UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

EDUARDO KOCHENBORGER DUARTE

Avaliação de Desempenho Temporal de Controlador Lógico Programável - Estudo de Caso

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof. Dr. João Cesar Netto
Co-orientador: Eng. João Ricardo Wagner de Moraes

Porto Alegre
2018

"Pode-se definir como ser vivo qualquer sistema que, de forma autônoma, possui a habilidade de reverter a lei natural do universo de aumento da entropia"

Jose Maria de Britto Duarte

AGRADECIMENTOS

Gostaria de agradecer à minha família por todo o apoio prestado durante todos estes longos anos de estudo. Agradeço a todos os colegas, amigos e professores que me acompanharam nesta longa caminhada. Em especial, gostaria de agradecer professor João Netto e ao engenheiro João de Moraes, pela orientação neste trabalho e diversas conversas, além de agradecer ao Roque Dapper e ao Tiago Dall'Agnol pelas muitas trocas de ideias e a todo pessoal do projeto SDCD pelo imprescindível suporte.

RESUMO

Em aplicações industriais, frequentemente são utilizados controladores lógicos programáveis, dispositivos desenvolvidos para tarefas de controle e para suportar o ambiente industrial hostil. Diferentemente de aplicações convencionais, o tempo de resposta e o determinismo são essenciais na indústria, sendo necessário que os dispositivos utilizados possuam desempenho bom o suficiente para a aplicação em questão. Este trabalho apresenta um método para realizar a avaliação de desempenho de um controlador lógico programável, utilizando como estudo de caso a série de controladores Nexto, fabricados pela empresa Altus. Baseado no método proposto, foram realizados experimentos através dos quais foram obtidos os dados necessários para as subsequentes análises de desempenho. Ao final do estudo, são propostas otimizações de desempenho fundamentadas nos resultados dos experimentos e no método proposto.

Palavras-chave: Controlador Lógico Programável, Avaliação de Desempenho, Protocolo de Comunicação, Informática Industrial.

ABSTRACT

In industrial applications, programmable logic controllers, devices designed for control tasks and to withstand hostile industrial environments, are frequently used. Unlike common applications, the response time and the determinism are essential in the industry, making it necessary for the devices used to have a good enough performance for the target application. This work presents a method for evaluating the performance of a programmable logic controller, using the Nexto series controllers, manufactured by Altus, as a case study. Based on the proposed method, experiments were conducted, from which the necessary data for the subsequent performance evaluations was obtained. At the end of the study, performance optimizations are proposed based on the results of the experiments and on the proposed method.

Keywords: Programmable Logic Controller, Performance Evaluation, Communication Protocol, Industrial Informatics.

LISTA DE ABREVIATURAS E SIGLAS

CLP Controlador Lógico Programável

I/O *Input/Output*

CPU *Central Processing Unit*

IEC *International Electrotechnical Commission*

LD *Ladder Diagrams*

IL *Instruction List*

SFC *Sequential Function Charts*

ST *Structured Text*

FBD *Function Block Diagram*

POU *Program Organization Unit*

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A automação de tarefas outrora executadas manualmente é recorrente na indústria e causa grande aumento de produtividade. Esse processo tem origem em meados da década de 40, cunhado sob as necessidades da indústria automobilística americana. Por um longo tempo, a automação existiu em escalas bastante diminutas, utilizando-se principalmente de dispositivos mecânicos. Com o surgimento e disseminação da utilização de computadores, a flexibilidade adquirida possibilitou a automatização de praticamente qualquer tarefa (GUPTA; ARORA, 2009).

Uma das formas de se automatizar tarefas é através da utilização de relés, que são interruptores eletromecânicos, ligados entre si de forma a se obter o comportamento desejado. Contudo, um grande sistema de controle desenvolvido com relés apresentaria uma grande complexidade. Essa complexidade dificulta muito a depuração de problemas, a implementação de mudanças e a reutilização de partes do sistema. Foi nesse contexto que surgiram os controladores lógicos programáveis (CLPs), trazendo grandes melhorias para os problemas até então apresentados.

Um CLP é, basicamente, um dispositivo que captura informações do processo e produz uma reação, baseada na forma como foi programado. As informações do ambiente vêm de sensores, como sensores de temperatura ou pressão, por exemplo. Usando esses dados, o CLP produzirá uma saída de acordo com a lógica programada pelo usuário. Por exemplo, pode-se ativar um atuador que controle um ventilador uma vez que a temperatura suba além de um limiar aceitável.

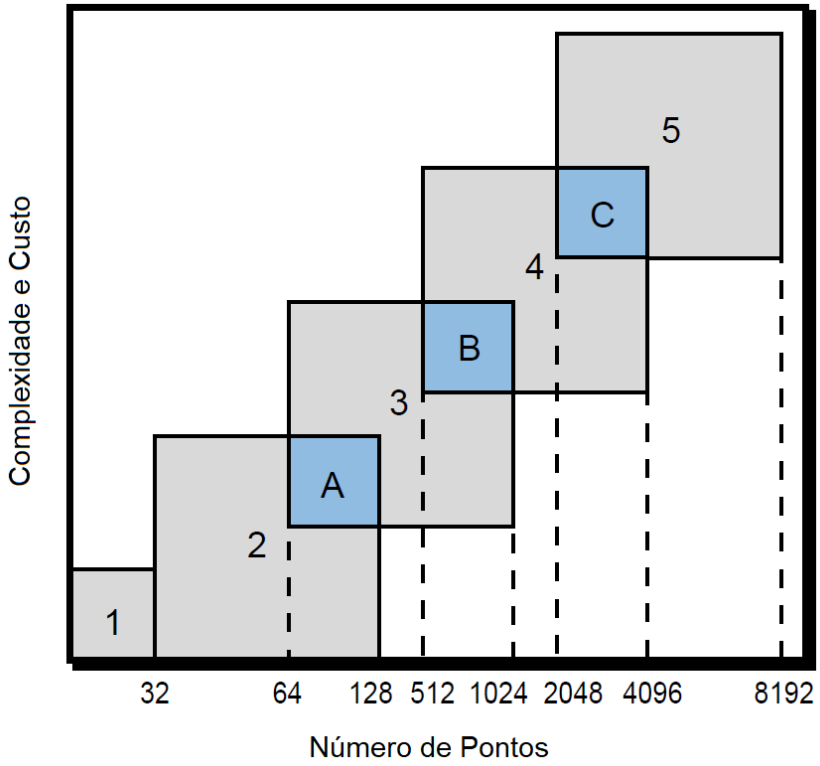
Existem diversos tipos de aplicações para as quais os CLPs são adequados a desempenhar. Naturalmente, é necessário fazer um levantamento de requisitos específicos à aplicação antes de se avaliar qual equipamento é o mais apropriado utilizando-se alguma métrica. O CLP de mercado da série Nexto, disponível no laboratório do Instituto de Informática, em função de suas características, não possui o desempenho necessário para aplicações com requisitos de desempenho muito altos, considerando o tempo de ciclo como métrica, como aplicações que exijam um tempo de ciclo tão rápido quanto 2 ms.

Cada tipo de CLP é desenvolvido com o objetivo de se adequar a uma determinada gama de aplicações. Algumas das características que devem ser levadas em conta na escolha de um produto são a quantidade de pontos de entrada e saída e sua capacidade de expansão. Na prática, isso pode ser um fator limitante no tamanho do sistema de controle a ser desenvolvido.

Existem equipamentos com suporte a uma pequena quantidade de pontos, mas também existem alguns com suporte a grandes quantidades. De acordo com (BRYAN; BRYAN, 1997), podemos classificar os CLPs baseado na quantidade de pontos de I/O da seguinte forma:

1. CLPs micro
2. CLPs pequenos
3. CLPs médios
4. CLPs grandes
5. CLPs muito grandes

Figura 1.1: Gamas de CLPs (Fonte: Adaptado de (BRYAN; BRYAN, 1997))



Existem áreas de intersecção entre cada classe, representadas na Figura 1.1 pelas letras 'A', 'B' e 'C'. Isso ocorre pois certas funcionalidades e opções podem ser adicionadas aos equipamentos levando em conta a aplicação específica a que eles serão destinados. Dessa forma, ocorre um casamento melhor entre os CLPs e a aplicação, não sendo necessário a troca por equipamentos de uma classe imediatamente superior.

A série Nexto, por exemplo, é orientada para alta disponibilidade com soluções de redundância e suporte a quantidades médias de pontos de entrada e saída. Assim, é possível dizer que a série Nexto apresenta um nicho de aplicação amplo, não focando

especificamente em um tipo de aplicação. Outros equipamentos podem ter mais recursos de hardware, ou serem desenvolvidos para aplicações mais específicas, podendo ser mais otimizados para essas aplicações do que CLPs mais gerais. Um exemplo disso é mostrado na Tabela 1.1.

Tabela 1.1: Comparativo de CLPs. (Fontes: (SÉRIE..., 2017), (BECKHOFF..., 2017b), (BECKHOFF..., 2017a), (MOTION..., 2017))

CLP	**CLP de Mercado**	**Tempo de Ciclo**	**Quantidade de Pontos**
Uso Geral	Nexto (Altus)	Ordem de 5 ms	> 100 pontos
Uso geral com processador mais poderoso	CX (Beckhoff)	Ordem de 3 ms	> 100 pontos
Motion Control	Q-Series (Mitsubishi)	< 1 ms	< 50 pontos

Tendo em vista as características da série, o que se deseja é poder utilizar os controladores Nexto em sistemas de controle com requisitos muito fortes de desempenho, onde são necessários um rápido tempo de ciclo e uma baixa variação de tempo (*jitter*) na atualização de entradas e saídas. Em aplicações industriais, é importante conseguir prever, com maior exatidão possível, o comportamento do sistema, conceito chamado de determinismo (KOPETZ, 2011). Em certos casos, vidas humanas podem ser afetadas pelo comportamento imprevisível de um sistema de controle, fazendo com que o determinismo seja de suma importância. Assim, um *jitter* de atualização pequeno é muito desejável, uma vez que aumenta a previsibilidade do sistema.

O objetivo deste trabalho é realizar um estudo sobre possíveis otimizações e melhorias que possibilitem o uso dos CLPs Nexto nesses sistemas. Para isso, serão apresentados testes que verifiquem o estado da situação atual no que diz respeito a desempenho, bem como uma análise sobre as possíveis causas e soluções.

2 CONTROLADORES PROGRAMÁVEIS

CLPs são, em essência, iguais aos computadores de uso pessoal: recebem uma entrada, que é processada, e produzem uma saída, de acordo com sua programação. A grande diferença se dá na tarefa atribuída ao dispositivo. Computadores de uso pessoal são otimizados para tarefas de uso geral. CLPs são otimizados para tarefas de controle, geralmente de tempo real, e para suportar ambientes industriais hostis. Isso inclui variações de temperatura, umidade, vibração mecânica, e ruído. (BOLTON, 2015)

2.1 Entradas e Saídas

A ideia principal de operação de um CLP é bastante simples: existe um programa, ou mais especificamente, um sistema de controle que foi desenvolvido para reagir/tomar decisões baseado em informações coletadas do processo em questão. Essas informações chegam ao controlador através de suas entradas, que estão conectadas a sensores, equipamentos que convertem grandezas elétricas e mecânicas, entre outras, em sinais padrão para serem tratados pelos controladores. Similarmente, as decisões tomadas pelo sistema de controle chegam ao mundo exterior através das interfaces de saída. As interfaces de saída são ligadas a atuadores, componentes que possuem uma alimentação e recebe um sinal de controle, convertendo a energia em movimento, ou a emissores de luz, válvulas, entre outros.

É importante notar como a robustez do controlador é influenciada por todas as partes. Por exemplo, é extremamente indesejável que um surto de tensão em uma das entradas danifique componentes centrais, ou qualquer outro componente. De fato, é necessário que os controladores sejam meticulosamente planejados para evitar resultados desfavoráveis.

Tanto as entradas quanto as saídas podem ser de dois tipos: digitais ou analógicas. Variáveis digitais possuem apenas dois níveis lógicos: nível alto e nível baixo. Internamente no CLP, variáveis analógicas possuem uma quantidade discreta de estados, representando um sinal analógico, que é contínuo. A quantidade de estados é dependente da resolução do respectivo *hardware*, podendo ter diversos valores intermediários entre o nível lógico alto e o baixo (BRYAN; BRYAN, 1997).

