POSE: A Mathematical and Visual Modelling Tool to Guide Energy Aware Code Optimisation

Stephen Roberts∗, Steven Wright∗, David Lecomber†, Christopher January†,
Jonathan Byrd†, Xavier Oró† and Stephen Jarvis∗
∗The University of Warwick, U.K.
{si.roberts, steven.wright, s.a.jarvis}@warwick.ac.uk
†Allinea Software Ltd., U.K.
{david,cjanuary,jbyrd}@allinea.com

Abstract—Performance engineers are beginning to explore software-level optimisation as a means to reduce the energy consumed when running their codes. This paper presents POSE, a mathematical and visual modelling tool which highlights the relationship between runtime and power consumption. POSE allows developers to assess whether power optimisation is worth pursuing for their codes.

We demonstrate POSE by studying the power optimisation characteristics of applications from the Mantevo and Rodinia benchmark suites. We show that LavaMD has the most scope for CPU power optimisation, with improvements in Energy Delay Squared Product ($ED^2P$) of up to 30.59%. Conversely, MiniMD offers the least scope, with improvements to the same metric limited to 7.60%. We also show that no power optimised version of MiniMD operating below 2.3 GHz can match the $ED^2P$ performance of the original code running at 3.2 GHz. For LavaMD this limit is marginally less restrictive at 2.2 GHz.

I. INTRODUCTION

Advances in processor design have delivered improvements in CPU performance for decades. As physical limits are reached, however, refinements to the same basic technologies are beginning to show diminishing returns [1]. One side-effect of this is an unsustainable rise in system power use, which the US Department of Energy has identified as a primary constraint for exascale systems [2].

Hardware manufacturers are increasingly prioritising energy efficiency in processor designs [3]. Research suggests that software modifications will be required to fully exploit the resulting improvements in modern processors [4]. The development of new energy-aware performance engineering tools and techniques will help developers to identify and capitalise on this new class of optimisation.

In this paper we present the Power Optimised Software Envelope (POSE). POSE is a mathematical and visual modelling tool which provides insight into the energy consumption characteristics of a code. Our work helps performance engineers understand whether power or runtime optimisation is the best strategy for improving the energy efficiency of their codes.

The contributions made in this research are:

• We introduce POSE, providing derivations for its constituent boundaries and an overview of the insights it provides;
• We show how POSE can be targeted to specific platforms and use-cases. Specifically, we investigate the trade-offs between runtime and CPU power consumption;
• We use POSE to study codes from the Mantevo and Rodinia benchmark suites. We assess the potential benefits of power optimisation for each code, showing that LavaMD offers the most scope for power optimisation while MiniMD offers the least;
• Finally, we investigate how opportunities for power optimisation vary in response to frequency scaling for these two codes.

The remainder of this paper is structured as follows: Section II presents a survey of related work; Section III details the construction of POSE along with the various insights it provides; Section IV demonstrates our new modelling tool with a study into the CPU power optimisation opportunities presented by a range of benchmark applications; and finally Section V concludes the paper and describes future research.

II. RELATED WORK

Performance modelling techniques enable the rapid exploration of large hardware and software design spaces. Table I categorises the performance modelling ecosystem based on model domain and underlying approach.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Domain</th>
<th>Energy</th>
</tr>
</thead>
</table>

This research was sponsored through a U.K. Technology Strategy Board project, number 131197 (Energy-Efficiency Tools For High-Performance Multi- and Many-core Applications).
A. Simulators

Performance simulators such as SST [5], WARRP [6] and PACE [7] gather performance data by executing a simplified representation of the original code. Using code as a modelling input reduces the burden of model construction placed on the user, meaning model accuracy depends instead on how faithfully the simulator is able to model an underlying system.

These approaches can be extremely insightful when searching for optimisations, however constructing and validating representative simulations is often limited by the need for numerous micro-benchmarks and also the time and state-space overheads of the underlying discrete simulator.

Tools such as Wattch [8] and McPAT [9] extend performance simulators with models of power draw. These models use the energy costs associated with particular hardware actions to estimate the power consumption characteristics of simulated code.

B. Analytical Models

Analytical models distil the structure and behaviour of a program into a set of parameterised mathematical expressions. Performance predictions are then obtained by solving these expressions for the required input parameters.

Analytical models produce results more quickly than simulations, making them particularly suitable for parameter studies. Ensuring the model is expressive enough to capture all possible program behaviours is challenging however and requires a deep understanding of the target application.

