Energy-Aware Dynamic Adaptation of Runtime Systems

Jordy Aaldering[0009−0001−3018−5152], Bernard van Gastel[0000−0002−0974−4634], and Sven-Bodo Scholz[0000−0002−8663−1043]

Radboud University, Nijmegen, Netherlands
{Jordy.Aaldering,Bernard.vanGastel,SvenBodo.Scholz}@ru.nl

Abstract. In recent years the energy-efficiency of software has become a key focus for both researchers and software developers, in an attempt to reduce greenhouse-gas emissions and operational costs. Despite this growing awareness, developers still lack effective strategies to improve the energy-efficiency of their programs beyond the well-established approaches that optimise for runtime performance. In this paper we present a dynamic adaptation algorithm that uses energy consumption feedback to optimise the energy-efficiency of data-parallel applications, by steering the level of parallelism during runtime through external control. This approach is especially suited to functional languages, whose side-effect-free nature and strong semantic guarantees allow for easier code generation and straightforward scalability of the parallelism of programs.
Through a series of experiments we evaluate the effectiveness of this approach. We measure how well the adaptation algorithm adapts to runtime changes, and we evaluate its effectiveness compared to a hypothesised oracle that always determines the optimal level of parallelism, as well as a runtime-optimising-based approach. We show that in a fixed-workload scenario we approach the theoretical best energy-efficiency, and that in changing workload scenarios the adaptation algorithm is able to react to changes in the energy consumption pattern and converges towards an optimal level of parallelism that minimising energy consumption.

Keywords: Dynamic Adaptation, Runtime Systems, Sustainability, Energy-Efficiency, High-Performance Computing, Parallel Programming, Functional Programming

1 Introduction

The importance of software sustainability has grown rapidly in recent years, alongside an increasing awareness of environmental concerns and the need for energy-efficient computing. From mobile applications to data centres, minimising the energy consumption of software has become essential for mitigating environmental impact, increasing battery life, and reducing operational costs. Traditional software systems often focus on maximising runtime performance without adequately considering energy consumption. However, as sustainability becomes a priority, there is a growing need for approaches that balance runtime performance and energy-efficiency.

The current landscape of computing hardware is characterised by its heterogeneity, with a shift towards multi-core and many-core hardware systems. High-performance computing applications typically aim to fully utilise all available resources on these systems, with the goal of maximising runtime performance. Determining efficient resource allocation for optimising performance – be it runtime or energy-efficiency – depends not only on an algorithm's implementation, but also on input data size, hardware characteristics, cache utilisation, and system-specific configurations such as thread pinning. This makes static approaches for determining resource allocation increasingly difficult, as they fail to account for variations in hardware capabilities and energy-efficiency [4]. Furthermore, it has been shown that optimising for runtime is not necessarily equivalent to optimising for energy-efficiency, and that optimising for runtime performance can even have a negative effect on the energy consumption of a program [34,3].

A key candidate for improving the energy-efficiency through more efficient resource allocation is the thread management of an application. Traditionally, thread management in software systems has focused primarily on performance metrics such as execution time and operations per second. A notable approach in the context of the Single assignment C (SaC) language uses a dynamic adaptation algorithm that adjusts the thread-count of a program by observing changes in runtime metrics [18]. This technique enables SaC applications to adapt to varying workloads and resource availability, improving runtime performance by efficiently utilising computational resources. This method optimises for runtime performance, but does not directly address energy consumption.

We present a method that extends existing runtime adaptation techniques by incorporating energy consumption as the primary factor in decision-making. By dynamically adjusting the number of threads based on real-time energy consumption metrics, we show that it is possible to achieve a more sustainable balance between runtime performance and energy-efficiency. Functional languages are an especially suitable target for this dynamic adaptation algorithm, as their side-effect-free nature and strong semantic guarantees allow for easier code generation and straightforward scalability of the parallelism of programs.

Our contributions are:

- A static analysis of data-parallel programs that investigates the relation between the level of parallelism of an application and its overall energy consumption. (Section 4)
- A dynamic adaptation algorithm that aims to minimise the energy consumption of data-parallel applications by steering level of parallelism during runtime. (Section 5)
- An implementation of a thread-steering mechanism in the data-parallel functional language SaC, based on the dynamic adaptation algorithm. (Section 6.1)
- An evaluation of how close the adaptation algorithm comes to the energy-efficiency of a theoretical optimum. (Section 6.3 and 6.4)

– An evaluation of the energy-efficiency improvements of the adaptation algorithm compared a runtime-based approach and static approaches. (Section 6.5 and 6.6)

2 Single assignment C

Single assignment C (SaC) is a functional array language that resembles the imperative syntax of languages such as C, whilst remaining side-effect-free [21,20]. It aims to generate high-performance parallel code for a wide range of multi-core and many-core architectures.

One of the key design choices that makes this possible is the use of a single language construct for all array operations, named a tensor comprehension [39]. These tensor comprehensions are side-effect-free mappings from index spaces to element values, based on the set builder notation. Take for example the following tensor comprehension, which increments the first 9×9 values of an array:

```
arr = { iv -> arr[iv] + 1 | [0,0] <= iv < [9,9] }
```

This tensor comprehension constitutes an index variable `iv` that ranges over the index set with lower bound `[0,0]` and upper bound `[9,9]`. If the lower or upper bound can be inferred from the use of `iv` in `arr[iv]`, they may be left out. For each index variable in this index space, the value at that index the array is incremented.

All array types in SaC consist of an element type followed by a shape specification in square brackets. In simple cases, such shape specifications are lists of numbers describing the length of each dimension of the array. To express relations between shape components, such shape extends can be replaced by variables, where identical variable names require identical extends in the corresponding positions. For example, an integer vector of length 100 can be defined as `int[100]`, and a square integer matrix of size $n \times n$ can be referred to as `int[n,n]`. An n-dimensional array of doubles with rank d and shape vector shp can be described by `double[d:shp]`. This notation is referred to as type patterns [2]. It allows not only to express constraints within shapes but also between different function arguments and return types. Take for example the shape constraints of a matrix multiplication. It requires that the multiplied matrices are of shapes $u \times v$ and $v \times w$, resulting in a $u \times w$ matrix.

This can be expressed as:

```
double[u,w] matmul(double[u,v] a, double[v,w] b)
```

Furthermore, shape variables such as `u`, `v`, and `w` can be used in function bodies as if they were user-defined variables.

2.1 Parallelism in Single assignment C

The SaC compiler is capable to generate parallel code for a range of parallel architectures, including multi-core machines [19], GPUs [23,26], and clusters [31]. In this paper, we extend the work on code generation for multi-core machines.

Before the actual generation of multi-threaded code, the high-level optimisation phase of the compiler extensively fuses tensor comprehensions in order to increase the granularity of parallelism and to improve the locality of code [21]. The multi-core back end then generates code which executes all top-level tensor comprehensions of sufficient size in parallel [19]. Each of these regions is executed in a fork-join style of bulk synchronous computing. To avoid the overhead of thread creation and termination for each such parallel region, the threads are being created only once upon program startup and kept active throughout the execution.

We refer to such a set of threads as *bees* in a *bee hive*. One designated thread, the *queen bee*, executes the entire program including the sequential sections, and it coordinates all the other bees. The other bees wait until the start of a parallel section is being signalled to them by the queen, and they return to the waiting state thereafter. More details on the multi-threaded back-end can be found in [19].

3 Defining "energy consumption"

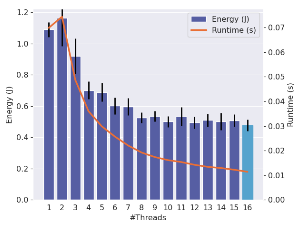
To be able to quantify the energy-efficiency of a program, it is essential that we accurately define what we mean by its "energy consumption". Ideally we would isolate the total amount of energy consumed by the hardware to run one specific program, separating it from other sources of energy consumption such as operating system overhead, background tasks, and the baseline power required to keep the hardware operational. However, fully isolating the energy consumption of a single process remains a challenge.

Currently, separating a program's energy consumption from that of other programs is only possible in certain environments. On Apple Silicon, for example, this is possible through the use of the `task_info` API, which returns a per-process energy consumption value [37]. For Apple's Intel-based machines the `diagCall64` function can be used instead. On Linux-based operating systems this is theoretically possible by extending the scheduler statistics, however to the best of our knowledge there is no implementation available. Although these methods provide potential solutions, they lack necessary documentation and would require specific hardware and elevated permissions, which go against our goal of making a broadly applicable adaptation algorithm. Furthermore, these methods still fail to exclude the baseline power required to keep the hardware operational, for which currently no solution exists.

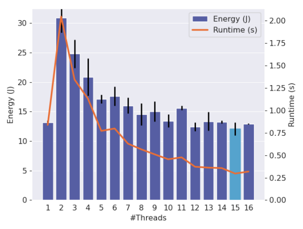
For the purposes of this study we define the energy consumption of a program as the total amount of energy consumed by the CPU throughout the entire runtime of that program, given that no other user-defined programs are running on that device. This implies that operating system overhead and baseline power required to keep the hardware operational are included in these measurements, as we cannot control or isolate these. We hypothesise that: if we succeed in improving the energy-efficiency of a program, this will result in a measurable decrease in total energy consumption across the runtime of that program.

