Variant-Frequency Semantics for Green Futures

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

Abstract

This paper describes an operational semantics for futures, with the primary target on energy efficiency. The work in progress is built around an insight that different threads can coordinate by running at different "paces," so that the time for synchronization and the resulting wasteful energy consumption can be reduced. We exploit several inherent characteristics of futures to determine how the paces of involving threads can be coordinated. The semantics is inspired by recent advances in computer architectures, where the frequencies of CPU cores can be adjusted dynamically. The work is a first-step toward a direction where variant frequencies are directly modeled as an essential semantic feature in concurrent programming languages.

1 Introduction

For software developers, adopting multi-core architectures is widely known to be a trade-off. On the benefit side, a programmable task – if written as a multi-threaded program and deployed on multi-core platforms – may potentially yield higher performance compared with a single-threaded implementation. On the cost side however, correct and efficient multi-threaded programming is a complex matter. The vast majority of today's research on multi-core programming and compilation can be viewed as efforts to tip this cost-benefit analysis favorably, improving *quality* of multi-core software:

$$\text{multi-core software quality} = \frac{\text{performance}}{\text{pain and horror of software development and use}}$$

Examples include designing new programming models to ease programming efforts and enforce invariants, new program analyses to find concurrency bugs, or new optimization techniques to further improve performance.

Multi-Core Software Energy Efficiency An additional form of cost – obvious but so far largely under the radar of multi-core programming and compilation research – is the energy consumption of multi-core architectures: the energy consumption of CPUs multiplies when platforms evolve from single-core architectures to multi-core ones due to the inherent nature of digital circuits. In this paper, we call for more research efforts to tip a new flavor of cost-benefit analysis favorably, improving *energy efficiency* of multi-core software:

$$\text{multi-core software energy efficiency} = \frac{\text{performance}}{\text{energy consumption}}$$

To come up with a software-centered solution for energy efficiency, it is important to identify energy inefficiencies as *introduced by software*. Indeed, when we deploy a multi-threaded program on a 20-core machine, we would have tolerated a 20x increase of energy consumption if our program yields 20x speed-up. The culprit that prevents this – according to the now famous Amdahl's law [1] – is really the program (algorithm) itself! The law tells us linear speed-up is impossible on multi-core executions for algorithms with any serial components (which by the way, apply to virtually all practical programs). Performance degrades the most when a parallel execution is stalled due to the need for executing the

program's serial components. To improve energy efficiency, it is thus the most profitable if we focus on minimizing energy consumption for these parallelism-stalling oprations.

On the programming language level, such operations are often realized through synchronization primitives. 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 [6] of the first thread. Unfortunately, neither spinning nor blocking is energy-efficient. Spinning – also known as busy waiting – consumes energy with no execution throughput directly related to program code. Blocking – the strategy that context-switches the first thread so that the CPU core can be occupied by other threads – increases overall CPU utilization but comes with the cost of context switch. This especially takes a toll on energy consumption: context switch usually leads to significant reduction on cache locality; the resulting cache misses are known to be one of the most expensive operations in terms of energy consumption.

Energy-Efficient Futures This paper puts the spotlight on one particular form of synchronization mechanism, futures [5], and argues that several of their distinct traits – if exploited – can potentially improve energy efficiency of multi-core program execution. Our key insight is that, to avoid the useless energy consumption of spinning or blocking, different threads can execute at different "paces," so that the thread likely to arrive early "saunters" to the synchronization point whereas the one likely to arrive late "sprints" to the synchronization point. To achieve the effect of sauntering and sprinting is not hard: modern CPUs are almost invariably equipped with abilities to dynamically adjust frequencies and voltages, a strategy widely known as Dynamic Voltage and Frequency Scaling (DVFS) [2]. The main challenge here is to *determine which thread should saunter, and which thread should sprint*, a question that will be answered in the next section.

