Energy-Aware Dynamic Adaptation of Runtime Systems

Jordy Aaldering, Bernard van Gastel, and Sven-Bodo Scholz

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, aiming 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 our approach. We measure how well the adaptation algorithm adapts to runtime changes, and we evaluate its effectiveness compared to a hypothesised oracle that knows 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 converges towards an optimal level of parallelism that minimises energy consumption.

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

1 Introduction

The importance of software sustainability has grown rapidly in recent years, alongside an increasing awareness of environmental concerns. 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 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 algorithms 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 [4]. Furthermore, it has been shown that optimising for runtime is not always equivalent to optimising for energy-efficiency, and that optimising runtime can have a negative effect on energy consumption [3, 34].

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 and overall energy consumption. (Section 4)
- A dynamic adaptation algorithm that aims to minimise energy consumption of data-parallel programs at runtime through external control. (Section 5)
- An implementation of a thread-steering mechanism in the data-parallel functional language SaC, based on this 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 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 [20, 21]. It aims to generate high-performance parallel code for a wide range of multi-core and many-core architectures from a single source code.

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:

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

It constitutes an index vector `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 usage of `iv` in the inner expression, they may be left out. In the example, the upper bound would be inferred as the shape of `arr`. For each index vector in this range, the result is a selection into the array `arr` incremented by one.

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, these shape lengths can be replaced by variables, where identical variable names require identical lengths 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]`. A multi-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 are user-defined variables. In combination with variable-rank type patterns, this allows for the embeddings of complicated shape expressions within type patterns, simplifying the function bodies.

2.1 Parallelism in Single assignment C

The SaC compiler is able 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 to increase the granularity of parallelism and to improve the locality of code [21]. The multi-core back-end then generates code to execute all top-level tensor comprehensions of sufficiently large 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, threads are 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 coordinates all the other bees. The other bees wait until the start of a parallel section is 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 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 programs 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, separating the energy consumption is theoretically possible by extending the scheduler statistics, however to 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 to measure and control 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.

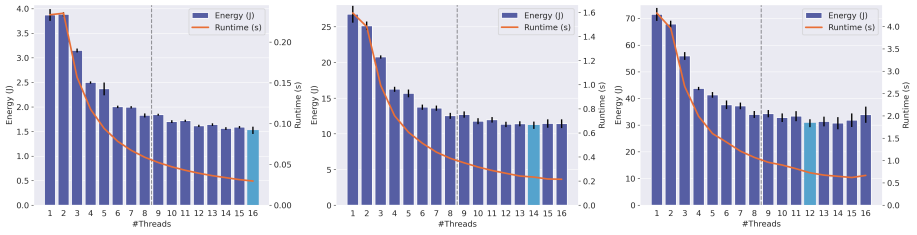
RAPL reports fine-grained and reliable energy consumption data across various CPU domains, such as the package, PSys, and DRAM [28]. RAPL operates using a running counter that tracks the cumulative energy consumed by each of these domains. The energy consumption in micro-Joules over a time interval is calculated as the difference in counter values at the start and end of that time period. RAPL is available on Intel CPUs since the Sandy Bridge architecture (2011), and AMD supports a similar feature starting with their Zen architecture (2017). In addition to its wide availability over the past decade, RAPL requires only slightly elevated privileges [42], making it accessible on most systems.

4 Energy consumption patterns

We perform a set of static experiments to determine which factors affect the energy consumption patterns of programs. 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 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, such as algorithm implementation, workload behaviour, and hardware characteristics. To illustrate this we investigate the effect these factors have on the energy consumption pattern of the N-body, nine-point stencil, and matrix multiplication 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.



(a) 10000 elements. (b) 25000 elements. (c) 40000 elements.

Fig. 1: Average energy consumption and runtime of 250 N-body simulation in SaC. Bars denote energy consumption, with corresponding values in the left y-axis. The line denotes runtime, corresponding to the right y-axis.

4.1 N-body simulation

As our first benchmark we measure the energy consumption and runtime of a SaC implementation of the N-body algorithm, given three different problem input sizes. 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. 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].

