Avaliação do Consumo de Energia de um CubeSat Brasileiro de Código Aberto

Júnior E. Bassani[1], **Marcos Corino**[1]

[1]Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul

junior.eduardo.bassani@gmail.com, marcos.corino@veranopolis.ifrs.edu.br

Abstract. *CubeSats are small satellites that have achieved popularity, mainly in the academic field, due to their low cost and shrunken development time. However, failures caused by poor energy management are still motives of recurrent worry, and may jeopardize the mission objective and lead to total loss of the spacecraft. This paper evaluates the energy consumption of the software of the FloripaSat-I Brazilian CubeSat. The tests were executed with the aid of an energy consumption analysis tool, evaluating the software modules when executing in isolation and concurrently. The obtained results may provide better comprehension regarding the behavior of the satellite once in space.*

Resumo. *CubeSats são pequenos satélites que alcançaram popularidade, principalmente no setor acadêmico, devido ao seu baixo custo e tempo de desenvolvimento reduzido. Entretanto, falhas causadas por mau gerenciamento de energia ainda são fontes de preocupação recorrente, podendo comprometer o objetivo da missão e levar à perda total do veículo espacial. Este trabalho avalia o consumo de energia do software do CubeSat brasileiro FloripaSat-I. Os testes foram executados com auxílio de uma ferramenta de análise de consumo de energia, avaliando os módulos do software quando executando isolada e concorrentemente. Os resultados obtidos podem prover melhor compreensão quanto ao comportamento do satélite uma vez no espaço.*

1. Introdução

Cubesats são pequenos satélites, criados em 1999 e padronizados por uma especificação de domínio público [University 2014]. Desde então, diversas missões com satélites desse tipo foram desenvolvidas e executadas com sucesso, especialmente no campo acadêmico. Não obstante, devido ao baixo custo de construção e lançamento dos CubeSats, o setor industrial está visualizando esse segmento de satélites como uma alternativa viável à construção de satélites grandes e caros quando se trata de missões de curto e médio prazo.

O tamanho de um CubeSat, entretanto, impõe diversos limites durante seu desenvolvimento, como demonstrado em [Martin-Mur et al. 2015], o que requer grande alocação de recursos para a etapa de testes de hardware e software, com a finalidade de garantir que o satélite operará sem falhas uma vez no espaço.

Apesar de existirem diversas opções para *benchmarks* baseados em hardware [Avery et al. 2011, Bonsu et al. 2019], ainda há escassez de ferramentas de software para a execução de testes em CubeSats em desenvolvimento. Adicionalmente, ferramentas para avaliação do consumo de energia em tempo de execução, um dos maiores desafios para veículos espaciais pequenos, são ainda mais raras.

De acordo com [Swartwout 2013], 17% das manifestações de falhas nos primeiros cem CubeSats foram causadas por mau gerenciamento de energia. Os dados no Nanosats Database [nan 2021] sugerem que o problema ainda persiste, com mais de 26 missões tendo reportado problemas ligados ao consumo de energia; dessas, 22 foram realizadas após 2010.

Esse trabalho tem como objetivo a avaliação do consumo de energia do software de um CubeSat brasileiro de código aberto, o FloripaSat-I. Diversos testes de consumo de energia foram elaborados e executados em um ambiente de simulação, utilizando hardware semelhante ao que foi utilizado na missão original. Os dados produzidos nesse ambiente foram, posteriormente, utilizados para a geração de gráficos, para melhor visualização. Todos os testes foram elaborados e executados em 2019, como parte de um projeto de pesquisa em que o autor participou [Bassani et al. 2020].

O restante desse trabalho está estruturado da seguinte forma: a seção 2 fornece uma visão geral dos CubeSats, bem como descreve arquiteturas de hardware e software comumente empregadas nesse tipo de satélite. A seção 3 examina abordagens de teste de software para CubeSats. A seção 4 explora o CubeSat FloripaSat-I, cujo software foi utilizado como estudo de caso para testes de consumo de energia. A seção 5 analisa a configuração utilizada no ambiente de testes, assim como detalha o método de trabalho utilizado. A seção 6 examina os resultados obtidos dos testes executados. Finalmente, a seção 7 conclui o trabalho.

2. Fundamentação Teórica

CubeSats são pequenos satélites, desenvolvidos em 1999 pela Universidade Estadual Politécnica da Califórnia e pela Universidade de Stanford. Esses veículos são padronizados por uma especificação em domínio público, estando na versão 13 atualmente [University 2014]. A especificação detalha os requisitos de dimensão e massa para CubeSats 1U, 1,5U, 2U, 3U e 3U+ (o sufixo significa *unidade*). O CubeSat 1U deve ter dimensão de $10cm^3$ e massa de até 1,33Kg. Os demais modelos permitem dimensão e massa maiores. O CubeSat 6U também é padronizado, porém por uma especificação distinta [University 2018]. Satélites de unidades maiores também foram desenvolvidos, porém estes, até a data deste trabalho, não são padronizados.

Além dos satélites propriamente ditos, a especificação também padroniza um sistema de lançamento para CubeSats: o *Poly Picosatellite Orbital Deployer* (P-POD) é um envoltório responsável por transportar o satélite com segurança até o momento em que é liberado no espaço. Diferentes P-PODs comportam diferentes tamanhos de CubeSats, e um foguete, dessa forma, é desvinculado do satélite que carrega; em vez disso, basta que o foguete seja compatível com a interface física do P-POD para também ser compatível com os CubeSats que esse P-POD comporta.