2 Green Futures: The General Approach

Futures have long known to be appealing for implicit thread management [5] and program optimization [3]. The idea was popularized in a functional setting (such as MultiLisp and Scheme), and later successfully adopted to object-oriented languages, both as research prototypes [8, 9] and mainstream language extensions (Java and C#). For example, the following pseudo-code demonstrates the use of futures in a Java-like method:

```
1 void procRequest(Socket s) {
2   Buffer in = future readBuf(s);
3   ...
4   int size = in.position();
5   ...
6 }
```

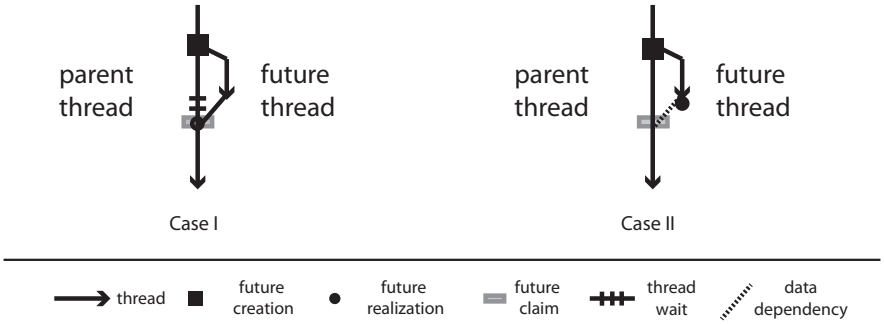
Here, keyword **future** signifies that the invocation to `readBuf` at L. 2 is an asynchronous thread creation (called a *future creation*); the method body of `readBuf` will be executed in a separate *future thread*, and the program counter in the thread executing `procRequest` (the *parent thread*) immediately moves on to the next statement. From this moment on, the two threads will run in parallel, with the future thread serving as a "producer" which eventually fulfills the return value of `readBuf` – called *future realization* (or *fulfillment*) – and the parent thread serving as a "consumer" when the return value of `readBuf` is needed – `in.position()` invocation here at L. 4 – called *future claim* (or *touch*). With the method body of `readBuf` and the omitted statements at L. 3 running in parallel, futures offer

an appealingly simple and incremental way to speed up previously serial code (*i.e.* the one when the keyword **future** is removed).

To improve energy efficiency, we design a variant-frequency execution strategy for the two threads involved in futures. Specifically, we adopt the following general strategy:

> **Main Strategy**: we set the parent thread executing at a *lower* frequency level than the future thread.

This strategy is designed thanks to two distinctive traits of futures. First, futures shapes up a fundamentally asymmetric relationship between two threads: one is a producer, and the other is a consumer; Second, the future thread terminates upon future realization. Let us now demonstrate why the **Main Strategy** is a sensible one. Not to lose generality, observe that there can only be two cases for a future-involved execution:



In Case I, despite having the parent thread executing at a lower frequency, it still reaches the future claim point before the future is realized. In this case, the parent thread does need to spin or block, but observe that the duration of spinning or blocking – hence useless energy consumption – is reduced compared with the scenario where the parent had chosen to run faster and reached the claim point even earlier. In Case II, as a result of the "faster" execution, the future thread successfully fulfills the future before the parent thread claims it. In this case, the future thread has accomplished its mission of being, and can be terminated. No spinning or blocking is needed. When the parent thread finally reaches the claim point, the value (such as `in` in the example) is ready. No spinning or blocking is needed for the parent thread either.

It should be noted that what really matters is the *relative* pace of the parent thread and the future thread, not the absolute one. For instance, instead of slowing down the parent thread, the reasoning above still stands if an implementation chooses to *speed up* the future thread, or slowing down the parent thread and speeding up the future thread at the same time. All variations are likely to improve on overall energy efficiency – in that the likelihood and duration of wasteful wait is reduced – but they might have different effects on performance and overall energy consumption. For instance, if one chooses to (absolutely) slow down the parent thread, Case II above has the potential to lead to performance penalties because the program may run longer as a result of the slower execution of the critical path (the parent thread). On the other hand, if one chooses to (absolutely) speed up the future thread, Case II will not have the aforementioned negative impact on performance. Overall, what this suggests is the same **Main Strategy** above may lead to different implementation choices in DVFS. In Sec. 3, we provide a more precise account of this approach.