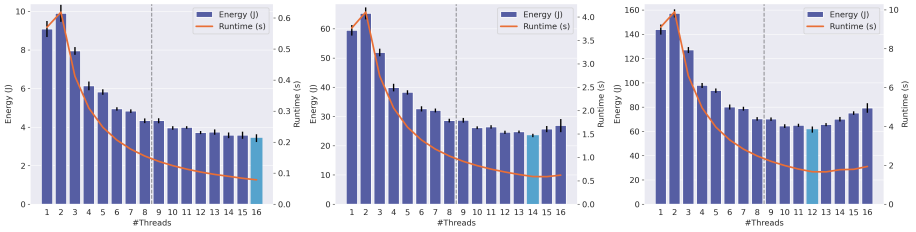
A single time step of the N-body simulation with delta-time `dt` on an array of bodies is defined as follows, where `acc` is a function that returns the acceleration vector between two bodies.

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

For these case studies only the relevant parallel codes are measured, namely the tensor comprehensions. The runtime and energy measurements start right before those parallel regions, and stop immediately after their synchronisation. The average runtime and energy consumption for the three input sizes are shown in Figure 1. Along the x-axis is the number of threads, ranging from one to sixteen. Bars denote the average energy consumption of a single time step, and the line denotes its average runtime.

For an N-body simulation with 10000 elements in Figure 1a we observe that although increasing the thread-count strictly decreases the runtime, for simulations with 25000 and 40000 elements in Figures 1b and 1c respectively, the energy consumption plateaus when using more than eight threads. It seems that any improvements in energy consumption gained by using hyper-threading are mitigated by the energy 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. This frees up the remaining threads to be used by other processes, without incurring a significant negative energy consumption impact on the N-body simulation. Although this increases the total runtime, it can conversely lead to energy savings.

An interesting observation, seen in the following benchmarks as well, is that increasing the thread-count has a greater effect on runtime performance than on energy-efficiency. This suggests that the overhead caused by having to manage multiple threads has a stronger negative effect on the energy consumption of a program than on its runtime performance, and that optimising for runtime performance has diminishing returns for decreasing energy consumption.



(a) 10000×10000 points. (b) 25000×25000 points. (c) 40000×40000 points.

Fig. 2: Average energy consumption and runtime of 250 nine-point stencil operations in SaC. Bars denote energy consumption, with corresponding values in the left y-axis. The line denotes runtime, corresponding to the right y-axis.

4.2 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 nowadays is in the training process of Convolutional 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.

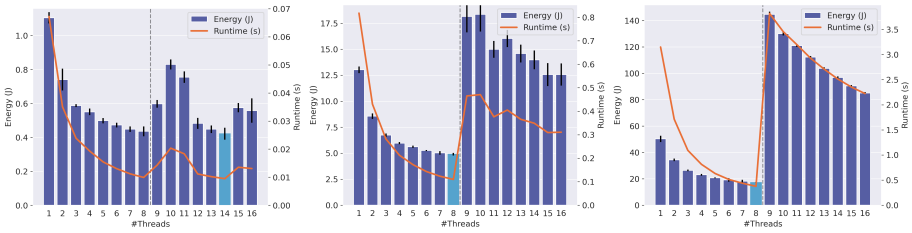
The nine-point stencil is defined as a weighted sum of each point in an array and its eight immediate neighbours, where each weight depends on the corresponding value in a 3×3 array of weights. It is defined in SaC as:

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

For a 10000×10000 input matrix in Figure 2a 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 both the energy-efficiency and runtime performance. We see this in Figures 2b and 2c respectively, where as the input size increases, the optimum thread-count decreases.

4.3 Matrix multiplication

The matrix multiplication algorithm is used across a variety of domains, from solving linear equations, to machine learning, to computer graphics. Whereas more sophisticated approaches aim to have consistent cache locality [41], the



(a) 500 × 500 matrix. (b) 1000 × 1000 matrix. (c) 1500 × 1500 matrix.

