate how energy efficiency of multi-core software can be improved through identifying synchronization patterns and applying pattern-specific DVFS strategies. In the future, we plan to validate the proposed ideas through constructing experiments on real-world multi-threaded benchmarks and measuring their energy consumptions. This includes measuring the cost of energy consumption as the result of spinning or blocking, comparing the relative effectiveness of different patterns, and study the impact of different VM-level and/or OS-level thread scheduling options.

Acknowledgment

We thank the anonymous reviewers for their useful comments. This work is being supported by NSF CAREER Award CCF-1054515 and a Google Faculty Award.

References

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.

[2] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26:769–804, September 2004.

[3] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[4] Thomas D. Burd and Robert W. Brodersen. Design issues for dynamic voltage scaling. In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 9–14, 2000.

[5] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, pages 519–538, 2005.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[7] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *In PLDI'00*, pages 219–232. ACM Press, 2000.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[9] www.grepcode.com.

[10] Flavius Gruian and Krzysztof Kuchcinski. Lenes: task scheduling for low-energy systems using variable supply voltage processors. In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, ASP-DAC '01, pages 449–455, 2001.

[11] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, October 1985.

[12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[13] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *MICRO'39*, pages 347–358, Washington, DC, USA, 2006. IEEE Computer Society.

[14] www.koders.com.

[15] Jian Li, Jos F. Martnez, and Michael C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *HPCA'04*, page 14. IEEE Computer Society, 2004.

[16] Yu David Liu. Variant-frequency semantics for green futures. In *Proceedings of the Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES'12)*, 2012.

[17] Matteo Monchiero, Gianluca Palermo, Cristina Silvano, and Oreste Villa. Power/performance hardware optimization for synchronization intensive applications in mpsocs. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, pages 606–611, 2006.

[18] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15:391–411, November 1997.

[19] Ching-Long Su and Alvin M. Despain. Cache design tradeoffs for power and performance optimization: a case study. In *ISLPED '95: Proceedings of the 1995 international symposium on Low power design*, pages 63–68, 1995.

[20] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 2, Berkeley, CA, USA, 1994. USENIX Association.

[21] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for java. In *OOPSLA '05*, pages 439–453, 2005.

[22] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, pages 374–, 1995.

[23] Yumin Zhang, Xiaobo Sharon Hu, and Danny Z. Chen. Task scheduling and voltage selection for energy minimization. In *Proceedings of the 39th annual Design Automation Conference*, DAC '02, pages 183–188, 2002.

[24] Dakai Zhu, Rami Melhem, and Bruce R. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems. *IEEE Trans. Parallel Distrib. Syst.*, 14(7):686–700, July 2003.