Patching software inefficiencies

Arnas Venskunas 6427553
Vincent Ruijgrok 5692210
Radu Andrei Vasile 5744792
Nick van Luijk 5559944
Calin Georgescu 5497035

***Abstract*—This paper explores a novel set of guidelines to practically apply energy-saving optimizations at runtime for desktop applications. Currently, many tactics exist to optimize energy efficiency, but a central, practical, and industry-oriented set of guidelines is lacking. To address this gap, we conduct an investigation of prior research describing profiler techniques and academic optimization methods. Then, based on the research, we develop and incorporate a novel solution of machine learning-based reverse engineering and devise a set of general practical guidelines for desktop application engineers to follow. This, in turn, should reduce energy consumption stemming from insufficient optimization in desktop applications. We then validate this by applying the workflow to the "Scrap Mechanic" video game, showing a significant reduction in CPU usage as well as an average of 10-15% savings in energy consumption, proving the efficiency of the suggested framework.**

I. Introduction

In a world driven by technology, software that powers technological devices is an irreplaceable element of modern society. Today, infrastructure, finances, healthcare, and almost every other area of life are defined by software [1]. It was shown that major software disruption can have large societal consequences in life threatening situations and economic disasters [2]. It was indicated that such a scale and importance of technology cause significant environmental operating costs, as it was estimated that ICT represented 4.7% of global energy consumption in 2021 [3]. Despite this broad impact, energy inefficiencies at the software level are often overlooked, particularly in everyday applications where suboptimal design choices can lead to unnecessary resource consumption. Recently, research focusing on software energy efficiency has expanded significantly, providing various tools to measure, analyze, and compare the energy efficiency of software applications [4]. However, these approaches are often fragmented and lack practical guidance for developers [5]. This work aims to address this gap by proposing a structured set of guidelines aimed at optimizing energy efficiency for desktop applications.

II. Related Work

A. Energy measurement tools

Significant research was focused on devising tools to effectively measure energy consumption. Several CPU profilers were proposed in academia such as eprof [6] or IgProf [7]. The common feature of these energy profilers is that they are able to map energy usage to a specific function or process (granularity). This may prove crucial in identifying inefficiencies. Currently, there exist a variety of industry profilers, in the context of desktops, the manufacturers' AMD μProf [8] and Intel® VTune™ [9] seem to be the most common choice.

B. Energy usage optimization methods

Interestingly, reducing software execution time does not always result in reduced energy consumption [5]. According to Jelschen et al. [10], software systems can be optimized in four abstraction layers: *Hardware*, *Low-Level Software*, *Operating System*, *Application Software*, the latter sitting at the highest abstraction layer. Based on a survey conducted in 2024 by Martinez et al. where 163 tactics were identified across 142 studies, most of them focus on source code optimizations and dynamic monitoring. The paper concluded that at the time of its publication, the tactics derived were mostly of academic origin and of little industrial relevance.

III. Proposed solution

We propose an application-level optimization framework that combines energy profiling, reverse engineering, and targeted optimization to fight the problem as previously described. The goal is to systematically identify energy inefficiencies in compiled desktop applications and provide actual improvements without needing access to the source code.

The framework is made to be iterative, allowing developers to step-by-step improve application behavior while evaluating the impact of each step.

The proposed framework consists of four steps: profiling, hotspot identification, analysis, and optimization.

1) Energy Profiling: The first step is to measure the energy usage of an app using profilers. Tools such as AMD μProf and Intel VTune allow one to find out where in the app functions are called and CPU time is used. This gives the user a basic idea of where possible solutions for efficiency lie.

2) Hotspot Identification: With the profiling data in hand, the next step is to find the energy hotspots in the app: functions or paths that use too much energy for what they do. Typically, these hotspots have a high CPU usage, call the same function very often or are inefficient in their synchronization patterns.

3) Reverse Engineering and Analysis: Often, source code is unavailable. Therefore, the framework uses reverse engineering techniques to analyze compiled binaries. Disassembly and intermediate representations are used to find out program logic and get to inefficient constructs.

To accelerate this process, large language models (LLMs) are used for interpreting low-level code, recovering semantic meaning, and suggesting potential optimizations.

The analysis phase focuses on detecting patterns such as:

- Inefficient parallelization strategies
- Redundant computations
- Suboptimal synchronization primitives
- Excessive system calls or idle waiting

4) Optimization and Patching: Based on the analysis, targeted optimizations are applied. These may include modifying execution parameters, replacing inefficient constructs, or introducing more efficient scheduling strategies.