Fig. 3: Average energy consumption and runtime of 250 naive matrix multiplications in SaC. Bars denote energy consumption, with corresponding values in the left y-axis. The line denotes runtime, corresponding to the right y-axis.

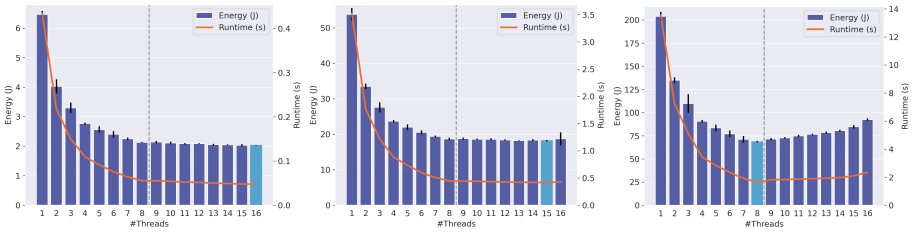
cache locality of the naive approach decreases as the input size increases, shifting the bottleneck from CPU to memory. We implement the naive algorithm, since this behaviour provides us with an interesting case study.

The 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] }) };
}
```

Due to cache behaviour of the naive implementation, manual tuning is required to ensure an efficient thread layout. When using eight or fewer threads, an interleaved mode is optimal, ensuring each thread executes on a different physical core. However, since the system only has eight physical cores, using nine or more threads requires multi-threading. At this point an interleaved mode is no longer beneficial as it increases the amount of false sharing, drastically decreasing performance. It should be noted that choosing a different thread pinning strategy leads to significantly different observations, and that choosing an efficient thread layout is crucial to the performance of this benchmark.

The results of a 500 × 500 matrix multiplication in Figure 3a align with our expectations; increasing the number of threads up to eight threads improves both the energy-efficiency and the runtime performance. Using more threads causes a decrease in performance, likely due to additional thread management overhead. This overhead is somewhat overcome by the increase in thread-count, making a thread-count of fourteen threads similar in performance to a thread-count of eight. However, as observed in Figures 3b and 3c, increasing the matrix size to 1000 × 1000 and 1500 × 1500 changes this energy consumption pattern, resulting from the shift of being CPU-bound to being memory-bound. Even with an efficient thread pinning strategy, using more than eight threads causes a significant increase in both the energy consumption and runtime, making those approaches worse for energy-efficiency than even a single-threaded approach.



(a) 500 × 500 matrix. (b) 1000 × 1000 matrix. (c) 1500 × 1500 matrix.

Fig. 4: Average energy consumption and runtime of 250 naive matrix multiplications in Rust. Bars denote energy consumption, with corresponding values in the left y-axis. The line denotes runtime, corresponding to the right y-axis.

4.4 Implementation language

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. In Figures 4a and 4b we observe that up until eight threads, increasing the thread-count improves both the energy-efficiency and runtime. When using more than eight threads, both the energy consumption and runtime plateau, suggesting that we gain no additional benefits by using hyper-threading. As the matrix size increases to 1500 × 1500, there is a shift in the energy consumption pattern. As the number of threads increases from 8 to 16, the energy consumption gradually increases again. This suggests that the overhead associated with hyper-threading outweighs the benefits of further parallelisation.

Interestingly, although the SaC and Rust implementations provide a similar level of performance when using nine or more threads, comparing the best-case scenarios we find that the SaC implementation performs roughly 4 times better in terms of energy consumption and roughly 5 times better in terms of runtime. As discussed previously, thread pinning and cache behaviour have a significant impact on the performance characteristics of this benchmark. Although we suspect that these are the main reasons for this performance difference, it is unclear why exactly this behaviour occurs, complicating the task of determining an optimal thread-count and emphasising the need for a dynamic approach.

5 Energy-Aware Dynamic Adaptation

In Section 4 we determine a multitude of running conditions that affect the selection of a thread-count for maximal energy-efficiency. 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 energy overhead. Consequently, we look for an solution that avoids the need for manual tuning by dynamically adapting to runtime changes.