2.1. Arquitetura de Hardware para CubeSats

Como o baixo despendimento de recursos é desejável ao desenvolvimento de CubeSats, a maior parte das missões emprega produtos de prateleira, cujo tempo de vida útil no espaço é muitas vezes incerto. Dentre as famílias de processadores mais populares, encontram-se Texas Instruments MSP430, Microchip PIC, ATMEL ATmega e controladores ARM. Esses processadores são geralmente integrados com outros subsistemas,

como os de comunicação e de fornecimento de energia. Essa abordagem possibilita o desenvolvimento paralelo e a criação de módulos de software que podem ser reutilizados em missões futuras [Carrara et al. 2017, Villa et al. 2014, McNutt et al. 2009].

Devido ao ambiente hostil em que CubeSats operam, os componentes de hardware e software devem garantir que o satélite possua resiliência à radiação. Duas abordagens são comumente empregadas para lidar com esse desafio. A primeira é a implementação de redundância ao nível de hardware. Dessa forma, se um subsistema sofrer dano irreparável, o(s) subsistema(s) sobressalente(s) pode(m) substituí-lo. Essa estratégia é bastante comum para satélites grandes e de alto orçamento, mas também foi utilizada em CubeSats [Aslan et al. 2013, Manyak 2011]. Todavia, essa solução apresenta duas desvantagens: o custo adicional de produção do satélite e o espaço extra necessário para armazenar o(s) componente(s) redundante(s). Um segunda abordagem leva em consideração o orçamento e o espaço limitados dos CubeSats, e emprega redundância e mecanismos de tolerância a falhas na camada de software. Embora o produto final possa apresentar menor nível de confiança, o resultado pode ainda ser satisfatório, como demonstrado em [Manyak 2011, Mehlitz and Penix 2005].

2.2. Arquitetura de Software para CubeSats

Muitos projetos de CubeSats elaboram a arquitetura de software como unidades modulares, de forma similar ao hardware, possibilitando o trabalho em isolamento dos diferentes componentes do sistema. Dessa forma, cada componente possui uma tarefa específica e continua a operar mesmo se outras partes do sistema falharem.

Muitas missões também fazem uso de um sistema operacional (SO), a maior parte deles adequados para sistemas de tempo real. Isso facilita o desenvolvimento do software, pois é possível dividi-lo em processos e/ou *threads* que executam diferentes tarefas, ao mesmo tempo em que é possível fazer uso dos recursos e abstrações providos pelo SO, como gerenciamento de processos, memória, armazenamento, sincronização e entrada e saída. Muitos projetos utilizam FreeRTOS [fre 2021] ou sistemas baseados em Linux. Apesar dos benefícios providos por um SO que gerencia mecanismos de baixo nível, algumas missões não necessitam de um, enquanto que o hardware de outras simplesmente não possui memória suficiente para ambos o SO e o software de aplicação, e acabam, dessa forma, tendo de implementar as primitivas de baixo nível de que necessitam.

3. Estado da Arte em Testes de Software de CubeSats

É possível dividir as atividades de testes de software em três categorias [Sommerville 2011]: testes unitários, testes de integração, e testes de sistema. Cada tipo de teste pode ser utilizado em diferentes etapas do desenvolvimento de um software, possuindo propósitos distintos, além de pontos fortes e fracos.

Testes unitários consistem em verificar a funcionalidade de segmentos isolados no código-fonte de um sistema, como funções e classes. Sua implementação é simples para sistemas com poucas dependências e que requerem pouca ou nenhuma configuração. Outra vantagem desse tipo de teste é que ele possui granularidade fina, i.e., os erros podem ser localizados em seu local origem em vez de em camadas de software superiores, para as quais são propagados.

Testes de integração focam em identificar falhas decorrentes da interação entre diferentes componentes de um sistema, e geralmente são executados após os testes unitários. Os casos de teste para esse método focam, portanto, nas interfaces públicas dos componentes. Erros podem ocorrer devido à passagem de parâmetros incorreta, à pressuposição incorreta do comportamento de um componente, ou a erros de temporização, por exemplo.

Por fim, os testes de sistema constituem a última etapa de testes para um software. Neste estágio, todos os componentes desenvolvidos são integrados e o sistema como um todo é testado. Essa etapa é importante pois pode revelar erros em um ou mais componentes cuja manifestação ocorre somente quando o sistema é totalmente integrado.

A figura 1 ilustra uma possível configuração de um sistema para satélites estruturado em módulos. Os módulos na parte mais baixa da figura são de baixo nível (i.e., são responsáveis por partes específicas do satélite, como armazenamento de dados em memória persistente), enquanto os módulos na parte mais alta da figura representam funcionalidades de alto nível, que atendem aos requisitos da missão científica do satélite (e.g., tirar uma foto da Terra a cada 30 minutos). Os módulos intermediários, neste exemplo, servem apenas como camada de interconexão.