When source code is not accessible, optimizations are applied by patching the code the target process executes. Patches are applied non-destructively by injecting a dynamically linked library into the process, which then overwrites instructions the process' executable sections in memory. This enables runtime modification of application behavior without altering the original executable.

The newly written instructions may modify parameters a function was called with, replace the return value, invert branches, or completely replace instructions with code that does nothing. Function hooking techniques such as Import Address Table (IAT) Hooking and Trampoline Functions are sometimes used to achieve this.

Each optimization is validated through repeated profiling to ensure that energy consumption is reduced without negatively impacting functionality or user experience.

A. Expected Impact

By combining profiling, reverse engineering, and targeted optimization, the framework enables developers to uncover inefficiencies. The framework is especially effective in scenarios involving complex, multi-threaded applications where energy inefficiencies arise from synchronization overhead and suboptimal workload distribution.

IV. Application

A. Scrap Mechanic

To show the effectiveness of our optimization flow, we've applied it to the video game Scrap Mechanic. This video game was chosen because of its unusually high CPU usage across all cores, even in empty worlds, indicating Scrap Mechanic is able to be optimized further.

1) Running the game with a profiler: We attached the profiler AMD μProf[1] to a running instance of Scrap Mechanic to discover where the CPU is spending large amounts of time. This profiler was chosen, as we used a CPU from AMD. For Intel CPUs, the Intel® VTune™ Profiler[2] could be used instead.

The computer used for the profiling and benchmarking contains an AMD Ryzen™ 9 7950X3D Gaming Processor[3] with 16 cores and 32 threads, an MSI GeForce RTX™ 5090 32G VENTUS 3X OC[4] for the GPU, and 64 gigabytes of RAM (G.SKILL Trident Z5 Neo RGB DDR5-6000 CL30-38-38-96[5]).

To create a reproducible test environment, we created a new Creative Mode world with terrain enabled. Once the world was created, the player was not moved. The frames per second (FPS) limit was set to 1000 using the `/frameratecap 1000` chat command, which is the maximum supported value. Measurements were done for a duration of 30 seconds at a resolution of 3840x2160 with all settings set to their highest value.

2) Interpreting profiler results: After running the profiler, we noticed that most of the time was spent inside the kernel at `ntoskrnl.exe!0x1406be1b4`. Immediately after that, the functions `ntdll.dll!0x180162114` and `ntoskrnl.exe!0x14033cc68` take a significant amount of CPU time, followed by numerous functions from `concrt140.dll`. This dynamically linked library (DLL) is distributed by the Microsoft Visual C++ (MSVC) runtime, and exports functions used for running other functions concurrently.

By attaching x64dbg[6], a debugger for native Windows applications, to the game's process, we were able to see the disassembly of the function at `ntdll.dll!0x180162114`. This turned out to be a wrapper for `ZwDelayExecution`, a syscall that tells the Windows kernel the thread is to be suspended until a condition is met.

3) Reverse engineering with LLMs: The profiler results also show a large portion of CPU time is spent inside `scrapmechanic.exe!0x1406df8e4`, `scrapmechanic.exe!0x1406df8c1` and `scrapmechanic.exe!0x1406df94e`. In order to understand what is going on in these functions, we used Binary Ninja[7] to disassemble the game's functions into x86_64 assembly instructions, and lifted them into HLIL, a pseudocode representation that's easier to understand. Other disassemblers such as IDA[8] or the open source Ghidra[9] could also be used.

These pseudocode representations are however still lacking any function names, variable names and structures. Reverse engineering these by hand is very time-consuming, so we used an LLM to speed this up.

The Rikugan plugin for Binary Ninja[10] was used for its rich integration with the Binary Ninja user interface. This plugin contains built-in tools and skills operated by an AI agent to aid with reverse engineering. It was chosen over other plugins, as it had an order of magnitute more stars on GitHub than all other Model Context Protocol (MCP) server plugins.