The overall idea is based on the observation that data-parallel codes typically perform several repetitions of the same parallel computation. This behaviour allows dynamic adaptation to tune the parallelism between such repetitions [18]. Our adaptation algorithm monitors changes in the energy consumption from one repetition to the next, and periodically adjusts the recommended thread-count 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 recomputes the recommended thread-count at a fixed interval; waiting for a fixed amount of energy measurements to arrive. Each of these energy measurements is the energy consumed during a single iteration of the parallel region, measured using RAPL (Section 3). The frequency is configurable, but we find 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. 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, corresponding to a value of 1 or -1. The step size gives the amount of threads that will be added or removed from the current thread-count, respectively.

When the median energy consumption of an iteration differs more than a factor α compared to the energy consumption of the previous iteration, we hypothesise that there is likely a change in workload behaviour or system configuration. To quickly converge to the new optimal thread-count, the step size is reset to half the number of maximum threads. For our benchmarks we observe that a value of 0.5 for α works well. Otherwise, if there was an increase in energy consumption that is less than factor α, this means that the previous iteration was more energy-efficient. We predict that the optimal thread-count is somewhere in between the current thread-count and the previous thread-count. Therefore the step direction is inverted, and 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 step size that nears zero as we get close to this optimum, as opposed to a positive integer step size that always results in a change to the thread-count. Whereas typically the step size is halved in each iteration, as it reaches zero we gradually decrease it by a lower amount, ensuring that changes to the thread-count do still occur occasionally. This change to the step size is defined by a function $\Delta t \,/\, (\Delta t + \beta)$, where a lower value for β results in smaller changes to the step size. For our benchmarks we find that a value of 0.85 for β works well.

Eventually the step size reaches zero, and the thread-count will stagnate. In an attempt to escape potential local optima, and to adapt to subtle changes to the energy consumption pattern of a program, the step size is reset to half the maximum number of threads when it becomes less than γ. The lower the value of γ, the more iterations are required until the step size is reset. Given our choice for β, a value of 0.155 for γ works well for our benchmarks.

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

Algorithm 1 Algorithm for repeatedly updating the thread-count n_t based on energy measurements S, using step direction $\hat{d}$ and step size Δt.

$\hat{d} \leftarrow -1$
$\Delta t \leftarrow m_t \,/\, 2$
$E_{old} \leftarrow 0$
loop
  $S \leftarrow$ Wait for samples
  $E_{new} \leftarrow median(S)$
  if $E_{new} \,/\, E_{old} < (1 - \alpha) \vee E_{new} \,/\, E_{old} > (1 + \alpha)$ **then**
    $\hat{d} \leftarrow sign(m_t - 2n_t)$
    $\Delta t \leftarrow m_t \,/\, 2$
  else
    if $E_{new} > E_{old}$ **then**
      $\hat{d} \leftarrow -\hat{d}$
    if $\Delta t > \gamma$ **then**
      $\Delta t \leftarrow max(\Delta t \,/\, 2,\ \Delta t \,/\, (\Delta t + \beta))$
    else
      $\hat{d} \leftarrow sign(m_t - 2n_t)$
      $\Delta t \leftarrow m_t \,/\, 2$
  $E_{old} \leftarrow E_{new}$
  $n_t \leftarrow n_t + \hat{d} \cdot \Delta t$
  $n_t \leftarrow max(1,\ min(m_t,\ n_t))$

A drawback of this approach is that it contains manually tuned variables, which might need reconfiguration in different contexts. The main difficulty lies in the variation in energy-efficiency between hardware systems, and in run-to-run variance due to input-dependent behaviour and changes to the software environment. In future work we aim to investigate whether, within this context, a control theory-driven design technique can help mitigate this drawback [9].

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 solution 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 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, respectively. 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 semantics of the program. Instead of SaC's standard busy wait, we use a semaphore to ensures that the threads of sleeping bees incur no additional energy consumption, and are free to be used by other processes on the system. The queen bee wakes sleeping bees sending a message to their semaphore, after which these bees enter the busy wait state again, waiting for the signal 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 with an empty body. The measured sources of overhead include 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 $48\mu J$ and a runtime overhead of $5\mu s$. Considering that even a single 500×500 matrix multiplication consumes around half a Joule in the best case, 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