Examples of this approach include LogP [10], LogGP [11] and PRAM [12], which provide model skeletons which must then be tailored to individual codes. This approach has also been applied to modelling energy consumption, with examples including BTL [13] and CAPE [14].

C. Heuristic Models

Heuristic models represent the most abstract category of performance models and the one to which our work belongs. Rather than attempting to faithfully represent an entire system, heuristic models provide a simplified analogy which helps developers reason about particular properties of a code. Ease of construction and the clarity of their insights mean heuristic models are well suited to the early stages of optimisation.

Ahmdal’s Law [16], arguably the best known heuristic model, states that the performance gains from parallelisation are limited by the serial portion of a parallel program. A further example of this approach is the Roofline model [15], which frames application performance in terms of its operational intensity and two system bottlenecks; off-chip memory bandwidth and floating point performance. This simplification limits Roofline’s use as a predictive model but does mean a developer can easily isolate the limiting factor of code performance and target their optimisation efforts accordingly.

POSE serves as a preliminary ‘first cut’ modelling technique intended to guide energy-aware optimisation efforts. Our model provides an asymptotic analysis of the scope for optimisation in the power and runtime domains, allowing performance engineers to focus their efforts wherever they will be most beneficial. POSE draws inspiration from the Roofline model in that its insights are also presented in an intuitive graphical format. POSE differs from the other models described in that it does not identify optimisation opportunities, but rather which tools and approaches are best suited to finding them.

Horowitz proposed the Energy Delay Product as a metric to evaluate the energy and performance trade-offs in low power design [18]. Brooks extended this by weighting each component to better reflect the demands of specific domains [19]. We refer to these as the $E^n t^m$ family of metrics, which includes power ($E^1 t^{-1}$), energy ($E^1 t^0$) and time ($E^0 t^1$) as members. Most of the examples found in this paper use Energy Delay Squared Product ($E^1 t^2$), described by Brooks et al. as the most appropriate $E^m t^n$ metric when considering a fixed micro-architecture [8].

III. POWER OPTIMISED SOFTWARE ENVELOPE

Energy is the integral of power consumed over time, or put simply $E = Pt$. As such, the energy efficiency of a code can be achieved either through shortening its runtime ($t$) with conventional program optimisations or reducing average power consumption ($P$) with power optimisations. POSE enables performance engineers to compare the potential benefits of each approach for a given code and thus focus their efforts on whichever offers the greatest rewards.

The POSE heuristic partitions the energy/runtime plane into areas with differing performance characteristics relative to some initial unoptimised code and input. POSE is a general purpose heuristic which applies to all members of the $E^m t^n$ group with $m > 0$ and $n \geq 0$, and indeed any metric that increases in line with runtime and energy consumption. We introduce the bounds which constitute POSE and provide their derivations for the $E^m t^n$ family of metrics. The only prerequisites of our model are that time and energy consumption can be accurately measured or calculated for the target platform.

A. Feasible Performance Envelope

POSE is built around the concept of a Feasible Performance Envelope. We construct this by plotting lines of gradient $P_{\text{min}}$
and \( P_{\text{max}} \) in Figure 1. These values represent the minimum and maximum rates of power consumption that can occur during normal operation of the target platform. The (Runtime, Energy) costs incurred by running any given code \( \theta \) under similar conditions must be represented within this envelope.

\[ E_\lambda = \frac{P_\lambda^m}{t_\lambda^{m+n}} = \frac{P_\theta^m}{t_\theta^{m+n}} \]

\[ P_\lambda = P_\theta \times t_\lambda^{m+n} \]

\[ E_\lambda = \frac{P_\theta^m}{t_\theta^{m+n}} \times t_\lambda^{m+n+1} \]

\[ E_\lambda = (\frac{P_\theta^m}{t_\theta^{m+n}} \times t_\lambda^{m+n+1})^{\frac{1}{m}} \] (2)

We seek to use the most appropriate tools while searching for optimisations. If an optimisation yields significant reductions in runtime, with only minor reductions in power consumption, then it is reasonable to classify it as a runtime optimisation. Conventional time-based profilers and performance engineering tools are therefore better suited to finding these optimisations. It is the Contribution Bound which enables our model to make this distinction.

### B. Optimisation Bound

To constrain our search space further we consider the metric we wish to reduce.

**Definition 1.** For logically equivalent codes \( \theta \) and \( \lambda \), the transformation \( \theta \rightarrow \lambda \) is a valid optimisation with respect to a cost metric \( M \) iff \( M(\lambda) \succeq M(\theta) \).