Rikugan supports multiple LLM providers and requires an API key to perform automated reverse engineering. The

[1]https://www.amd.com/en/developer/uprof.html

[2]https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[3]https://www.amd.com/en/products/processors/desktops/ryzen/7000-series/amd-ryzen-9-7950x3d.html

[4]https://www.msi.com/Graphics-Card/GeForce-RTX-5090-32G-VENTUS-3X-OC

[5]https://www.gskill.com/product/165/390/1661410135/F5-6000J3038F16GX2-TZ5NR

[6]https://x64dbg.com/

[7]https://binary.ninja/

[8]https://hex-rays.com/

[9]https://github.com/NationalSecurityAgency/ghidra

[10]https://rikugan.reversing.codes/

OpenAI API-based pricing and Anthropic API-based pricing are, however, far more expensive to use than their subscription counterparts. We already had a ChatGPT Plus subscription, providing access to Codex, an AI agent capable of performing software engineering tasks. The Codex CLI exposes an API through `codex app-server` that allows it to be embedded inside of other tools[11]. We used Codex to create a fork of Rikugan that uses the Codex app-server instead of its own agent such that we were able to use the subscription-based pricing. This fork has been published at https://github.com/TechnologicNick/Rikugan.

GPT 5.4 was given the profiler results and tasked with reverse engineering the most expensive functions, figuring out why they are expensive, and suggest how we can optimize them. It quickly discovered the hottest instructions inside of ScrapMechanic.exe are `lock inc` and `lock xadd` inside of a parallel for loop.

We verified this by manually going through the Xrefs of this function to see what other functions cross reference or call `scrapmechanic.exe!0x1406df8e4`, `scrapmechanic.exe!0x1406df8c1` and `scrapmechanic.exe!0x1406df94e`. All Xrefs discovered by Binary Ninja call functions from `CONCRT140.dll` such as `Concurrency::details::_SpinWait<1>::_SpinOnce` and `Concurrency::_Trace_ppl_function(Concurrency::PPLParallelForEventGuid, 4, 2)`, confirming GPT 5.4's findings.

4) Creating a patch: Our hypothesis was that the overhead of splitting up the work and joining the threads is higher than doing the work on fewer threads.

We tested this hypothesis by changing the CPU affinity mask using `start /affinity F ScrapMechanic.exe`. This allows the entire process to use only cores 0, 1, 2 and 3, instead of all 32 cores of the CPU.

This showed immediate results, as the amount of frames per second rendered at 1440p increased from approximately 400 to 800. We later discovered this was partially due to the game forcefully lowering the graphics settings when it detects only 4 cores are available. CPU usage also reduced from approximately 70% on 32 cores to almost 100% on 4 cores, significantly reducing the core average.

For a more permanent solution, we created a dynamically linked library (DLL) that is automatically injected into the Scrap Mechanic process using SMInjector[12]. When our DLL is loaded, it performs and Import Address Table (IAT) hook by replacing the pointer to `Concurrency::GetProcessorCount` with a pointer to a custom implementation. Every time the game attempts to get the processor count, it now hits our custom implementation instead.

[11]https://developers.openai.com/codex/app-server

[12]https://github.com/TechnologicNick/SMInjector

We only want to override the processor count in the code path that creates the worker threads, as we do not want the game to forcefully lower the graphics settings. Through manual reverse engineering with Binary Ninja, we discovered that only when `Concurrency::GetProcessorCount` is called from the `TaskManager` constructor, should the return value be overwritten. This function is identified at runtime by scanning the executable sections for a reference to a string that's uniquely used by this function. When our hook is reached, and the return address lies within the `TaskManager` constructor, do we overwrite the return value. This way, we only limit the amount of worker threads for the parallel for loops, allowing other subsystems, such audio, to keep using the other cores.

To test if our patch was successful, we attached AMD μProf again with the same test setup. This time, however, with our patch applied, set to use a single thread. Functions from `ntoskrnl.exe` are no longer found in the top 300 of most expensive functions. The most expensive function is now `ntdll.dll!0x180162114`, which we previously annotated as a wrapper for `ZwDelayExecution`. The amount of CPU time spent inside functions from `concrt140.dll` has also been reduced. Notably, a new DLL has shown up in the top 10, `nvwgf2umx.dll!0x18059defa`. This is a driver for the graphics card. Overal, these findings, comibined with the performance improvements and reduced CPU usage, show that the patch was successful.

The final patch and benchmarking scripts have been published under the MIT license at https://github.com/TechnologicNick/SMInjector/tree/main/PluginDevFolder/Scraptifine

V. Results

In order to determine the effectiveness of our proposed solution, we will now compare the performance of Scrap Mechanic with different configurations of the developed patch.

