Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU

Edwin Chan Kinshuk Govil Hal Wasserman

Computer Science Division
University of California, Berkeley

December 15, 1994

Abstract

In order to take advantage of the full potential of ubiquitous computing devices, we will need systems which minimize power consumption. Weiser *et al.* and others have suggested that this may be accomplished in part by a CPU which dynamically changes speed and voltage, thereby saving energy by spreading run cycles into idle time. Here we continue this research, using a simulation to compare a number of policies for dynamic speed-setting. Our work clarifies a fundamental power vs. delay tradeoff, as well as the role of prediction and of speed-smoothing in dynamic speed-setting policies. We conclude that success seems to depend more on simple smoothing algorithms than on more sophisticated prediction techniques, but defer to the eventual replication of these results on actual multiple-speed systems.

1 INTRODUCTION

Recent developments in ubiquitous computing make it likely that the future will see a proliferation of cordless computing devices. Clearly it will be advantageous for such devices to minimize power consumption. The top power-consumers in a computer system are the display (68%), the disk (20%), and the CPU (12%) [4]. There is seemingly little which can be done to minimize screen power-consumption, beyond employing a screen-saver and waiting for hardware improvements. Disk power-consumption may be optimized by spinning down the disk whenever it has been inactive for several seconds; [2, 4, 5] have researched this topic.

It is certainly possible to imagine ubiquitous computing devices with neither disks nor conventional displays; and, for such devices, minimizing the power-consumption of the CPU will be particularly critical. Methods for saving CPU power have been suggested by [1, 3, 7]. They point out that it is possible to build CPUs which can run at several different speeds: and voltage may be decreased approximately linearly as speed decreases. A CPU, regarded as a capacitor-based system, satisfies the physical law

$$\text{Energy/sec} \propto \text{Voltage}^2 \cdot \text{speed}.$$

Thus, as voltage and speed are linearly decreased,

$$\text{Energy/task} \propto \text{Voltage}^2 \propto \text{speed}^2.$$

And so it is possible to save overall energy usage by reducing speed.

Therefore, it is advantageous to have a CPU capable of dynamic speed-setting. Such a CPU could well decrease power usage without inconvenience to the user. For example, a CPU might normally respond to a user's command by running at full speed for 0.001 seconds, then waiting idle for the next command; running at one-tenth speed, the CPU could complete the same task in 0.01 seconds, thereby saving energy without generating noticeable delay.

The essential performance factors of a dynamic speed-setting policy are **power-savings** and **delay.** To save power, a CPU would ideally run at a flat, average speeed. But this would result in unacceptable delay; hence a tradeoff between the two factors must be accomplished. The question of how to measure delay is found to be non-trivial, as is the question of how much delay is acceptable. Ideally, we would have specific knowledge about allowable delays for the various processes of a given application; but such information is not currently available.

In seeking to strike an optimal balance between low power-consumption and low delay, an algorithm must consider issues of **prediction** and **smoothing.** Given that there may be pragmatic limits on the frequency with which CPU speed can be changed, a speed-setting policy must predict how busy the CPU will be in the near future. Given this prediction, the policy will then have to make a decision aimed at smoothing speed. For example, if a peak in CPU usage is predicted, the policy might increase speed, but it might also keep speed low, thereby evening out speed at the cost of increasing delay.

Note that the above conceptual distinction between prediction and smoothing is not quite objective. For instance, a speed-setting algorithm which strongly attempts to set a flat, average speed may be thought of in terms of prediction (it always predicts that the near future will be like the average) or in terms of smoothing (it smoothes to the greatest extent possible). Nevertheless, our goal here will be to separate the two functions to some extent, trying to understand the utility (or lack thereof) of several algorithms for prediction and for smoothing.

Weiser *et al.* [7] present just one practical speed-setting policy, PAST. PAST's prediction algorithm is elementary, and its smoothing is somewhat ad hoc. Hypothesizing that more sophisticated prediction methods will allow for substantially improved performance, we here set out to compare the performance of several new policies.

In Section 2, we review the assumptions, measures, and simulation model employed by Weiser *et al.* In Section 3, we indicate how we have altered this model. In Section 4, we present a number of new speed-setting policies. In Section 5, we analyze the performance of these algorithms. Finally, in Section 6, we present our conclusions and suggest avenues for further research.

2 PREVIOUS WORK

Weiser *et al.* evaluate their policies via a simulator running on trace-data. Traces record CPU-usage for a workstation running standard applications; no attempt was made to capture the unique workload (if any) of a ubiquitous computing device.

It is assumed that speed may be set to any real number on range [min_speed, 1], where 1 represents full speed. Weiser *et al.* compile data for min_speed values of 0.2, 0.44, and 0.66 (corresponding to imagined CPUs with full voltage 5.0 V and minimum voltages 1.0 V, 2.2 V, or 3.3 V).

Trace data is first divided into uniform-length **intervals;** for each interval, one computes the percent of time in which the CPU was active (**run_percent**). Figures 1 and 2 give examples of such data for **interval_lengths** 0.01 seconds and 0.05 seconds, respectively. Not surprisingly, at the smaller interval_length the run_percent values are far more bursty.

The Weiser *et al.* policies recompute CPU speed at the start of each interval that contains a process-start or process-stop event; thus speed is not recomputed for intervals in the midst of long runs or long idles. When the speed is not fast enough to complete an interval's work, **excess_cycles** spill over into the next interval.

Each Weiser *et al.* simulation runs a given policy on a given trace ten times, for interval_lengths 0.001, 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds. Total energy consumption is then plotted as a function of interval_length. **Delay_penalty,** a somewhat subjective measure of delay computed by reference to the excess_cycles values, is also recorded.

If an optional flag is set, the simulator attempts to divide idle time into **hard_idle,** which must be left intact, and **soft_idle,** into which run_cycles may allowably be stretched. For example, it is valid to stretch an application's run_cycles into time spent waiting for the user's next command (soft_idle),

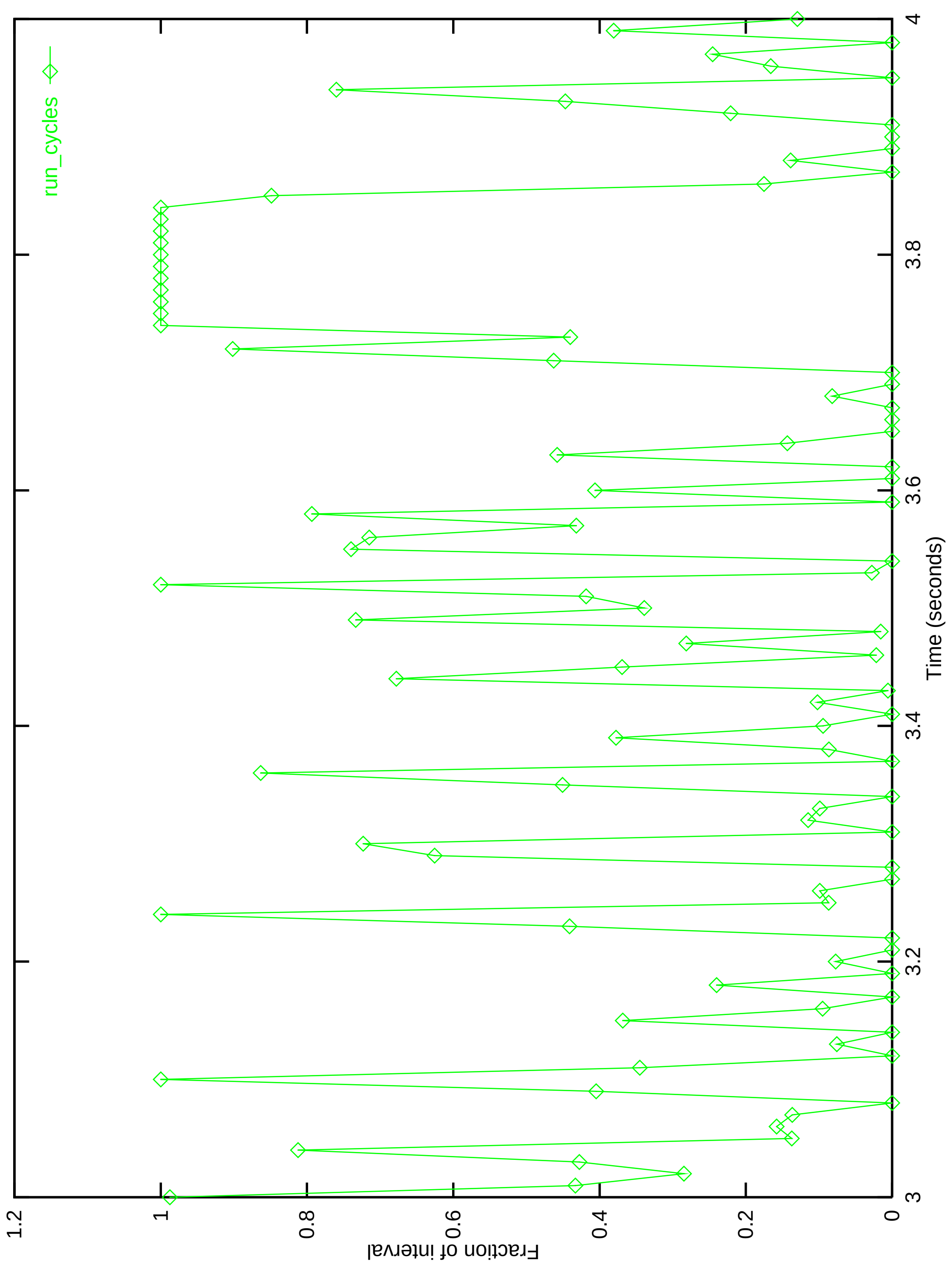

Figure 1: Run_cycles per interval. Trace emacs1, interval_length 0.01 seconds.