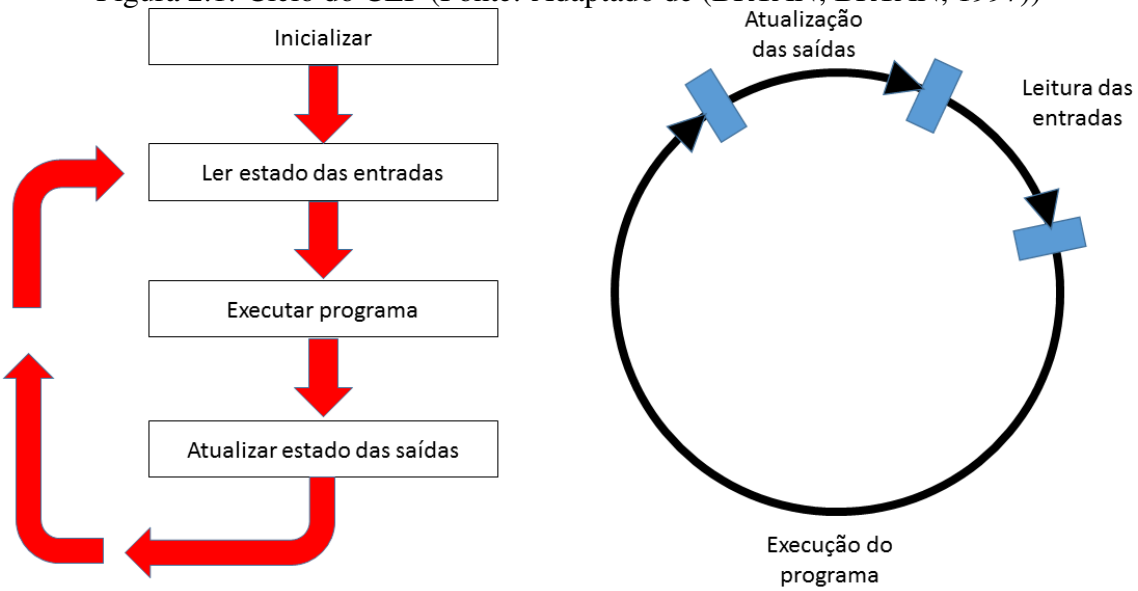
A decisão de quando deve ser utilizado cada tipo depende da aplicação em questão. Entradas e saídas digitais são adequadas para representar sinais discretos. Um interruptor,

por exemplo, que pode estar ligado ou desligado, mas nunca em um estado intermediário. Entradas e saídas analógicas são adequadas para representar sinais contínuos, como uma tensão, por exemplo, cujo valor será discretizado para ser interpretado por aplicações desenvolvidas para rodar no CLP.

Uma outra possível aplicação para entradas analógicas é um controle de temperatura. Temperatura é uma grandeza contínua, que possui tantos valores quanto se desejar (ou quanto a resolução do *hardware* permitir). Para que a informação de temperatura possa ser utilizada pelo CLP, é preciso transformar a temperatura em um sinal que possa ser interpretado. Para isso, é usado um sensor de temperatura. Por exemplo, o sensor conectado a uma entrada analógica pode gerar um sinal de tensão, onde o valor corresponde a uma determinada temperatura. O CLP poderá, então, realizar leituras a essa entrada, convertendo o valor lido para uma escala que seja conveniente ao programa de aplicação.

A leitura de entradas, execução do programa e escrita de saídas são feitas de forma cíclica no tempo. O ciclo do CLP é apresentado a seguir, na Figura 2.1.

Figura 2.1: Ciclo do CLP (Fonte: Adaptado de (BRYAN; BRYAN, 1997))



2.2 Tipos de CLPs

Usualmente, CLPs são classificados quanto ao seu design mecânico: os compactos e os modulares. Ambos os tipos apresentam as mesmas funcionalidades básicas.

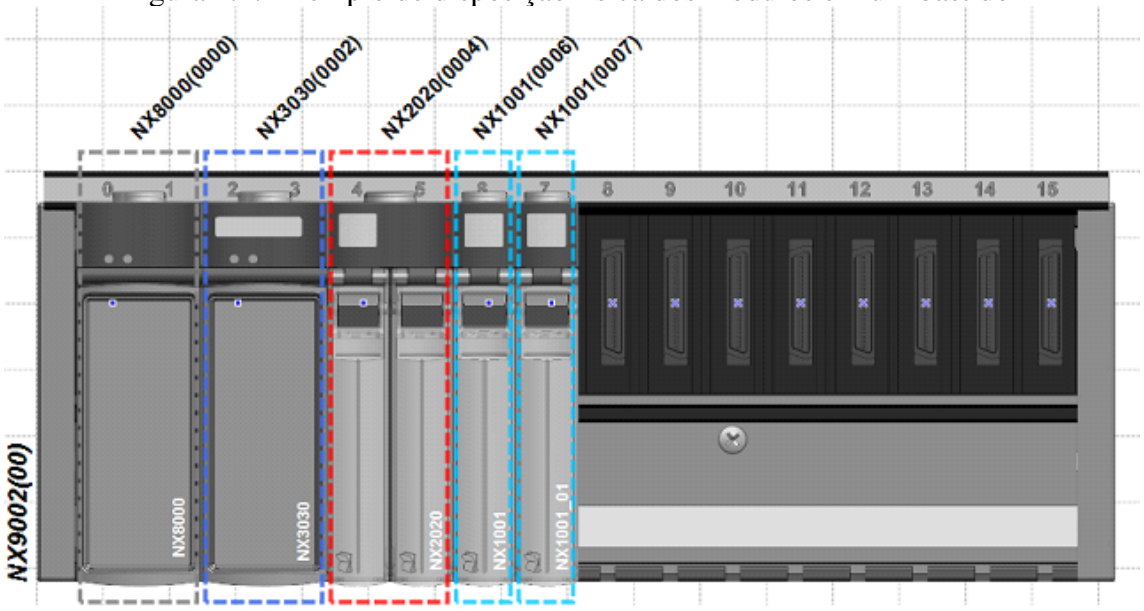
Os chamados "compactos"contêm todos os componentes que formam um CLP (alimentação, processador, memória e unidades de entrada/saída) integrados em uma

único equipamento. Esse tipo geralmente é utilizado para sistemas de menor porte e que necessitem de menos recursos.

CLPs do tipo modular possuem os mesmos componentes que os compactos, mas cada um deles é uma unidade separada, chamada de módulo. Os módulos são conectados a um bastidor, permitindo que se escolha exatamente quais módulos são adequados para o sistema de controle a ser desenvolvido.

Sistemas que necessitam de uma maior quantidade de recursos geralmente acabam por usar CLPs do tipo modular. Em um CLP modular, é muito mais fácil expandir a quantidade de entradas e saídas, ou a quantidade de memória, simplesmente adicionando-se novos módulos ao bastidor.

Figura 2.2: Exemplo de disposição física dos módulos em um bastidor



Na Figura 3.3, um exemplo de CLP modular, disponível no laboratório, utilizando os seguintes componentes: bastidor, fonte externa, CPU, módulo de saída digital e dois módulos de entrada digital.

2.3 Norma IEC 61131-3

A norma IEC 61131-3 é vista como um conjunto de orientações para programação de CLP. Fabricantes de CLPs precisam provar em quais partes o padrão foi cumprido ou não. Atualmente, essa norma é aceita pela maioria dos fabricantes. Várias vantagens são obtidas com o seu uso, e.g., reusabilidade de código, portabilidade, diminuição na curva

de aprendizagem dos usuários (JOHN; TIEGELKAMP, 2010).

Dentre as definições contidas na norma, está a das linguagens de programação adequadas para a programação de CLPs. As seguintes linguagens estão definidas:

- *Ladder diagrams* (LD): Descrição visual de circuito de automação por relé. Por isso, é de fácil compreensão por equipes de manutenção que estejam acostumadas a lidar com diagramas de relé. Além disso, pelo mesmo motivo, é de fácil utilização para a elaboração de lógicas de intertravamento.
- *Instruction list* (IL): Linguagem textual de baixo nível, onde cada linha representa uma operação simples. Pode ser considerado como uma versão textual da linguagem Ladder (BOLTON, 2015). Geralmente utilizado para programas de pequenos, bastante diretos e simples.
- *Sequential function charts* (SFC): Linguagem gráfica, de fácil visualização. É conveniente para controles de atividades sequenciais, onde seja necessário implementar uma máquina de estados.
- *Structured text* (ST): Esta linguagem possui grande semelhança com Pascal. Programas são escritos como uma série de sentenças separadas por um ponto e vírgula. Através de sentenças e sub-rotinas predefinidas, as variáveis do programa podem ser usadas ou terem seus valores trocados. Variáveis são, neste caso, valores definidos internamente, ou valores associados a entradas ou saídas.
- *Function block diagrams* (FBD): Linguagem gráfica onde o programa é descrito através de diversos blocos funcionais interligados. Um bloco funcional é um objeto que, quando executado, produz uma ou mais saídas. Também facilita a reutilização de código devido a sua modularidade.

Neste trabalho, para fins de testes de desempenho, será utilizada a linguagem ST. A escolha é justificada pela facilidade de programação da linguagem, além de que grande parte das bibliotecas internas do CLP foram implementadas em ST.

2.4 Execução: Tarefas

Uma tarefa é um elemento de controle de execução capaz de invocar a execução de um conjunto de unidades de organização de programas (POUs - *program organization units*), que incluem programas e blocos funcionais. A execução pode ser de forma periódica ou na ocorrência de eventos, como a detecção da transição do valor de uma variável

booleana, por exemplo (International Electrotechnical Commission and others, 1993).

Essas definições fazem parte da norma IEC 61131-3, e são válidas para todos os equipamentos que a seguem, incluindo os CLPs da série Nexto. Um dispositivo que segue a norma pode, ainda, ter funcionalidades extras, como outros tipos de tarefa, por exemplo.

Para as tarefas de execução periódica, é necessário definir um intervalo de ativação. A cada vez que transcorrer um tempo igual ao intervalo de ativação, a tarefa deverá ser escalonada. A resolução do intervalo é dependente de implementação, não sendo definida pela norma.

Cada tarefa deve receber um valor que representará sua prioridade, cujo objetivo é definir a prioridade de escalonamento das POUs associadas a esta tarefa. O valor zero representa a maior prioridade possível, enquanto valores maiores representam prioridades menores.

A prioridade de uma tarefa pode ser utilizada em dois tipos de escalonamento. São eles:

- Não-preemptivo: os recursos são liberados uma vez que a execução da tarefa esteja completa. Neste momento, a tarefa com maior prioridade começará sua execução. No caso de haver mais de uma tarefa com a mesma prioridade, aquela que estiver mais tempo esperando pelo recurso deverá começar sua execução.
- Preemptivo: quando uma tarefa é escalonada, ela pode interromper a execução de uma tarefa de menor prioridade que tenha alocado o mesmo recurso. Isso significa que a tarefa menos prioritária aguardará pelo término da execução da tarefa mais prioritária. Uma tarefa não pode interromper outra com prioridade igual ou maior do que a sua própria.

2.4.1 Tarefas na Série Nexto

A série Nexto é compatível com a norma IEC 61131-3, mas existem certos graus de liberdade de implementação em pontos que não são definidos na norma. O tipo de escalonamento implementado é o preemptivo, onde uma tarefa pode interromper outra com menor prioridade.

Os tipos de tarefas implementados estão de acordo com a norma. Contudo, existem algumas peculiaridades específicas desta implementação. Na série Nexto, os seguintes tipos de tarefas estão disponíveis:

- Tarefa Cíclica (*Cyclic Task*): Executada em intervalos regulares de tempo, periodicamente. Este tipo de tarefa é equivalente à tarefa periódica, definida pela norma.
- Tarefa por Evento (*Event Task*): Executada, apenas uma vez, na ocorrência da borda de subida da variável booleana especificada. No caso da série Nexto, estes eventos podem ser de *software* ou de *hardware*.
- Tarefa Contínua (*Continuous Task*): Tarefa com execução contínua, também chamada de execução livre, onde não existe um intervalo de ativação definido. A tarefa executa e é escalonada o mais rápido possível, sem qualquer restrição de tempo. Na prática, isso significa que o período de execução pode ser variável, e será mais longo caso a tarefa necessite de mais tempo para finalizar seu processamento.