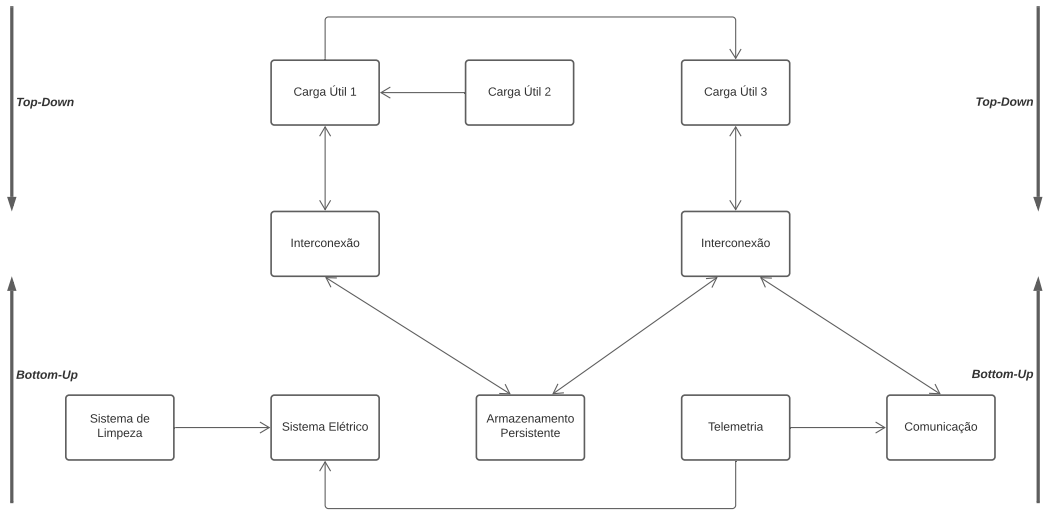

Figura 1. Exemplo de configuração de módulos para um sistema de satélites

Esse exemplo melhor elucida a importância de testes de integração. Os módulos de software não funcionam em isolamento absoluto; ao contrário, em certos momentos precisam se comunicar uns com os outros, na figura 1 demonstrado visualmente pelas setas de um módulo a outro. A confiança de que os módulos se comunicam de forma correta e segura é um objetivo a ser alcançado pelos testes de integração.

Dentre os vários métodos para testes de integração [Sommerville 2011], é possível citar a estratégia *big bang*, na qual os módulos são interconectados e testados de uma só vez. Esse método pode ser ineficiente, pois dificulta a identificação do módulo defeituoso. Outra estratégia é a *incremental*, em que a integração dos módulos é feita de modo progressivo, ou seja, grupos de dois ou mais módulos são integrados e testados. Quando todos os testes dessa integração reportarem êxito, a integração expande para abranger os

demais módulos. Esse processo se repete até que todos os módulos sejam integrados e testados com sucesso. A abordagem incremental pode ainda ser desestruturada em uma abordagem *top-down* ou *bottom-up*. Em *bottom-up*, primeiro testa-se os módulos de baixo nível, expandindo a integração em direção aos módulos de alto nível; em *top-down* ocorre o inverso.

Em *bottom-up*, quando os módulos de baixo nível precisam ser testados e os módulos de alto nível ainda não foram desenvolvidos, pode-se utilizar *drivers*, programas simuladores que farão a integração com os módulos de baixo nível. Esse método facilita a detecção de falhas nos módulos de baixo nível, porém, potenciais imperfeições de arquitetura de alto nível só serão descobertas em um estágio posterior. A estratégia *bottom-up* está representada na figura 1 pelas setas direcionadas para cima. Os módulos de outros níveis, caso ainda não tenham sido desenvolvidos, mas precisem ser integrados, podem ser substituídos por *drivers*.

No método *top-down*, são utilizados *stubs* para simular os módulos de baixo nível que ainda não foram desenvolvidos, mas que precisam ser integrados com os módulos de alto nível. A implementação de *stubs* pode apresentar complicações caso estes simulem módulos complexos. A abordagem de testes *top-down* está ilustrada na figura 1 pelas setas direcionadas para baixo. Os módulos de outros níveis, caso ainda não tenham sido desenvolvidos, mas precisem ser integrados, podem ser substituídos por *stubs*.

Por fim, uma terceira forma de testes, o *Sandwich Testing*, determina que as estratégias *top-down* e *bottom-up* sejam empregadas concomitantemente, ou seja, módulos de baixo nível são integrados e testados ao mesmo tempo em que módulos de alto nível são integrados e testados, resultando, assim, em uma abordagem de testes híbrida. Essa estratégia está ilustrada na figura 1 quando considerada em sua forma completa. Essa abordagem permite tanto a utilização de *drivers* quanto de *stubs*, conforme necessário.

Adicionalmente, é possível fazer uso de simulações como *hardware-in-the-loop* e *software-in-the-loop*. Na simulação *hardware-in-the-loop*, o hardware final é efetivamente utilizado, mas em um ambiente de simulação. Já em *software-in-the-loop*, o software é testado em um ambiente que simula o hardware.

As práticas de verificação e validação de software incluem uma variedade de técnicas utilizadas para garantir que o software opere da forma pretendida e que se adéque aos requisitos estipulados. Essas técnicas têm sido aprimoradas ao longo dos anos e sua utilização no desenvolvimento de CubeSats é agora substancial. Uma pesquisa conduzida por [Jacklin 2015] encontrou que simulação é a prática mais utilizada, embora outros métodos também sejam empregados, como verificação baseada em modelos e injeção de falhas. A maior parte das missões com satélites, incluindo CubeSats, geralmente emprega uma combinação de verificação e validação tanto na camada de hardware quanto na de software. Nas missões SUCHAI [Gonzalez et al. 2019], testes de unidade e de integração automatizados foram utilizados após cada modificação de software para garantir que as partes de código adicionadas ou modificadas não tivessem impacto negativo no sistema. Da mesma forma, uma ferramenta para rastrear dependências entre módulos foi utilizada para garantir que a arquitetura do software permanecesse razoavelmente consistente com a evolução do mesmo. Testes também têm sido empregados em estações terrestres que se comunicam com os satélites. Em [Yuhaniz and Hamzah 2015], após o software da

estação terrestre ser finalizado, um programa foi escrito apenas para simular o CubeSat. Ele podia enviar e receber dados de e para a estação, de forma a verificar se ela estava operando corretamente ou não.

O consumo de energia também é um tópico muito importante para sistemas de software, crescendo em popularidade [Pinto et al. 2014] e particularmente interessante para pequenos satélites. Entretanto, pesquisadores revelaram que programadores ainda carecem do conhecimento para criar aplicações com consumo de energia eficiente [Pinto et al. 2014, Pang et al. 2016]. Isso pode estar ligado ao fato de que aprimorar o consumo de energia de uma aplicação requer esforço extra quando comparado com outros problemas de desenvolvimento [Chowdhury and Hindle 2016]. Entretanto, aprimoramentos de consumo de energia, em alguns casos, podem ser alcançados indiretamente a partir do aprimoramento de performance [Johann et al. 2012]. Adicionalmente, em cenários específicos, geralmente sistemas embarcados, otimizações no consumo de energia são factíveis ao simplesmente reduzir a frequência de relógio do processador, sem ser necessário modificar a base de código [Normann 2016].