We plot curve \( B \rightarrow E \) in Figure 1, which passes through \( \theta \) linking all points where \( M(\lambda) = M(\theta) \). By Definition 1 any optimised versions of \( \theta \) must exist below this bound. Naturally the equation for the Optimisation Bound depends on the metric we are optimising for. Figure 1 shows the Optimisation Bound for \( E^1t^2 \). The general form of this bound for the \( E^m t^n \) family of metrics is derived as follows:

\[ E_\lambda = \frac{P_\lambda^m}{t_\lambda^{m+n}} = \frac{P_\theta^m}{t_\theta^{m+n}} \]

\[ E_\lambda = (\frac{P_\theta^m}{t_\theta^{m+n}} \times t_\lambda^{m+n+1})^{\frac{1}{m}} \] (1)

### C. Contribution Bound

Our second bound considers what it means to optimise for reduced power draw.

**Definition 2.** An optimisation \( \theta \rightarrow \lambda \) with respect to metric \( M \) is a power optimisation iff the reduction in power draw it delivers is responsible for the majority of the improvement in terms of \( M \).

We plot curve \( C \rightarrow \theta \) in Figure 1 linking all points for which power and runtime factors contribute to \( M \) in the same ratio as our original code. By Definition 2 any power-optimised versions of \( \theta \) must lie below this Contribution Bound. Again the equation for the Contribution Bound depends on the metric chosen. Figure 1 shows the bound for \( E^1t^2 \) while the general form for \( E^m t^n \) metrics is derived as follows:

\[ P_\lambda^m \]

\[ t_\lambda^{m+n} = \frac{P_\theta^m}{t_\theta^{m+n}} \]

\[ P_\lambda^m = \frac{P_\theta^m}{t_\theta^{m+n}} \times t_\lambda^{m+n} \]

\[ E_\lambda = \frac{P_\theta^m}{t_\theta^{m+n}} \times t_\lambda^{m+n+1} \]

\[ E_\lambda = (\frac{P_\theta^m}{t_\theta^{m+n}} \times t_\lambda^{m+n+1})^{\frac{1}{m}} \] (2)

The first of these categories relates to the benefit offered by power optimisation. Taking the difference in energy between point \( \theta \) and \( D \) gives us an upper limit on the amount of energy which can be saved by reducing power consumption. Similarly, the difference in value between \( M(\theta) \) and \( M(C) \) gives an upper bound on the improvement in our metric we can expect to see from power optimisation.

The second category indicates the scope a code has for power optimisation. The difference in runtime between intersect \( E \) and \( \theta \) represents the maximum increase in runtime we could feasibly trade off to achieve a slower yet more energy efficient code. The value \( t_\theta/t_E \) represents the smallest speed up which guarantees a code that outperforms \( \theta \) with respect to \( M \). Finally, \( t_\theta/t_A \) is the smallest speed up guaranteed to outperform any power optimised version of \( \theta \).

The figures produced by POSE are all upper bounds, and the benefits of power optimisation will be more modest in practice. Even so, these figures are useful as they allow performance insights.
activity factor and denotes load capacitance, $C$.

Equation 4 is an approximation for dynamic power in which dissipation exist, though their effects are relatively minor [25].

Dynamic power is consumed when logic gates change state. Other forms of power draw of CMOS chips into component parts, of which semiconducting properties of silicon break down, allowing current to escape even when gates remain inactive. Other forms of power dissipation are the most significant.

Power Limit (RAPL) technology [24]. The choices of platform, for example. The cpufrequtils package allows us to override DVFS and manually set the desired P-state $S$. We control the number of active cores $C$ by specifying the number of threads used by our benchmarking routines and pinning each one to its own core to prevent migration. The remaining property is activity factor, which is influenced by benchmark code.

We define the range of values that $A$ and $C$ depend on the code and the nature of the optimisations being considered. POSE models for inherently serial codes should be constructed using single threaded benchmarks where $C_{min} = C_{max} = 1$, for example.

The first step in applying POSE is to construct a feasible performance envelope. Many manufacturers publish power dissipation figures for their hardware, however for safety reasons these are usually conservative estimates. POSE works best when the power bounds are as tight as possible, therefore we determine $P_{min}$ and $P_{max}$ empirically.