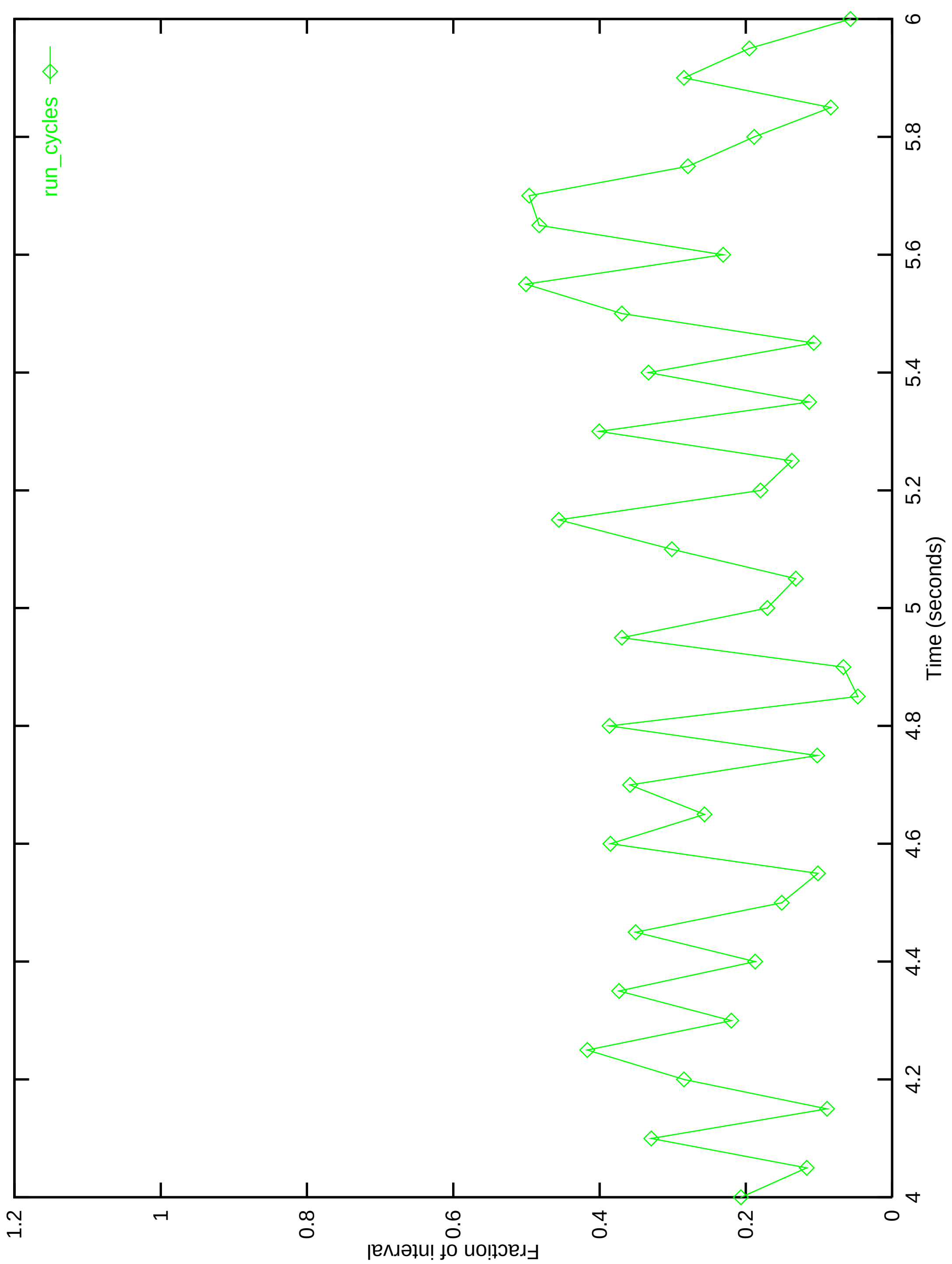

Figure 2: Run_cycles per interval. Trace emacs1, interval_length 0.05 seconds.

but not valid to stretch a process into time which it must spend waiting after requesting data from disk (hard_idle).

The realism of this simulation is somewhat limited: it is not possible to model event reordering due to speed changes or to identify situations in which delays could be invidiously additive. This seems an inherent difficulty of simulating on simple trace data.

3 SIMULATION MODEL

We employed the same traces used by Weiser *et al.* Our simulator is based on theirs, but has been changed in several respects.

• We recompute speed at the beginning of each interval, even if the interval is in the midst of a long run or idle. This could be regarded as less efficient than the Weiser *et al.* method, as its implementation would require additional interrupts. On the other hand, we feel the Weiser *et al.* model to be unrealistic: in particular, it gives its policy premature knowledge that a long run has begun. Moreover, we wished to create and analyze our policies in a way more suited to a uniform, per-interval speed-setting.[1]

• The Weiser *et al.* option of dividing idle into hard and soft seemed degenerate, often failing to identify any significant amount of soft_idle. Some of our simulations are thus run without making use of this mode. However, we later confirm that runs with the hard/soft option yield similar results.

• All of our simulations are for min_speed = 0.2.

• To speed up the simulation, we removed the calculation for the 0.001 second interval_length.

• An error was corrected: due to a programming bug, the Weiser *et al.* simulator was overly optimistic about the amount of work which could be completed in an interval.

• Rather than plotting our results as power vs. interval_length, we plotted power vs. a delay measure. Thus we attempted to focus more clearly on the power vs. delay tradeoff, regarding interval_length as a merely internal parameter.

Theoretically the two plotting methods are not unrelated, as the Weiser *et al.* PAST policy is intended to limit its delay of work essentially to an interval_length. However, as excess_cycles are allowed to spill over into future cycles, true delay if thus an unclear function of interval_length and delay_penalty. The importance of delay_penalty is clearly indicated by the fact that Weiser *et al.*'s algorithm FUTURE, an artificial algorithm which has perfect knowledge of the next interval, still uses *more energy* than PAST

[1]We also used this revised simulation method when running Weiser *et al.*'s policy for purposes of comparison. We intended this as a measure friendly to their policy: when run on our modified simulator, its performance improved somewhat (though only when measured with the revised delay metric described below).

because it is not allowed to push excess_cycles into future intervals.

Instead of trying to combine interval_length and delay_penalty into a meaningful composite number, we substituted our own measure of delay, which is illustrated in Figure 3. For a given CPU task, consider plotting the amount of work that remains to be done as a function of time. The lower line in Figure 3 illustrates what this plot would look like when the CPU is running at full speed; the upper line illustrates the same task being run slowly, belatedly, and intermittently on a variable-speed CPU. We take the area between the two lines as a measure of the task's delay; we then divide the sum of all the "delay areas" in the trace by the sum of all the "full speed" areas to derive a figure for average delay. This measure, while still arbitrary, is rather sophisticated: it fairly represents delay both within and between intervals, it is not unduly sensitive to arbitrary time-slicing of tasks into smaller tasks, and it considers a task which is nearly allowed to complete before a delay (so that it may already have generated most of its child processes) as having a lower delay figure that a task which is delayed at its start.

4 SPEED-SETTING POLICIES

4.1 PAST

PAST is the only practical speed-setting policy presented by Weiser *et al.* We employ it for purposes of comparison.[2]

- PAST:
  - **Prediction:** PAST calculates how busy the previous interval was (including excess_cycles brought into the interval). It then predicts that the coming interval will be identical to this previous interval.
  - **Speed-setting:** If the prediction is for a busy interval, PAST increases speed; if for an idle interval, PAST decreases speed. Some smoothing is accomplished by limiting the amount by which speed can change (except if excess_cycles rises particularly high).

Since PAST makes an effort to complete work within the interval after that which generated it, it is not surprising that delay rises and power-usage falls as the interval_length increases. Weiser *et*

[2]It should be noted, however, that PAST was evidently only intended as a reasonable first-version policy. Moreover, our correction of a simulator bug, as noted in the previous section, may have affected the relative performance of the policy in ways which Weiser *et al.* did not have the opportunity to correct for.