4. Estudo de Caso: A Missão FloripaSat-I

O FloripaSat-I [Villa et al. 2014, Martins et al. 2016] é um projeto de CubeSat de código aberto que iniciou em 2012 como uma parceria entre estudantes e pesquisadores da Universidade Federal de Santa Catarina e o Instituto Federal de Santa Catarina para desenvolver um CubeSat 1U. O satélite foi lançado no fim de 2019 e continua operacional durante a escrita deste trabalho.

O CubeSat carrega duas cargas úteis. A primeira tem o propósito de analisar danos causados por radiação em memórias do tipo *static random-access memory* (SRAM); a outra tem a missão de estudar alternativas para o armazenamento de energia em componentes eletrônicos e a viabilidade para aplicações de baixo consumo de energia no segmento espacial.

O hardware do satélite é dividido em múltiplos módulos, cada qual responsável por uma tarefa específica e por executar seu respectivo módulo de software. O módulo *On-Board Data Handling* (OBDH) utiliza o microcontrolador MSP430F6659IPZ, com 64KB de memória volátil (RAM) e 512KB de memória persistente (flash). Ele serve como ponto de interconexão entre os demais módulos e pode se comunicar com a estação terrestre a partir do módulo *Telemetry, Tracking and Command* (TTC). Seu software é dividido em 10 *threads* (chamadas de *tarefas* pelo FreeRTOS, o SO utilizado por esse módulo). A tarefa *Watchdog* tem a única responsabilidade de, periodicamente, reiniciar os componentes *watchdog* interno e externo (redundância de hardware), prevenindo possíveis *deadlocks* em tempo de execução. A tarefa *Communications* se encarrega da comunicação com a Terra, recebendo comandos e enviando suas respectivas respostas. A tarefa *Housekeeping* tem como dever garantir o correto funcionamento do satélite. Para isso, ela monitora sensores, contendo o status do sistema, além de reiniciar o software do módulo OBDH a cada 12 horas. A tarefa *Data Storage* é responsável por armazenar dados gerados pelas outras tarefas em memória persistente. As demais tarefas, TTC, *Electrical Power System*, *Inertial Measurement Unit*, *Solar Panels*, *Payload Rush* e *Payload Brave* são todas interfaces de software para os seus respectivos módulos de hardware.

O software foi escrito na linguagem C, utilizando a *integrated development envi-*

ronment (IDE) *Code Composer Studio* (CCS)[1] [flo 2021]. Para o gerenciamento de todas as tarefas supracitadas, o SO FreeRTOS foi escolhido para o módulo OBDH, sobre o qual as tarefas executam concorrentemente e com diferentes prioridades. Recursos adicionais providos pelo SO, como filas de mensagens e semáforos, também foram utilizados pelos desenvolvedores.

5. Configuração de Ambiente e Método de Testes

O microcontrolador utilizado durante este trabalho foi o MSP430FR5969, embutido em uma placa de desenvolvimento provida pelo fabricante. O processador atinge velocidade de até 16MHz, possuindo 2KB de RAM e 64KB de *ferroelectric random-access memory* (FRAM), um tipo de memória não volátil. Dessa forma, modificações foram necessárias para compilar e executar o código com sucesso[2]. Primeiramente, o módulo OBDH foi o único utilizado durante os testes, e o número de tarefas foi reduzido a seis, devido a restrições de tempo: *Watchdog*, *Housekeeping*, *Communications*, *Data Storage*, TTC, e *Payload Rush Interface*. Diversas portas do microcontrolador utilizadas para tráfego de dados e pela comunicação com outros módulos também foram modificadas para atingir compatibilidade com a especificação do produto [Instruments 2012], ou para simplesmente simular uma interface não existente no hardware utilizado.

Para testar a versão modificada, a ferramenta embutida no CCS, *Energy Trace*, foi utilizada. Ela executa o código compilado para a plataforma alvo, ao mesmo tempo em que registra dados sobre corrente, tensão e consumo de energia em um arquivo, com precisão de tempo de nanossegundos. A placa do microcontrolador interagiu com essa ferramenta por meio de uma entrada USB, tanto para fornecer energia à placa quanto para a transferência de dados. Programas utilitários foram escritos para processar e reduzir o tamanho dos dados coletados pela ferramenta *Energy Trace*, já que o conjunto de dados armazenados se mostrara demasiado grande para ferramentas de visualização de dados genéricas.

O método de testes teve como base a permutação de tarefas habilitadas para escalonamento preemptivo. Os resultados dos cenários testados foram posteriormente comparados entre si. Nos primeiros cenários, cada tarefa foi habilitada individualmente. Em seguida, todas as tarefas foram habilitadas para execução concorrente. Por último, diferentes conjuntos de tarefas foram testadas. Cada cenário de teste foi executado quatro vezes para verificar a variação nos resultados de uma execução à outra. Ademais, as tarefas habilitadas para cada cenário foram executadas por um período ininterrupto de dez minutos.