Using Scrap Mechanic, we analyse the framework from multiple perspectives, measuring the performance benefits in game, total CPU time spent in processes as well as energy and power usage. We define performance as the number of Frames Per Second (FPS) observed while the game is running. First, we measure the impact of the number of logical cores on the performance of the application, where we demonstrate that the optimal performance is achieved from limiting to a single core, rather than keeping the default value of 32. Second, we further compare the optimized result with the default value, by capping performance impacting metrics (the FPS and resolution), resulting in less CPU usage and energy consumed. Overall, we demonstrate that using the appropriate number of cores eliminates a large part of CPU usage, by requiring less synchronisation and the ability to allocate more resources towards actual computing tasks. A direct implication of this result is less energy used by the machine while executing tasks, which can have a large impact on the world when generalised to many applications.

1) Impact of number of cores on game performance: We perform this analysis by choosing multiple values for the number of cores, namely: 1, 2, 3, 4, 5, 6, 8, 16, and 32. The results show a clear, mostly linear relationship between thread count and all metrics. We prove that, when using a small amount of cores (6 or less), the CPU performs most efficiently, relying on less thread synchronisation, thus reducing the number of system calls for this matter. There is evidence of a bottleneck when using only one core, but it is hardly noticeable in terms of all metrics. The best overall performance is noted for 4 cores, balancing both performance and energy consumption. When comparing to the default version, which uses all 32 threads, we notice an increase in FPS from 570 to 680, almost 20% on creative mode, and from 337 to 366, an 8.6% performance boost in survival. A larger difference is observed in CPU usage, where a substantial, almost sixfold increase is recorded for the non-patched version, from 11% to 61%. In terms of energy consumption, the difference is also clear, showing an almost 60% increase in measured CPU energy and power when running the game on the default version, compared to the optimised one.

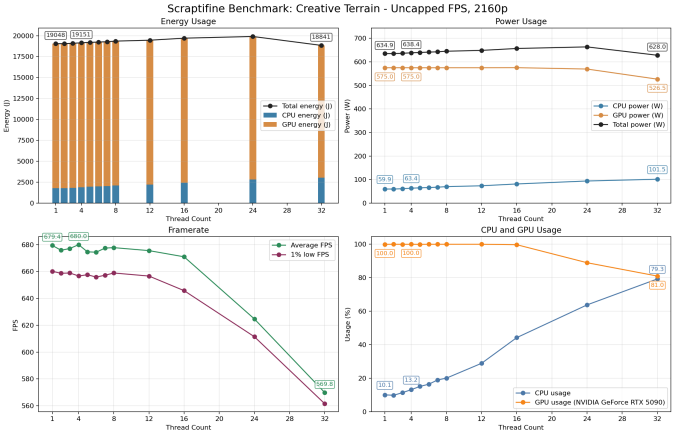

Fig. 1. Creative Mode Performance Metrics For Experiment 1: FPS, Resource Usage, Energy & Power Consumption

In terms of GPU usage, there are not many differences recorded, indicating that the true efficiency gains are at the CPU level. An interesting result is that GPU usage actually drops from 100% when using more cores, which is another indicator of the CPU being bottlenecked, confirming our initial hypothesis.

2) Fixed-parameter resource usage impact when running on optimal versus default settings: We also analyze the resource usage while fixing parameters with most influence on performance, namely FPS and resolution, to the values of 144 FPS and 1080p resolution. The results here confirm our expectation, resulting in almost twice as much CPU usage in creative mode, from 8% to 16%, and an almost three times increase on survival settings, from 8% to 24%, when comparing the 4 cores optimised version with the 32 core default. Meanwhile, GPU usage remains almost the same, which is an expected observation according to our hypothesis. One

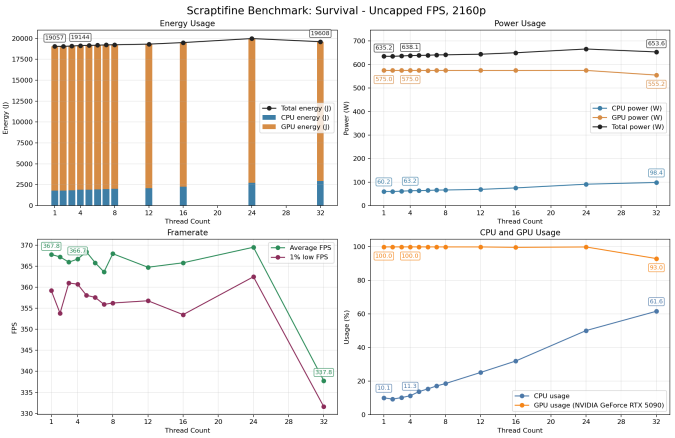

Fig. 2. Survival Mode Performance Metrics For Experiment 1: FPS, Resource Usage, Energy & Power Consumption