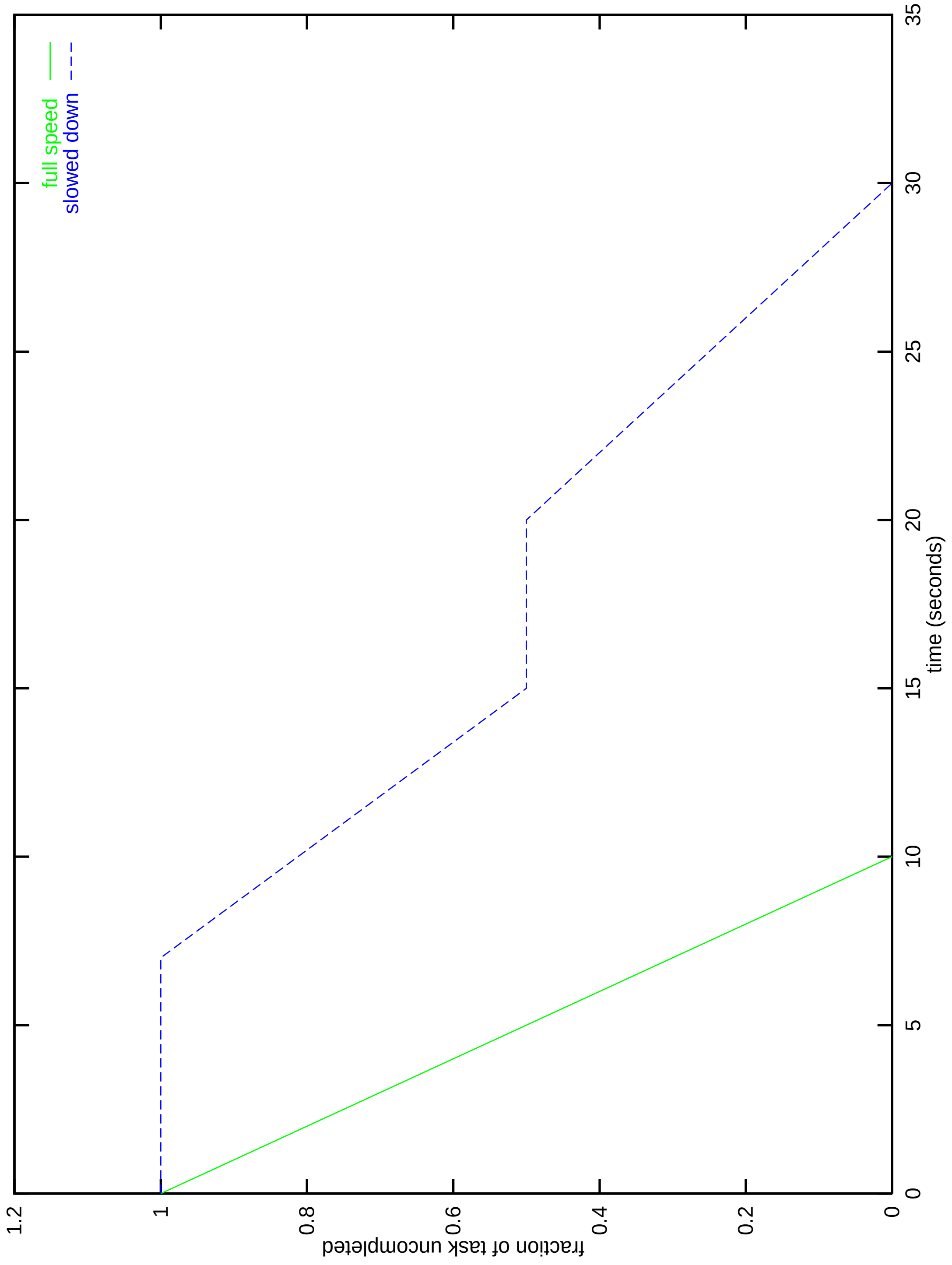

Figure 3: Graph of work still to be done as a function of time for a hypothetical process that normally takes 10 seconds to execute. The bottom line shows the situation when the CPU executes the process at full speed; the top line, the situation when the process is executed at a slower, intermittent speed.

al. identify the range of interval_lengths from 0.01 seconds to 0.05 seconds as one in which delay and energy-savings both seem reasonable. For a simple algorithm, PAST does surprisingly well.

Nevertheless, we felt that there was considerable room for improvement here. We particularly disagree with PAST's prediction algorithm: in a bursty trace such as that illustrated in Figure 1, the assumption that adjacent intervals will be similar is in fact almost certainly wrong. Moreover, it seems non-optimal that PAST only considers the excess_cyles that went into the previous interval, ignoring the seemingly more valuable figure of excess_cycles coming out of that interval into the new interval. Predicting by looking back only one interval seems bad from the point of view of smoothing; and the attempt to patch this problem with an arbitrary limit on speed-change seems ad hoc—and dangerous in that a process can be delayed several intervals while the system slowly banks up speed. The behavior of PAST can be downright strange: given uniform input data, it can thrash speed without coming to a limit; furthermore, due to mistaken speed-setting decisions, it saves little more energy when min_speed is 0.2 than when it is 0.44. Finally, we feel that the role of interval_length is confused, as it influences the outcome of the simulation in three different ways: it determines (1) the frequency with which speed can be corrected, (2) the acceptable amount of delay, and (3) how far into the past PAST looks when making its predictions. We would wish to see a clearer separation of these distinct functionalities.

In short, we felt that, particularly with more sophisticated prediction methods, it should be possible to create a better policy.

4.2 FLAT

Our first policy is FLAT, a simple algorithm which tries to smooth out speed to a global average. FLAT takes an input $\langle const \rangle$, which must be a real number on range [0,1].

- FLAT $\langle const \rangle$:
  - **Prediction:** We predict the new run_percent to be $\langle const \rangle$.
  - **Speed-setting:** Set speed fast enough to complete the predicted new work *plus* the excess_cycles being pushed into the new interval.

While FLAT is strong on smoothing, it also responds effectively to excess_cycles, and so should not generate bad delays. However, it employs no smart prediction techniques.

4.3 LONG_SHORT

LONG_SHORT attempts to outdo FLAT with a more predictive policy, one which attempts to find a golden mean between local behavior and a global average:

- LONG_SHORT $\langle const \rangle$:
  - **Prediction:** The policy maintains two averages of previous run_percents (*including* any excess_cycles added into each interval). One average is short-term (last 3 intervals); the other is long-term (last 12). The predicted new run_cycles is then a weighted sum of the two averages. $\langle const \rangle$ determines this weighting: its use is somewhat arbitrary, but, essentially, higher values give more weight to the local average.
  - **Speed-setting:** Set speed fast enough to complete the predicted work.

Note that LONG_SHORT, an early algorithm, is less elegant than FLAT and is more like PAST (particularly in that it uses excess_cycles only in an indirect way). Our hope was that this algorithm would work best for some ideal value of $\langle const \rangle$ at which it predicted accurately by giving some, but not too much, weight to local behavior. This can alternatively be thought of in terms of smoothing: LONG_SHORT attempts to smooth to a global average, but still shows some respect for local peaks.

4.4 AGED_AVERAGES

A perhaps cleaner variant of LONG_SHORT, AGED_AVERAGES attempts to predict via a weighted average: one which smoothly reduces the weight given to each previous interval as we go back in time.

- AGED_AVERAGES $\langle const \rangle$:
  - **Prediction:** The predicted new run_percent is equal to a weighted average of all previous run_percents, where the weight given to an interval's data is decreased by a factor $\langle const \rangle$ for each 0.01 seconds that we go back into the past.
  - **Speed-setting:** Set speed fast enough to complete the predicted new work plus excess_cycles.

For example, if interval_length is 0.01 seconds, $\langle const \rangle$ equals $\frac{2}{3}$, and the previous run_percents are $R(t-1), R(t-2), \ldots$, then the predicted new run_percent would be

$$\frac{1}{3} \cdot R(t-1) + \frac{2}{9} \cdot R(t-2) + \frac{4}{27} \cdot R(t-3) + \cdots$$

Note that $\langle const \rangle$ is defined so that aging will be essentially independent of interval_length; this we regard as one step toward reducing the confusing multiple effects of the interval_length figure.

We hoped that AGED_AVERAGES, like LONG_SHORT, would work best for an ideal value of $\langle const \rangle$, at which its aged averages would optimally balance the long-term and the short-term past.

4.5 CYCLE

We now experiment with more sophisticated prediction algorithms. The CYCLE policy was inspired by run_percent plots such as Figure 2. Observe that these run_percent values look quite cyclical. Can we take advantage of such cycling to predict?

- CYCLE $\langle const \rangle$:
  - **Prediction:** Examine the last 16 values of run_percent. Does there exist X such that the last $2X$ values seem to approximately repeat a cycle of length X? If so, predict by extending this cycle. If no good cycle is found, just predict the new run_percent to be a flat $\langle const \rangle$.
  - **Speed-setting:** Set speed fast enough to complete the predicted new work plus excess_cycles.

Observe that CYCLE behaves like FLAT except that at times it prefers to make a "smarter" guess by reference to a discovered cycle.