3.1 Measuring energy consumption

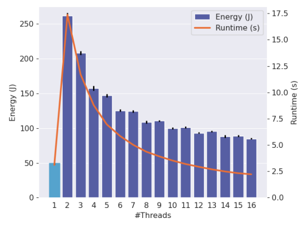
Running Average Power Limit (RAPL) is a technology developed by Intel for measuring and controlling the energy consumption of the CPU. It can be used to measure the total amount of energy consumed by the CPU throughout the runtime of a program. Although initially based on estimation models, thanks to improvements in hardware support RAPL has evolved into a precise energy measurement tool. RAPL reports fine-grained and reliable energy consumption data across various CPU domains, such as the package, PSys, and DRAM [28]. RAPL operates using a running counter that tracks the cumulative energy consumed by each of these domains. The energy consumption in micro-Joules over a time interval is calculated as the difference in counter values at the start and end of that time period. Although RAPL is only available on Intel CPUs since the Sandy Bridge architecture (2011), AMD supports a similar feature starting with their Zen architecture (2017). In addition to its broad availability, RAPL requires only slightly elevated privileges [42], making it accessible on many systems.



(a) **500** × **500** matrix multiplication.



(b) **1000** × **1000** matrix multiplication.



(c) **1500** × **1500** matrix multiplication.

Fig. 1: Average energy consumption and runtime of 200 naive matrix multiplications in SaC. Energy consumption is denoted by bars, with corresponding values in the left y-axis. Runtime is denoted by a line, corresponding to the right y-axis.

4 Energy consumption patterns

We perform a set of static experiments to determine which factors affect the energy consumption patterns of programs, and whether this leads to cases where using all available resources is not optimal. We aim to determine whether there are cases where using all available resources is not optimal, and whether optimising for energy-efficiency always correlates with optimising for runtime. We do this to gain new insight that informs the design process of the dynamic adaptation algorithm in Section 5.

Presumably, using all available resources of a system does not necessarily lead to the lowest energy consumption. Energy consumption patterns of programs are influenced by a wide range of factors, including programming language choice, algorithm implementation, workload behaviour, and hardware characteristics. To illustrate this we investigate the effect of different implementations, workload behaviours, and hardware characteristics on the energy consumption pattern of the matrix multiplication, N-body, and nine-point stencil algorithms. Within the context of data-parallel programming, these algorithms provide an interesting mix of CPU-bound and memory-bound applications.

Benchmarks are conducted on an Intel Xeon E-2378 server, maintained by the Radboud University. The system is accessible through Slurm, and contains a single 8 core 16 thread CPU, running at a base clock of 2.60GHz. The CPU has 16MB of level three cache, and the system itself provides 32GB of RAM.

4.1 Matrix multiplication

As our first benchmark we measure the runtime and energy usage of a single matrix multiplication, for different input matrix sizes. The runtime and energy measurements start right before the parallel region, and stop immediately after synchronisation. The matrix multiplication algorithm is used across a wide variety of domains, ranging from solving linear equations, to machine learning, to computer graphics. Although more sophisticated algorithms exist [41], we use the naive algorithm because it contains only a single tensor comprehension that SaC parallelises over. Whereas more sophisticated approaches aim to have consistent cache locality, the cache locality of the naive approach decreases as the input size increases, shifting the bottleneck from CPU to memory. This naive SaC implementation closely resembles the mathematical definition:

```
double[u,w] matmul(double[u,v] a, double[v,w] b)
{
    return { [i,j] -> sum({ [k] -> a[i,k] * b[k,j] }) };
}
```

The average runtime and energy consumption of this matrix multiplication algorithm on three different input matrix sizes are shown in Figure 1. Along the x-axis is the number of threads, ranging from one to sixteen. Bars represent the average energy consumption of a single matrix multiplication, whereas the line represents the average runtime. The results of a 500×500 matrix multiplication in Figure 1a align with our expectations; increasing the number of threads improves both the energy-efficiency and the runtime performance. When increasing the number of threads from one thread to two threads we do observe a small increase in both energy consumption and runtime, likely caused by the overhead introduced by having to manage multiple threads. By default, SaC uses a busy wait for the synchronisation of threads, which is known to have a negative effect on the energy consumption of a program [9].

An interesting observation, that we see in the following benchmarks as well, is that whereas the speedup in energy-efficiency and runtime performance is similar for low thread-counts, as the number of threads increases the runtime performance changes at a faster rate than the energy-efficiency. This suggests that the overhead caused by hyper-threading and having to manage multiple threads has a stronger negative effect on the energy consumption of a program than on its runtime performance.

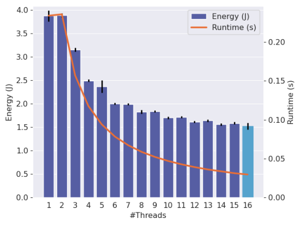
Increasing the matrix sizes past 500×500 introduces a significant shift in the optimal thread-count. This can be observed in Figure 1c, and to a lesser degree in Figure 1b. For 1000×1000 inputs there is a lot of variation in the observed energy measurements, potentially due to inefficient work distribution and inconsistent cache locality between threads. Whereas the best runtime performance is still obtained when using all available threads, the optimal thread-count in terms of energy consumption has shifted to only a single thread. This surprising result potentially arises from the fact that the memory required for these larger matrices exceeds the CPU's 16MB L3 cache capacity. When the combined size of the input and output matrices surpasses the available cache, the bottleneck shifts from compute to memory transfer. Although increasing the number of threads does still decrease the runtime, it has an over-proportional effect on the energy consumption. This reinforces the idea that selecting an optimal thread-count is a non-trivial task that does not always coincide with optimising for runtime, and that it is difficult to predict the precise reasons for such discrepancies.

4.2 N-body simulation

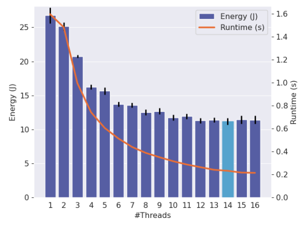
The N-body algorithm is typically applied in physics and astronomy to simulate the effect of physical sources, like gravity, on systems of bodies such as particles and celestial objects. For example, it can be used to simulate the movement of the planets of the solar system. Each body is defined by a record type, containing its current position, velocity, and mass. Although records in SaC use a similar notation as in C, they are fully flattened into arrays of the record's fields, improving data locality [25].

A time-step of the N-body simulation on an array of bodies and delta-time `dt` is defined as follows, where `acc` is a function that returns the acceleration vector from one body to another.

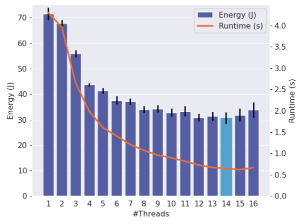
```
struct Body[n] nbody(struct Body[n] bodies, double dt)
{
```



(a) **10000** element N-body.



(b) **25000** element N-body.



(c) **40000** element N-body.

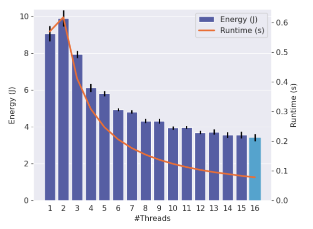
Fig. 2: Average energy consumption and runtime of 200 N-body simulation in SaC. Energy consumption is denoted by bars, with corresponding values in the left y-axis. Runtime is denoted by a line, corresponding to the right y-axis.

```
    accel = { [i] -> sum({
                [j] -> acc(bodies[i], bodies[j]) }) };
    bodies.pos += bodies.vel * dt;
    bodies.vel += accel * dt;
    return bodies;
}
```

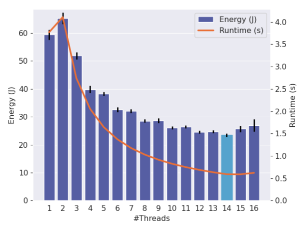
For an N-body simulation with 10000 elements in Figure 2a we observe that although increasing the thread-count strictly decreases the runtime, for simulations with 25000 and 40000 elements in Figures 2b and 2c respectively, the energy consumption plateaus after using more than eight threads. It seems that any improvements in energy consumption gained by using hyper-threading are mitigated by the overhead it introduces. Consequently, on a system with shared resources and multiple running processes it might be beneficial to select a lower thread-count for the N-body simulation, in order to free up the remaining threads for other processes without incurring a significant negative impact on the energy consumption of the N-body simulation.

4.3 Nine-point stencil

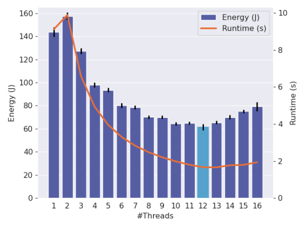
The nine-point stencil operation is used for image processing, computer simulations, and for solving partial differential equations. Perhaps the most common application of the stencil operation is in the training process of Convolutional



(a) **10000** × **10000** nine-point stencil.