surprising result noticed from this experiment is that, despite the significant difference in resource usage and performance, as well as the total amount of energy consumed being lower for the optimised version, the difference between the two is hardly noticeable, registering only a 10-15% increase on average. This result can, however, be explained by the fact that the figures for CPU usage for the default version are still considerably lower when compared to the first experiment, so we should not expect such a similar increase.

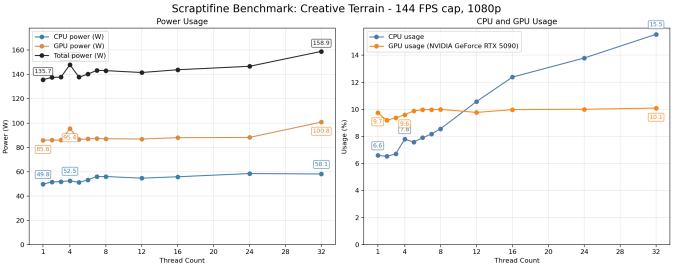

Fig. 3. Creative Mode Performance Metrics For Experiment 2: Resource Usage & Power Consumption

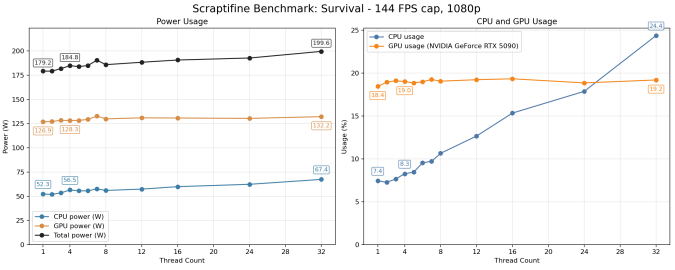

Fig. 4. Survival Mode Performance Metrics For Experiment 2: Resource Usage & Power Consumption

We also measure the impact of only fixing the FPS parameter to 144, while varying the resolution from 1080p to 4K, on the creative mode of the game. The results here show that the resolution primarily impacts GPU usage, which gains no

benefit from changing the amount of CPU cores used. For the CPU usage, we arrive at the same conclusion as above, namely that the usage is significantly increased in the default version compared to the optimised one, from 5% to 15% for all resolutions, while the power consumption also increases accordingly, by around 10% on average.

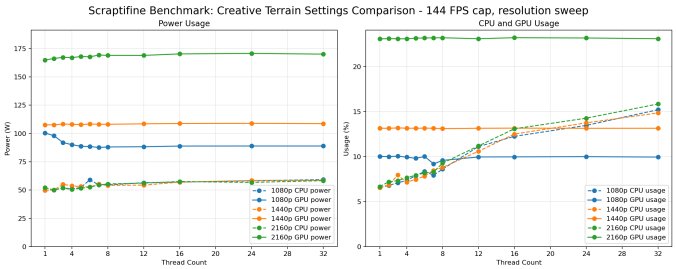

Fig. 5. Creative Mode Performance Metrics With Varying Resolution: Resource Usage & Power Consumption

By having applied our proposed solution to Scrap Mechanic and producing a patch that significantly improves performance and reduces energy usage, we can conclude that our optimisation flow is effective at finding and patching software inefficiencies. No features that are only applicable to Scrap Mechanic were used, allowing our proposed solution to be used in the wider software engineering industry.

VI. Discussion

The results demonstrate that limiting the number of logical cores can significantly improve both performance and energy efficiency. Across all experiments, the optimal configuration was consistently achieved with four cores, outperforming the default configuration of 32 cores.

The first experiment shows that increasing the number of threads does not lead to better performance. Instead, configurations with fewer cores (six or less) performed more efficiently, with four cores providing the best balance. Compared to the default configuration, this resulted in performance improvements of up to 20% in creative mode and 8.6% in survival mode. At the same time, CPU usage was drastically reduced, dropping from 61% to 11%, alongside a reduction of approximately 60% in CPU energy and power consumption.