We specify power benchmarks using $(S, A, C)$ tuples, with P-state $S$, activity factor $A$ and active core count $C$. Our $P_{min}$ and $P_{max}$ benchmarks should reflect the range of values these properties can take for a given code $\theta$. This notion is formalised by Equation 6.

$$P_{min} = (S_{min}, A_{min}, C_{min} | \theta)$$
$$P_{max} = (S_{max}, A_{max}, C_{max} | \theta)$$

The values of $S$, $A$, and $C$ depend on the code and the nature of the optimisations being considered. POSE models for inherently serial codes should be constructed using single threaded benchmarks where $C_{min} = C_{max} = 1$, for example.

The cpufrequtils package allows us to override DVFS and manually set the desired P-state $S$. We control the number of active cores $C$ by specifying the number of threads used by our benchmarking routines and pinning each one to its own core to prevent migration. The remaining property is activity factor, which is influenced by benchmark code.

We define the range of values that $A$ can take for some fixed $S$ and $C$ as $[0, \alpha, \beta]$. Our $\alpha$ benchmark executes a single jmp instruction each clock cycle, preventing instruction pipelining. It performs no calculations or memory accesses while keeping control logic to a minimum. Supporting benchmark code can be downloaded from the link given in the Appendix.

Non-trivial codes perform more work per unit time than our minimal benchmark. This additional work means more transistors changing state per cycle, and hence a higher activity factor. The only exception occurs when applications are blocked for long periods, allowing the processor to enter an idle state. This can be addressed by adding delays to the benchmark.

FIREFISTER [27] serves as our benchmark for activity factor $\beta$. This tool is designed to trigger near-peak power
Prime95 and Linpack were also evaluated as beta factors above the level achievable with high level languages. It consists however they were outperformed by FIRESTARTER.

fully characterise our processor by measuring all the RAPL registers. Techniques described by Hahnel et al. [28] periodically to identify and compensate for any wraparound of power consumption figures. Our tool polls the RAPL interface optimisations which reduce parallelism (C < 4) or processor negligible time waiting for resources. We therefore disregard by running each code is given in Table III.

Applications were run with the default configuration given by the other applications. The energy and runtime costs incurred defaults, parameters were chosen to yield runtimes in line with their documentation where available. In the absence of suitable suites. These codes were compiled with ICC version 14.0.0.

Having characterised our system we now proceed to build POSE models for benchmarks in the Mantevo and Rodinia suites. These codes were compiled with ICC version 14.0.0. Applications were run with the default configuration given by their documentation where available. In the absence of suitable defaults, parameters were chosen to yield runtimes in line with the other applications. The energy and runtime costs incurred by running each code is given in Table III.

All codes ran in parallel across four cores and spent negligible time waiting for resources. We therefore disregard optimisations which reduce parallelism (C < 4) or processor consumption across a range of x86_64 processors. It consists of hand optimised assembly routines which raise the activity factor above the level achievable with high level languages. Prime95 and Linpack were also evaluated as beta benchmarks however they were outperformed by FIRESTARTER.

C. POSE Models for Code Optimisation

TABLE II: Feasible Performance Envelope Parameters (W)

<table>
<thead>
<tr>
<th>P-state (GHz)</th>
<th>Pα</th>
<th>Pβ</th>
<th>P C</th>
<th>Pα</th>
<th>Pβ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.50</td>
<td>8.6</td>
<td>11.29</td>
<td>10.47</td>
<td>15.13</td>
<td>12.01</td>
</tr>
<tr>
<td>1.70</td>
<td>9.11</td>
<td>11.62</td>
<td>10.73</td>
<td>15.76</td>
<td>12.38</td>
</tr>
<tr>
<td>1.80</td>
<td>9.59</td>
<td>11.98</td>
<td>11.02</td>
<td>16.29</td>
<td>12.77</td>
</tr>
<tr>
<td>2.30</td>
<td>10.47</td>
<td>14.04</td>
<td>12.74</td>
<td>19.92</td>
<td>15.10</td>
</tr>
<tr>
<td>2.40</td>
<td>10.71</td>
<td>14.56</td>
<td>13.17</td>
<td>20.81</td>
<td>15.73</td>
</tr>
<tr>
<td>2.50</td>
<td>11.04</td>
<td>15.15</td>
<td>13.65</td>
<td>21.72</td>
<td>16.34</td>
</tr>
<tr>
<td>2.60</td>
<td>11.36</td>
<td>15.67</td>
<td>14.16</td>
<td>22.83</td>
<td>17.14</td>
</tr>
<tr>
<td>2.70</td>
<td>11.71</td>
<td>16.34</td>
<td>14.64</td>
<td>23.95</td>
<td>17.82</td>
</tr>
<tr>
<td>2.90</td>
<td>12.47</td>
<td>17.79</td>
<td>15.92</td>
<td>26.37</td>
<td>19.46</td>
</tr>
<tr>
<td>3.00</td>
<td>13.00</td>
<td>18.64</td>
<td>16.68</td>
<td>27.91</td>
<td>20.45</td>
</tr>
<tr>
<td>3.10</td>
<td>13.39</td>
<td>19.38</td>
<td>17.44</td>
<td>29.55</td>
<td>21.49</td>
</tr>
<tr>
<td>3.20</td>
<td>13.93</td>
<td>20.43</td>
<td>18.26</td>
<td>31.07</td>
<td>22.83</td>
</tr>
</tbody>
</table>