(b) **25000** × **25000** nine-point stencil.



(c) **40000** × **40000** nine-point stencil.

Fig. 3: Average energy consumption and runtime of 200 nine-point stencil operations in SaC. Energy consumption is denoted by bars, with corresponding values in the left y-axis. Runtime is denoted by a line, corresponding to the right y-axis.

Neural Networks (CNN). In this context, the stencil operation is applied for edge-detection and other image processing tasks, as well as for providing a means for spatial awareness. Most relevant to us is the application of the stencil operation for resizing arrays in CNNs, which highlights a real-world scenario where the input data size changes during the runtime of a program.

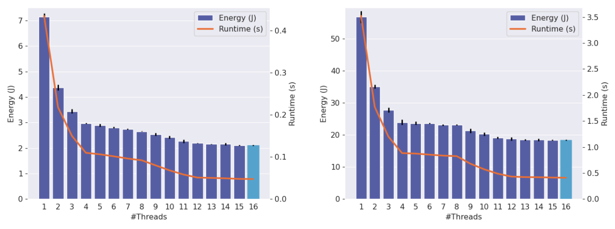
The nine-point stencil is defined in SaC as:

```
double[2:shp] stencil(double[2:shp] a, double[3,3] w)
{
    return { iv -> sum({
                jv -> w[jv] * a[mod(iv+jv-1, shp)] })
           | iv < shp };
}
```

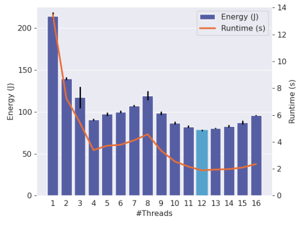
It is represented as a weighted sum of each point in an array and its eight immediate neighbours, where each weight depends on the corresponding value in a 3×3 array of weights.

For a 10000×10000 input matrix in Figure 3a we observe that both the runtime and energy consumption decrease as the number of threads increases. However for 25000×25000 and 40000×40000 inputs, using all available threads is no longer optimal for energy-efficiency and runtime performance. We see this in Figures 3b and 3c respectively, where as the input size increases the optimum thread-count decreases. Whereas the change in the energy consumption pattern of the matrix multiplication algorithm could be explained by cache-behaviour, that is not the case in the nine-point stencil operation. It is unclear why this behaviour occurs, which highlights again the difficulty in determining an optimal thread-count.

4.4 Implementation language



(a) **500 × 500** matrix multiplication in Rust. (b) **1000 × 1000** matrix multiplication in Rust.

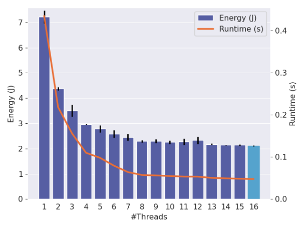


(c) **1500 × 1500** matrix multiplication in Rust.

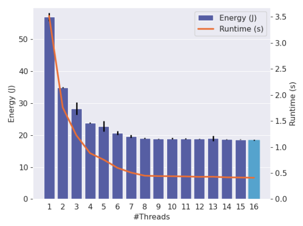
Fig. 4: Average energy consumption and runtime of 50 naive matrix multiplications in Rust. Energy consumption is denoted by bars, with corresponding values in the left y-axis. Runtime is denoted by a line, corresponding to the right y-axis.

To validate that these observations are not specific to SaC, we also evaluate a manually parallelised implementation of the matrix multiplication algorithm in Rust. We observe in Figure 4 that switching the implementation language from SaC to Rust produces a significantly different energy consumption pattern. Although for the 500×500 and 1000×1000 inputs in Figures 4a and 4b respectively the optimum thread-count remains unchanged, the single-threaded SaC implementation performs significantly better than the Rust implementation in terms of both runtime and energy consumption. Furthermore, the runtime and energy consumption of the Rust implementation plateau already when using four or more threads, whereas the runtime and energy consumption of the SaC implementation strictly decrease as the number of threads increases. In Figure 4c we observe that, whereas for a 1500 × 1500 input the optimal thread-count for the SaC implementation is at only a single thread, the optimal thread-count for the Rust implementation lies at twelve threads.

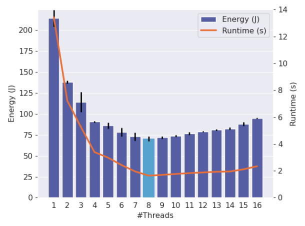
4.5 Thread-pinning & Scheduling



(a) **500** × **500** matrix multiplication in Rust.



(b) **1000** × **1000** matrix multiplication in Rust.



(c) **1500** × **1500** matrix multiplication in Rust.

Fig. 5: Average energy consumption and runtime of 50 naive matrix multiplications in Rust. Energy consumption is denoted by bars, with corresponding values in the left y-axis. Runtime is denoted by a line, corresponding to the right y-axis.

Besides changing the input data size, we investigate whether a change in system configuration can also introduce a change in the energy consumption pattern of a program. To this end we investigate the effect of disabling thread pinning, which was enabled for the previous benchmarks. When disabling thread pinning, the responsibility of thread management moves from the application to the operating system. This makes it difficult to reason about the way in which threads are scheduled, as it becomes dependent on the specific operating system, and its configuration.

Disabling thread pinning in SaC has no significant impact on the energy consumption pattern of the chosen benchmarks on the system under test, besides slightly increasing the energy consumption overall. Therefore we omit those results here. In the case of the Rust implementation, it can be observed in Figure 5c that disabling thread pinning has a significant impact on the energy consumption profile. Disabling thread pinning has almost no effect on the 500×500 and 1000×1000 matrix multiplications, in Figures 5a and 5b respectively. However, for 1500×1500 input matrices there is a significant change in the energy consumption pattern. As the number of threads increases from 8 to 16, the energy consumption gradually starts increasing again. This behaviour suggests that the overhead associated with hyper-threading may outweigh the benefits of parallelism, further complicating the task of optimising for energy-efficiency.

5 Energy-Aware Dynamic Adaptation

In Section 4 we observe that the selection of an optimal thread-count to maximize energy-efficiency depends on the running conditions. Factors such as input data size, cache limits, memory transfer overhead, thread pinning, and implementation choices can heavily influence the energy consumption profile of an application. Given the multitude of effects, a static cost-model-based approach seems infeasible. A profiling-based approach might provide better data but would incur a significant overhead, which goes against our goal of optimising for energy. Consequently, we look for a dynamic approach to avoid the need for manual tuning, minimise overall energy consumption, and to be able to adapt to runtime changes.

The overall idea is based on the observation that data-parallel codes typically perform several repetitions of the same parallel computation before moving on to the next set of repetitions. This behaviour allows dynamic adaptation to tune the parallelism between such repetitions [18]. The adaptation algorithm monitors changes in the energy consumption from one repetition of a parallel region to the next, and periodically adjusts the recommended thread-count in an attempt to find the thread-count with the lowest energy consumption. Based on measured changes in energy consumption, using power meters built in to the processor, the algorithm determines whether to increase or decrease the thread count for the next iteration, and by what amount. The hypothesis is that with this approach, we can adapt the thread-count during runtime in correspondence to changes in energy consumption, converging to a (local) optimum for energy-efficiency.

The algorithm uses two variables for steering the thread-count of an application: a step direction, and a step size. The step direction describes whether the number of threads will be increased or decreased, where the step size gives the amount of threads that will be added or removed. The algorithm is executed at a fixed frequency, and is supplied with an array of energy consumption samples of the total amount of energy consumed by each iteration op the parallel region. The frequency is configurable, but we found that for our benchmarks that a frequency of ten iterations per thread-count adjustment strikes a balance between a high adaptation speed whilst also being resilient against noise such as short-running background tasks and operating system overhead. To filter noise, the median value of the given energy consumption samples is taken.

When the measured energy consumption of an iteration differs more than a factor α compared to the previous iteration, it is likely that there was a change in workload behaviour. To be able to quickly find the (potentially different) optimal thread-count, the step size is reset to half the number of maximum threads. If there was an increase in energy consumption of less than factor α, we assume that the optimal thread-count is somewhere in between the current thread-count and the previous thread-count. In that case the step direction is inverted, and as to not overshoot the optimum the current step size is halved.

As we observe in Section 4, changing the thread-count by even a single thread can have a significant negative impact on the energy consumption of a program. In an attempt to minimise this overhead, alongside the intrinsic scheduling overhead for making changes to the thread-count, we aim to settle into a fixed thread-count when we are close to the (local) optimum. To this end we use a real-valued thread-count and step size. Instead of decreasing the step size at each iteration by a fixed amount, updating the step size is defined by a function: $\max(\frac{3}{5}\mathit{\Delta} t,\ \mathit{\Delta} t \,/\, (\beta + \mathit{\Delta} t))$, for $\beta \in [0.5, 1]$. Using this function, step sizes greater than one are decreased by a fixed fraction $\frac{3}{5}$. For step sizes near one the function is less steep, ensuring that we do not settle in a local optimum too quickly by decreasing the step size too much. For our benchmarks we have found that a value of 0.85 for β works well.