É importante ressaltar que as tarefas escritas para o FloripaSat-I abdicam de tempo de processamento, por um período de tempo fixo, quando não mais possuem trabalho a realizar. Quando não há nenhuma tarefa de aplicação apta a ser executada, uma tarefa ociosa é escalonada. Em todos os cenários, a única ação desempenhada por essa tarefa foi a de configurar o microcontrolador para operar em um estado de baixo consumo de energia. Essa tarefa ociosa é substituída pelo FreeRTOS logo que uma tarefa de aplicação está pronta para executar novamente. É plausível que, em alguns casos, a tarefa ociosa

[1]Código fonte: `https://github.com/floripasat/obdh`

[2]Código fonte da versão modificada: `https://github.com/juniorbassani/floripasat-obdh-msp430fr5969`

tenha sobrepujado outras tarefas em termos de utilização absoluta de processamento. Os tempos de suspensão de execução são: as tarefas *Watchdog* e *Communications* têm 500ms de tempo de espera até entrar para a fila de escalonamento novamente; *Payload Rush Interface*, *Housekeeping* e *Data Storage*, 1000ms; TTC *Interface*, 5000ms.

6. Resultados Obtidos

A análise dos resultados inicia-se com a tabela 1, que apresenta a quantidade de memória persistente e a quantidade estimada de consumo máximo de memória volátil quando da execução do programa no microcontrolador. A primeira coluna expressa as tarefas habilitadas para cada teste. Esses dados foram providos pela IDE CCS. Como esperado, a utilização mínima de memória, tanto volátil como persistente, ocorre quando apenas a tarefa ociosa é habilitada, enquanto que o uso máximo é observado quando todas as tarefas são habilitadas. Quando esses dois cenários são ignorados, a tarefa *Watchdog* é a que consome a quantidade mínima de memória volátil e persistente, e a tarefa *Communications* é a que consome a maior quantidade.

Tabela 1. Utilização de memória volátil e persistente

Tarefa	RAM utilizada (bytes)	FRAM utilizada (kilobytes)
Nenhuma (Ociosa)	692	21,08
Watchdog	696	21,51
Housekeeping	696	23,70
Communications	1615	32,41
Data Storage	910	25,26
TTC *Interface*	900	24,81
Payload Rush Interface	736	23,74
Todas	1685	39,95

A tabela 2 apresenta o consumo total de energia, em média aritmética, bem como o desvio padrão, quando cada cenário é testado por dez minutos ininterruptos. Novamente, os cenários em que apenas a tarefa ociosa está habilitada, e aquele em que todas as tarefas estão habilitadas, manifestaram o menor e o maior consumo de energia, respectivamente. Desconsiderando esses cenários, a tarefa *Payload Rush Interface* foi a que consumiu a menor quantidade de energia, enquanto que a *Data Storage* foi a que consumiu a maior quantidade. Quanto ao desvio padrão, novamente desconsiderando os cenários supracitados, os valores mínimo e máximo se manifestaram com as tarefas *Watchdog* e *Communications*, respectivamente.

Em seguida, a figura 2 ilustra a média de consumo de energia quando nenhuma tarefa está habilitada a executar, com exceção da tarefa ociosa. A primeira característica notável é o consumo de energia inconsistente que o gráfico apresenta. De fato, essa é, até certo ponto, uma característica exibida por todas as configurações testadas. É possível que os declínios de consumo sejam em razão da troca de escalonamento de uma tarefa de aplicação para a tarefa ociosa, enquanto que os picos de consumo, quando da ocorrência do inverso. Esse teste, de forma esperada, foi o que consumiu a menor quantidade de energia.

A figura 3 ilustra a energia consumida pela tarefa *Watchdog*, quando executada em isolamento. Esse gráfico é consideravelmente diferente do anterior. Ele também apresenta

Tabela 2. Consumo de energia em média aritmética

Tarefa	Consumo de energia (joules)	Desvio padrão (joules)
Nenhuma (Ociosa)	0,70	0,02
Watchdog	1,37	0,35
Housekeeping	1,41	0,43
Communications	3,29	0,56
Data Storage	9,09	0,54
TTC *Interface*	1,47	0,42
Payload Rush Interface	1,15	0,44
Todas	10,28	0,46

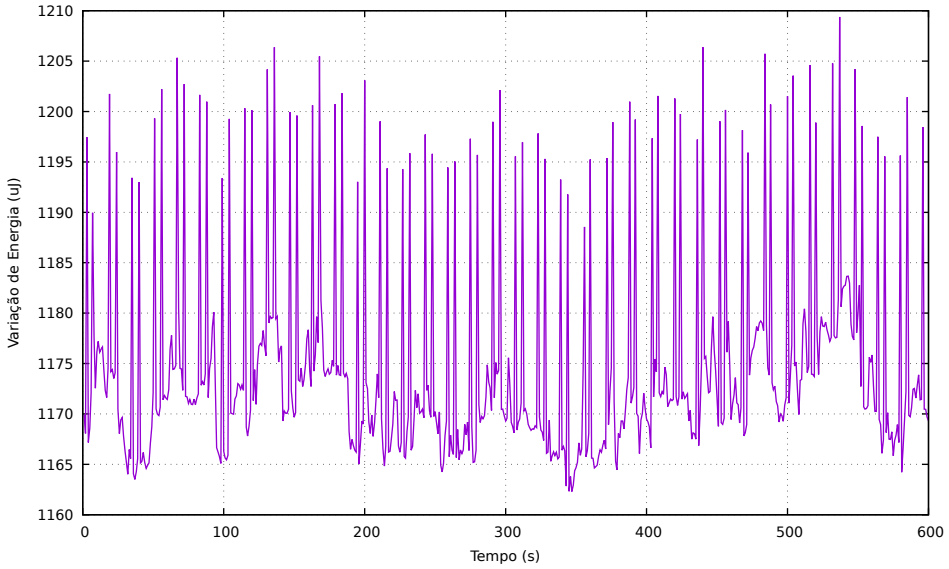

Figura 2. Teste da tarefa ociosa

variações de consumo ao longo de todo o eixo horizontal, mas estas são menos frequentes e levam tempo maior até atingir o pico de consumo. É importante salientar que essa tarefa simplesmente reinicia os *watchdogs* interno e externo; portanto, é possível que a tarefa ociosa tenha usufruído de maior tempo de processamento.