3 SISTEMAS DE TEMPO REAL E DETERMINISMO

Um sistema de tempo real é definido como uma atividade ou sistema de processamento que precisa responder a entradas geradas externamente dentro de um período finito e especificado (FIELD-RICHARDS, 1983) (conforme citado em (BURNS; WELLINGS, 2001)). A dimensão deste tempo depende do tipo de aplicação. Por exemplo, um sistema de controle de um míssil provavelmente terá um intervalo de tempo menor dentro do qual o sistema precisa responder do que um sistema de montagem de automóveis.

No contexto de sistemas de tempo real, determinismo temporal significa que o comportamento do sistema seja previsível em termos de valores e tempo (KOPETZ, 2011). Em um sistema de frenagem, por exemplo, não é suficiente garantir que a ação de frenagem vai ser executada em algum momento posterior. Neste caso, seria necessário impor mais restrições, como definir que a ação de frenagem ocorrerá 2 ms após o seu acionamento.

Assim, relacionando esses conceitos com os tipos de tarefas anteriormente apresentados, é possível perceber que tarefas contínuas não são adequadas para aplicações que necessitem de determinismo. Uma vez que não existe um tempo definido entre cada execução, não é possível saber antecipadamente em que momentos a tarefa executará.

3.1 *Jitter* de Atualização de I/O

Jitter é definido como a diferença entre o menor e o maior valor de atraso. Neste caso, mais especificamente, esse atraso é relativo ao momento de atualização de I/O. O ciclo de atualização de I/O foi explicado na Seção 2.1. Analisando-se a Figura 2.1, podemos notar um inerente problema: a atualização das saídas ocorre após a execução do programa do usuário. Ou seja, o determinismo da atualização das saídas ficará comprometido, já que dependerá do tempo de execução da lógica de usuário.

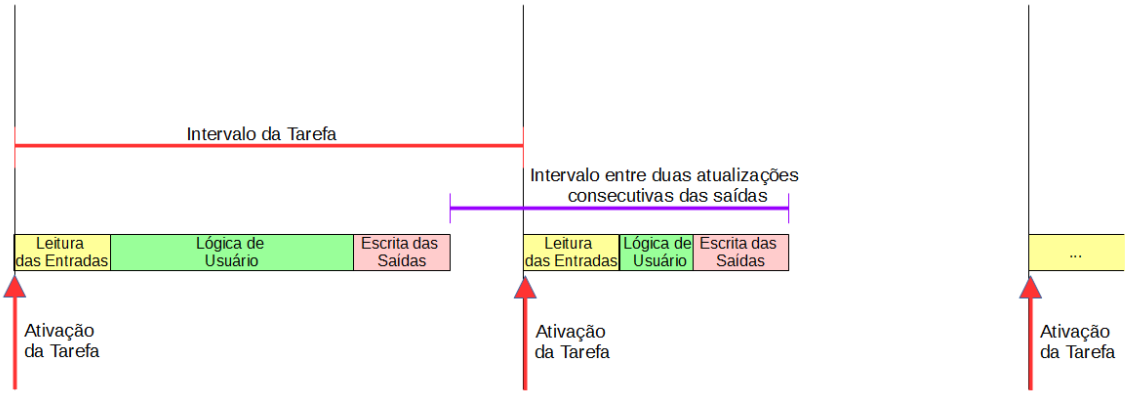
É possível visualizar esse problema com o auxílio da Figura 3.1. Supondo uma tarefa cíclica, i.e., que possui um intervalo entre duas ativações consecutivas I_{tarefa} predefinido, e supondo que a tarefa inicie sua execução imediatamente após ser escalonada (sem qualquer perda de generalidade); idealmente, assim como o tempo entre duas execuções da tarefa, o tempo entre duas escritas consecutivas deveria ser igual ao intervalo de ativação da tarefa. Contudo, sendo a atualização das saídas realizada imediatamente

após a execução da lógica de usuário, haverá um *jitter* de saídas J_o tal que:

$$0 \leq J_o < 2I_{tarefa}$$

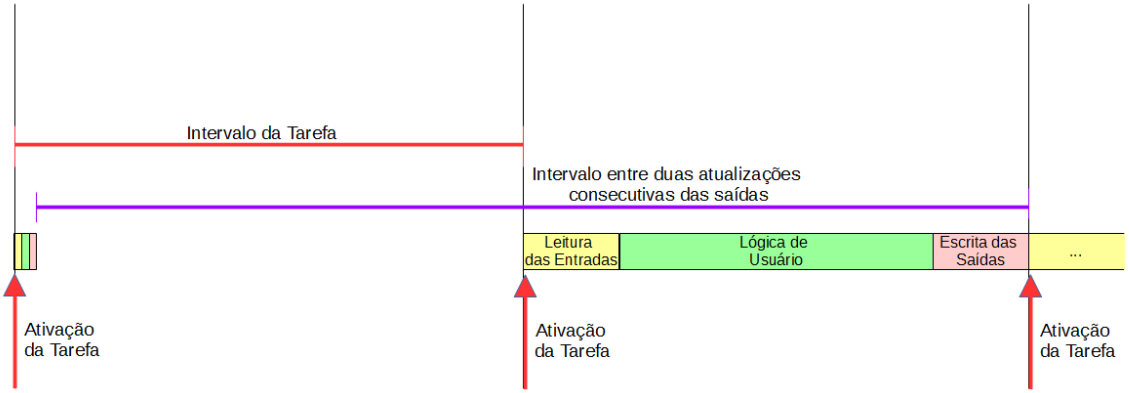
No caso específico onde o final da atualização das saídas coincidir no mesmo ponto de

Figura 3.1: Execução hipotética de uma tarefa com atualização de entradas e saídas



tempo dentro do intervalo de ativação da tarefa, não haverá *jitter*. Contudo, salvo esse caso, em todos os outros existirá essa variação no tempo. Considerando o pior caso, onde em um ciclo tanto os tempos de atualização de I/O quanto o tempo de execução de lógica de usuário são mínimos no primeiro ciclo, e no ciclo seguinte esses tempos são máximos, teremos um *jitter* que não excede duas vezes o tempo do intervalo de ativação da tarefa. Além disso, mesmo esses tempos sendo mínimos, eles existirão (mesmo que arbitrariamente pequenos), e isso justifica o porquê o valor do *jitter* de atualização estar contido no intervalo semi-aberto à direita $[0, 2I_{tarefa})$. Isso pode ser visualizado na Figura 3.2.

Figura 3.2: Exemplo de pior caso de *jitter* na atualização das saídas



Nos casos apresentados nas Figuras 3.1 e 3.2, os comandos de leitura de entradas e escrita de saídas são enviados em momentos separados. É possível agrupá-los e transmití-los juntos, restando apenas decidir qual o melhor momento para isso: na ativação da tarefa, ou após a execução da lógica de usuário. Ambas as opções apresentam vantagens e desvantagens, conforme analisado a seguir:

- Envio dos comandos após a execução da lógica de usuário: neste caso, apresentado na Figura 3.2, o determinismo de atualização de entradas e saídas será afetado devido a variação no tempo de execução da lógica de usuário. Além disso, na primeira execução, as entradas não estarão com seus valores atualizados. Este método é mais adequado para quando o intervalo de ativação da tarefa é grande, pois suas saídas serão escritas logo após o término da execução do programa, independentemente do tempo de intervalo.
- Envio dos comandos no momento de ativação da tarefa: a atualização das saídas ocorrerá na mesma cadência de ativação da tarefa, ou seja, manterá seu determinismo. Além disso, as entradas serão lidas antes do início da execução da lógica de usuário, o que significa que seus valores estarão atualizadas desde a primeira execução. A desvantagem aparece no momento em que o intervalo de ativação da tarefa é bastante grande. Ao final da execução do programa, embora os valores das saídas já tenham sido calculados, somente na próxima ativação da tarefa é que esses valores serão de fato escritos.

3.2 Atualização de I/O na Série Nexto

A comunicação entre os módulos Nexto gerada pela atualização de entradas e saídas é feita através de um protocolo de comunicação. No equipamento disponível, o protocolo usado é o EtherCAT, um protocolo de comunicação industrial de tempo real e alta performance baseado em Ethernet.

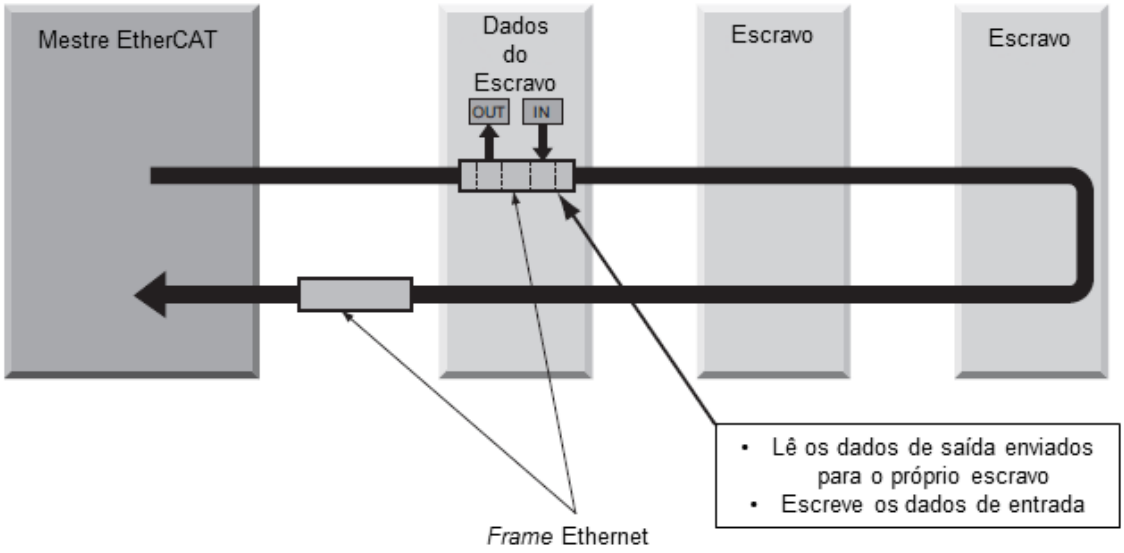
O EtherCAT funciona em uma configuração mestre/escravo, onde o mestre envia comandos aos nodos escravos, que podem ler e/ou escrever os dados (OMRON Automation Pvt Ltd, 2016). Através deste mecanismo, a atualização de I/O acontece ciclicamente no tempo. O objetivo é maximizar a utilização do canal de comunicação e minimizar os tempos de resposta.

Os comandos são inseridos no segmento de dados de um telegrama EtherCAT.

Um telegrama é composto por um ou mais endereços de escravos, dados e alguns bits de controle. Esses telegramas são inseridos como *payload* em um *frame* Ethernet. Cada um desses *frames* pode conter diversos telegramas EtherCAT.

Um *frame* enviado pelo mestre é repassado apenas para o primeiro nodo escravo, que o repassa ao próximo, e assim sucessivamente. A leitura ou inserção de dados se dá neste momento, quando o *frame* está sendo transmitido, ou seja, não ocorre uma pausa na transmissão: os dados são lidos/inseridos *on the fly*. Uma vez que o *frame* chegue no último nodo escravo, este o enviará de volta ao mestre, utilizando-se da comunicação *full-duplex* do Ethernet. Assim, o tempo de processamento de cada escravo é praticamente desconsiderável.