When using such a real-valued step size, the step size will eventually near zero and consequently the thread-count will no longer change. In an attempt to escape local optimums, the step size is reset to half the maximum amount of threads when it becomes less than $\gamma \in (0, 1]$. The lower the value of γ, the larger the amount of iterations that is required until the step size is reset. For our benchmarks we have found that a value of 0.155 for γ works well.

A formal definition of this algorithm is described in Algorithm 1.

6 Case-studies

We conduct a series of experiments to evaluate the effectiveness of the dynamic adaptation algorithm. First we evaluate the ability of the algorithm to adapt to runtime changes and reach an optimal thread-count, by comparing the thread-count given by the adaptation algorithm to an optimal thread-count that can be derived from the observations in Section 4. To evaluate how close the adaptation algorithm comes to the performance of a theoretical optimum, we compare its effectiveness against a hypothesised oracle that always determines the most energy-efficient number of threads. To investigate whether there are differences between our energy-optimising approach and a runtime-optimising approach, we similarly compare our approach to the runtime-based adaptation algorithm described by Gordon et al. [18]. We investigate whether the adaptation algorithm introduces significant energy overhead, and finally we measure how much energy the adaptation algorithm saves compared to multiple static approaches.

Both the energy-based and runtime-based dynamic adaptation algorithms, the benchmark scripts, and the benchmark results are publicly available on

Algorithm 1 Algorithm for updating the thread-count based on energy measurements.

```
Require: Maximum number of threads $m_t$, current number of threads $n_t$, direction of
    change $\hat{d} \in \{-1, 1\}$, rate of change $\Delta t$, array of energy samples $S$, and parameters
    $\alpha, \beta, \gamma \in \mathbb{R}$.
Ensure: A continuous updating of $n_t$, $\hat{d}$, $\Delta t$, and $E_{old}$.
 1: $\hat{d} \leftarrow -1$
 2: $\Delta t \leftarrow \frac{1}{2} m_t$
 3: loop
 4:     Wait for samples $S$
 5:     $E_{new} \leftarrow median(S)$
 6:     if $(E_{new} < E_{old} \cdot (1 - \alpha)) \vee (E_{new} > E_{old} \cdot (1 + \alpha))$ then
 7:         $\hat{d} \leftarrow sign(m_t - 2n_t)$
 8:         $\Delta t \leftarrow \frac{1}{2} m_t$
 9:     else
10:         if $E_{new} > E_{old}$ then
11:             $\hat{d} \leftarrow -\hat{d}$
12:         if $\Delta t > \gamma$ then
13:             $\Delta t \leftarrow max(\frac{3}{5} \Delta t,\ \Delta t \,/\, (\beta + \Delta t))$
14:         else
15:             $\hat{d} \leftarrow sign(m_t - 2n_t)$
16:             $\Delta t \leftarrow \frac{1}{2} m_t$
17:     $E_{old} \leftarrow E_{new}$
18:     $n_t \leftarrow n_t + \hat{d} \cdot \Delta t$
19:     $n_t \leftarrow clamp(n_t,\ 1,\ m_t)$
```

GitHub [1]. The implementation of the thread-switching beehive system and adaptation algorithm in SaC are available on the SaC GitLab project [22].

6.1 Thread-management in Single assignment C

We hook into SaC's beehive system, explained in Section 2.1, to control the number of threads of SaC programs dynamically. Before sending a signal to the worker bees to start processing, the queen requests the recommended thread-count and decreases or increases the size of the hive by putting worker bees to sleep or waking up sleeping bees. Since the actual thread-count is fixed, we selectively put bees to sleep through the use of a semaphore. The side-effect-free nature of the parallel code enables the required workload redistributions through the queen bee without any potential impact on the overall semantic of the program. Using a semaphore, instead of the standard wait, ensures that the threads of sleeping bees are free to be used by other processes on the system. Sleeping bees are awoken by the queen bee by posting to this semaphore, after which these bees enter their wait state again, waiting for the queen to signal them to start working.

6.2 Energy overhead

With our goal of minimising the energy consumption of programs, it is crucial that the adaptation algorithm itself does not introduce a significant amount of energy overhead. We measure the overhead introduced by the adaptation algorithm by applying it to a function that does nothing, and simply returns zero. The measures sources of overhead then are the starting and stopping of each energy measurement, and the re-computation of the suggested thread-count when ten energy measurements have been supplied. The energy consumption and runtime overhead are measured by repeating this process one million times. We find that on average the algorithm has an energy overhead of $48.49 \mu J$ and a runtime overhead of $4.84 \mu s$. From this we conclude that the adaptation algorithm does not introduce significant energy overhead.

While there might be additional effects of the adaptation that cause overhead, such as context switches or other operating system effects, we fail to identify experiments that would quantify these effects. We study the overhead that stems from the adjustment process itself in a series of experiments where we look at an execution where the problem-size is being changed over time.

6.3 Adaptation quality

Factors such as input data size typically change during the execution of a program. A prominent real-world example this is observed in the training process of Convolutional Neural Networks, where each layer of the network operates on differently sized arrays. We have observed in Section 4 that these changes can cause shifts in the optimal thread-count. A key requirement for the adaptation algorithm is its ability to adjust to such changes during the runtime of programs. The objective is to dynamically converge towards the optimal thread-count under changing workload behaviours and system configurations. We evaluate the adaptation quality of the algorithm by gradually changing the input data size on the examples from Section 4, and assessing how effectively the adaptation algorithm converges to the thread-count with minimal energy consumption.

Matrix multiplication The matrix multiplication algorithm in SaC presents an interesting challenge for our adaptation algorithm. For 1500×1500 input matrices using only a single thread consumes the least amount of energy, whereas using two threads already consumes the most amount of energy. Since the energy consumption strictly decreases for each subsequent thread-count after two threads, it is likely that the adaptation algorithm ends up in the local optimum at sixteen threads.

Figure 6 shows the actual thread-count suggested by the adaptation algorithm across the runtime of a single program, alongside the optimum thread-count as found in Section 4. The x-axis represents the runtime of the program, where the input data size – denoted by the x-axis labels – changes at each third of the program's runtime. The orange dashed line shows the optimal thread-count that can be derived from the observations in Section 4. The blue line shows the thread-count recommended by the adaptation algorithm. The closer the suggested thread-count comes to the derived optimal thread-count, the better the adaptation quality of the algorithm. In Figure 6 we see that the adaptation algorithm is able to find the optimum thread-count for 500×500 and 1000×1000 input matrices. However, as expected, the algorithm is not able to find the optimum at a single thread for 1500×1500 input matrices. It is however able to remain in the local optimum at sixteen threads.

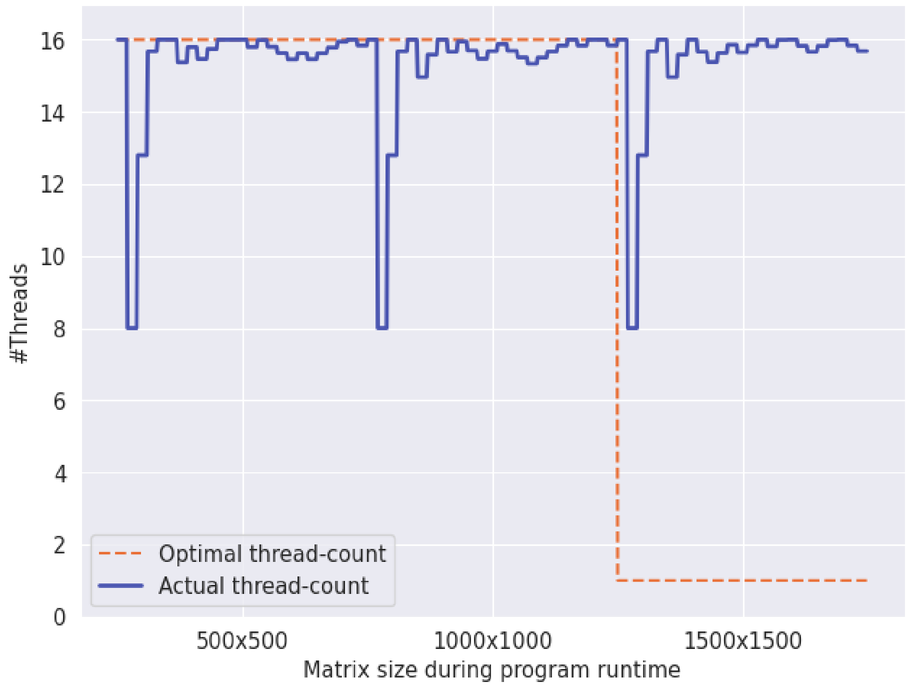


Fig. 6: Adaptation quality on the SaC implementation of the matrix multiplication algorithm.

N-body simulation In the case of the N-body simulation we have observed in Section 4 that the energy consumption plateaus when using eight or more threads. Any thread-count between eight and sixteen threads can be considered as sufficient, since the energy consumption at each of these thread-counts lies within a few percent of each other. Figure 7 shows that the adaptation algorithm is able to find the optimum thread-count for the N-body simulation. In fact, even though there is only a small difference in the energy consumption between eight and sixteen threads, the adaptation algorithm still finds the optimum at sixteen, fourteen, and twelve threads for all three input sizes respectively. For brevity we exclude the N-body simulation from the remaining benchmarks, as it presents no further noteworthy results or insights.