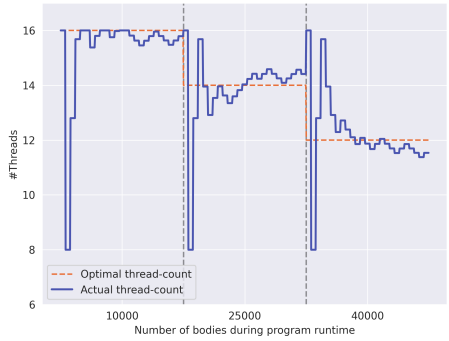


Fig. 5: Adaptation quality on the SaC implementation of the N-body simulation.

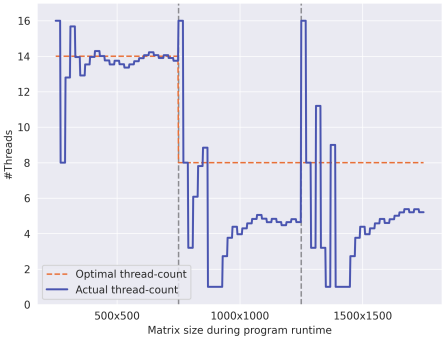


Fig. 6: Adaptation quality on the SaC implementation of the matrix multiplication algorithm.

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 often 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 changes to the optimal thread-count. A key requirement for the adaptation algorithm is its ability to adjust to those 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 to the examples from Section 4, and assessing how effectively the adaptation algorithm converges to the thread-count with minimal energy consumption.

Figure 5 shows the actual thread-count suggested by the adaptation algorithm across the runtime of the N-body simulation, alongside the optimum thread-count determined in Section 4. The x-axis represents the iterations of the program, where the input data size, denoted by the x-axis labels, changes at fixed intervals. The orange dashed line shows the optimal thread-count derived from the observations in Section 4, and the blue line shows the thread-count suggested by the adaptation algorithm. The closer the suggested thread-count comes to the derived optimum, the better the adaptation quality of the algorithm.

In Figure 5 we observe that the energy consumption plateaus when using eight or more threads. Thread-counts between eight and sixteen threads are close in terms of energy-efficiency. Even so, we see that the adaptation algorithm is able to find the optimum thread-count for the N-body simulation for each

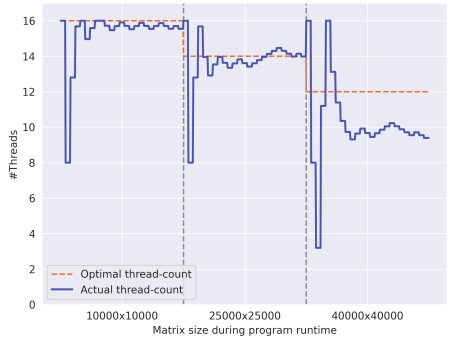


Fig. 7: Adaptation quality on the SaC implementation of the nine-point stencil operation.

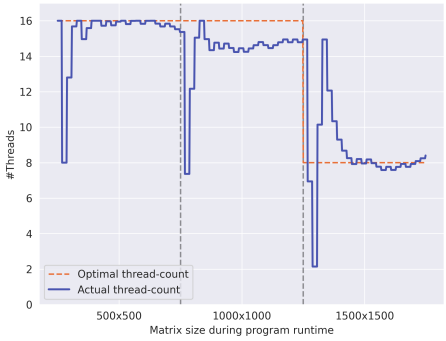


Fig. 8: Adaptation quality on the Rust implementation of the matrix multiplication algorithm.

of the three input sizes. For brevity we exclude the N-body simulation from the remaining benchmarks, as it provides a slightly better but similar level of performance compared to the nine-point stencil benchmark, and presents no further noteworthy results or insights.

The SaC implementation of the matrix multiplication algorithm presents an interesting challenge. As discussed in Section 4, choosing an effective thread pinning strategy is crucial to performance. Whereas an interleaved pinning strategy is optimal when using eight or fewer threads, a contiguous strategy is more effective when using more than eight threads. A drawback of our approach is that threads are disabled in a contiguous order, which results in a sub-optimal pinning strategy for this benchmark. In Figure 6 we see that for 500×500 inputs we converge to the optimal thread-count at fourteen threads, even though the local optimum at eight threads only consumes slightly more energy. However, for the other two input sizes the recommended thread-count around four threads, instead of eight. This happens because the dynamic algorithm disables threads in a contiguous order, resulting in a poor thread pinning strategy.

In Section 4 we observe for the nine-point stencil benchmark that as the input data size increases, the optimal thread-count gradually decreases. Applying the adaptation algorithm we see in Figure 7 that it is able to converge to the optimal thread-count in these cases. Although for 40000×40000 inputs the algorithm suggests a thread-count of ten instead of twelve threads, Figure 2c shows that the energy consumption at these two thread-counts are within margin of error.

Figure 8 shows the adaptation quality for the Rust implementation. 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 1500×1500 matrix multiplication in SaC where we do however find the local optimum given the sub-optimal thread pinning strategy. 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.

Input	Energy	Runtime
10000^2	5%	4%
25000^2	5%	5%
40000^2	15%	33%

(a) Nine-point stencil.

Input	Energy	Runtime
500^2	8%	8%
1000^2	46%	93%
1500^2	68%	100%

(b) Matrix multiplication.

Input	Energy	Runtime
500^2	1%	1%
1000^2	1%	0%
1500^2	10%	16%

(c) Rust implementation.

Table 1: Energy and runtime difference of the energy-based approach compared to the theoretical best performance of the oracle-based approach.