4.6 PATTERN

CYCLE is generalized somewhat in PATTERN. Here we divide run_percent values into four possible "magnitudes": 0 to 0.25, 0.25 to 0.5, 0.5 to 0.75, and 0.75 to 1. We may then identify the last $\langle const \rangle$ run_percents with one of $4^{\langle const \rangle}$ possible patterns of successive magnitudes. Can we then match with a previous occurrence of the same pattern to predict?

- PATTERN $\langle const \rangle$:
  - **Prediction:** Find the most recent sequence of $\langle const \rangle$ previous run_percents that matches the last $\langle const \rangle$ intervals. Predict that the coming run_percent will have the same magnitude as that which followed the matching previous intervals.
  - **Speed-setting:** Set speed fast enough to complete the predicted new work plus excess_cycles.

This model of pattern-discovery is evidently somewhat partial and arbitrary. Nevertheless, we hoped that, when $\langle const \rangle$ was set optimally, repeated patterns—e.g., repeated peaks of a certain common width—would be picked out to good effect.

4.7 PEAK

PEAK is a more specialized version of PATTERN. It looks specifically for narrow peaks, such as those which occur frequently in Figure 1.

- PEAK ⟨*const*⟩:
  - **Prediction:** The policy uses several heuristics based on the expectation of narrow peaks. For example, if the run_percent is falling, it is expected to fall farther; when it is high, it is expected to pass its peak and fall somewhat; when it is low and flat, it is expected to stay low and flat.
  - **Speed-setting:** Set speed fast enough to complete the expected new work plus a ⟨*const*⟩ fraction of the excess_cycles.

Observe that we have modified our speed-setting policy of always trying to complete all excess_cycles; this policy was eminently logical but perhaps too cautious.

5 PERFORMANCE OF OUR POLICIES

In Section 5.1, we run each of our policies in turn, comparing the results to those of PAST and finding the optimal value of each policy's constant. All these runs are done on trace emacs1, a relatively short trace of text being typed into an emacs buffer. In Section 5.2, we then compare the various algorithms, double-check with runs on a substantially different trace, and draw conclusions.

5.1 Runs of each policy

Figure 4 shows the performance of FLAT running with several possible values of its constant; PAST is also provided for comparison. Observe that, for each policy, results are presented for a selection of nine interval_lengths; usually these form a curve, slanting off toward more delay and less energy as interval_length increases. The optimal algorithm that works within a given delay limit is found by starting on the x-axis at the desired delay figure and then moving vertically until one reaches the lowest curve. Thus policies whose results curve closer to the origin are superior, while sets of data points that seem "shifted" along a single curve represent different ranges of possible energy-usage and delay but a similar energy vs. delay tradeoff.

FLAT clearly outdoes PAST.[3] We also note that FLAT seems to achieve optimality around a ⟨*const*⟩

[3]This is not an artifact of our new delay measure. If one uses Weiser *et al.*'s delay_penalty figure instead, the difference is only more pronounced.

value of 0.4.

Figure 5 describes the performance of LONG_SHORT. This algorithm is shifted toward lower energy usage and higher delay. Again it outdoes Weiser *et al.* easily. Results improve as $\langle const \rangle$ increases, leveling out around $\langle const \rangle = 3$ to 5. This seems to indicate that paying attention to the local average is particularly advantageous.

Figure 6 describes the performance of AGED_AVERAGES. We consider this result a disappointment, since, rather than indicating an optimal value of $\langle const \rangle$, the plot demonstrates that the higher $\langle const \rangle$ is, the better. Thus AGED_AVERAGES works best with $\langle const \rangle = 1$, in which case it simply predicts by calculating the unweighted average of all run_percents so far. Thus, contrary to our hypothesis, it seems here that the best one can do is to predict via the global average, in which case AGED_AVERAGES is little different from FLAT.

Figure 7 describes the performance of CYCLE. Results seem lackluster; $\langle const \rangle$ values around 0.5 are optimal.

Figure 8 describes the performance of PATTERN. The results here are particularly disappointing: there is no significant change as $\langle const \rangle$ varies, suggesting the meaningful patterns are not being found.

Figure 9 describes the performance of PEAK. Results seem strong, with PEAK achieving optimality at a $\langle const \rangle$ value of 0.2. Note, however, that PEAK at 0.2 performs little better than PEAK at 1.0. This indicates that our experiment of being lazier about the completion of excess_cycles has had little effect. Thus, if PEAK proves to be a strong policy, we should attribute this primarily to the prediction algorithm rather than to the experimental smoothing.

5.2 Comparison of the policies

Figure 10 summarizes the performance of the best policies from Figures 4 through 9. Surprisingly, the simplest policy, FLAT 0.4, is optimal for the delay values below 8, while LONG_SHORT 3, which is scarcely more complex, is optimal for the higher delay values. Of our more sophisticated predicting algorithms, only PEAK 0.2 comes close to equaling FLAT and LONG_SHORT at medium delay. AGED_AVERAGES, CYCLE, and PATTERN all have disappointing performance. It is particularly telling that CYCLE is consistently worse than FLAT. For CYCLE imitates FLAT except when it is "trying to be clever"; and so this result would suggest that when CYCLE tries to be clever, the result is generally for the worst.

To indicate that the above data is not specific to emacs1, we have duplicated the runs in Figure 10 on a quite different trace, kestrel.mar1; nearly ten hours long, this trace is on a workload including

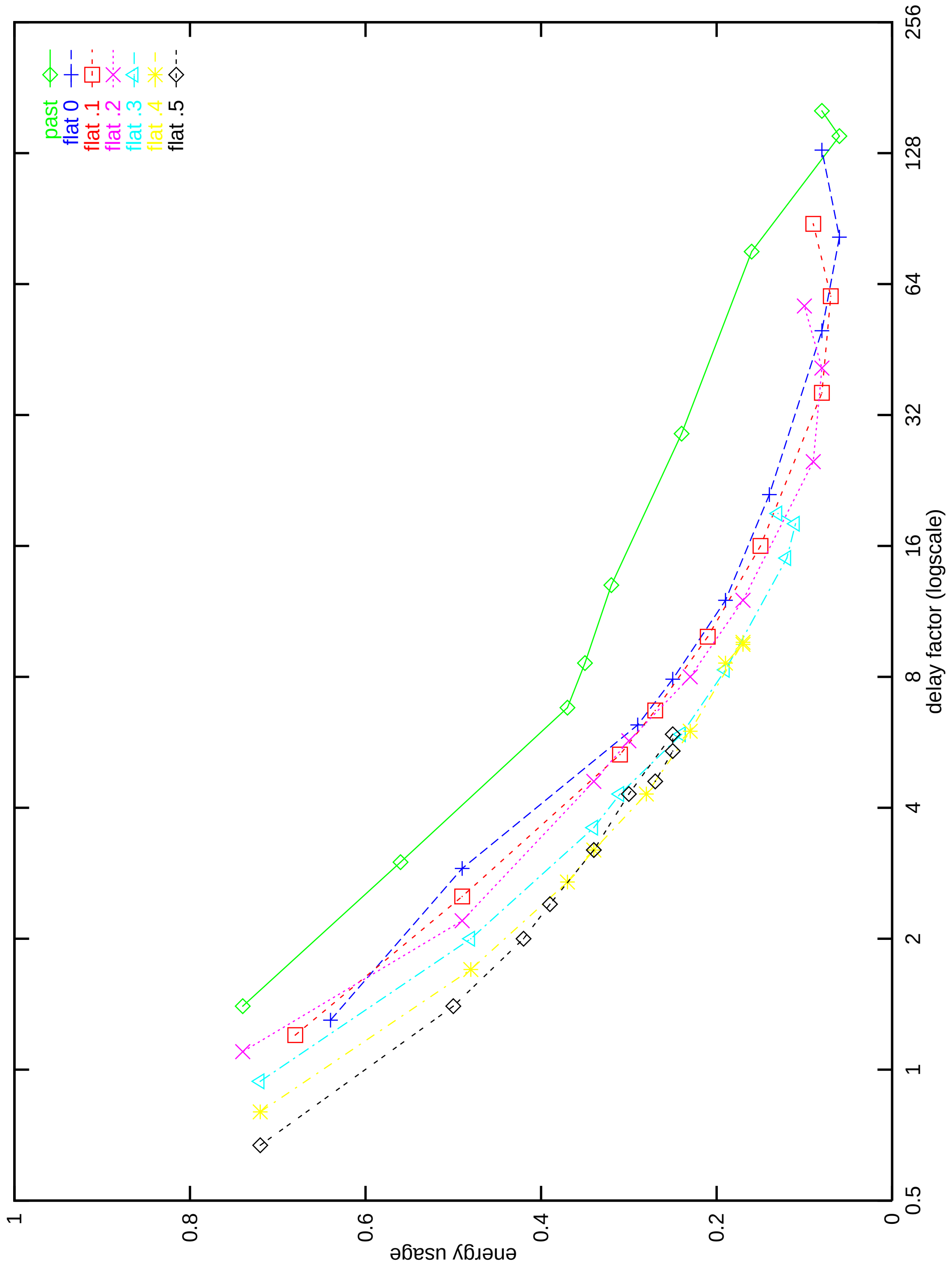

Figure 4: Performance of policy FLAT on trace emacs1. FLAT, run with several possible values of its input constant, is compared to PAST. For each policy, interval lengths of 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds are displayed connected.