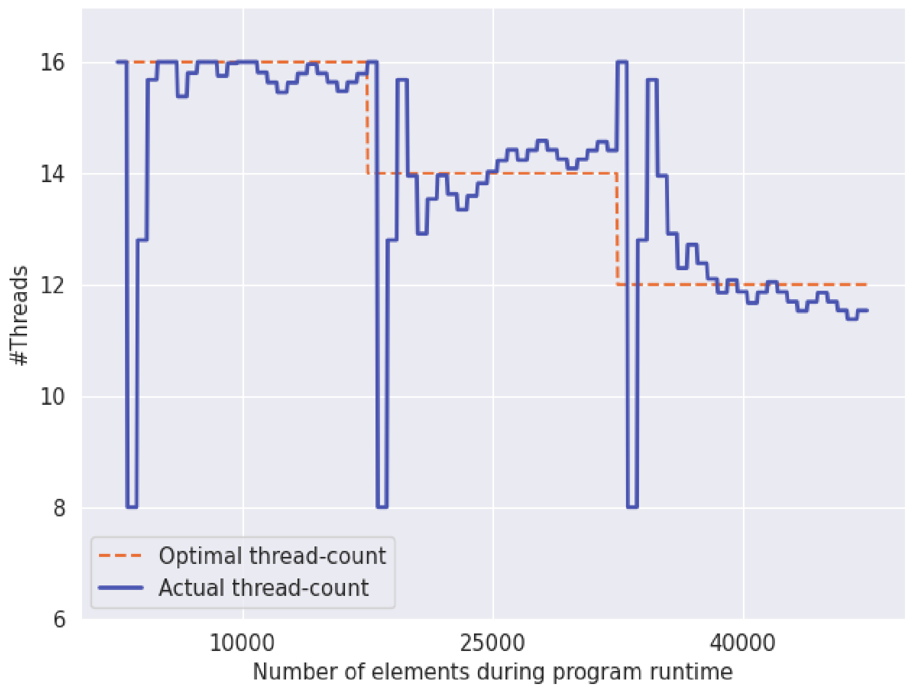


Fig. 7: Adaptation quality on the SaC implementation of the N-body simulation.

Nine-point stencil For the nine-point stencil operation we have observed in Figure 3 that as the input data size increases, the optimal thread-count gradually decreases. Applying the adaptation algorithm we see in Figure 8 that it is able to converge to the optimal thread-count in these cases. Although for a 40000×40000 input the algorithm suggests a thread-count of ten instead of twelve, Figure 3c shows that the energy consumption at these two thread-counts is within a few percent of each other, so we consider this thread-count sufficient.

Rust implementation Figure 9 shows the adaptation quality for the Rust implementation. In Section 4 we saw the most variation in this benchmark, with optimum thread-counts at eight, twelve, and sixteen threads, as well as local optimums. Even in a scenario with varying optimal thread-counts, the adaptation algorithm is able to quickly find the optimal thread-count after a change in the matrix size or system configuration.

These experiments show that the dynamic adaptation algorithm is capable of adequately adapting to runtime changes, except in outlier cases such as the

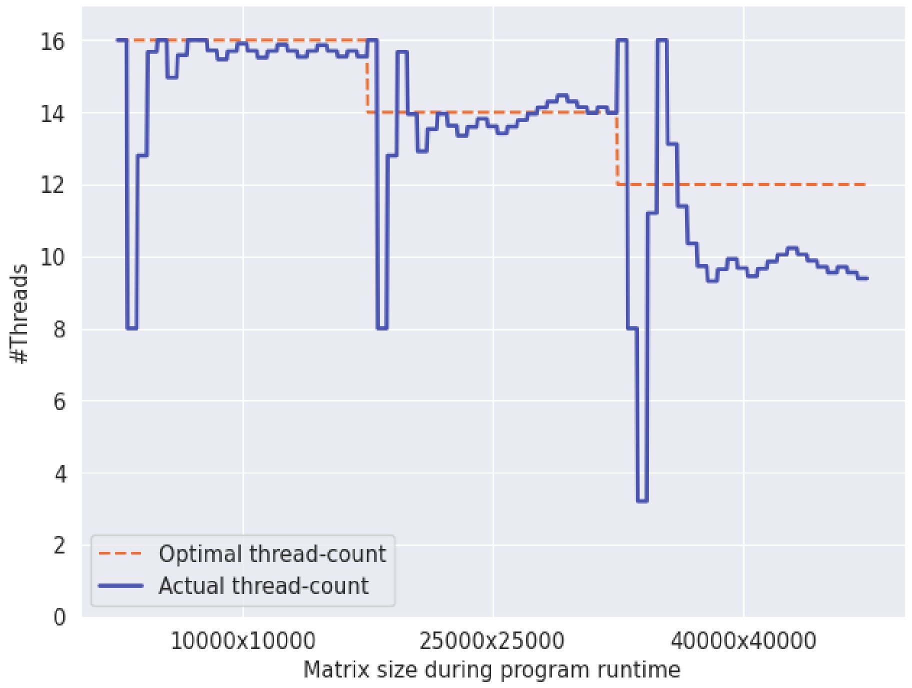


Fig. 8: Adaptation quality on the SaC implementation of the nine-point stencil operation.

1500 × 1500 matrix multiplication in SaC where we do however find the local optimum. When a change in the energy consumption pattern occurs, the adaptation algorithm is able to convert to the optimal thread-count within a few thread-count adjustments.

6.4 Comparison to oracle-based approach

To further evaluate the effectiveness of the dynamic adaptation algorithm we compare its performance against a theoretical "oracle" approach. This theoretical oracle knows which thread-count will result in the lowest energy consumption for the current implementation, workload behaviour, and hardware characteristics. Although in general this level of foresight is unattainable without an analysis like the one in Section 4, it provides a theoretical upper bound for the performance of the chosen benchmarks. These optimums serve as the baseline to simulate the decision-making process of the oracle. This oracle represents the best possible energy consumption, so any solution that closely approaches its performance can be considered highly effective.

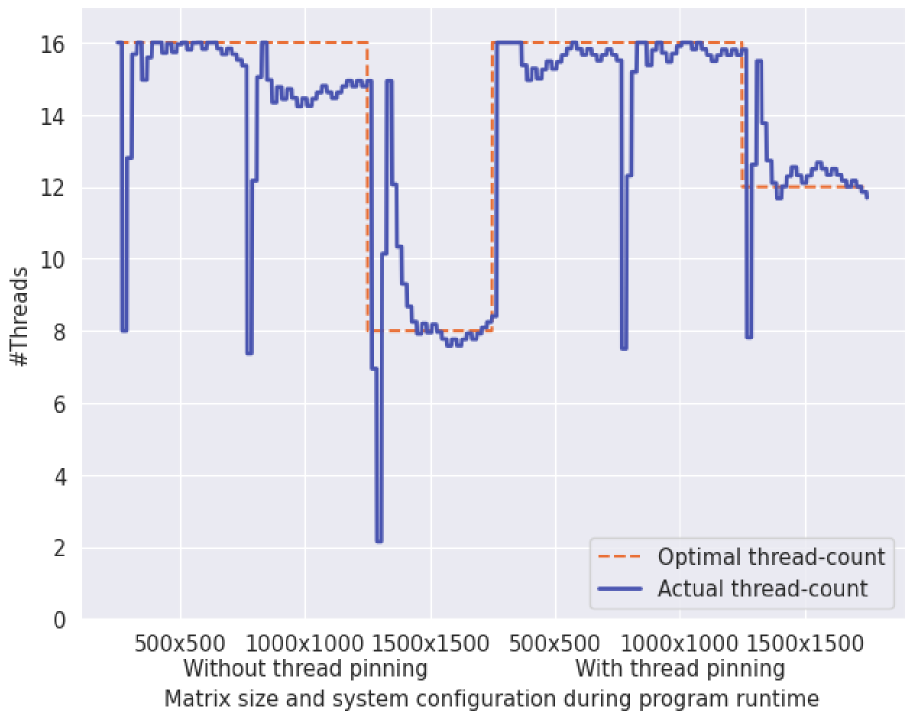


Fig. 9: Adaptation quality on the Rust implementation of the matrix multiplication algorithm.

Matrix multiplication Table 1 shows how close the adaptive algorithm comes to the theoretical best performance on the matrix multiplication algorithm, provided by the oracle. For 500×500 inputs the dynamic approach is on par with the oracle in terms of energy consumption, although it introduces a 10% increase in runtime. Given a 1000×1000 matrix these results suggest that the dynamic approach consumes 7% less energy, however this is likely a consequence of the large amount of variation that was observed in Figure 1b, potentially due to inefficient work distribution and inconsistent cache locality between threads. This does however show that for 1000×1000 inputs the dynamic approach provides a similar level of performance. For a 1500×1500 matrix size the dynamic approach consumes 55% more energy than the oracle-based approach, which is a result of the inability of the adaptation algorithm the reach the optimal thread-count at a single thread. This does however lead to a 27% shorter runtime compared to the oracle-based approach.