Figura 3.3: Comunicação EtherCAT (Fonte: Adaptado de (OMRON Automation Pvt Ltd, 2016))

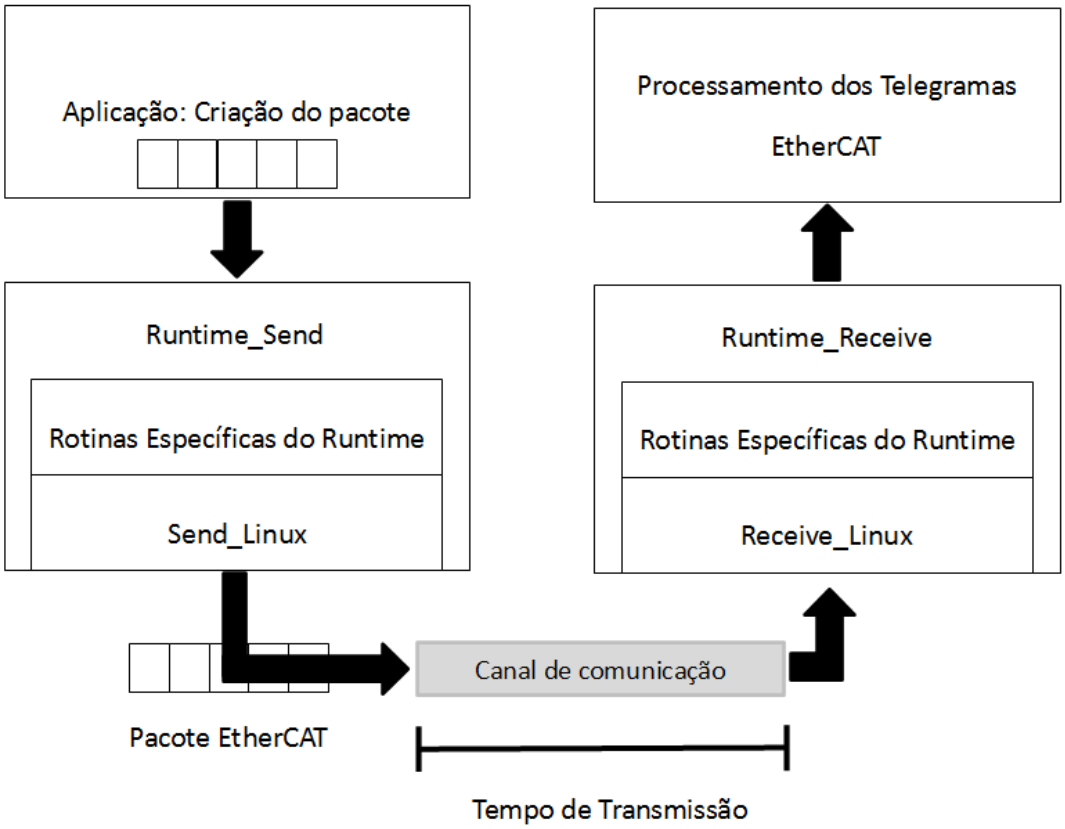


Embora o protocolo EtherCAT em si possua alta performance e determinismo temporal, existem outros fatores externos ao protocolo que podem diminuir o desempenho, como as rotinas de processamento dos pacotes, o gerenciamento do envio dos pacotes, entre outros. No caso dos CLPs Nexto, a atualização de I/O ocorre pelo menos na tarefa principal (*MainTask*). Outras tarefas também podem desempenhar esta função, mas isso depende de cada aplicação (ALTUS, 2016). Na prática, isso significa que mais de uma tarefa pode ser responsável pelo envio e recebimento de pacotes. Deste modo, dependendo de quão eficiente for a implementação, o envio e recebimento de pacotes pode ter um grande impacto no desempenho.

No Nexto, o sistema operacional usado é o Linux, onde existe um *runtime* que é responsável pela execução de aplicações compatíveis com a norma IEC 61131. Para a

comunicação, especificamente, existem rotinas implementadas pelo *runtime* para o envio e recebimento de pacotes. Contudo, por esta solução se tratar de uma solução proprietária, não é possível ter acesso à sua implementação, mas é possível estimar qual o tempo gasto nestas rotinas através de medições dos diversos segmentos que compõem a comunicação. O fluxo de envio e recebimento de um pacote pode ser mais facilmente compreendido com o auxílio da Figura 3.4, a seguir:

Figura 3.4: Fluxo de envio e recebimento dos Pacotes



Assim, caso seja desejável avaliar o desempenho da comunicação, é possível tentar isolar cada um dos blocos apresentados na Figura 3.4. Isolar o grande bloco "comunicação"em blocos menores permite uma avaliação mais precisa do estado atual.

3.3 Tempo de Ciclo e Latência de Execução de Tarefa

O tempo de ciclo é definido como o tempo entre a ativação da tarefa até o final da execução da mesma. Isso inclui o tempo gasto pela atualização de I/O e o tempo de execução da lógica do usuário, conforme explicado na Figura 2.1.

Os CLPs da série Nexto apresentam como tempo de intervalo mínimo 5 ms. Tanto aplicações como o sistema de frenagem mencionado na Seção 3, quanto aplicações de *motion control*, necessitam de tarefas que executem mais rapidamente. Atualmente, não é possível o uso desses CLPs nessas aplicações devido às características da série Nexto.

As medições dos tempos de ciclo médio, mínimo e máximo são apresentadas em uma tela de monitoração na própria ferramenta de programação. Existem monitores implementados em *software* pelo *runtime* que extraem estas informações através de medições.

Outra funcionalidade dos monitores é medir a diferença de tempo entre o momento em que uma tarefa foi invocada e o momento em que ela inicia sua execução propriamente dita. Esta diferença é chamada de latência de execução de tarefa (3S-Smart Software Solutions GmbH, 2012). Esta latência diminui o determinismo do sistema, e os monitores são ferramentas importantes que permitem o supervisionamento do sistema.

4 OBJETIVOS E TRABALHOS RELACIONADOS

Este capítulo apresenta os objetivos do trabalho, além de alguns trabalhos relacionados à área de avaliação de desempenho.

4.1 Objetivos

A meta principal do trabalho é compreender o estado da implementação utilizada atualmente na série Nexto, realizando a avaliação de desempenho temporal a fim de descobrir quais os fatores limitantes do tempo de ciclo e causadores de *jitter* de atualização de I/O, caso existam. Será utilizada uma abordagem *top-down*, começando a análise em tópicos de mais alto nível, isolando os componentes e permitindo a compreensão do sistema.

Sabendo-se que existem diversos componentes do sistema que podem ser otimizados em relação ao seu desempenho, o que se deseja é identificar qual deles é o gargalo. Naturalmente, melhorias podem muito provavelmente ser feitas em qualquer um deles, mas deseja-se identificar o componente cuja otimização terá o maior impacto possível. Sintetizando o que foi apresentado nas seções anteriores, os principais pontos a serem observados para avaliar a situação atual são:

- *Jitter* de atualização de I/O;
- Tempo de ciclo;

Mais especificamente, deseja-se verificar:

- Se existe dependência entre o tempo gasto na execução da lógica de usuário e o momento em que a atualização de entradas e saídas ocorre, conforme Seção 3.1;
- Qual a ordem de grandeza do tempo de ciclo dada a implementação atual, e determinar o quanto cada funcionalidade/componente contribui para o total deste tempo;

O estudo conduzido neste trabalho primeiro verificará a existência de *jitter* de atualização de I/O e a ordem de grandeza do tempo de ciclo. Estas informações serão obtidas através da realização de experimentos com monitoração. Em seguida, o método proposto será utilizado para facilitar o isolamento dos componentes e a identificação de suas respectivas contribuições para o tempo de ciclo total observado.

4.2 Trabalhos Relacionados

Este trabalho consiste em uma avaliação de desempenho de uma implementação única, utilizada em equipamentos específicos. Por este motivo, não existem outros trabalhos especificamente relacionados com a série Nexto. Contudo, existem outros trabalhos de avaliação de desempenho que utilizam diferentes ambientes, ferramentas e equipamentos, ou mesmo trabalhos de cunho teórico.

Em (JARP, 2002), o autor propõe identificar gargalos em programas através do uso de *performance counters*. Com esse mecanismo de monitoração, são obtidas informações úteis sobre o programa em execução, como: quantidade de ciclos de processamento, número de instruções, ciclos de *stall* (causados por eventos como *cache miss*, predição de desvios incorreta, entre outros) e atividade dos diversos níveis de cache. Essas informações por si só não conduzem a um melhor desempenho de execução, mas são muito úteis para ajudar na identificação de gargalos.

Em (JAIN, 1990), são descritos critérios para a seleção de parâmetros para serem testados a fim de se identificar o gargalo do sistema. Parâmetros que não dizem respeito ao componente cujo desempenho se deseja melhorar podem ser omitidos, simplificando a análise. Também é descrito o uso de monitores, que são ferramentas utilizadas para observar atividades de uma determinada parte do sistema. Um monitor observa o desempenho do sistema, coleta dados, forma estatísticas, analisa e exibe os dados.

Também existem trabalhos que avaliam o desempenho baseados em aplicação de *workloads* e coleta de resultados, como em (MARIESKA; KISTIJANTORO; SUBAIR, 2011). Neste trabalho, o objetivo é avaliar o desempenho de um sistema operacional de tempo real quando usado em dispositivos que realizam comunicação na rede. Para tanto, são utilizadas três métricas: tempo de processamento dos pacotes, *jitter* (variação de tempo gasto para processar cada pacote) e a vazão, medida em número de pacotes processados em um determinado período.

5 MÉTODO PROPOSTO

Para apresentar o método proposto, este capítulo será dividido em duas seções: avaliação da situação atual e investigação de problemas.

5.1 Avaliação da Situação Atual

Para averiguar se os equipamentos Nexto estão de acordo com os requisitos, será utilizada uma abordagem que consiste em experimentos que simulem uma carga de trabalho semelhante a uma aplicação real, mas contando com um ambiente controlado e podendo ser repetidos inúmeras vezes. Esta carga de trabalho aplicada a estudos de *performance* é chamada de *workload* de teste. No estudo, como as cargas de trabalho não são provenientes de aplicações reais, a carga de trabalho é chamada de *workload* sintético (JAIN, 1990).

Para a programação dos experimentos, será utilizada a linguagem ST, uma das linguagens definidas na norma Norma IEC 61131-3, conforme discutido na Seção 2.3. As justificativas para a escolha são a facilidade de programação, devido a semelhança da linguagem com Pascal, e o fato de a biblioteca que implementa comunicação nos dispositivos estudados ter sido programada nesta linguagem. Assim, os experimentos podem ser mais facilmente desenvolvidos e seus resultados mais facilmente monitorados e analisados.

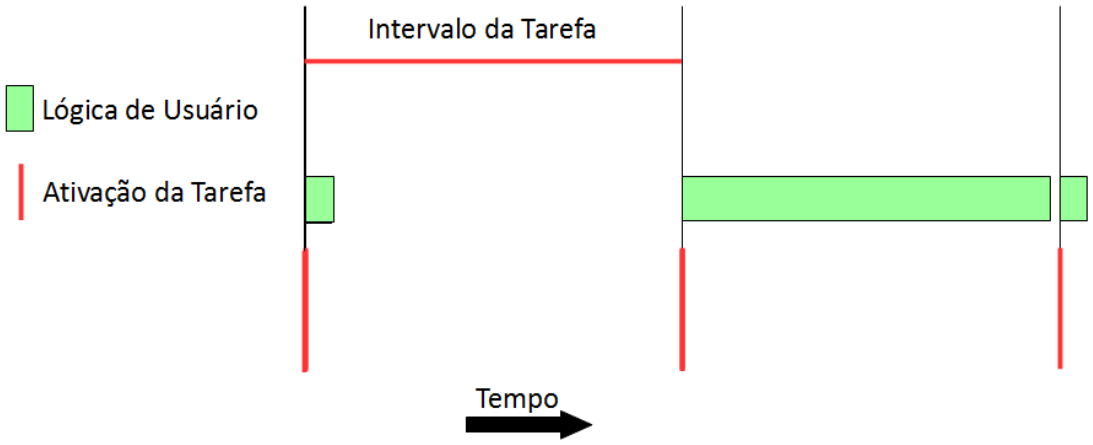
A avaliação da situação atual será dividida em duas partes:

- Teste para verificar a existência de *jitter* na atualização de I/O;
- Teste para verificar se o intervalo de ativação mínimo disponibilizado na ferramenta de programação é adequado;

5.1.1 Teste para Verificar Existência de *jitter* na Atualização de I/O

O primeiro teste proposto consiste em criar uma tarefa cíclica, definida conforme a Seção 2.4, com um intervalo de ativação bastante grande para a tarefa. Bastante grande, neste caso, significa que o tempo de duração da tarefa será praticamente desprezível quando comparado ao intervalo de ativação. Um exemplo da execução de uma tarefa pode ser visualizado com auxílio da Figura 5.1, a seguir.

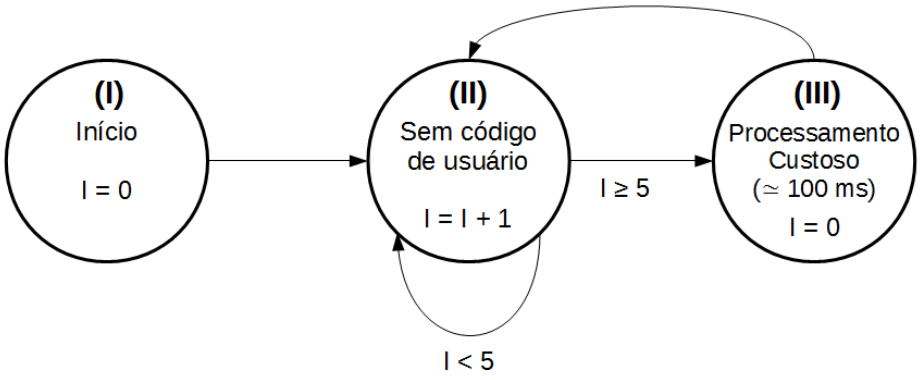
Figura 5.1: Execução de uma tarefa cíclica



O objetivo do teste é verificar se existe dependência entre o tempo gasto na execução de código de usuário e o momento em que ocorre a atualização de I/O. Deste modo, é possível avaliar essa dependência através da variação do tempo de ciclo da tarefa e subsequente monitoração do sistema.