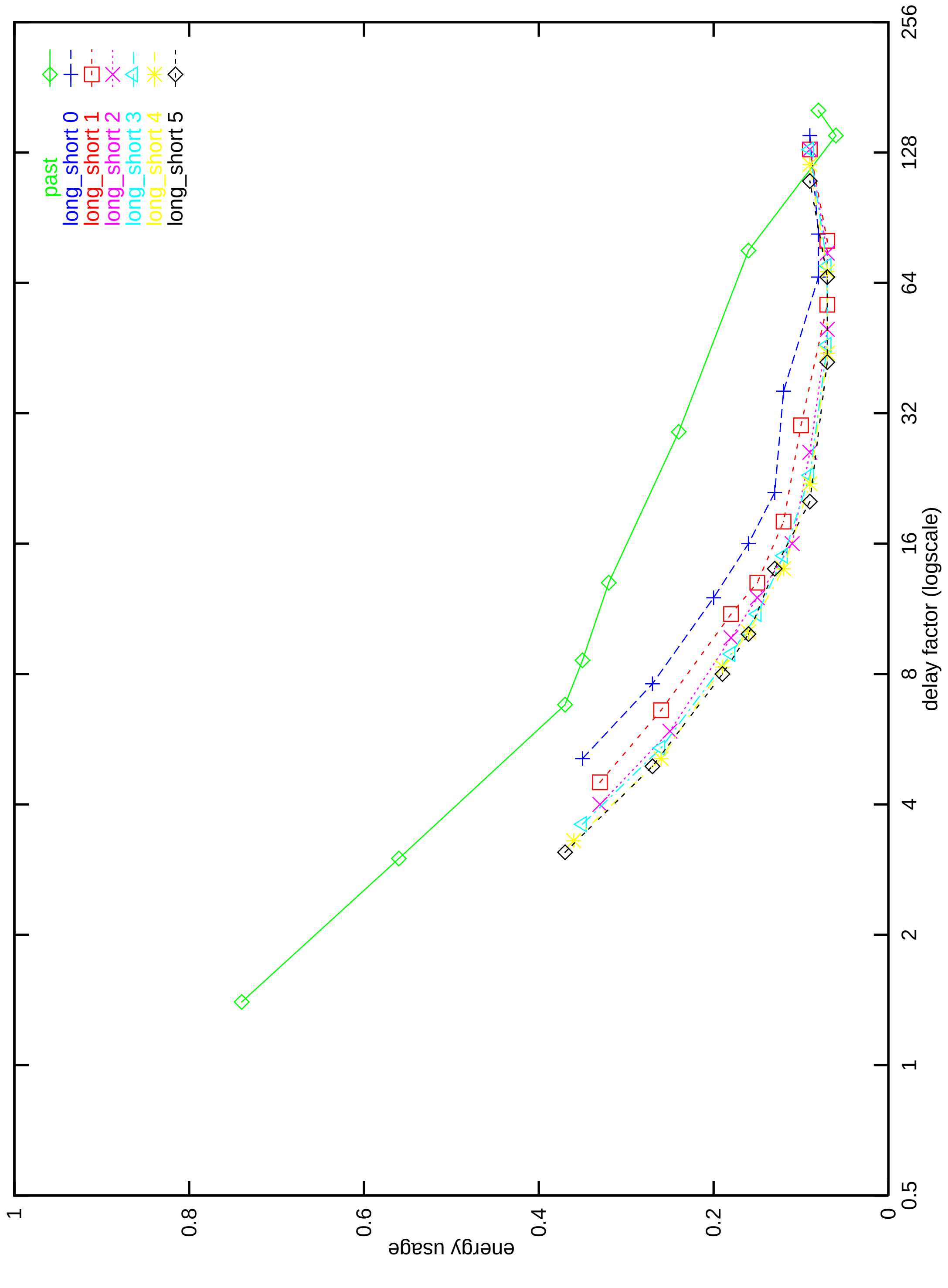

Figure 5: Performance of policy LONG_SHORT on trace emacs1. LONG_SHORT, run with several possible values of its input constant, is compared to PAST. For each policy, interval lengths of 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds are displayed connected.

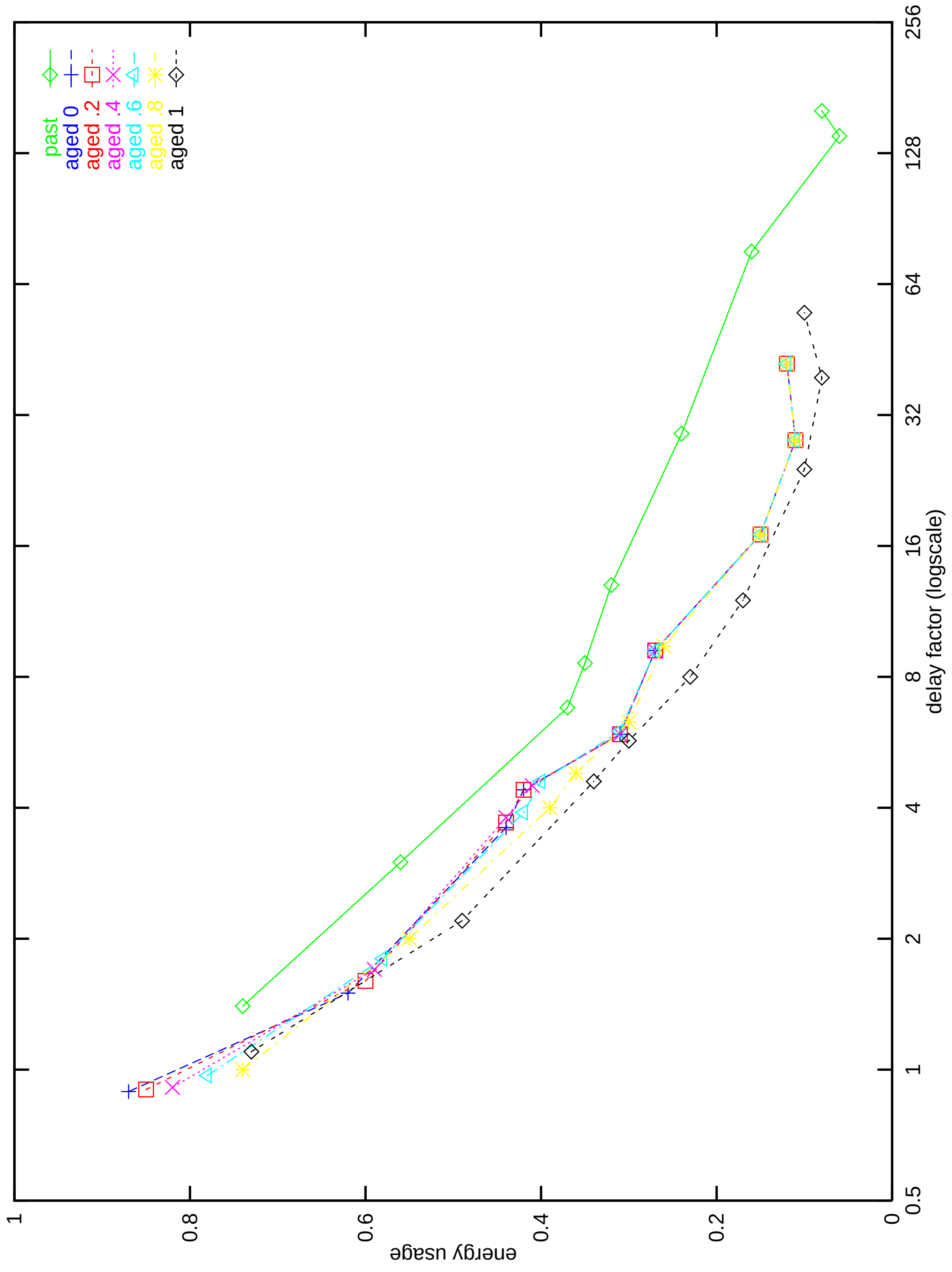

Figure 6: Performance of policy AGED_AVERAGES on trace emacs1. AGED_AVERAGES, run with several possible values of its input constant, is compared to PAST. For each policy, interval lengths of 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds are displayed connected.