Nine-point stencil Table 2 shows the overhead of the dynamic approach on the nine-point stencil operation. For 10000×10000 and 25000×25000 inputs the dynamic approach provides a similar level of performance, with only a 5% in-

Matrix size	Energy	Runtime
500×500	0%	10%
1000×1000	-7%	-5%
1500×1500	55%	-27%

Table 1: Energy difference between the adaptive approach and the theoretical best performance of the oracle for the matrix multiplication algorithm.

crease in energy consumption compared to the theoretical optimum. However for 40000×40000 inputs the energy consumption is 15% worse compared to the oracle. In Figure 8 we observed that for this input size the adaptation algorithm recommends ten threads, instead of the optimal thread-count of twelve threads. Although this thread-count has a similar energy consumption compared to running at twelve threads, it does have significantly worse runtime, leading to a 33% increase in runtime. Part of the increase in energy consumption can be attributed to additional energy overhead inherent to an increase in runtime, as we discuss in Section 3.

Matrix size	Energy	Runtime
10000×10000	5%	4%
25000×25000	5%	5%
40000×40000	15%	33%

Table 2: Energy difference between the adaptive approach and the theoretical best performance of the oracle for the nine point stencil operation.

Rust implementation For the Rust implementation of the matrix multiplication algorithm in Table 3 we see that the overhead of our dynamic approach is minimal in cases where thread pinning is enabled. With thread pinning disabled, the dynamic approach consumes up to 8% more energy and has an up to 18% longer runtime, which is potentially due to additional scheduler overhead inherent to thread management.

These results demonstrate that while this theoretical oracle is not feasible in practise, in the typical case our dynamic approach provides a practical solution for reasonably approximating the theoretical optimal energy-efficiency. For outliers such as the 1500×1500 matrix multiplication in SaC the dynamic approach might not be able to find the optimal thread-count, instead finding a local optimum that results in an increase in energy consumption.

6.5 Comparison to runtime-based approach

We repeat the same benchmark as in Section 6.4, however we now compare the energy-based dynamic adaptation algorithm against an implementation of the dynamic runtime-optimising algorithm described by Gordon et al. [18].

Matrix size	Thread pinning	Energy	Runtime
500×500	Enabled	3%	6%
1000×1000	"	1%	3%
1500×1500	"	8%	10%
500×500	Disabled	1%	2%
1000×1000	"	1%	1%
1500×1500	"	8%	18%

Table 3: Energy difference between the adaptive approach and the theoretical best performance of the oracle for the matrix multiplication algorithm in rust.

Matrix multiplication Table 4 shows that for 500×500 and 1500×1500 inputs the energy-optimising and runtime-optimising approaches provide a similar level of performance. For 1000×1000 inputs we even observe an 11% decrease in energy consumption, however this could again be partially due to the large amount in variation for this example. Although the most energy-efficient threat-count for a 1500×1500 input does not correlate with the optimal thread-count for runtime, in Figure 6 we observed that the energy-optimising approach is not able to find this optimum, instead converging to a threat-count of sixteen. This is the best thread-count for runtime, so consequently both approaches perform similarly on 1500×1500 inputs.

Matrix size	Energy	Runtime
500×500	-1%	0%
1000×1000	-11%	-17%
1500×1500	-2%	-7%

Table 4: Energy difference between the energy-based and runtime-based adaptation algorithms on the matrix multiplication algorithm.

Nine-point stencil In Table 5 we see that for the two smaller input sizes the difference in energy consumption between the energy-optimising and runtime-optimising approach is within a few percent of each other. However, the energy-optimising approach does have a 25% greater runtime than the runtime-optimising approach for a 40000×40000 input. As we discuss in the comparison to the oracle-based approach, this is likely due to the fact that the energy-aware algorithm recommends a threat-count of ten, which is similar to using twelve threads in terms of energy consumption, but is significantly worse in terms of runtime. Keeping this in mind, a 7% increase in energy consumption seems reasonable.

Rust implementation Table 6 shows that in the case of the Rust implementation of the matrix multiplication algorithm, the energy-optimising approach and runtime-optimising approach have similar performance, with the energy-based algorithm always performing slightly better in terms of energy consumption.

Matrix size	Energy	Runtime
10000×10000	-2%	-8%
25000×25000	-4%	-3%
40000×40000	7%	25%

Table 5: Energy difference between the energy-based and runtime-based adaptation algorithms on the nine-point stencil operation.

Matrix size	Thread pinning	Energy	Runtime
500×500	Enabled	-1%	-3%
1000×1000	"	-2%	-5%
1500×1500	"	-1%	-6%
500×500	Disabled	-2%	-1%
1000×1000	"	0%	0%
1500×1500	"	0%	2%

Table 6: Energy difference between the energy-based and runtime-based adaptation algorithms on the matrix multiplication algorithm in Rust.

These results show that, while a runtime-based approach can provide a similar level of performance in cases where energy consumption scales linearly with the number of threads. In cases where the energy consumption or runtime is not strictly decreasing with respect to the thread-count, the runtime-based approach is not able to achieve the same level of energy-efficiency – or even runtime performance – as the energy-based approach.

6.6 Comparison to static approaches

Finally, we evaluate how much energy can be saved when using the dynamic approach compared to static approaches that always run at a fixed thread-count. For a set of fixed thread-counts we measure the average difference in energy consumption compared to using a dynamic approach. Besides running single-threaded, typical choices for threat-count are chosen based on the number of physical cores or the maximal number of available threads. Therefore we include the results of running at one, eight, and sixteen threads respectively. In Section 4 we observed that for our examples choosing twelve threads often also leads to lower energy consumption, therefore we include a twelve-threaded application as well.

Matrix multiplication Table 7 describes the difference in energy consumption and runtime between the dynamic adaptation algorithm and static thread-count approaches. Compared to always running at sixteen threads the adaptive approach provides a similar level of performance, however when a poor choice in thread-count is made the adaptive algorithm can save up to 21% of energy. Although the adaptive approach is 85% faster in terms of runtime compared to a single threaded approach, the energy reduction is 12%. This is due to the outlier in 1500×1500 inputs for which our adaptation algorithm is not able to find the optimal thread-count, where using a single thread has a lower energy consumption but a greater runtime.

#Threads	Energy	Runtime
1	-12%	-85%
8	-21%	-57%
12	-7%	-21%
16	-1%	4%

Table 7: Energy consumption of the dynamic adaptation algorithm compared to static approaches on the matrix multiplication algorithm.

Nine-point stencil If a reasonable, but ineffective, choice in thread-count is made for the nine-point stencil operation we observe in Table 8 that 10% of energy can be saved with our dynamic approach. Compared to a single-threaded approach, our dynamic approach consumes 79% less energy on average. Statically choosing twelve threads leads to a slightly lower energy consumption, however in practise this is not a thread-count that is typically chosen without an extensive analysis of the optimal thread-count for energy consumption, as we have performed in Section 4.

#Threads	Energy	Runtime
1	-79%	-137%
8	-10%	-39%
12	4%	0%
16	-5%	7%

Table 8: Energy consumption of the dynamic adaptation algorithm compared to static approaches on the nine-point stencil operation.

Rust implementation For the Rust implementation of the matrix multiplication algorithm, Table 9 shows that with thread pinning enabled we save a significant amount of energy compared to statically choosing eight threads and are on par when choosing sixteen threads. However with thread pinning disabled we save a significant amount of energy compared to running at sixteen threads, whereas we are on par when choosing eight threads. This shows that even for the same algorithm, it might be unclear which thread-count is optimal, and that in either case a dynamic approach can aid in decreasing energy consumption.

From this evaluation we conclude that improvements in energy consumption can be significant in cases where a poor choice was made for the static thread-count. Even in cases where the static choice was the optimal one, the dynamic

#Threads	Thread pinning	Energy	Runtime
8	Enabled	-28%	-67%
12	"	2%	3%
16	"	-3%	-1%
8	Disabled	-2%	-6%
12	"	-2%	-1%
16	"	-6%	-5%

Table 9: Energy consumption of the dynamic adaptation algorithm compared to static approaches on the matrix multiplication algorithm in Rust.

approach results in a similar energy consumption, excluding the outlier case of a 1500×1500 matrix multiplication in SaC, for which our approach found a local optimum. Compared to static approaches, the dynamic approach can save up to 79% of energy. In cases where a reasonable, but incorrect, choice was made for the thread-count our approach saves up to 21% energy in the tested benchmarks.

7 Related Work

7.1 Dynamic adaptation

Besides the runtime-based dynamic adaptation algorithm by Stuart et al. [18], there exist other approaches that aim to steer the resource allocation of a system through dynamic control, in an attempt to improve the runtime performance of programs.