Para provocar esta variação no tempo de ciclo da tarefa, propõe-se uma aplicação cujo comportamento pode ser representado pela seguinte máquina de estados:

Figura 5.2: Máquina de estados da aplicação do teste de *jitter* de I/O



Após o início, representado pelo estado (I), a tarefa passará para o estado (II), onde nenhum código de usuário será executado. Após cinco ciclos, número arbitrariamente escolhido, a tarefa passará para o estado (III), onde executará um código bastante custoso. Este código ocupará a maior parte do tempo de intervalo da tarefa. Neste caso, 100 ms.

Propõe-se utilizar uma saída digital, cujo valor será alterado a cada ciclo. Assim, a monitoração do sistema pode ser feita com uso de um osciloscópio, por exemplo. Caso o momento de envio dos *frames* de atualização de I/O dependa do tempo de execução da

tarefa, analisando-se o sinal da saída monitorada, será possível detectar algum deslocamento no eixo do tempo quando a aplicação estiver no estado (III), devido ao tempo de processamento.

O intervalo de ativação da tarefa escolhido para o teste foi o máximo permitido pela ferramenta de programação, 100 ms. Uma justificativa é que deseja-se minimizar a possível influência de fatores externos na tarefa, como interrupção por tarefas mais prioritárias, por exemplo. No estado (II), a tarefa permanecerá a maior parte do tempo entre ativações sem realizar qualquer processamento, diminuindo possíveis interrupções que poderiam causar variações no momento de envio dos *frames*.

Outra justificativa é a preocupação com a precisão das medidas. Como o método de monitoração será através de um osciloscópio, é necessário avaliar se a precisão do equipamento é alta o suficiente para se atingir o objetivo do teste. Através da escolha de um intervalo de ativação bastante grande, como 100 ms, podemos minimizar os erros de precisão do instrumento utilizado, pois a ordem de grandeza do intervalo será proporcionalmente maior em relação à ordem de grandeza da precisão do equipamento.

As especificações da precisão do osciloscópio utilizado, conforme (AGILENT..., 2017), são:

- Precisao da base de tempo: 100 ppm, ou 0,01%;
- Taxa de amostragem: 500 milhões de amostras por segundo;

Assim, para um intervalo de ativação de 100 ms, teremos:

$$(500 \cdot 10^6)\ amostras/s \cdot 100\ ms = 50 \cdot 10^6\ amostras$$

Portanto, de acordo com as especificações, poderá haver um erro de:

$$0,01\% \cdot 50 \cdot 10^6\ amostras = 5000\ amostras$$

A essa taxa de amostragem, o erro do osciloscópio em segundos é:

$$\frac{5000\ amostras}{500 \cdot 10^6\ amostras/s} = 0,01\ ms$$

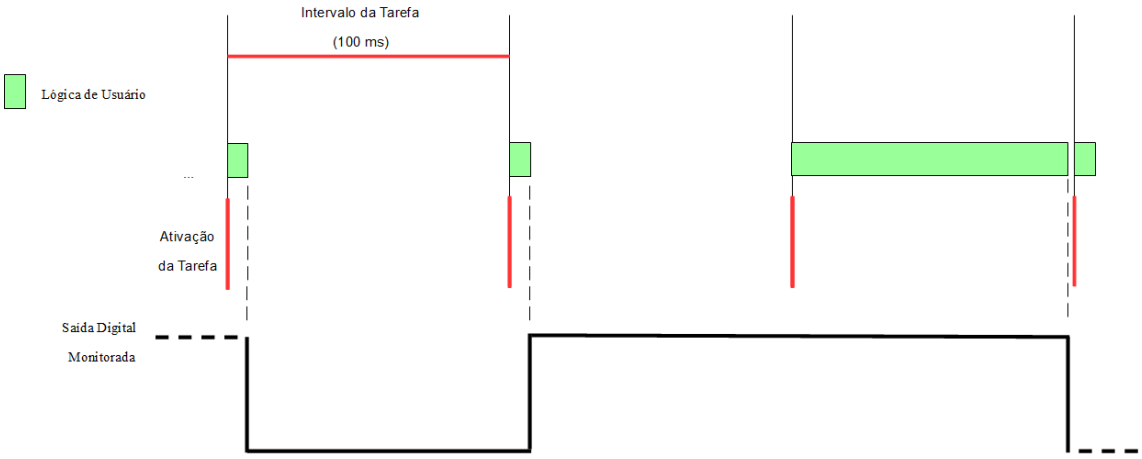
Em relação ao tempo do intervalo, o erro introduzido é:

$$\frac{0,01\ ms}{100\ ms} = 0,01\%$$

Portanto, embora o instrumento introduza uma parcela de erro, a magnitude do mesmo é desprezível. Assim, é possível dizer que o equipamento atende aos requisitos de precisão desejados.

Caso o *jitter* de fato exista, levando em conta a imprecisão associada ao equipamento de medição, haverá um deslocamento do sinal da entrada digital monitorada em relação ao eixo do tempo. Isso pode ser visualizado mais claramente com auxílio da Figura 5.3, apresentada a seguir:

Figura 5.3: Representação temporal com *jitter* da onda da saída digital monitorada

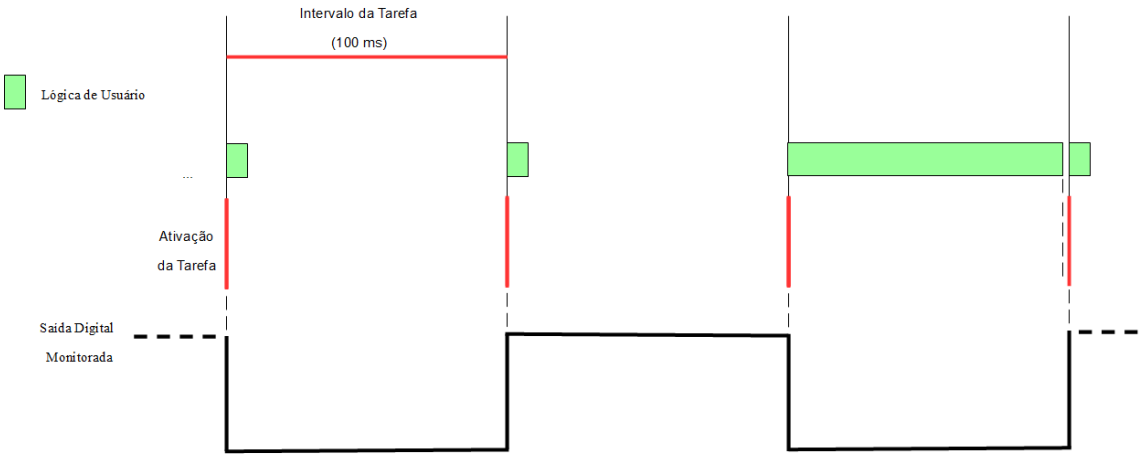


Para facilitar a compreensão, foram omitidas as parcelas de tempo correspondentes à leitura de entradas e à escrita de saídas. No caso específico da Figura 5.3, existe um *jitter* de aproximadamente 100 ms, que é inaceitável. Assim, é possível perceber o quanto essa situação afetaria o determinismo.

Idealmente, conforme discutido na Seção 3.1, a atualização de I/O deve ser independente do tempo de execução da tarefa. Caso a implementação atual obedeça a esse requisito, a medição deverá apresentar um resultado bastante semelhante à Figura 5.4, a seguir.

Para estimular o sistema, será aplicado um *workload* sintético. O seu código é composto por: um laço que é repetido um grande número de vezes, levando um tempo pouco menor do que o intervalo da tarefa. Esse laço é executado uma vez a cada 6 ciclos da tarefa. Nos outros ciclos, não existe nenhuma carga computacional intensiva. O princípio de funcionamento do *workload* é semelhante ao programa apresentado na Figura

Figura 5.4: Representação temporal sem *jitter* da onda da saída digital monitorada



5.3. O programa foi escrito em ST, linguagem pertencente à norma IEC 61131-3, e seu código fonte é apresentado na Figura 5.5. O número de iterações do laço foi definido empiricamente, especificamente para o equipamento utilizado nos testes.

Figura 5.5: Programa implementado para avaliação do *jitter* de atualização de I/O

```
PROGRAM UserPrg
VAR
        out : BOOL;
        load : INT := 0;
        count : DINT := 0;
        temp : INT := 0;
END_VAR

IF (load = 5) THEN
        FOR count := 0 TO 3160000 DO
                temp := temp+1;
        END_FOR
        load := 0;
END_IF
out := NOT out;
load := load + 1;
```

5.1.2 Teste para Avaliar o Tempo de Ativação Mínimo da Tarefa Principal

No segundo teste, o objetivo é verificar se o tempo de ativação da tarefa principal, *MainTask*, é adequado. Atualmente, a ferramenta de programação permite um intervalo de ativação mínimo de 5 ms para a tarefa principal. É possível que este tempo seja desnecessariamente grande, o que poderia restringir o uso dos equipamentos em certas aplicações. Portanto, este teste tem como meta avaliar esta possibilidade.

A avaliação será feita a partir da coleta do seu tempo de ciclo, utilizando-se os monitores presentes no sistema. Será criada uma tarefa vazia (ou seja, sem código de usuário), que realize comunicação, com tempo de ativação em 5 ms, e amostrado diversas vezes o seu tempo de ciclo. Serão amostrados os tempos mínimo, médio e máximo. A diferença entre o tempo máximo e o tempo mínimo será um dado importante para os próximos passos da avaliação. Mesmo para uma tarefa que não execute nenhum código de usuário, o tempo de ciclo não será nulo, uma vez que ainda se faz necessário a atualização de I/O, além de outras funções, como diagnósticos, que detectam e informam a ocorrência de falhas, por exemplo.

Dessa forma, se o tempo de ciclo da tarefa for próximo do tempo limite inferior definido, estará justificado o limite, pois a tarefa não pode ter um tempo de ciclo maior do que seu intervalo de ativação. Além disso, será considerada, para fins de avaliação, a recomendação do manual da ferramenta, que sugere um intervalo de ativação de pelo menos duas vezes o tempo de ciclo da tarefa (ALTUS, 2016).

Ainda, a tarefa criada no teste é como as tarefas utilizadas em aplicações reais, exceto pelo fato de que esta não realiza o controle de nenhum sistema. Logo, é possível extrair uma estimativa dos tempos de ciclo de uma tarefa completa, que realize todos os passos da comunicação. Este dado possibilitará comparações com casos mais restritos, onde certos componentes serão isolados, o que auxilia na detecção do gargalo.

5.2 Investigação de Problemas

Uma vez que tenham sido realizados os testes propostos, precisa-se investigar os componentes do sistema com o objetivo localizar possíveis problemas e possibilitar o aumento de desempenho. Suspeita-se que a comunicação seja um gargalo do sistema. Contudo, não podemos descartar a possibilidade de que o desempenho esteja sendo afetado por um conjunto de fatores. Para auxiliar nesta investigação, é proposto um método.

Primeiramente, deseja-se saber qual a ordem de grandeza do tempo mínimo, dada a implementação atual. Uma vez obtido este tempo, pode-se avaliar se são necessárias otimizações e estimar a magnitude das mudanças. Este passo já foi contemplado no teste apresentado na Seção 5.1.2.

Então, pode-se prosseguir para a identificação dos gargalos. O método proposto é apresentado na Figura 5.6 e tem o objetivo de servir como ferramenta para o processo de otimização. Os testes e as ações foram numeradas na Figura para auxiliar a compreensão.

Avaliar o desempenho sem comunicação (I): para determinar se o desempenho é de fato limitado pela comunicação, realiza-se um teste para avaliar o caso sem comunicação. Caso o desempenho não atinja a meta estabelecida, significa que a comunicação não é o principal gargalo. Sugere-se investigar funções de *callback* de outras funcionalidades do sistema que possam estar interrompendo a execução do programa. Além disso, sugere-se a utilização de um *profiler* para monitorar e analisar a execução. Se o desempenho for satisfatório, prosseguir para a avaliação (II).