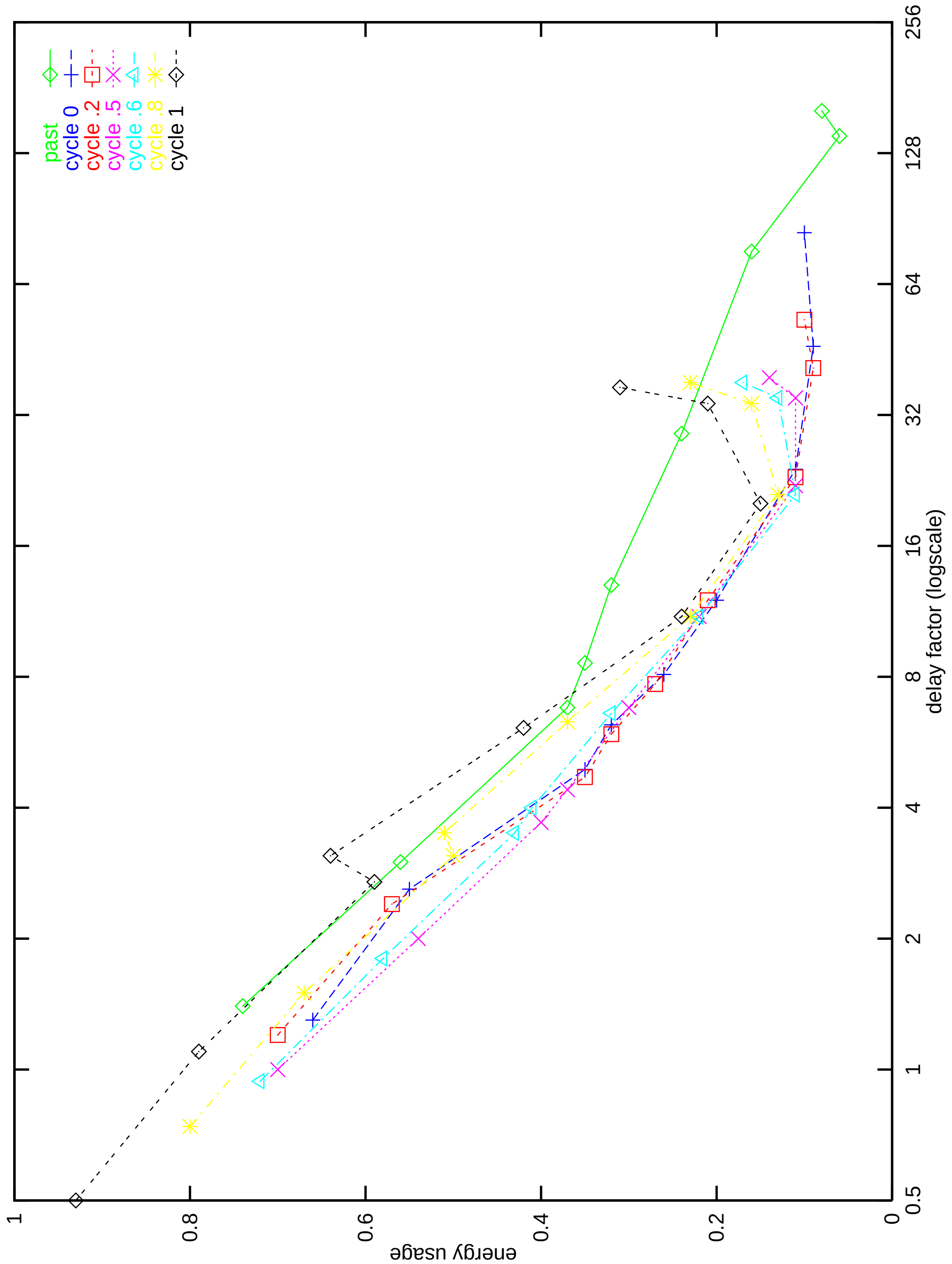

Figure 7: Performance of policy CYCLE on trace emacs1. CYCLE, run with several possible values of its input constant, is compared to PAST. For each policy, interval lengths of 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds are displayed connected.

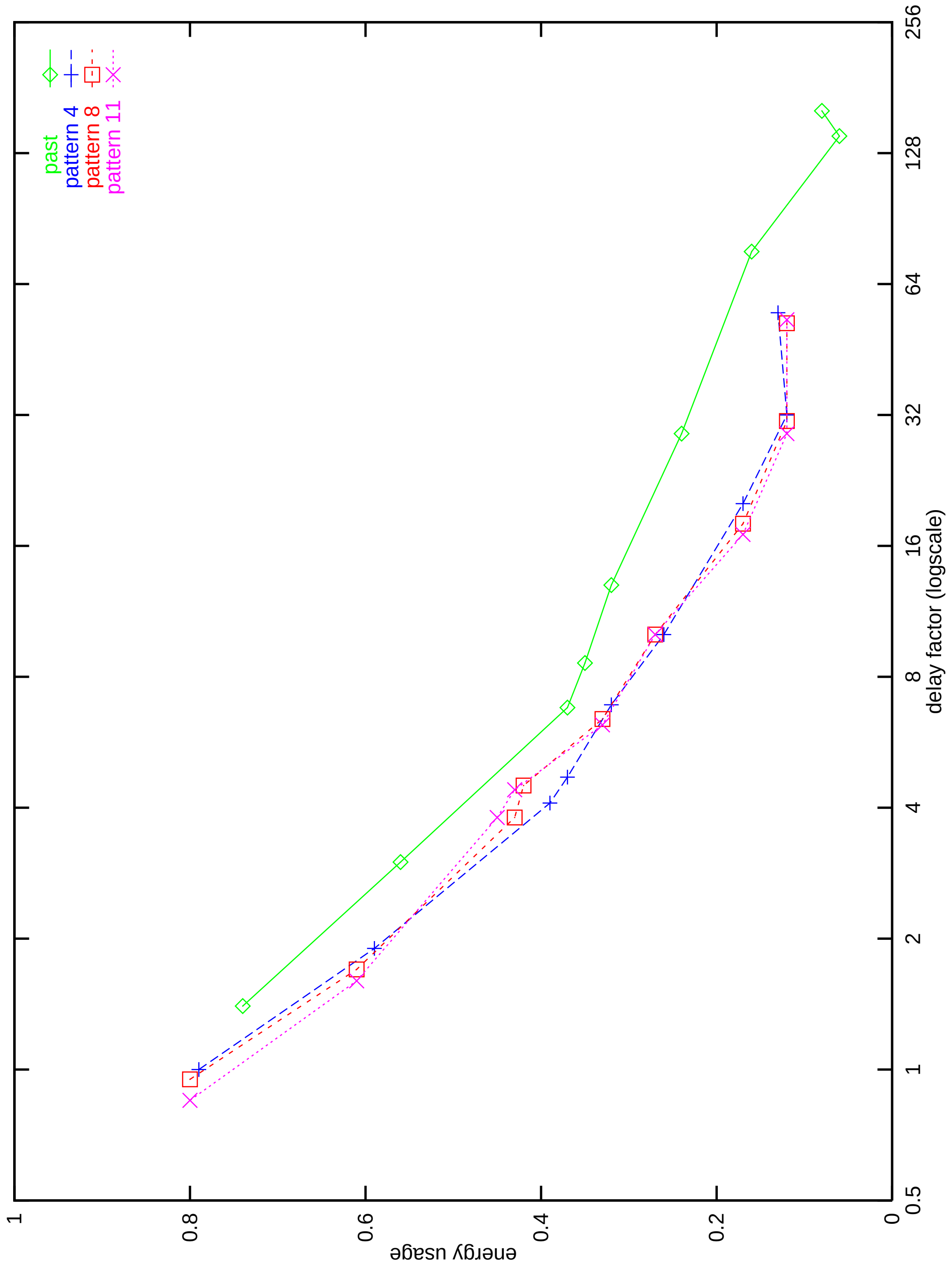

Figure 8: Performance of policy PATTERN on trace emacs1. PATTERN, run with several possible values of its input constant, is compared to PAST. For each policy, interval lengths of 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds are displayed connected.