Grand Central Dispatch (GCD) is a technology developed by Apple that aids developers in writing parallel programs [38,33,17]. GCD is integrated into the host operating system, and provides a holistic approach for thread management and execution, shifting this load from the developer to the scheduler. However, GCD still requires some manual instrumentation from developers. They must define blocks of code that they dispatch to GCD synchronously or asynchronously, using one of several types of waiting queues. Parallel Dispatch Queues (PDQ) are a similar programming abstraction [8]. By synchronizing messages in a queue, it aims to reduce the overhead caused by acquiring and releasing synchronization primitives, as well as preventing busy waiting within handlers. These approaches however only optimise for runtime performance and do not consider energy-efficiency.

A related approach is Dynamic Voltage Frequency Scaling (DVFS). Instead of trying to increase energy-efficiency or runtime performance by instrumenting the software, this approach aims to instrument the hardware by scaling the voltage supplied to the CPU. Dynamic programming approaches have been applied to further improve the effectiveness of DVFS [24].

7.2 Profile-based optimisation

Profile-based optimizations use the results of profiling test runs to select an optimal implementation of an algorithm for the given input data or hardware configuration. A prominent example of this is FFTW3, which selects one of many discrete Fourier transformation implementations through profiling [10]. They achieve this with the use of a special-purpose compiler, that generates optimised code for the given hardware in a planning phase. It would be interesting to investigate whether similar results can be obtained when optimising for energy-efficiency during this planning phase.

Profile-based optimisation of virtual machine scheduling is a popular avenue of research in the context of data-centres [7,5,6,32]. These approaches focus on the development of resource allocation policies and scheduling algorithms that aim to decrease the carbon footprint of data-centres without compromising on the required quality of service.

7.3 Static methods for optimising energy

Although there exists a multitude of compiler optimisations that aim to minimise the runtime of programs, not many optimisations exist that aim to specifically reduce energy consumption [34]. Pallister et al. have provided several optimisation that specifically aim to reduce the energy consumption of programs, in the context of embedded devices [35,36]. As have we, they have found that for optimal energy-efficiency gains a vertical integration process that exploits hardware-specific energy characteristics is needed, to bridge the gap between hardware and software.

Bangash et al. have investigated byte-code transformations in the context of Android applications, determining whether certain (combinations of) transformations lead to in increase or decrease in energy consumption [3]. They found that certain combinations of byte-code transformations can actually increase energy consumption, whereas choosing the right combination of transformation can lead to a reduction of up to 11% in energy consumption.

7.4 Energy visualisation & analysis

One of the key requirements in allowing developers to increase the energy-efficiency of their software is the ability to visualise and analyse the energy consumption of their code [40]. There exist semantics for programming languages that can calculate the energy consumption of their programs, and allow for calculation of break even points of algorithms [15,13,12]. Making these semantics work for a full language is hard, as can be seen in [14].

Another approach is to use model checking to detecting energy bugs and hotspots in control software [16]. The control software targetted with this model checking does have key interaction between software and hardware made explicit in the source code This is not the case for our target domain, in which all interactions are implicit. This makes using model checking harder.

Using the same explicit interaction is used in [30], in which energy consumption graphs are generated next to control software source code inside an IDE. Although it allows for near instant feedback (couple of seconds delay) to programmers, which is great for improving the code, it does not fit our target domain.

7.5 Energy-efficiency programming strategies

There is a handful of programming patterns that are known to improve the energy-efficiency of programs. These include application-level techniques such as caching, buffering, and batching, but also system-wide techniques such as retention policies and data compression. The effectiveness and adoption of such techniques have been previously investigated [27,11].

Another prominent example of a well-known pattern for reducing energy consumption is load balancing. Multiple approaches exist, however these make generalized assumptions about the energy-efficiency of the hardware. Kistowski et al. have compared a variety of load distribution strategies, and propose a new strategy that reduces a system's energy consumption even further [29].

8 Conclusion

In this paper we present a dynamic adaptation algorithm aimed at minimising the energy consumption of data-parallel applications. It is based on the observation that data-parallel applications typically apply the same parallel operations in a repetitive fashion. This property allows for a dynamic feedback loop that reacts on changes in energy consumption from one repetition to the next. Our experiments show that such a non-intrusive approach can be very effective while requiring no in-depth program analysis, involving no programmer effort at all, and causing negligible overhead. Looking at three different use cases with dynamically changing behaviour, we can see that our adaptation approach in most cases gets within 10% of a manually generated, ideally adapting runtime. The only remarkable outlier is the case of `matmul` where there exists a global minimum in the extreme case which our adaptation algorithm fails to find. We also observe that the dynamic approach for any chosen fixed number of threads yields energy improvements. Depending on the number of threads chosen, the gains are up to 79%. Surprisingly, we also notice that our energy-optimising algorithm improves slightly over the runtime-optimising dynamic algorithm of [18]. This reflects our static analysis which show that while there is a correlation between overall runtime and energy use, the energy minima usually are reached before the runtime minima when increasing the number of threads. Overall, we show that our approach provides a feasible method for lowering the energy footprint of functional data-parallel applications solely based on code generation without the need of any user intervention.

Whilst our adaptation algorithm proves effective across a range of data-parallel workloads, future work could explore the capability of the algorithm to adapt to changes in runtime conditions caused by other processes running in the background. Furthermore it might be interesting to investigate whether the same approach can be applied so switch between devices in a heterogeneous system, for example to switch between a CPU and GPU implementation of an algorithm. Beyond the array based domain, there is potential for applying this approach in other contexts that handle a large amount of similar computational complexity. For example, by adjusting resources dynamically in response to workload demands, web-servers could decrease their energy consumption, potentially without compromising on responsiveness.

Acknowledgements

We thank Thomas Koopman for proofreading the paper and providing helpful suggestions to improve it.

References

1. Aaldering, J.: Dynamic adaptation controller repository (2024), `https://github.com/JordyAaldering/mtdynamic`, branch ifl24, commit f54a630
2. Aaldering, J., Scholz, S.B., van Gastel, B.: Type patterns: Pattern matching on shape-carrying array types. In: Proceedings of the 35th Symposium on Implementation and Application of Functional Languages. IFL '23, Association for Computing Machinery, New York, NY (2024). `https://doi.org/10.1145/3652561.3652572`, `https://doi.org/10.1145/3652561.3652572`
3. Bangash, A.A., Ali, K., Hindle, A.: A black box technique to reduce energy consumption of android apps. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results. pp. 1—-5. ICSE-NIER '22, Association for Computing Machinery, New York, NY (2022). `https://doi.org/10.1145/3510455.3512795`, `https://doi.org/10.1145/3510455.3512795`
4. Buchty, R., Heuveline, V., Karl, W., Weiss, J.P.: A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators. Concurren cy and Computation: Practice and Experience **24**(7), 663–675 (2012). `https://doi.org/10.1002/cpe.1904`, `https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1904`
5. Buyya, R., Beloglazov, A., Abawajy, J.: Energy-efficient management of data center resources for cloud computing: a vision, architectural elements, and open challenges (2010), `https://arxiv.org/abs/1006.0308`
6. Dai, X., Wang, J.M., Bensaou, B.: Energy-efficient virtual machines scheduling in multi-tenant data centers. IEEE Transactions on Cloud Computing **4**(2), 210–221 (2016). `https://doi.org/10.1109/TCC.2015.2481401`, `https://doi.org/10.1109/TCC.2015.2481401`
7. Ding, Z.: Profile-based virtual machine placement for energy optimization of data centers. Ph.D. thesis, Queensland University of Technology (2017)
8. Falsafi, B., Wood, D.: Parallel dispatch queue: a queue-based programming abstraction to parallelize fine-grain communication protocols. In: Proceedings Fifth