C. POSE Models for Code Optimisation

Having characterised our system we now proceed to build POSE models for benchmarks in the Mantevo and Rodinia suites. These codes were compiled with ICC version 14.0.0. Applications were run with the default configuration given by their documentation where available. In the absence of suitable defaults, parameters were chosen to yield runtimes in line with the other applications. The energy and runtime costs incurred by running each code is given in Table III.

All codes ran in parallel across four cores and spent negligible time waiting for resources. We therefore disregard optimisations which reduce parallelism (C < 4) or processor throughput (S < 3.2 GHz). This corresponds to the feasible performance envelope given by Equation 7.

\[
P_{\text{min}} = (3.2 \text{ GHz}, \alpha, 4) = 26.88W, \]
\[
P_{\text{max}} = (3.2 \text{ GHz}, \beta, 4) = 49.61W
\]

Table IV summarises the POSE models constructed for each code. The time and energy costs of each code are given in the θ columns of Table IVa and Table IVb respectively. The POSE model coordinates given in the remaining columns are obtained by solving Equations 1 and 2 for these values, as described in Section III.

The remainder of this section focusses on MiniMD and LavaMD as the two codes representing the extremes of power consumption. POSE models for these two codes are reproduced graphically in Figure 3a and Figure 3b, and the results of these models are presented in Table V. Results for the remaining codes can be found in the Appendix.

C. POSE Models for Code Optimisation

Having characterised our system we now proceed to build POSE models for benchmarks in the Mantevo and Rodinia suites. These codes were compiled with ICC version 14.0.0. Applications were run with the default configuration given by their documentation where available. In the absence of suitable defaults, parameters were chosen to yield runtimes in line with the other applications. The energy and runtime costs incurred by running each code is given in Table III.

All codes ran in parallel across four cores and spent negligible time waiting for resources. We therefore disregard optimisations which reduce parallelism (C < 4) or processor

D. POSE Models for Frequency Scaling

The relationship between P-state and energy consumption is non-linear and workload dependent. Operating in low power
states can increase runtime, offsetting any energy savings from reduced power draw [29]. Application-aware DVFS can save energy by selecting the optimal P-state schedule for a given code [30]. This implies that code changes may effect the optimal P-state assignment. We now use POSE to reason about this class of optimisation.

The performance of MiniMD and LavaMD was measured for each P-state supported by our system. Figure 4 shows that the energy consumption of both codes can be reduced at the cost of increased runtime. Despite this, the lowest $E^1 t^2$ value for both codes occurs at 3.2 GHz, meaning race-to-halt is the optimal strategy in terms of this metric.

While useful, this simple analysis fails to account for potential co-optimisation of activity factor and P-state. It is possible that different optimisations may be required to achieve optimal performance in different P-states. The flexibility of POSE allows us to model this scenario by considering optimisations which can impact P-state as well as activity factor.

Equation 8 gives the feasible performance envelope corresponding to this class of optimisation. We choose 3.2 GHz as our initial ‘unoptimised’ baseline because this is the P-state our system defaults to when running MiniMD or LavaMD.

$$P_{min} = (1.6 \text{ GHz}, \alpha, 4) = 13.51W;$$
$$P_{max} = (3.2 \text{ GHz}, \beta, 4) = 49.61W$$

(8)

If two P-states have overlapping POSE models then it may be possible for a power optimised code running at the lower frequency P-state to outperform an unoptimised code running at a higher frequency. Conversely, if their POSE models do not overlap then switching to the higher performance P-state dominates any possible power optimisation at the weaker state. This analysis allows the weaker state to be excluded from any search for power optimisations.