According to our preliminary experiments, the overhead of DVFS – usually within tens of microseconds in existing architectures – can be ignored. Indeed, threads usually execute at a duration magnitudes longer, otherwise the cost of thread management would have invalidated their *raison d'ête*.

3 Operational Semantics

In this short presentation, we illustrate our ideas through a mini-language, a multi-threaded addition calculator:

$$
\begin{array}{rcll}
e & ::= & e + e \mid \textbf{future}\ e \mid v & \textit{expressions} \\
v & ::= & i \mid \mathit{fv} & \textit{values} \\
c & ::= & \textbf{cl}(\mathsf{fq}, e)^{\mathit{fv}} \parallel c & \textit{configurations}
\end{array}
$$

Values are either integers or future values (*fv*). A new thread can be created via the **future** e expression, and future claim may happen for the $+$ expression when either of its arguments is a a future value. A parallel configuration is formed by concatenating single-threaded computations together, via commutative $\parallel$. Each single-threaded computation takes the form of $\textbf{cl}(\mathsf{fq}, e)^{\mathit{fv}}$, denoting expression e is currently evaluated on a CPU with frequency fq, for realizing a future value *fv*. In addition, let us define the frequencies supported by each CPU core as a finite well-ordered set $W = \{\mathsf{fq}_1, \ldots, \mathsf{fq}_n\}$ where $\mathsf{fq}_1 < \mathsf{fq}_2 \cdots < \mathsf{fq}_n$ (as in hertz). Since this set is fixed given a hardware environment, the rest of the definitions are implicitly parameterized by W.

The small-step operational semantics is defined by the transitive reduction relation $\Longrightarrow$ over configurations. The reduction rules are defined as follows:

$$
\begin{array}{rrcll}
\text{(R-Create)} & \textbf{cl}(\mathsf{fq}, \mathbf{E}[\textbf{future}\ e])^{\mathit{fv}'} & \Longrightarrow & \textbf{cl}(\uparrow \mathsf{fq}, e)^{\mathit{fv}} \parallel \textbf{cl}(\downarrow \mathsf{fq}, \mathbf{E}[\mathit{fv}])^{\mathit{fv}'} & \text{if} \quad \mathit{fv} \text{ fresh} \\
\text{(R-Claim)} & \textbf{cl}(\mathsf{fq}, \mathbf{C}[\mathit{fv}])^{\mathit{fv}'} \parallel \textbf{cl}(\mathsf{fq}', v)^{\mathit{fv}} & \Longrightarrow & \textbf{cl}(\Uparrow \mathsf{fq}, \mathbf{C}[v])^{\mathit{fv}'} & \\
\text{(R-Add)} & \textbf{cl}(\mathsf{fq}, \mathbf{E}[i + i'])^{\mathit{fv}} & \Longrightarrow & \textbf{cl}(\mathsf{fq}, \mathbf{E}[i''])^{\mathit{fv}} & \text{if} \quad i'' \text{ sum of } i, i' \\
\text{(R-Cxt)} & c \parallel c'' & \Longrightarrow & c' \parallel c'' & \text{if} \quad c \Longrightarrow c'
\end{array}
$$

$$
\begin{array}{rcll}
\mathbf{E} & ::= & \bullet \mid \mathbf{E} + e \mid v + \mathbf{E} & \textit{evaluation context} \\
\mathbf{C} & ::= & \bullet \mid \mathbf{C} + v \mid i + \mathbf{C} & \textit{claim context}
\end{array}
$$