These results indicate that Scrap Mechanic suffers from excessive parallelization. Profiling and analysis suggest that a large portion of execution time is spent on thread synchronization rather than useful computation. By reducing the number of active threads, synchronization overhead is minimized, allowing the CPU to allocate more resources to actual workload execution.

The second experiment further supports this observation by fixing performance-related parameters such as FPS and resolution. Even under controlled conditions, the default configuration exhibited significantly higher CPU usage—up to three times greater in survival mode—while GPU usage remained largely unchanged. This confirms that the primary bottleneck lies at the CPU level rather than the GPU.

Interestingly, while CPU usage differences were substantial, the increase in total energy consumption was more moderate (approximately 10–15%). This can be explained by the lower absolute CPU utilization in this setup compared to the first experiment, reducing the overall impact on energy consumption.

Overall, the findings highlight that more hardware resources do not necessarily translate to better performance or efficiency. Instead, excessive parallelization introduces synchronization overhead that degrades both. Limiting concurrency to an appropriate level can significantly improve performance while reducing unnecessary energy consumption.

Although the proposed solution of an optimisation flow is independent from Scrap Mechanic and thus allowing it to be generalized to the wider software engineering industry, the proposed solution is backed up by only a single proof of viability. It could very well be the case that Scrap Mechanic was an outlier with the amount of performance still being left on the table that ready to be optimised with a patch. Due to time limitations, it was not possible to evaluate the proposed solution on multiple pieces of software.

References

[1] Min Ouyang. “Review on modeling and simulation of interdependent critical infrastructure systems”. In: *Reliability Engineering amp; System Safety* 121 (Jan. 2014), pp. 43–60. ISSN: 0951-8320. DOI: 10.1016/j.ress.2013.06.040. URL: http://dx.doi.org/10.1016/j.ress.2013.06.040.

[2] Linn Svegrup, Jonas Johansson, and Henrik Hassel. “Integration of Critical Infrastructure and Societal Consequence Models: Impact on Swedish Power System Mitigation Decisions”. In: *Risk Analysis* 39.9 (2019), pp. 1970–1996. DOI: https://doi.org/10.1111/risa.13272. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/risa.13272. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/risa.13272.

[3] International Telecommunication Union and World Bank. *Measuring the Emissions and Energy Footprint of the ICT Sector: Implications for Climate Action*. International Telecommunication Union and World Bank, 2024. URL: https://www.itu.int/en/ITU-D/Environment/Documents/Publications/2024/ITU-World%20Bank%20Measuring%20the%20Emissions-Energy%20Footprint%20of%20the%20ICT%20Sector%202024.pdf.

[4] Javier Mancebo, Félix García, and Coral Calero. “A process for analysing the energy efficiency of software”. In: *Information and Software Technology* 134 (June 2021), p. 106560. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2021.106560. URL: http://dx.doi.org/10.1016/j.infsof.2021.106560.

[5] Jose Balanza-Martinez, Patricia Lago, and Roberto Verdecchia. “Tactics for Software Energy Efficiency: A Review”. In: *Advances and New Trends in Environmental Informatics 2023*. Springer Nature Switzerland, 2024, pp. 115–140. ISBN: 9783031469022. DOI: 10.

1007/978-3-031-46902-2_7. URL: http://dx.doi.org/10.1007/978-3-031-46902-2_7.

[6] Simon Schubert et al. “Profiling Software for Energy Consumption”. In: *2012 IEEE International Conference on Green Computing and Communications*. IEEE, Nov. 2012, pp. 515–522. DOI: 10.1109/greencom.2012.86. URL: http://dx.doi.org/10.1109/GreenCom.2012.86.

[7] Kashif Nizam Khan et al. “Energy Profiling Using IgProf”. In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, May 2015, pp. 1115–1118. DOI: 10.1109/ccgrid.2015.118. URL: http://dx.doi.org/10.1109/CCGrid.2015.118.

[8] Advanced Micro Devices. *AMD uProf*. https://www.amd.com/en/developer/uprof.html. Accessed: 2026-03-24. 2026.

[9] Intel Corporation. *Intel VTune Profiler*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html. Accessed: 2026-03-24. 2026.

[10] Jan Jelschen et al. “Towards Applying Reengineering Services to Energy-Efficient Applications”. In: *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, Mar. 2012, pp. 353–358. DOI: 10.1109/csmr.2012.43. URL: http://dx.doi.org/10.1109/CSMR.2012.43.