The comparison to the nine-point stencil operation is shown in Table 1a. It shows that, for 10000×10000 and 25000×25000 inputs, the dynamic approach provides a similar level of performance compared to the oracle, with only a 5% increase in energy consumption. However for 40000×40000 inputs the energy consumption is increased by 15% compared to the oracle. In Figure 7 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. The decrease in energy-efficiency can be attributed to the additional energy cost introduced by the operating system, as well as the power required to keep the hardware operational (Section 3), which increases as the runtime increases. Considering these additional energy requirements due to an increase in runtime, a 15% increase in energy consumption compared to a 33% increase in runtime seems reasonable.

Table 1b 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 we are within 8% of the energy consumption and runtime of

the oracle-based approach. However, for the larger input sizes the performance deteriorates. As discussed in Section 6.3, this decrease in performance is due to the inability of our controller to switch its pinning strategy. A more sophisticated method might be able to provide a level of performance similar to the 500×500 input, and to the other three benchmarks.

For the Rust implementation of the matrix multiplication algorithm we see in Table 1c that the energy overhead of the dynamic approach is minimal. Even in the case where the dynamic approach has an 18% longer runtime, the increase in energy consumption is only 8%. In the other two cases, the energy-efficiency is within margin of error of the oracle.

These results demonstrate that while this theoretical oracle requires a level of manual tuning that is not feasible in practise, our dynamic approach provides a practical solution for reasonably approximating the theoretical optimal energy-efficiency in most cases.

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].

Input	Energy	Runtime
10000^2	-2%	-8%
25000^2	-4%	-3%
40000^2	7%	25%

(a) Nine-point stencil.

Input	Energy	Runtime
500^2	-7%	-13%
1000^2	10%	41%
1500^2	-3%	14%

(b) Matrix multiplication.

Input	Energy	Runtime
500^2	-2%	-3%
1000^2	-1%	-1%
1500^2	1%	-2%

(c) Rust implementation.

Table 2: Energy and runtime difference of the energy-based approach compared to the runtime-based approach.

In Table 2a 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. 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. Consequently, the increase in runtime introduces inherent additional energy consumption of the operating system and the hardware itself. Keeping this in mind, compared to the stark increase in runtime, a 7% increase in energy consumption seems reasonable.

In Table 2b we see that for the matrix multiplication benchmark our approach fares better against the runtime-based approach than compared to the

oracle. This is due to the fact that the runtime-based implementation is unable to select an effective pinning strategy. Interestingly, for 1500×1500 we observe that the energy-optimising approach decreases energy consumption, but increases runtime, showing that optimising for energy consumption is not necessarily equivalent to optimising for runtime.

In Section 4 we observe that the energy consumption and runtime performance of the Rust implementation of the matrix multiplication algorithm are strongly correlation. As expected, in Table 2c we see that the energy-based approach and runtime-based approach provide a similar level of performance, both in terms of energy-efficiency and runtime performance.

These results demonstrate that, while a runtime-based method 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 our energy-based approach.

6.6 Comparison to static approaches

Finally, we compare the performance of our dynamic approach to static approaches that run at typically effective thread-counts. Besides running single-threaded, typical choices for threat-count are based on the number of physical cores or the number of available threads. In Section 4 we observe that for our examples, choosing twelve threads often also leads to lower energy consumption.

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

(a) Nine-point stencil.

Threads	Energy	Runtime
1	-55%	-110%
8	41%	67%
12	-58%	-40%
16	-49%	-29%

(b) Matrix multiplication.

Threads	Energy	Runtime
1	-100%	-157%
8	1%	-3%
12	-2%	-4%
16	-7%	-7%

(c) Rust implementation.

Table 3: Energy and runtime difference of the energy-based approach compared to four fixed thread-count approaches.

Table 3a describes the difference in energy consumption and runtime between the dynamic adaptation algorithm and static thread-count methods. If a reasonable, but ineffective, choice in thread-count is made for the nine-point stencil operation we observe that 10% of energy can be saved with our dynamic approach. Compared to the single-threaded case, 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.