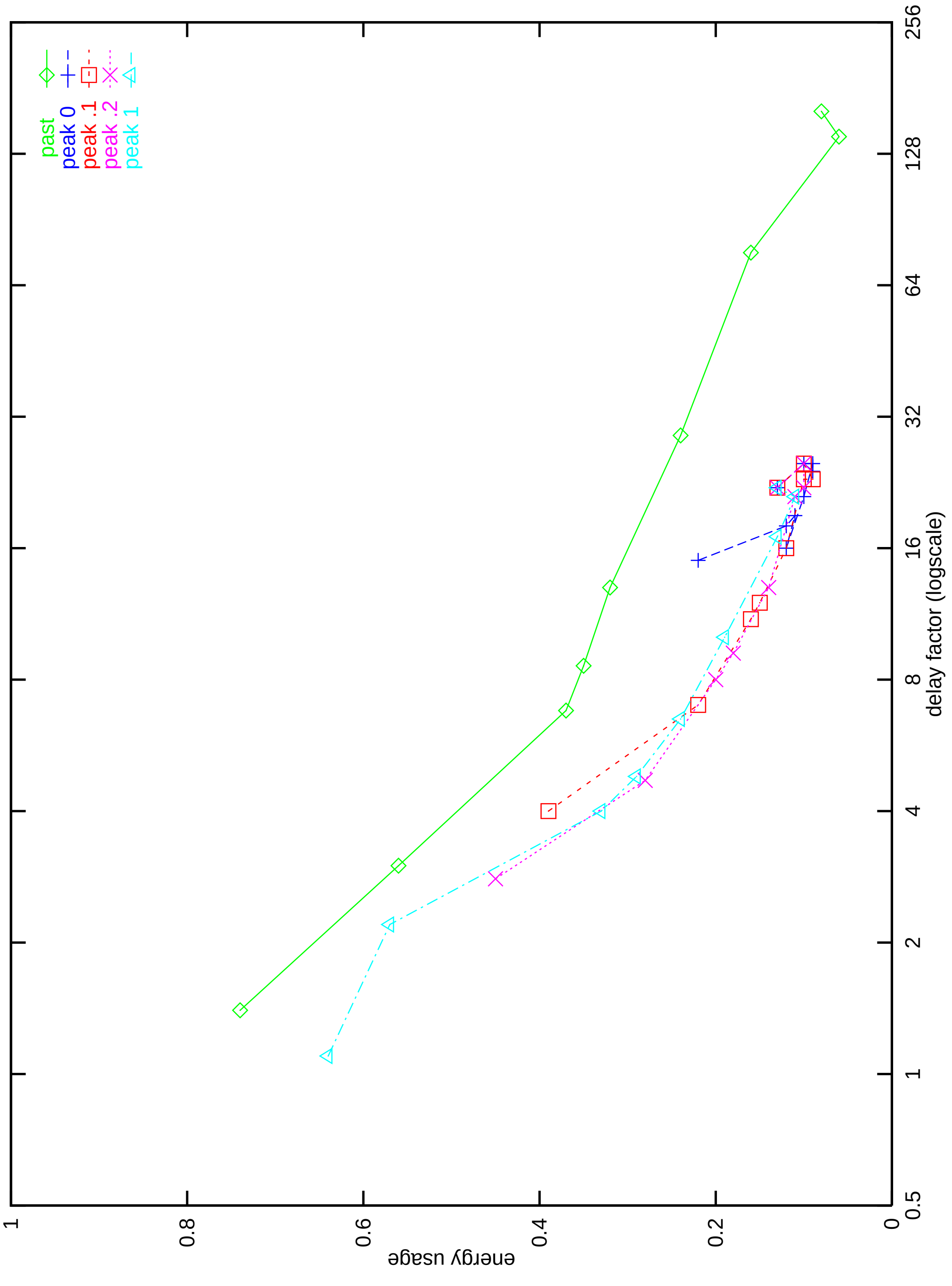

Figure 9: Performance of policy PEAK on trace emacs1. PEAK, run with several possible values of its input constant, is compared to PAST. For each policy, interval lengths of 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds are displayed connected.

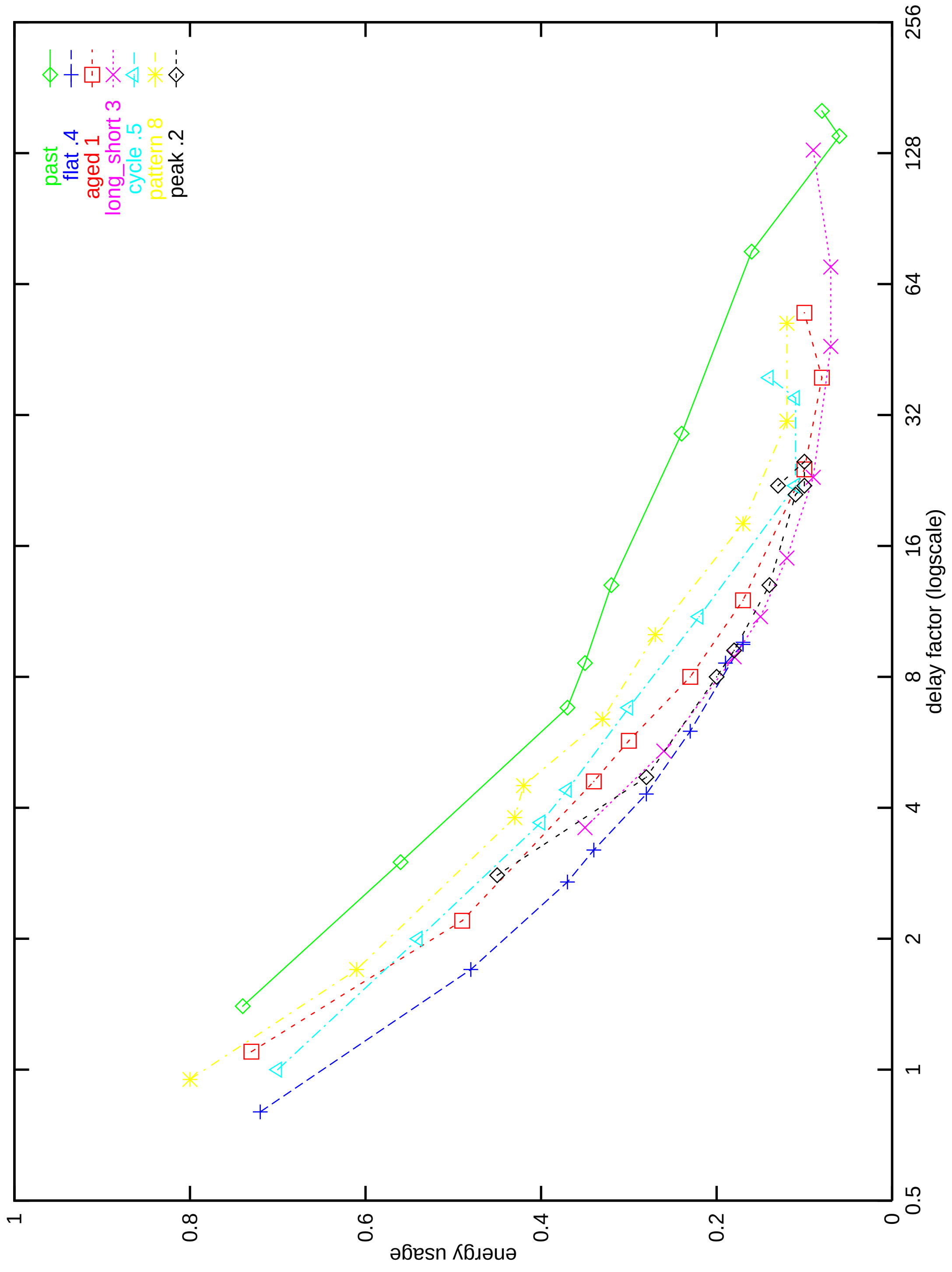

Figure 10: Performance of various policies on trace emacs1. The best policies from Figures 4 through 9 are here compared to PAST. For each policy, interval lengths of 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds are displayed connected.

"software development, documentation, e-mail, simulation, and other typical activities of engineering workstations" [7]. The results are shown in Figure 11. Since the simulator was capable of identifying much of the idle time in kestrel.mar1 as soft, we were also able to run these traces with run_cycles stretched only into soft_idle. Also note that delay factors are unusually small for this trace, apparently because a long block of full-run intervals, which our algorithms handled near-optimally, dominated the delay measure.

In spite of these substantial differences, comparative results for the various algorithms are quite similar to those on emacs1. The main difference is that PEAK 0.2 has edged ahead of FLAT 0.4 and LONG_SHORT 3 to become the optimal algorithm for medium delay values. It is notable that PEAK, having been designed to work well with thin peaks, proves particularly effective for small interval_lengths, at which such bursty peaks are common. Figure 12 illustrates the superior behavior of PEAK 0.2 relative to PAST by tracking the speeds they respectively set on a stretch of kestrel.mar1 for the 0.005 second interval_length. These speeds are graphed along with the effective run_percent—that is, run_cycles divided by (run_cycles + soft_idle). Note the comparative smoothness and the greater correspondence to run_percent of the PEAK speeds.

6 CONCLUSIONS AND DIRECTIONS FOR FUTURE RESEARCH

We found that several of our predictive algorithms performed poorly; only PEAK exhibited strong performance. We might then conclude that simple algorithms which place their emphasis on rational smoothing rather than "smart" predicting may be most useful after all.

Nevertheless, further possibilities for prediction remain to be tried. A policy might divide past information per process and use its knowledge of the expected run-loads of various types of processes to deepen its understanding of the system's current and near-future computational needs. Moreover, each application could provide the system with useful information—both about how much it expects to be loading the system and about how much delay of a given process it would regard as acceptable. Indeed, communication of straightforward deadlines to the system (a keystroke must be processed in 0.01 seconds, and so on) would be an obvious component of an optimal speed-setting policy.

Testing out such theories, however, would quickly go beyond the limits of a simulation. Finally, the hypothesis that a computer's speed may be dynamically changed without inconveniencing the user must be tested on a real system, so that the user-level effects of CPU speed-changes may be unambiguously observed. As multiple-speed CPUs seem consistent with the technological capabilities of the near future, the day when this research may be further advanced is to be expected shortly.