A figura 4 mostra o resultado do consumo de energia da tarefa *Housekeeping*, cujo aspecto se assemelha ao da figura 2. Ambos os testes apresentaram o mesmo padrão de picos regulares de consumo de energia e, apesar de as ocorrências de tais picos serem mais frequentes na figura 2, a variação em joules foi similar.

As variações súbitas de consumo de energia também estiveram presentes nos testes das tarefas TTC *Interface* e *Payload Rush Interface* (não mostrados). O teste da tarefa *Communications*, na figura 5, por sua vez, também apresentou picos e declínios de consumo, mas estes são menos abruptos do que os obtidos com os testes anteriores.

A figura 6 pertence à tarefa que consumiu a maior quantidade de energia, a *Data Storage*. Este teste apresentou um padrão de consumo oposto ao dos anteriores, com breves declives de consumo a intervalos regulares. De todo modo, essa tarefa consumiu uma quantidade de energia considerável. O resultado obtido nesse estudo de caso, en-

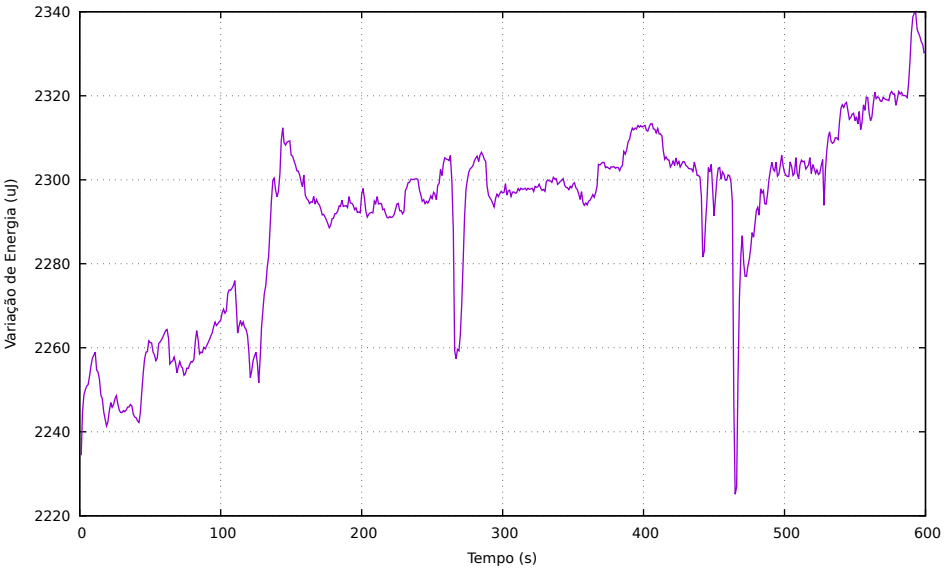

Figura 3. Teste da tarefa *Watchdog*

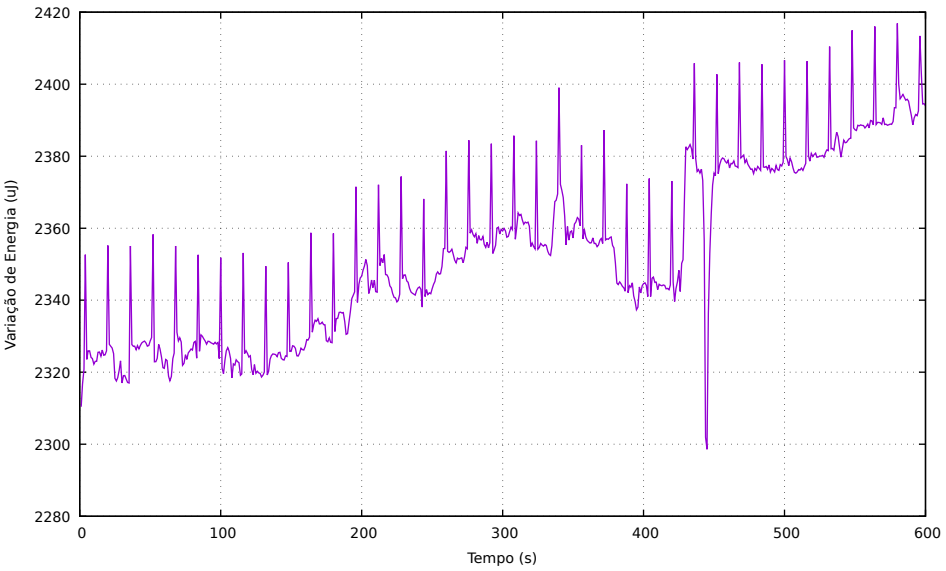

Figura 4. Teste da tarefa *Housekeeping*

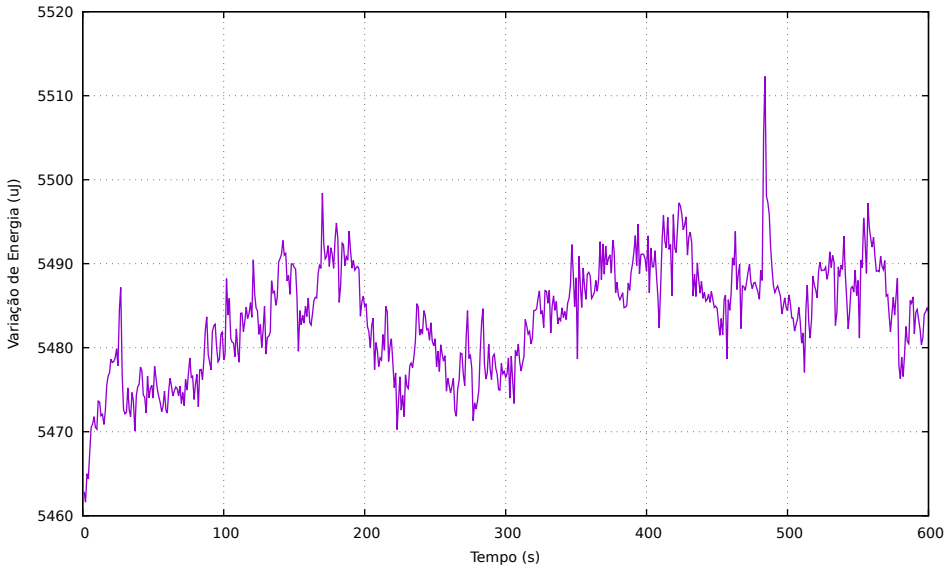

Figura 5. Teste da tarefa *Communications*

tretanto, pode não ser reprodutível em outros microcontroladores da família MSP430. O produto utilizado nesse trabalho, o MSP430FR5969, utiliza tecnologia de memória persistente FRAM, que não está presente no microcontrolador utilizado na missão FloripaSat-I. Dessa forma, diferenças de consumo de energia são factíveis.