The results of the matrix multiplication example are shown in Table 3b. If a poor choice is made for the static thread-count, we save around 50% of energy, even without an optimal pinning strategy. However, when a static thread-count of eight threads is chosen, which we observed in Section 4 is always optimal or near-optimal, our approach consumes 41% more energy.

For the Rust implementation of the matrix multiplication algorithm, Table 3c shows that we save a significant amount of energy compared to a single-threaded approach or when running at sixteen threads, whereas the performance difference is within margin of error when using eight or twelve threads.

From this evaluation we conclude that improvements in energy consumption can be significant in cases where a reasonable but non-optimal choice was made for the static thread-count. When a program-specific and system-specific analysis is applied to determine an optimal static thread count, our dynamic approach still results in similar energy consumption, except for outlier cases such as the naive matrix multiplication where we observe an increase in energy consumption due to an inefficient thread pinning strategy. However, we argue that in general such an analysis is infeasible, and that our approach in most cases provides a reasonable solution for approximating these optimums.

7 Related Work

Dynamic adaptation algorithms. Besides the runtime-based dynamic adaptation algorithm [18], there are more approaches that steer the resource allocation of a system through dynamic control, in an attempt to improve runtime performance.

Grand Central Dispatch (GCD) is a technology developed by Apple that aids developers in writing parallel programs [17, 33, 38]. GCD is integrated into the host operating system and provides an integrated approach for thread management, 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 are dispatched 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 do not consider energy-efficiency, and only optimise for runtime performance.

Related 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].

Profile-based optimisation. Profile-based optimizations use the results of profiling runs to select an optimal implementation of an algorithm for the given input 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 through 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 by optimising for energy-efficiency instead.

Profile-based optimisation of virtual machine scheduling is a popular avenue of research in the context of data-centres [5–7, 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.

Static methods for optimising energy. Although there exists a myriad 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 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.

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 [12,13,15]. However, making these semantics work for a full language is hard [14].

Another approach uses model checking to detecting energy bugs and hotspots in control software [16]. The control software targetted with this model 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, energy consumption graphs can be generated next to control software source code inside an IDE [30]. Although it allows for near instant feedback to programmers, which is great for improving the code, it does not fit our target domain.

Energy-optimising 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 [11, 27].

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

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 programs typically apply the same parallel operations in a repetitive fashion. This property allows for a dynamic feedback loop that reacts to changes in energy consumption from one repetition to the next. Our experiments show that such a non-intrusive method, in a functional context where scaling the level of parallelism is straightforward, can be very effective while requiring no in-depth program analysis, involving no programmer effort at all and causing negligible overhead. Looking at four different use cases with dynamically changing behaviour, we can see that our adaptation approach in most cases gets within 15% of a manually generated, ideally adapting runtime. The only remarkable outlier is the case of matrix multiplication where the ideal case depends on the thread pinning strategy, which our adaptation algorithm fails to find. However, we also find that optimising these cases is highly context-dependent and that slight variations in configuration or implementation can introduce significant changes to the energy consumption characteristics of a program.