The main novelty here is that the frequency of each thread execution can be explicitly adjusted, through unary operators for upscaling ($\uparrow$ and $\Uparrow$) and downscaling ($\downarrow$). How these operators are defined concretely is the standard problem of scaling factor selection in DVFS. Some design choices, including the difference between $\uparrow$ and $\Uparrow$, will be discussed shortly. (R-Create) and (R-Claim) correspond to future creation and future claim, respectively. At future creation time, the frequency for the "future" thread is scaled up, whereas the 'parent" thread is scaled down. This design choice reflects our main principle of frequency adjustment: the "future" thread should hurry up to realize the future values, whereas the "parent" thread should leisurely proceed. (R-Claim) shows that future claims are fundamentally an operation of synchronization. Here, the blocking semantics is used, whenever a future value needs to be claimed, the reduction cannot progress until the future is realized. After future claim, the frequency of the "parent" thread needs to scale up – intuitively, the reason for the "parent" thread to saunter no longer exists. In addition to the standard evaluation context $\mathbf{E}$, a separate claim context $\mathbf{C}$ is defined, for execution configurations where a future must be realized. Note that the definition here is able to support "futures of futures": it is possible that a future thread realizes its future with another future value – in which case the v metavariable in (R-Claim) is a future value in its own.

For example, if a (somewhat contrived) program $3 + \textbf{future future}\ (3 + 4)$ starts its execution at frequency $\mathsf{fq}_{\texttt{init}}$, the following reduction sequence is possible, where $\mathit{fv}_{\texttt{init}}$ is a trivial future value for the initial configuration, and $\mathit{fv}_1, \mathit{fv}_2$ are fresh:

$$
\begin{array}{rll}
 & \mathbf{cl}(\mathsf{fq}_{\mathtt{init}}, 2 + \mathbf{future}\ (\mathbf{future}\ (3+4)))^{fv_{\mathtt{init}}} & \text{(R-Create)} \\
\Longrightarrow & \mathbf{cl}(\uparrow \mathsf{fq}_{\mathtt{init}}, \mathbf{future}\ (3+4))^{fv_1} \ \parallel\ \mathbf{cl}(\downarrow \mathsf{fq}_{\mathtt{init}}, 2 + fv_1)^{fv_{\mathtt{init}}} & \text{(R-Create), (R-Cxt)} \\
\Longrightarrow & \mathbf{cl}(\uparrow (\uparrow \mathsf{fq}_{\mathtt{init}}), 3+4)^{fv_2} \ \parallel\ \mathbf{cl}(\downarrow (\uparrow \mathsf{fq}_{\mathtt{init}}), fv_2)^{fv_1} \ \parallel\ \mathbf{cl}(\downarrow \mathsf{fq}_{\mathtt{init}}, 2 + fv_1)^{fv_{\mathtt{init}}} & \text{(R-Add), (R-Cxt)} \\
\Longrightarrow & \mathbf{cl}(\uparrow (\uparrow \mathsf{fq}_{\mathtt{init}}), 7)^{fv_2} \ \parallel\ \mathbf{cl}(\downarrow (\uparrow \mathsf{fq}_{\mathtt{init}}), fv_2)^{fv_1} \ \parallel\ \mathbf{cl}(\downarrow \mathsf{fq}_{\mathtt{init}}, 2 + fv_1)^{fv_{\mathtt{init}}} & \text{(R-Claim), (R-Cxt)} \\
\Longrightarrow & \mathbf{cl}(\Uparrow (\downarrow (\uparrow \mathsf{fq}_{\mathtt{init}})), 7)^{fv_1} \ \parallel\ \mathbf{cl}(\downarrow \mathsf{fq}_{\mathtt{init}}, 2 + fv_1)^{fv_{\mathtt{init}}} & \text{(R-Claim), (R-Cxt)} \\
\Longrightarrow & \mathbf{cl}(\Uparrow (\downarrow \mathsf{fq}_{\mathtt{init}}), 2+7)^{fv_{\mathtt{init}}} & \text{(R-Add)} \\
\Longrightarrow & \mathbf{cl}(\Uparrow (\downarrow \mathsf{fq}_{\mathtt{init}}), 9)^{fv_{\mathtt{init}}} &
\end{array}
$$