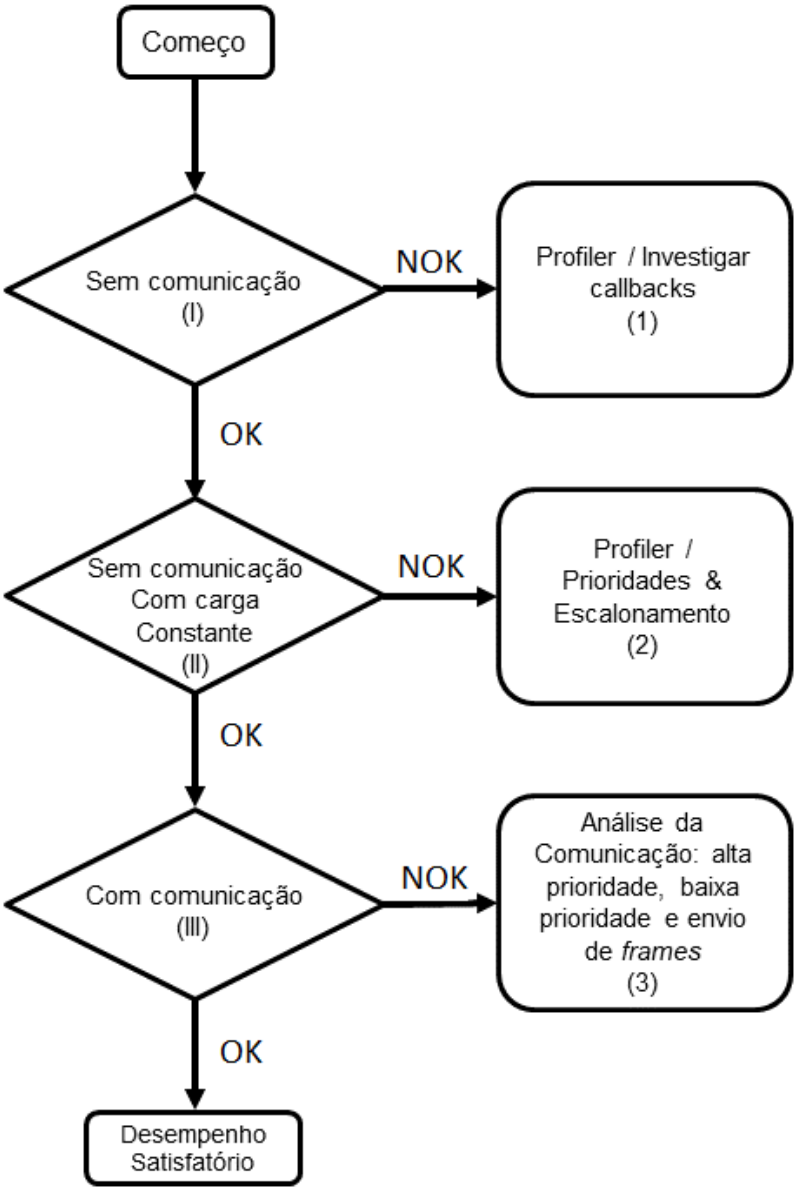
Avaliar o desempenho com uma carga computacional constante (II): executar um programa de usuário sem comunicação com carga computacional constante, que ocupe uma parcela do tempo de execução da mesma ordem de grandeza do tempo ocupado pela comunicação. Observar a magnitude da diferença entre o tempo de ciclo máximo e o tempo de ciclo mínimo, comparando com o valor da execução normal, com comunicação, obtido no teste apresentado na Seção 5.1.2. Através desta comparação, é possível determinar se existe uma variação inerente no tempo de ciclo causada pela comunicação, ou causada por outros fatores. Caso a execução apresente uma variação grande, também é possível que existam outras *threads* de usuário executando com prioridade maior causando interrupções no programa. Sugere-se investigar o escalonamento de tarefas, utilizando um *profiler* para determinar se a perda de desempenho é causada por problemas de prioridade. Se o desempenho for satisfatório, isto é, for constatado que o *jitter* é majoritariamente causado pela comunicação, prosseguir para a avaliação (III).

Avaliar o desempenho com comunicação (III): se o desempenho for degradado apenas quando utilizado comunicação, significa que a comunicação é realmente um dos gargalos. Portanto, é importante tentar isolar qual das funcionalidades implementadas através de comunicação é o problema. Sugere-se uma abordagem que trabalhe com quatro diferentes partes da comunicação:

- Troca de I/O, responsável pela atualização de entradas e saídas e, portanto, de alta prioridade;

- Diagnósticos do CLP, de baixa prioridade;
- Comunicação relacionada ao *discovery cycle*, que é responsável pelo descobrimento de novos módulos inseridos quando o barramento já está em execução;
- Interface de rede: possíveis problemas de gerenciamento de envio dos *frames* EtherCAT, como desempenho insatisfatório das rotinas de envio e recebimento implementadas pelo *runtime*, confome explicado na Seção 3.2. Possível compartilhamento de recursos com a interface de rede que realiza a comunicação com o *software* de programação.

Figura 5.6: Fluxograma do método proposto



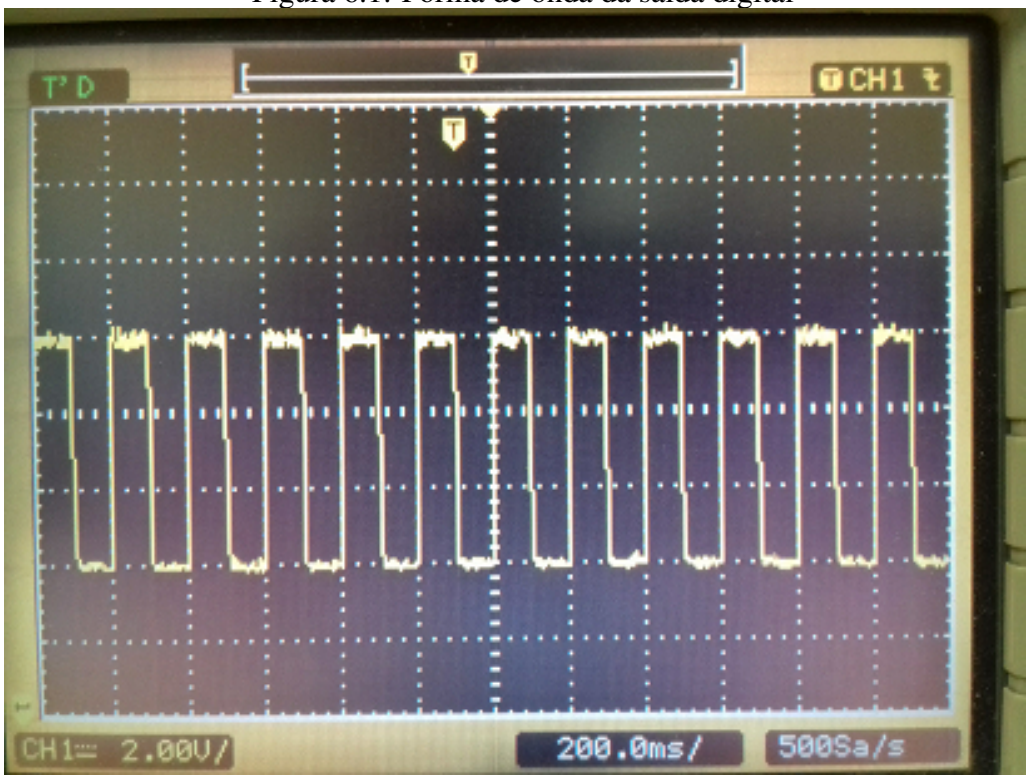
6 RESULTADOS OBTIDOS

Este capítulo apresenta os resultados obtidos a partir dos testes e métodos propostos, além de uma análise dos dados.

6.1 *Jitter* de Atualização de I/O

Assim, conforme proposto na Seção 5.1.1, o teste para verificar a existência de *jitter* na atualização de I/O foi executado. A saída digital foi monitorada com a utilização de um osciloscópio, e a sua forma de onda é apresentada a seguir, na Figura 6.1.

Figura 6.1: Forma de onda da saída digital



O experimento foi realizado dez vezes. Cada experimento compreendeu a execução de 24 ciclos, ou seja, a execução do código com processamento custoso quatro vezes, representado pelo estado (III) na Figura 5.2. Assim, em cada experimento, o tempo de monitoração foi de 2400 ms, que é o tempo correspondente ao número de ciclos da tarefa.

A Figura 6.1 apresenta o resultado da monitoração de um dos experimentos. Não houve diferenças notáveis no resultado dos outros experimentos, então, por simplicidade, é apresentado apenas o resultado de uma monitoração.

Analisando-se a forma de onda, percebe-se que a saída digital teve seu valor alte-

rado a cada 100 ms, o que é condizente com o intervalo de ativação definido. Além disso, mesmo nos ciclos onde o laço com grande custo computacional foi executado, a mudança na saída permaneceu corretamente cadenciada, não havendo indícios de variação no tempo de atualização, como os ilustrados pela Figura 5.3. Portanto, pode-se concluir que o controlador cumpre com o requisito de *jitter* de atualização de I/O, significando que o tempo de execução da lógica de usuário não interfere nas entradas e saídas.

6.2 Tempo de Ativação Mínimo da Tarefa Principal

Assim, conforme proposto na Seção 5.1.2, o teste para avaliar o intervalo de ativação mínimo da tarefa principal foi realizado. O objetivo do teste é determinar se o tempo de ativação mínimo é adequado através da avaliação do tempo de ciclo da tarefa.

Foram realizados diversos experimentos variando-se o número de amostras, com o objetivo de minimizar o tempo de teste sem perder confiabilidade nos resultados. Deseja-se utilizar uma quantidade de amostras que proporcione tempos de ciclo estáveis.

Realizando o experimento dez vezes com 3.500 amostras, obteve-se os tempos de ciclos mostrados na Tabela 6.1. O mesmo experimento foi repetido para o 7.000 amostras com o objetivo de comparar os resultados e avaliar se são necessárias mais do que 3.500 amostras. O resultado está na Tabela 6.2.

Comparando-se as médias dos tempos mínimo, médio e máximo de ambos os experimentos, percebe-se que há uma diferença percentual menor ou aproximadamente igual a 1%. Assim, para a análise dos dados, esta diferença não será considerada significativa. Como este caso de execução é o mais completo no sentido de que a tarefa está com todas suas funcionalidades habilitadas, assume-se que para os experimentos mais restritos, com algumas funcionalidades desabilitadas, o mesmo número de amostras será o suficiente. Portanto, os experimentos seguintes serão conduzidos com 3.500 amostras.

Considerando que o tempo de ciclo máximo observado foi de aproximadamente 2,6 ms, é justificável que o intervalo de ativação mínimo da tarefa seja 5 ms, levando em conta a recomendação do fabricante, conforme apresentado na Seção 5.1.2. É preciso manter uma certa fatia de tempo para a execução do programa do usuário. Com esta configuração, considerando o pior caso, tem-se aproximadamente metade do tempo de intervalo para execução de lógica.

Tabela 6.1: Resultados do experimento com 3.500 amostras para medir o tempo de ciclo de uma tarefa que realiza comunicação

		Tempo de Ciclo (μs)		
		Mínimo	Médio	Máximo
Num de Amostras	3.500	1.718	1.845	2.600
		1.747	1.894	2.548
		1.764	1.888	2.600
		1.721	1.859	2.536
		1.704	1.843	2.407
		1.793	1.906	2.637
		1.702	1.833	2.599
		1.756	1.873	2.544
		1.787	1.904	2.571
		1.760	1.912	2.607
Média		1.745,2	1.875,7	2.564,9
Máximo - Mínimo (μs)	819,7			
Desvio Padrão		32,682	29,143	64,226
IC (95%)		20,256	18,063	39,807

Tabela 6.2: Resultados do experimento com 7.000 amostras para medir o tempo de ciclo de uma tarefa que realiza comunicação

		Tempo de Ciclo (μs)		
		Mínimo	Médio	Máximo
Num de Amostras	7.000	1.741	1.871	2.534
		1.777	1.886	2.530
		1.766	1.869	2.452
		1.740	1.889	2.606
		1.781	1.894	2.664
		1.789	1.887	2.570
		1.796	1.908	2.642
		1.783	1.899	2.647
		1.707	1.891	2.595
		1.761	1.895	2.670
Média		1.764	1.888,9	2.591
Máximo - Mínimo (μs)	826,9			
Desvio Padrão		27,557	11,845	70,142
IC (95%)		17,080	7,341	43,474

Tabela 6.3: Resultados do experimento para medir o tempo de ciclo de uma tarefa que não realiza comunicação

		Tempo de Ciclo (μs)		
		Mínimo	Médio	Máximo
Num de Amostras	3.500	247	272	500
		251	276	442
		252	277	442
		244	268	494
		247	269	389
		241	266	498
		238	267	403
		236	268	425
		245	268	460
		244	273	455
Média		244,5	270,5	450,8
Máximo - Mínimo (μs)	206,3			
Desvio Padrão		5,147	3,807	38,789
IC (95%)		3,190	2,360	24,041

6.3 Investigação de Problemas

Tendo conhecimento do desempenho do CLP em circunstâncias usuais, podemos prosseguir conforme o método proposto na Figura 5.6. Os experimentos conduzidos serão compostos por 3.500 amostras, conforme justificado na Seção 6.2, e seguirão o mesmo modelo do teste proposto na Seção 5.1.2, mas desta vez isolando os componentes.