Furthermore, we observe that the dynamic approach yields energy improvements over static approaches. Depending on the number of threads chosen, the gains are up to 79% for any reasonable choice, and up to 100% if a poor decision is made. We also notice that in many cases our energy-optimising algorithm improves slightly over the runtime-optimising approach. This is reflected by our static analyses, which show that, while there is a correlation between overall runtime and energy use, the energy minima usually are reached before runtime minima when increasing the number of threads. Overall, we show that our method provides a feasible method for lowering the energy footprint of data-parallel applications solely based on automatic code instrumentation.

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 would be interesting to investigate whether the same approach can be applied to switch between devices in a heterogeneous system, such as CPU and GPU implementation of an algorithm, or to switch between thread pinning strategies. 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 and the anonymous reviewers for proofreading the paper and providing helpful suggestions to improve it.

References

1. Aaldering, J.: Dynamic adaptation controller repository (2025), https://github.com/JordyAaldering/mtdynamic/tree/tfp25, branch tfp25, commit fea397d
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
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
4. Buchty, R., Heuveline, V., Karl, W., Weiss, J.P.: A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators. Concurrency and Computation: Practice and Experience **24**(7), 663–675 (2012). https://doi.org/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
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
9. Filieri, A., Maggio, M., Angelopoulos, K., D'Ippolito, N., Gerostathopoulos, I., Hempel, Andreas Berndt nd Hoffmann, H., Jamshidi, P., Kalyvianaki, E., Klein, C., et al.: Software engineering meets control theory. In: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 71–82. IEEE (2015). https://doi.org/10.1109/SEAMS.2015.12
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
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: 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 (Apr 2017). 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 (Apr 2017). https://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: Foundational and Practical Aspects of Resource Analysis. pp. 20–39. FOPARA '15, Springer, Springer International Publishing, New York, NY (Apr 2016). 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
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
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
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
20. Grelck, C.: Single assignment C (SaC) High Productivity Meets High Performance, pp. 207–278. CEFP '11, Springer Berlin Heidelberg, Berlin, Heidelberg (Jun 2012). 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 (Aug 2006). 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
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
25. Huijben, R., Aaldering, J., Achten, P., Scholz, S.B.: Flattening combinations of arrays and records. In: International Symposium on Trends in Functional Programming. pp. 220–240. TFP '24, Springer Nature, Cham (Jan 2025). https://doi.org/10.1007/978-3-031-74558-4_10

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
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
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
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
30. Klinik, M., van Gastel, B., Kop, C., van Eekelen, M.: Skylines for symbolic energy consumption analysis. In: Formal Methods for Industrial Critical Systems. pp. 93–112. FMICS '20, Springer, Springer International Publishing, New York, NY (Sep 2020). 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 (May 2019). https://doi.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
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
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
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 (Nov 2013). https://doi.org/10.1093/comjnl/bxt129
37. Quèze, F.: Power profiling with the firefox profiler (fosdem'23) (Feb 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

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
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
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