Scaling Factor Selection One possible way of defining the upscaling/downscaling operators is to apply the standard functions of computing successive and preceding elements over $W = [\mathsf{fq}_1, \ldots, \mathsf{fq}_n]$:

$$
\begin{array}{rcll}
\uparrow \mathsf{fq}_k & \stackrel{\text{def}}{=} & \Uparrow \mathsf{fq}_k \stackrel{\text{def}}{=} \mathsf{fq}_{k+1} & 1 \le k \le n-1 \\
\uparrow \mathsf{fq}_n & \stackrel{\text{def}}{=} & \Uparrow \mathsf{fq}_n \stackrel{\text{def}}{=} \mathsf{fq}_n & \\
 & & \downarrow \mathsf{fq}_k \stackrel{\text{def}}{=} \mathsf{fq}_{k-1} & 2 \le k \le n \\
 & & \downarrow \mathsf{fq}_1 \stackrel{\text{def}}{=} \mathsf{fq}_1 &
\end{array}
$$

Here, the future thread is literally scaled up and the parent is scaled down. If a parent thread is going to create two future threads, the second future thread is going to execute at a lower frequency than the first (because the parent thread itself has been scaled down after the first future creation). Assuming all futures created by the parent thread will be claimed, the parent thread eventually will return to the original frequency. As another strategy, we can adjust the frequency of the parent thread only:

$$
\begin{array}{rcll}
\uparrow \mathsf{fq}_k & \stackrel{\text{def}}{=} & \mathsf{fq}_k & 1 \le k \le n \\
\Uparrow \mathsf{fq}_k & \stackrel{\text{def}}{=} & \mathsf{fq}_{k+1} & 1 \le k \le n-1 \\
\Uparrow \mathsf{fq}_n & \stackrel{\text{def}}{=} & \mathsf{fq}_n & \\
\downarrow \mathsf{fq}_k & \stackrel{\text{def}}{=} & \mathsf{fq}_{k-1} & 2 \le k \le n \\
\downarrow \mathsf{fq}_1 & \stackrel{\text{def}}{=} & \mathsf{fq}_1 &
\end{array}
$$

4 Future Work

This paper describes a work in progress, focusing on the ideas. A full-fledged semantics is under development. In particular, it remains to be seen how future safety [9] interacts with the proposed ideas in an imperative setting. The fact that multiple scaling factor selection strategies exist clearly demonstrates the importance of experimental methods in this project. For each selection strategy, we are interested in exploring its impact on both performance – including both the spinning/blocking time and the overall execution time of the program – and energy consumption, and measuring it in a more rigorous setting *e.g.* through Energy-Delay Product [4] and Energy-Delay Squared Product [7].

This paper demonstrates that a compiler decision on DVFS can be made to improve the energy efficiency of multi-threaded programs without the knowledge of their logical/execution details. Like most optimization problems, the more knowledge one has on the optimization space, the more effective/optimal the solution will be. An interesting direction is to see how the general principle discussed in this paper can be further combined with static/dynamic information of programs to contribute to additional energy efficiency. For example, instead of viewing the described algorithm here as one where all scaling points and scaling factors are entirely fixed at compile time and oblivious of the run-time behaviors, an adaptive algorithm can be designed where the concrete definitions of $\uparrow$, $\Uparrow$, and $\downarrow$ can be adjusted at run time based on run-time environment information and program profiling data.

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

References

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, 1967.

[2] 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, New York, NY, USA, 2000. ACM.

[3] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. In *POPL '95*, pages 209–220, 1995.

[4] R. Gonzalez and M. Horowitz. Energy Dissipation in General Purpose Microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–84, 1996.

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

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

[7] Alain J. Martin, Mika Nyström, and Paul I. Pénzes. *ET2: a metric for time and energy efficiency of computation*, pages 293–315. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[8] Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent proxies for java futures. In *OOPSLA '04*, pages 206–223, 2004.

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