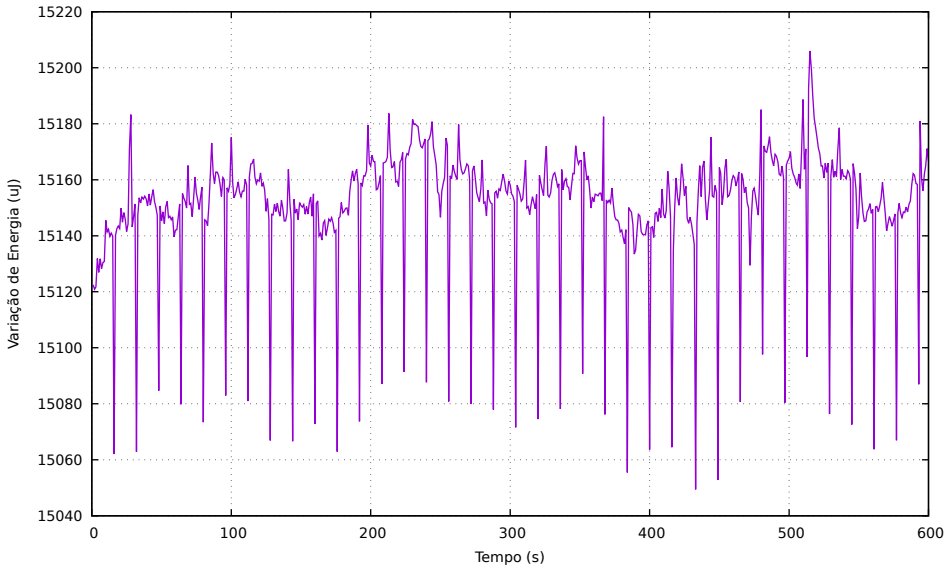

Figura 6. Teste da tarefa *Data Storage*

Por último, a figura 7 apresenta a média de consumo de energia quando todas as tarefas estão habilitadas para escalonamento e executam concorrentemente, cenário condizente com a missão do FloripaSat-I propriamente dita. De maneira expectável, esse teste resultou no maior consumo de energia da unidade de processamento. Além disso, o gráfico possui aparência mais estável do que os anteriores, com uma tendência de aumento de consumo ao longo do tempo.

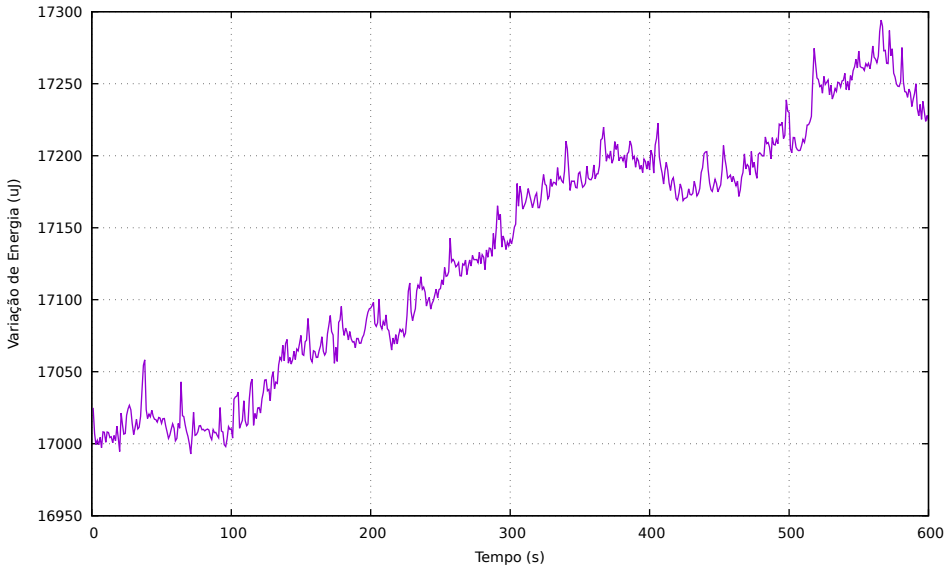

Figura 7. Teste de todas as tarefas

7. Conclusões

CubeSats são uma classe de satélites cuja popularidade deve ampliar ainda mais nas próximas décadas. Seu sucesso se deve principalmente ao custo reduzido de produção, o que permitiu que muitas universidades fizessem experimentos e documentassem seus resultados antes de a indústria também o adotar para missões de baixa e média escala.

Esse tipo de satélite, entretanto, impõe limitações físicas que, por sua vez, afetam a capacidade de gerenciamento de energia. Essas deficiências foram responsáveis por falhas leves e críticas de diversas missões. Ferramentas como o *Energy Trace*, disponível na IDE CCS são de suma importância para a execução de testes de consumo de energia de maneira simplificada, e que podem fornecer a desenvolvedores estimativas realistas para o consumo de energia de satélites pequenos quando no espaço.