International Symposium on High-Performance Computer Architecture. pp. 182–192. IEEE, Piscataway, NJ (1999). https://doi.org/10.1109/HPCA.1999.744362, https://doi.org/10.1109/HPCA.1999.744362
9. Falsafi, B., Guerraoui, R., Picorel, J., Trigonakis, V.: Unlocking energy. In: 2016 USENIX Annual Technical Conference (USENIX ATC 16). pp. 393–406. USENIX Association, Denver, CO (6 2016), https://www.usenix.org/conference/atc16/technical-sessions/presentation/falsafi
10. Frigo, M., Johnson, S.G.: The design and implementation of fftw3. Proceedings of the IEEE **93**(2), 216–231 (2005). https://doi.org/10.1109/JPROC.2004.840301, https://doi.org/10.1109/JPROC.2004.840301
11. Funke, M., Lago, P., Adenekan, E., Malavolta, I., Heitlager, I.: Experimental evaluation of energy efficiency tactics in industry: Results and lessons learned. In: 21st IEEE International Conference on Software Architecture (ICSA). pp. 1–12. IEEE, Piscataway, NJ (2024)
12. van Gastel, B.: Assessing sustainability of software: analysing correctness, memory and energy consumption. Ph.D. thesis, Open University, the Netherlands (2016)
13. van Gastel, B., van Eekelen, M.: Towards practical, precise and parametric energy analysis of it controlled systems. In: Bonfante, G., Moser, G. (eds.) Proceedings 8th Workshop on Developments in Implicit Computational Complexity and 5th Workshop on Foundational and Practical Aspects of Resource Analysis. EPTCS '17, vol. 248, pp. 24–37. Open Publishing Association (4 2017). https://doi.org/10.4204/EPTCS.248.7, https://doi.org/10.4204/EPTCS.248.7
14. van Gastel, B., van Eekelen, M.: Towards practical, precise and parametric energy analysis of it controlled systems. Electronic Proceedings in Theoretical Computer Science **248**, 24—-37 (4 2017). https://doi.org/10.4204/eptcs.248.7, http://dx.doi.org/10.4204/EPTCS.248.7
15. van Gastel, B., Kersten, R., van Eekelen, M.: Using dependent types to define energy augmented semantics of programs. In: van Eekelen, M., Dal Lago, U. (eds.) Foundational and Practical Aspects of Resource Analysis. pp. 20–39. FOPARA '15, Springer, Springer International Publishing, New York, NY (4 2016). https://doi.org/10.1007/978-3-319-46559-3_2, https://doi.org/10.1007/978-3-319-46559-3_2
16. van Gastel, P., van Gastel, B., van Eekelen, M.: Detecting energy bugs and hotspots in control software using model checking. In: Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming. pp. 93–98. Programming '18, Association for Computing Machinery, New York, NY (2018). https://doi.org/10.1145/3191697.3213805, https://doi.org/10.1145/3191697.3213805
17. Geeraerts, G., Heußner, A., Raskin, J.F.: Queue-dispatch asynchronous systems. In: 2013 13th International Conference on Application of Concurrency to System Design. pp. 150–159. IEEE, Piscataway, NJ (2013). https://doi.org/10.1109/ACSD.2013.18, https://doi.org/10.1109/ACSD.2013.18
18. Gordon, S., Scholz, S.B.: Dynamic adaptation of functional runtime systems through external control. In: Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages. pp. 10:1–10:13. IFL '15, Association for Computing Machinery, New York, NY (2015). https://doi.org/10.1145/2897336.2897347, https://doi.org/10.1145/2897336.2897347
19. Grelck, C.: Shared memory multiprocessor support for functional array processing in SaC. Journal of Functional Programming **15**(3), 353—401 (2005). https://doi.org/10.1017/S0956796805005538, https://doi.org/10.1017/S0956796805005538

20. Grelck, C.: Single assignment C (SaC) High Productivity Meets High Performance, pp. 207–278. CEFP '11, Springer Berlin Heidelberg, Berlin, Heidelberg (6 2012). https://doi.org/10.1007/978-3-642-32096-5_5, https://doi.org/10.1007/978-3-642-32096-5_5
21. Grelck, C., Scholz, S.B.: SaC – a functional array language for efficient multithreaded execution. International Journal of Parallel Programming **34**(4), 383–427 (8 2006). https://doi.org/10.1007/s10766-006-0018-x, https://doi.org/10.1007/s10766-006-0018-x
22. SaC group: SaC repository (2024), https://gitlab.sac-home.org/JordyAaldering/sac2c-mtdynamic, branch mtdynamic, commit 6cddce74
23. Guo, J., Thiyagalingam, J., Scholz, S.B.: Breaking the gpu programming barrier with the auto-parallelising SaC compiler. In: 6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11). pp. 15–24. ACM Press, New York, NY, USA (2011). https://doi.org/10.1145/1926354.1926359, https://doi.org/10.1145/1926354.1926359
24. Hajiamini, S., Shirazi, B., Crandall, A., Ghasemzadeh, H.: A dynamic programming framework for dvfs-based energy-efficiency in multicore systems. IEEE Transactions on Sustainable Computing **5**(1), 1–12 (2020). https://doi.org/10.1109/TSUSC.2019.2911471, https://doi.org/10.1109/TSUSC.2019.2911471
25. Huijben, R., Aaldering, J., Achten, P., Scholz, S.B.: Flattening combinations of arrays and records. In: Chang, S., Hemann, J. (eds.) Trends in Functional Programming. Springer, Springer International Publishing, New York, NY (2024)
26. Janssen, N., Scholz, S.B.: On mapping n-dimensional data-parallelism efficiently into gpu-thread-spaces. In: Proceedings of the 33rd Symposium on Implementation and Application of Functional Languages. pp. 54—66. IFL '21, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3544885.3544894, https://doi.org/10.1145/3544885.3544894
27. Jin, C., de Supinski, B.R., Abramson, D., Poxon, H., DeRose, L., Dinh, M.N., Endrei, M., Jessup, E.R.: A survey on software methods to improve the energy efficiency of parallel computing. The International Journal of High Performance Computing Applications **31**(6), 517–549 (2017). https://doi.org/10.1177/1094342016665471, https://doi.org/10.1177/1094342016665471
28. Khan, K.N., Hirki, M., Niemi, T., Nurminen, J.K., Ou, Z.: Rapl in action: Experiences in using rapl for power measurements. ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS) **3**(2), 1–26 (2018). https://doi.org/10.1145/3177754, https://doi.org/10.1145/3177754
29. von Kistowski, J., Beckett, J., Lange, K.D., Block, H., Arnold, J.A., Kounev, S.: Energy efficiency of hierarchical server load distribution strategies. In: 2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems. pp. 75–84. IEEE, Piscataway, NJ (2015). https://doi.org/10.1109/MASCOTS.2015.11, https://doi.org/10.1109/MASCOTS.2015.11
30. Klinik, M., van Gastel, B., Kop, C., van Eekelen, M.: Skylines for symbolic energy consumption analysis. In: ter Beek, M.H., Ničković, D. (eds.) Formal Methods for Industrial Critical Systems. pp. 93–112. FMICS '20, Springer, Springer International Publishing, New York, NY (9 2020). https://doi.org/10.1007/978-3-030-58298-2_3, https://doi.org/10.1007/978-3-030-58298-2_3
31. Macht, T., Grelck, C.: SaC goes cluster: Fully implicit distributed computing. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 996–1006. IEEE Computer Society, Los Alamitos, CA, USA (5 2019). https://

doi.org/10.1109/IPDPS.2019.00107, https://doi.ieeecomputersociety.org/10.1109/IPDPS.2019.00107

32. Minarolli, D., Freisleben, B.: Utility-based resource allocation for virtual machines in cloud computing. In: 2011 IEEE Symposium on Computers and Communications (ISCC). pp. 410–417. IEEE, Piscataway, NJ (2011). https://doi.org/10.1109/ISCC.2011.5983872, https://doi.org/10.1109/ISCC.2011.5983872
33. Nutting, J., Olsson, F., Mark, D., LaMarche, J.: Grand Central Dispatch, Background Processing, and You, pp. 455–487. Apress, Berkeley, CA (2014). https://doi.org/10.1007/978-1-4302-6023-3_15, https://doi.org/10.1007/978-1-4302-6023-3_15
34. Pallister, J.: Exploring the fundamental differences between compiler optimisations for energy and for performance. Ph.D. thesis, University of Bristol (2016), https://jpallister.com/documents/thesis_final.pdf
35. Pallister, J., Eder, K., Hollis, S.J.: Optimizing the flash-ram energy trade-off in deeply embedded systems. In: 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 115–124. IEEE, Piscataway, NJ (2015). https://doi.org/10.1109/CGO.2015.7054192, https://doi.org/10.1109/CGO.2015.7054192
36. Pallister, J., Hollis, S.J., Bennett, J.: Identifying compiler options to minimize energy consumption for embedded platforms. The Computer Journal **58**(1), 95–109 (11 2013). https://doi.org/10.1093/comjnl/bxt129, https://doi.org/10.1093/comjnl/bxt129
37. Quèze, F.: Power profiling with the firefox profiler (fosdem'23) (2 2023), https://archive.fosdem.org/2023/schedule/event/energy_power_profiling_firefox/, last accessed: 11-11-2024
38. Sakamoto, K., Furumoto, T.: Grand Central Dispatch, pp. 139–145. Apress, Berkeley, CA (2012). https://doi.org/10.1007/978-1-4302-4117-1_6, https://doi.org/10.1007/978-1-4302-4117-1_6
39. Scholz, S.B., Šinkarovs, A.: Tensor comprehensions in SaC. In: Proceedings of the 31st Symposium on Implementation and Application of Functional Languages. IFL '19, Association for Computing Machinery, New York, NY (2021). https://doi.org/10.1145/3412932.3412947, https://doi.org/10.1145/3412932.3412947
40. van der Steen, R., van Gastel, B.: The organizational hurdles of structurally reducing the energy consumption of software. In: BENEVOL. pp. 25–32 (2023)
41. Šinkarovs, A., Koopman, T., Scholz, S.B.: Rank-polymorphism for shape-guided blocking. In: Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing. pp. 1—-14. FHPNC 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3609024.3609410, https://doi.org/10.1145/3609024.3609410
42. Zhang, Z., Liang, S., Yao, F., Gao, X.: Red alert for power leakage: Exploiting intel rapl-induced side channels. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security. pp. 162—-175. ASIA CCS '21, Association for Computing Machinery, New York, NY (2021). https://doi.org/10.1145/3433210.3437517, https://doi.org/10.1145/3433210.3437517