O primeiro passo é avaliar o desempenho sem comunicação, segundo a avaliação (I) do método. Isto significa que não existe nenhum dos tempos associados aos passos do fluxo de envio e recebimento de pacotes, mostrado na Figura 3.4, e nem os tempos associados a outras funcionalidades, como diagnósticos, por exemplo. Os resultados do experimento são apresentados na Tabela 6.3. Estes números são considerados como o custo fixo de uma tarefa, livre de qualquer *overhead* causado por comunicação.

Percebe-se que, em média, aproximadamente 15% do tempo de ciclo da tarefa é gasto com outras funcionalidades não relacionadas à comunicação. Além disso, é bastante notável como a comunicação aumenta muito a diferença entre os tempos máximo e mínimo (820 μs contra 206 μs). Este último é o custo fixo de uma tarefa.

Prosseguindo para a avaliação (II) do método proposto, deseja-se verificar se o aumento da diferença entre os tempos máximo e mínimo está relacionado à comunicação ou a outros fatores, como escalonamento de tarefas. Para poder determinar qual a causa, o experimento foi repetido com uma tarefa sem comunicação acrescida de um código com

Tabela 6.4: Resultados do experimento para medir o tempo de ciclo de uma tarefa que não realiza comunicação e com uma carga computacional constante

		Tempo de Ciclo (μs)		
		Mínimo	Médio	Máximo
Num de Amostras	3.500	1.701	1.738	1.935
		1.679	1.741	1.889
		1.706	1.758	1.996
		1.679	1.723	1.897
		1.703	1.731	1.918
		1.703	1.730	1.913
		1.705	1.730	1.918
		1.704	1.736	1.938
		1.701	1.751	1.982
		1.702	1.731	2.001
Média		1.698,3	1.736,9	1.938,7
Máximo - Mínimo (μs)	240,4			
Desvio Padrão		10,296	10,650	40,524
IC (95%)		6,381	6,601	25,116

carga computacional constante. Os resultados obtidos são apresentados na Tabela 6.4.

A diferença entre a média dos tempos máximo e mínimo para o caso sem comunicação com uma carga computacional constante foi de aproximadamente 240 μs, enquanto que, no caso geral, com comunicação, foi de aproximadamente 820 μs. Ainda, no caso sem comunicação, foi de aproximadamente 206 μs. A partir destes dados, percebe-se que a comunicação é o principal fator que introduz esta variabilidade. Logo, é necessário prosseguir para a avaliação (III), com o objetivo de detectar possíveis problemas na comunicação.

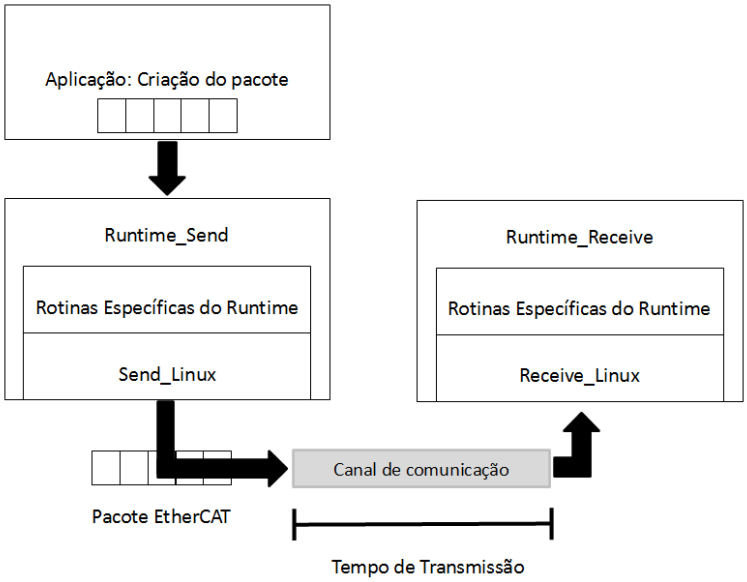
A comunicação, de modo geral, pode ser dividida em subpartes menos complexas, conforme Figura 3.4. Assim, foram realizados experimentos para determinar a contribuição ao total do tempo de ciclo das seguintes partes:

- Processamento dos telegramas EtherCAT;
- Funções de *send* e *receive* implementadas pelo *runtime*, que, entre outras rotinas, utilizam as rotinas de *send* e *receive* do Linux;
- Funções de *send* e *receive* do Linux propriamente dito, descartando a implementação específica das funções do *runtime*.

Para determinar o tempo gasto com o processamento dos telegramas EtherCAT, foi realizado um experimento com comunicação, mas sem o processamento dos pacotes. É importante notar como ainda existe neste experimento todo o tráfego de comunicação

explicitado na avaliação (III) da Seção 5.2, e somente o processamento dos telegramas EtherCAT propriamente ditos é que não foi executado. O fluxo do envio e recebimento de pacotes deste experimento pode ser visualizado na Figura 6.2. Os dados extraídos da execução estão na Tabela 6.5.

Figura 6.2: Fluxo de envio e recebimento de pacotes sem o processamento de telegramas EtherCAT



É notável como uma grande parte do tempo de ciclo é gasto com o processamento dos datagramas EtherCAT comparando os resultados das Tabelas 6.1 e 6.5. Em média, aproximadamente 744 μs do tempo de ciclo são gastos neste processo. Além disso, considerando a diferença entre a média do tempo de ciclo máximo e o mínimo, é possível perceber que o processamento também introduz picos de pouco mais de 500 μs.

Prosseguindo a análise, realizou-se um experimento onde foi desabilitada todas as comunicações nativas do CLP, similarmente ao experimento sem comunicação. Contudo, na aplicação de usuário, foram inseridas chamadas das funções de *send* e *receive* implementadas pelo *runtime*. O objetivo deste experimento é justamente avaliar qual a parcela de tempo associada ao envio e recebimento propriamente ditos, não levando em conta outros custos, como a criação do pacote ou o seu processamento. O pacote criado na aplicação teve seu tamanho arbitrariamente escolhido como 64 *bytes*, pois seu pequeno tamanho minimiza o tempo de transmissão. Como a conexão do equipamento apresenta uma taxa de transmissão de 100 Mbps, o tempo de transmissão destes pacotes será considerado desprezível neste experimento. Os dados obtidos são apresentados na Tabela 6.6.

Percebe-se, comparando as Tabelas 6.3 e 6.6, que o envio e recebimento de um

Tabela 6.5: Resultados do experimento para medir o tempo de ciclo de uma tarefa que realiza comunicação mas não processa os telegramas EtherCAT

		Tempo de Ciclo (μs)		
		Mínimo	Médio	Máximo
Num de Amostras	3.500	1.090	1.182	1.387
		1.064	1.134	1.340
		1.054	1.100	1.278
		1.054	1.132	1.472
		1.071	1.120	1.332
		1.054	1.106	1.300
		1.059	1.140	1.401
		1.075	1.206	1.447
		1.057	1.127	1.331
		1.056	1.144	1.422
Média		1.063,4	1.139,1	1371,0
Máximo - Mínimo (μs)	307,6			
Desvio Padrão		11,927	32,579	64,554
IC (95%)		7,392	20,192	40,010

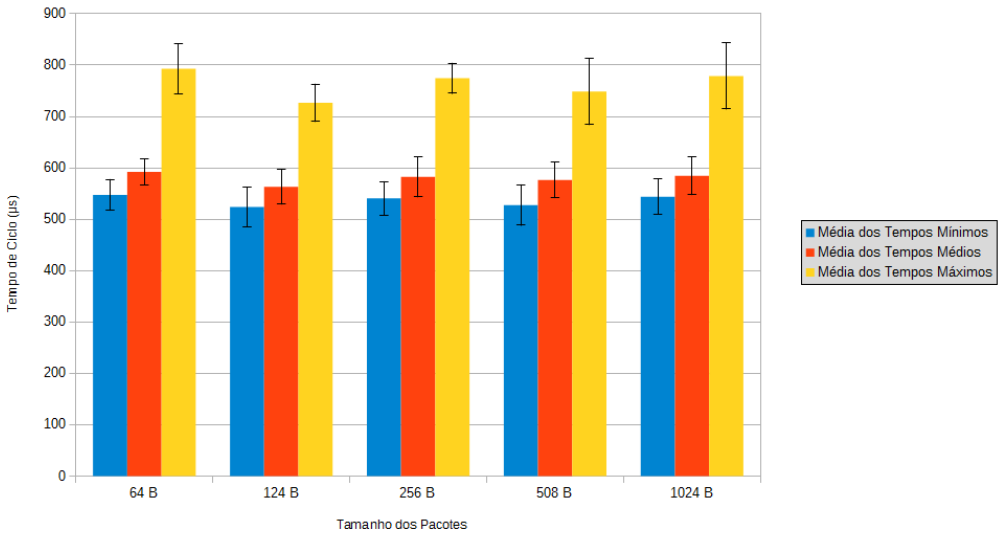
Tabela 6.6: Resultados do experimento para medir o tempo de ciclo de uma tarefa que apenas envia pacotes de 64 *bytes*

		Tempo de Ciclo (μs)		
		Mínimo	Médio	Máximo
Num de Amostras	3.500	558	588	823
		523	593	757
		562	593	761
		563	599	763
		498	570	770
		570	599	756
		573	627	838
		572	599	876
		497	535	737
		560	618	850
Média		547,6	592,1	793,1
Máximo - Mínimo (μs)	245,5			
Desvio Padrão		29,952	25,383	48,667
IC (95%)		18,564	15,732	30,163

pacote contribuem com aproximadamente 320 μs em média, mas contribuem pouco com a diferença entre as médias do tempo de ciclo máximo e mínimo, cerca de 40 μs. É importante notar que este experimento considera apenas um pacote. Para cada pacote a mais enviado e recebido durante um ciclo, este tempo seria somado novamente. Isso significa que, embora este tempo possa ser bastante menor do que o tempo gasto no processamento de datagramas, dependendo do sistema em execução, este custo pode ser muito considerável.

Ainda, o mesmo experimento foi repetido para outros tamanhos de pacote a fim de verificar o tempo gasto para pacotes maiores. Os tamanhos escolhidos foram 124, 256, 508 e 1024 *bytes*. Para facilitar a leitura, é apresentado apenas um comparativo entre as médias dos tempos de ciclo mínimo, médio e máximo para estes outros casos, conforme Figura 6.3. Notou-se que não existe um grande impacto nos tempos de ciclo com o aumento do tamanho do pacote. Portanto, no restante dos experimentos, não serão considerados cenários com outros tamanhos de pacote além de 64 *bytes*.

Figura 6.3: Comparativo dos tempos de ciclo para diferentes tamanhos de pacotes



Por fim, o último experimento a ser realizado tem o objetivo de isolar os tempos gastos pelo *overhead* de gerenciamento de pacotes do próprio Linux. Sabendo-se destes tempos, também será possível determinar o tempo gasto pelas funções de envio e recebimento oferecidas pelo *runtime*, cuja implementação é proprietária e não se tem acesso.