For MiniMD, Figure 4a shows that the first POSE model which does not overlap with that for our baseline occurs at 2.2 GHz. This means that no power optimisations exist at P-states 2.2 GHz and below which can match the $E^1 t^2$ performance of our unoptimised baseline. Conversely, such an optimisation may exist at frequencies between 3.2 GHz and 2.2 GHz as shown by the overlapping of the respective POSE models. For LavaMD this optimisation threshold is lower at 2.1 GHz, lending support to the claim that of these two codes LavaMD is more amenable to power optimisation.

Dynamic Concurrency Throttling has also been proposed as a means to reduce energy consumption [31]. POSE could be used to model such optimisations in a similar manner to our P-state investigation; the only difference being the parameterisation of the feasible performance envelope ($C_{min} = 1$).

V. CONCLUSION

This paper presents POSE, a mathematical and visual modelling tool which captures the trade off between software power consumption and runtime. POSE provides insights regarding the scope a code has for power optimisation as well as the level of improvement which can be expected. These insights help developers to determine whether power or runtime optimisation is the best approach for improving the efficiency of a code.

POSE works by partitioning the energy/runtime plane into areas corresponding to runtime and power optimised versions of an initial code with respect to an optimisation metric. We provide derivations of POSE’s boundaries for the Energy Delay Product family of metrics. We also discuss the various insights our model provides.

We demonstrate POSE by modelling the CPU power consumption of a number of codes taken from the Rodinia and Mantevo benchmark suites. Our results illustrate that runtime optimisation is the preferred approach to reducing the energy consumption of MiniMD; power optimisation is limited to improving the $E^1 t^2$ of this code by at most 7.60%.
LavaMD shows more scope for power optimisation, offering improvements of up to 30.59% in the same metric.

Our investigation into frequency scaling highlights the ability of POSE to rule out dominated configurations and hence reduce the optimisation search space. We show that no power optimised version of MiniMD operating at P-states below 2.3 GHz can match the $E^2P^2$ performance of the original unoptimised code running at 3.2 GHz. Once again LavaMD shows more scope for optimisation, with this limit falling at the marginally less restrictive level of 2.2 GHz.

We believe our results are of interest to performance engineers and serve to demonstrate the practical utility of POSE. POSE is being engineered for inclusion into Allinea MAP [32], a well-known state-of-the-art application analytics tool for HPC clusters and applications.

Future Work

Work is ongoing to develop the hardware and software required to measure power consumption at scale. This will allow us further validate POSE by applying it to a broader selection of scientific codes running on a range of architectures. The quantitative nature of our technique makes it particularly well suited to comparison studies. As such we intend to investigate the power optimisation opportunities presented by a range of different platforms.

Our ultimate aim is to demonstrate how POSE may be used to identify specific optimisations. This will involve developing feasible performance envelopes for individual subsystems including memory, file systems and processors. We also intend to profile specific classes of code and establish $P_{min}$ baselines for each. Doing so would allow POSE to highlight optimisation opportunities at a per-kernel, per-subsystem level and hence facilitate targeted optimisation.

APPENDIX

Table VI summarises the results of POSE models for the remaining codes documented in this paper. These results were generated automatically. Software for building POSE models and producing these energy performance reports can be downloaded from http://warwick.ac.uk/pose.

<table>
<thead>
<tr>
<th>Code</th>
<th>Best Case Energy Saved by Reducing Power Consumption</th>
<th>Worst Case Slowdown as a Result of Power Optimisation</th>
<th>Best Case Improvement in $E^2t^2$ from Power Optimisation</th>
<th>Minimum Speed Up Guaranteed to Outperform $\theta$</th>
<th>Speed Up Required to Dominate Power Optimisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>leukocyte</td>
<td>151.77J</td>
<td>1.80s</td>
<td>23.73%</td>
<td>5.73s; 1.17x</td>
<td>8.59s; 1.28x</td>
</tr>
<tr>
<td>CFD</td>
<td>134.52J</td>
<td>1.58s</td>
<td>26.75%</td>
<td>4.20s; 1.16x</td>
<td>6.72s; 1.29x</td>
</tr>
<tr>
<td>Heartwall</td>
<td>125.40J</td>
<td>1.47s</td>
<td>29.32%</td>
<td>3.35s; 1.16x</td>
<td>5.68s; 1.30x</td>
</tr>
<tr>
<td>streamcluster</td>
<td>176.64J</td>
<td>1.06s</td>
<td>29.88%</td>
<td>4.57s; 1.16x</td>
<td>7.84s; 1.30x</td>
</tr>
</tbody>
</table>

REFERENCES