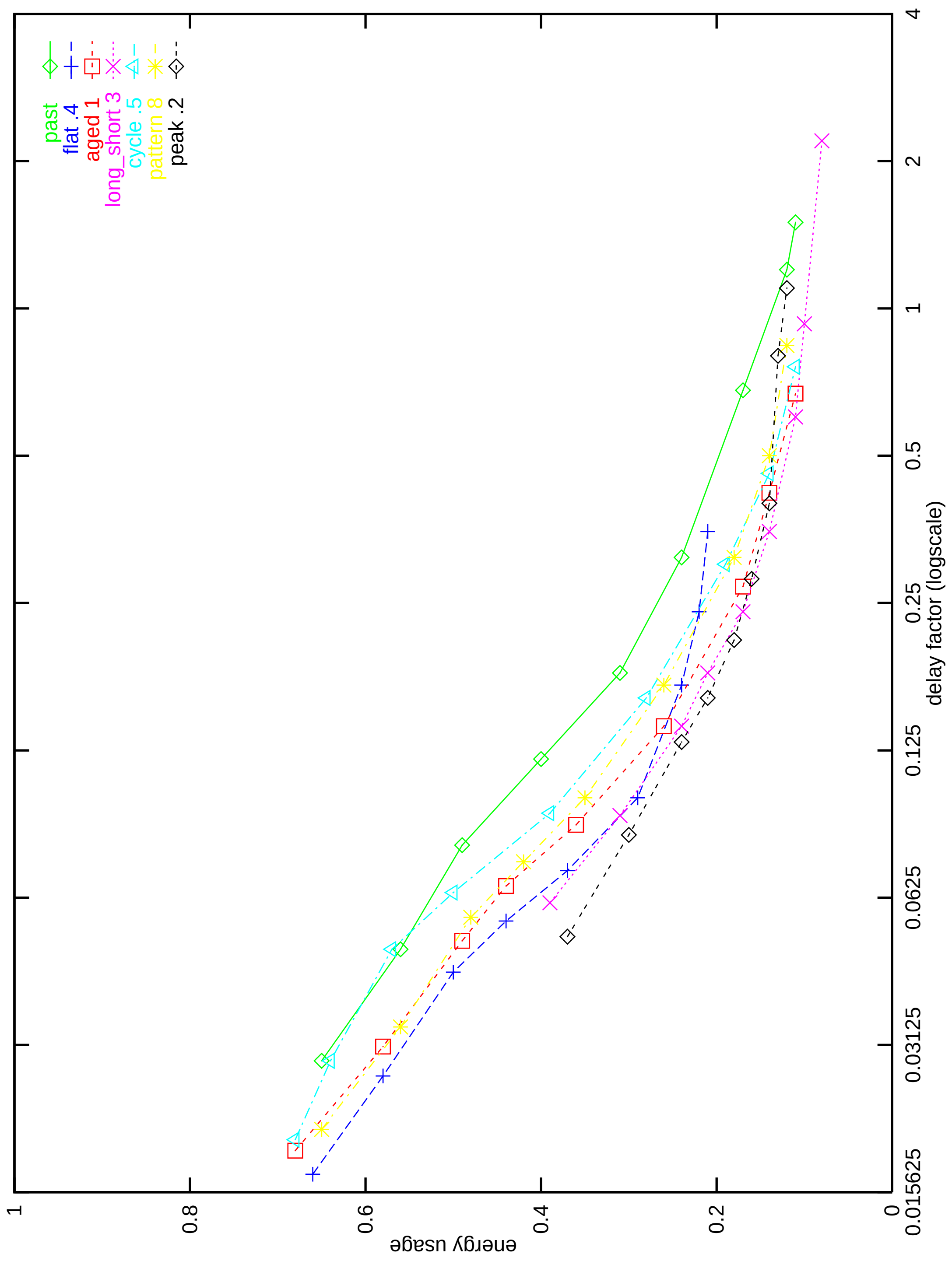

Figure 11: Performance of various policies on trace kestrel.mar1. The best policies from Figures 4 through 9 are here compared to PAST. For each policy, interval lengths of 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.25, 0.5, and 1.0 seconds are displayed connected.

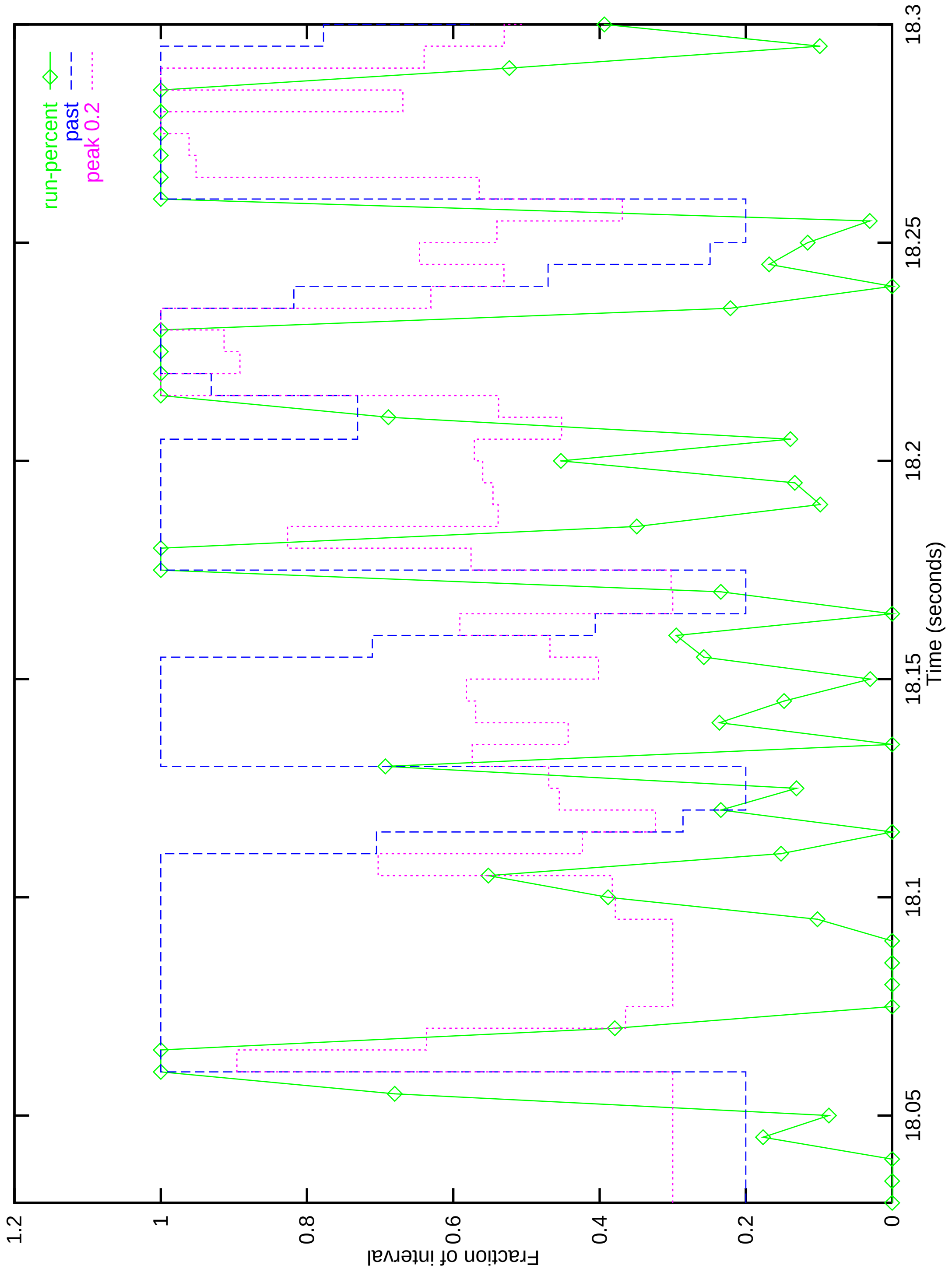

Figure 12: A stretch of the kestrel.mar1 trace, with effective run_percent (= run_cycles / (run_cycles + soft_idle)) graphed alongside the resulting speeds set by PEAK 0.2 and by PAST. Interval_length is 0.005 seconds.

We thank Mark Weiser, Alan Demers, Scott Shenker, and particularly Brent Welch, without whose help this project would not have been possible.

References

[1] A. P. Chandrakasan, S. Sheng, & R. W. Brodersen, "Low-power CMOS digital design," *IEEE Journal of Solid-State Circuits,* Vol. 27, pp. 473–484, April 1992.

[2] Fred Douglis, P. Krishnan, & Brian Marsh, "Thwarting the power-hungry disk," *Proc. Winter 1994 USENIX Conference,* pp. 293–306, January 1994.

[3] Mark A. Horowitz, "Self-clocked structures for low power systems," ARPA semi-annual report, Computer Science Laboratory, Stanford University, December 1993.

[4] Kester Li, Roger Kumpf, Paul Horton, & Thomas Anderson, "A quantitative analysis of disk drive power management in portable computers," *Proc. Winter 1994 USENIX Conference,* pp. 279–292, January 1994.

[5] Kester Li, "Towards a low power file system," CS Tech Report 94-814, University of California, Berkeley, May 1994.

[6] Mark Weiser, "Some computer science issues in ubiquitous computing," *Communications of the ACM,* Vol. 36, pp. 74–83, July 1993.

[7] Mark Weiser, Brent Welch, Alan Demers, & Scott Shenker, "Scheduling for reduced CPU energy," manuscript, Xerox PARC, 1994.