Este trabalho apresentou o estudo de caso de uma missão de CubeSat brasileira de código aberto, o FloripaSat-I, considerando as características de consumo de energia de parte de seu software, executando em uma plataforma de hardware semelhante ao utilizado na missão original. Os resultados obtidos podem equipar os desenvolvedores com o conhecimento do perfil de consumo de energia de seu produto, bem como direcionar possíveis otimizações para as partes do programa que mais demandaram do subsistema de fornecimento de energia.

Quanto a trabalhos futuros, a reavaliação dos testes aqui apresentados, porém executados na plataforma de hardware original pode ser útil para averiguar a ocorrência de possíveis disparidades de consumo de energia. É esperado que as maiores divergências se concentrem na tarefa *Data Storage*, visto que, como explicado na seção 6, o microcontrolador original possui tecnologia de armazenamento persistente diferente daquele utilizado neste trabalho.

Referências

(2021). Floripasat-i. https://github.com/floripasat/obdh. Acesso em: 15/10/2021.

(2021). Freertos. https://www.freertos.org/. Acesso em: 15/10/2021.

(2021). Nanosats database. https://www.nanosats.eu/. Acesso em: 15/10/2021.

Aslan, A., Yağcı, H., Umit, M., Sofyalı, A., Bas, M., Uludag, M., Ozen, O., Aksulu, M., Yakut, E., Oran, C., et al. (2013). Development of a leo communication cubesat. In *2013 6th International Conference on Recent Advances in Space Technologies (RAST)*, pages 637–641. IEEE.

Avery, K., Finchel, J., Mee, J., Kemp, W., Netzer, R., Elkins, D., Zufelt, B., and Alexander, D. (2011). Total dose test results for cubesat electronics. In *2011 IEEE Radiation Effects Data Workshop*, pages 1–8. IEEE.

Bassani, J. E., Villa, P. R. C., Bezerra, E., and Vargas, F. (2020). Energy consumption evaluation aimed at cubesat software development and test. In *IV IAA Latin American CubeSat Workshop*.

Bonsu, B., Masui, H., and Cho, M. (2019). Demonstration of lean satellite (1u cubesat) testing using pett vacuum chamber. In *2019 9th International Conference on Recent Advances in Space Technologies (RAST)*, pages 959–966. IEEE.

Carrara, V., Januzi, R. B., Makita, D. H., Santos, L. F. d. P., and Sato, L. S. (2017). The itasat cubesat development and design. *Journal of Aerospace Technology and Management*, 9(2):147–156.

Chowdhury, S. A. and Hindle, A. (2016). Characterizing energy-aware software projects: Are they different? In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 508–511.

Gonzalez, C. E., Rojas, C. J., Bergel, A., and Diaz, M. A. (2019). An architecture-tracking approach to evaluate a modular and extensible flight software for cubesat nanosatellites. *IEEE Access*, 7:126409–126429.

Instruments, T. (2012). Msp430fr596x, msp430fr594x mixed-signal microcontrollers.

Jacklin, S. A. (2015). Survey of verification and validation techniques for small satellite software development.

Johann, T., Dick, M., Naumann, S., and Kern, E. (2012). How to measure energy-efficiency of software: Metrics and measurement results. In *Proceedings of the First International Workshop on Green and Sustainable Software*, GREENS '12, pages 51–54, Piscataway, NJ, USA. IEEE Press.

Manyak, G. D. (2011). Fault tolerant and flexible cubesat software architecture.

Martin-Mur, T. J., Gustafson, E. D., and Young, B. T. (2015). Interplanetary cubesat navigational challenges.

Martins, V., Villa, P., Slongo, L., Salamanca, J., Sabino, F., Martinez, S., Mariga, L., Vidal, I., Eiterer, B., Baldini, M., et al. (2016). The experience of designing and developing the on-board electronics of a cubesat in brazil. In *1st IAA Latin American CubeSat Workshop*, volume 2, pages 69–73.

McNutt, C. J., Vick, R., Whiting, H., and Lyke, J. (2009). Modular nanosatellites–(plug-and-play) pnp cubesat. In *7th Responsive Space Conference*.

Mehlitz, P. and Penix, J. (2005). Expecting the unexpected-radiation hardened software. In *Infotech@ Aerospace*, page 7088.

Normann, M. A. (2016). Software design of an onboard computer for a nanosatellite. Master's thesis, NTNU.

Pang, C., Hindle, A., Adams, B., and Hassan, A. E. (2016). What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89.

Pinto, G., Castor, F., and Liu, Y. D. (2014). Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 22–31, New York, NY, USA. ACM.

Sommerville, I. (2011). *Software Engineering*. International Computer Science Series. Pearson.

Swartwout, M. (2013). The first one hundred cubesats: A statistical look. *Journal of Small Satellites*, 2(2):213–233.

University, C. P. S. (2014). Cubesat design specification rev. 13. *The CubeSat Program, Cal Poly SLO*.

University, C. P. S. (2018). 6u cubesat design specification rev. 1.0. *The CubeSat Program, Cal Poly SLO*.

Villa, P., Slongo, L., Salamanca, J., Martins, V., Silva, F., Martinez, S., Mariga, L., Eiterer, B., Vidal, I., Menegon, V., et al. (2014). A complete cubesat mission: the floripa-sat experience. In *1st IAA Latin American Cubesat Workshop*, volume 2, pages 307–314.

Yuhaniz, S. S. and Hamzah, N. (2015). Development of mission control station software for a cubesat mission. In *2015 International Conference on Space Science and Communication (IconSpace)*, pages 33–37.