Para atingir este objetivo, foi implementado um programa em C que envia e recebe pacotes através de *sockets*. A diferença é que o programa executa no CLP diretamente, como um programa de usuário, no Linux, descartando possíveis influências da implementação do *runtime*. Foram amostrados os tempos gastos durante o processo de envio e

Tabela 6.7: Resultados do experimento para medir o tempo de envio e recebimento de pacotes de 64 *bytes* por *sockets* Linux

		Tempo de Ciclo (μs)		
		Mínimo	Médio	Máximo
Num de Amostras	3.500	183	223	327
		183	222	361
		180	222	331
		183	223	332
		183	223	357
		183	223	363
		183	222	353
		182	222	331
		183	222	346
		183	223	351
Média		182,6	222,5	345,2
Máximo - Mínimo (μs)	162,6			
Desvio Padrão		0,966	0,527	13,782
IC (95%)		0,598	0,326	8,542

recebimento de um pacote de 64 *bytes*. Os tempos amostrados estão na Tabela 6.7.

Podemos perceber que a maior parte do tempo gasto no envio e recebimento de pacotes é, de fato, em função do custo do gerenciamento do Linux. Assim, utilizando os dados deste experimento em conjunto com os experimentos sem comunicação (Tabela 6.3) e com envio/recebimento de pacotes de 64 *bytes* (Tabela 6.6), podemos calcular o tempo gasto com rotinas específicas do *runtime* nas suas funções de *send* e *receive*. Baseando-se na Figura 3.4, temos:

$$T_{Tarefa64B} = T_{SemComunicacao} + T_{Linux} + T_{RotinasRuntime}$$

Onde

$$T_{Tarefa64B}$$

representa o tempo gasto pela tarefa. Portanto:

$$T_{RotinasRuntime} = T_{Tarefa64B} - T_{SemComunicacao} - T_{Linux}$$

Baseado nos resultados dos experimentos, obteve-se os tempos apresentados na Tabela 6.8. De fato, o tempo mínimo calculado analiticamente não está condizente com a realidade, pois não faz sentido ter um tempo negativo. Este resultado deve-se a imprecisões do experimento ao se trabalhar com tempos médios. Assim mesmo, embora estes

Tabela 6.8: Tempo gasto com rotinas do *runtime* obtido analiticamente

		Tempo de Ciclo (μs)		
		Mínimo	Médio	Máximo
Média		-2,9	99,1	120,5
Máximo - Mínimo (μs)	123,4			
Desvio Padrão		71,167	25,592	32,843
IC (95%)		44,109	15,862	20,356

números não sejam absolutos, ainda é possível ter uma boa estimativa da porcentagem do tempo total gasto com a comunicação que é associado às rotinas específicas do *runtime*.

Deste modo, após os experimentos realizados, podemos discriminar a contribuição de tempo de cada componente.

Figura 6.4: Contribuição de cada componente considerando a comunicação como um grande bloco (em μs)

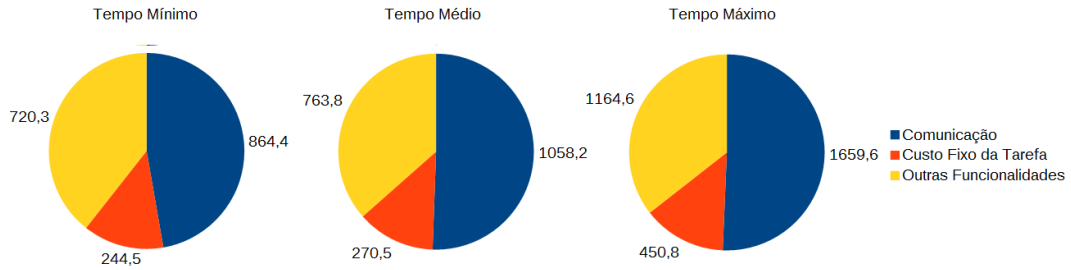


Figura 6.5: Contribuição de cada componente considerando o envio/recebimento de pacotes como um grande bloco (em μs)

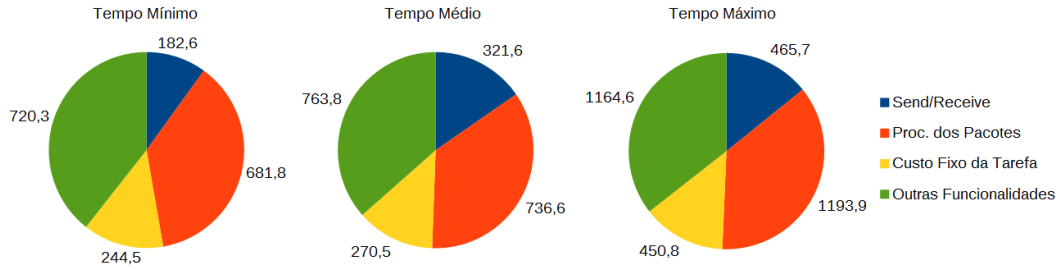
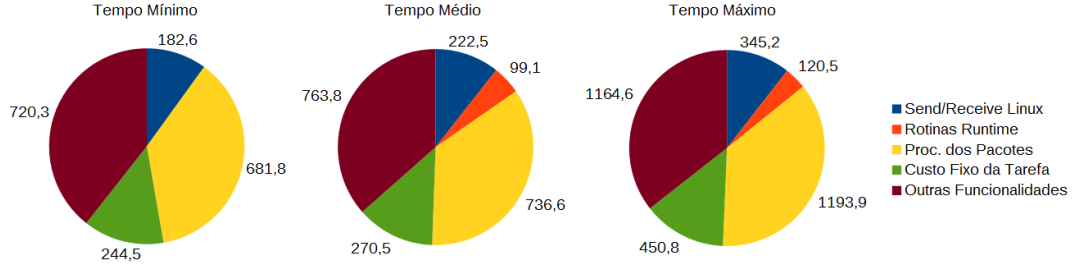


Figura 6.6: Contribuição de cada componente subdividindo o envio/recebimento de pacotes (em μs)



7 CONCLUSÃO

A análise da implementação de um CLP pode ser uma tarefa bastante complexa e maçante, possivelmente não se tendo acesso a partes de sua implementação. Por vezes, tem-se como objetivo alcançar um certo nível de desempenho para viabilizar o uso de determinados equipamentos em alguma aplicação específica. Neste trabalho, foi proposto um método que tem como objetivo facilitar o processo de avaliação de desempenho de um CLP. Embora o trabalho tenha sido desenvolvido como sendo um estudo de caso dos CLPs da série Nexto, o método proposto pode ser facilmente adaptado para equipamentos de outros fabricantes.

A sistemática proposta foi utilizada como alicerce para traçar um panorama, através de sucessivos experimentos, da situação atual do desempenho da série Nexto. Os resultados da aplicação do método foram satisfatórios. A avaliação de desempenho dos CLPs realizada neste trabalho demonstrou que grande parte do tempo de processamento é gasto com comunicação, representando aproximadamente metade do tempo total.

Mais precisamente, através dos experimentos, foi possível isolar diversos componentes que compõem a comunicação. A partir da análise dos dados, foi determinado o gasto de tempo com cada um destes componentes. Isso viabiliza novos estudos com o objetivo de aprimorar os pontos fracos da implementação atual.

Foram identificados dois pontos principais que são passíveis de melhorias. O primeiro deles é o envio e recebimento de pacotes. Detectou-se que a maior parte do tempo gasto nestas funções é devido ao *overhead* da pilha de rede do próprio Linux. O outro ponto é o processamento dos pacotes, cuja contribuição ao tempo gasto total é muito significativa. Além disso, constatou-se que este ponto é o principal causador de picos de tempo quando comparado aos outros componentes da comunicação.

Como trabalhos futuros, pode-se utilizar os resultados obtidos para efetivamente implementar otimizações e melhorias no sistema atual. Sugere-se a utilização de bibliotecas que implementem soluções de kernel *bypassing*. Deste modo, será possível diminuir muito o *overhead* causado pelo Linux. Além disso, uma outra frente de trabalho é buscar otimizações na função de processamento dos pacotes. Embora a implementação desta função não seja trivial, como o tempo de processamento gasto por ela é muito significativo, certamente ainda existe espaço para otimizações em seu algoritmo. Não somente é possível melhorar o seu desempenho diminuindo o tempo gasto nesta função, mas também é possível aumentar o determinismo do sistema através de um algoritmo

que apresente tempos mais constantes e estáveis. Ao mesmo tempo pode-se realizar um estudo sobre a comunicação com um enfoque em balanceamento de carga analisando-se a origem dos pacotes enviados, buscando entender a qual funcionalidade implementada através de comunicação aquele pacote está associado. Isso poderia ser detectado através de algum mecanismo de marcação de pacotes, realizando a avaliação (III) do método proposto na Seção 5.2, visando evitar *bursts* de telegramas, por exemplo.

REFERÊNCIAS

3S-Smart Software Solutions GmbH. *CODESYS Control V3 Manual*. [S.l.], 2012.

AGILENT Technologies 3000 Series Oscilloscopes Datasheet. 2017. <http://literature.cdn.keysight.com/litweb/pdf/5989-2235EN.pdf?id=638687>. Acessado em 19/11/2017.

ALTUS. **Manual de Utilização MasterTool IEC XE MT8500**. [S.l.], 2016.

BECKHOFF CX9020 Hardware documentation. 2017. <https://infosys.beckhoff.com/content/1033/cx9020_hw/9007202790037387.html>. Acessado em 29/06/2017.

BECKHOFF Fieldbus Components: Product Overview. 2017. <https://infosys.beckhoff.com/english.php?content=../content/1033/tcsample_crestron/html/tckb_crestron_performance.htm>. Acessado em 27/06/2017.

BOLTON, W. **Programmable logic controllers**. [S.l.]: Newnes, 2015.

BRYAN, L. A.; BRYAN, E. A. **Programmable controllers: theory and implementation**. [S.l.]: Industrial Text Company, 1997.

BURNS, A.; WELLINGS, A. J. **Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX**. [S.l.]: Pearson Education, 2001.

FIELD-RICHARDS, H. S. Real-time languages—design and development: Sj youngjohn wiley (1982)£ 29.50, pp 352. **Microprocessors and Microsystems**, Elsevier, v. 7, n. 4, p. 184, 1983.

GUPTA, A.; ARORA, S. **Industrial automation and robotics**. [S.l.]: Laxmi Publications, 2009.

International Electrotechnical Commission and others. Iec 61131-3. **Programmable Controllers-Part**, v. 3, 1993.

JAIN, R. **The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling**. [S.l.]: John Wiley & Sons, 1990.

JARP, S. **A methodology for using the Itanium 2 performance counters for bottleneck analysis**. [S.l.], 2002.

JOHN, K.-H.; TIEGELKAMP, M. **IEC 61131-3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids**. [S.l.]: Springer Science Business Media, 2010.

KOPETZ, H. **Real-time systems: design principles for distributed embedded applications**. [S.l.]: Springer Science & Business Media, 2011.

MARIESKA, M. D.; KISTIJANTORO, A. I.; SUBAIR, M. Analysis and benchmarking performance of real time patch linux and xenomai in serving a real time application. In: IEEE. **Electrical Engineering and Informatics (ICEEI), 2011 International Conference on**. [S.l.], 2011. p. 1–6.

MOTION Controllers. 2017. <http://dl.mitsubishielectric.com/dl/fa/document/catalog/ssc/l03014/qmotion.pdf>. Acessado em 09/06/2017.

OMRON Automation Pvt Ltd. **EtherCAT Communication Manual**. [S.l.]: OMRON, 2016.

SÉRIE Nexto - Altus. 2017. <www.altus.com.br/ftp/Public/Portugues/Produtos/Nexto/00DocSerie/ManuaiseApostilas/MU214000.pdf>. Acessado em 29/06/2017.

ANEXO A — TRABALHO DE GRADUAÇÃO - I

- Título: Avaliação de Desempenho Temporal de Controlador Lógico Programável - Estudo de Caso
- Data: Junho de 2016

Avaliação de Desempenho Temporal de Controlador Lógico Programável - Estudo de Caso

Eduardo Kochenborger Duarte[1],
João Cesar Netto[1],
João Ricardo Wagner de Moraes[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Av. Bento Gonçalves, 9500 - Porto Alegre, RS - Brasil

{eduardo.duarte,netto,joao.moraes}@inf.ufrgs.br

Abstract. *In industrial applications, programmable controllers, devices designed for control tasks and to withstand hostile industrial environments, are used. In some of those applications, the response time and the determinism are essential, making it necessary for the devices used to have a good performance. The Nexto series controllers are suitable for various processes, including the ones with a large number of points and redundancy. However, the series' current performance is not enough for motion control systems, or other systems that need a very fast response. This work has the objective of studying the implementation of Nexto series controllers and determining how to optimize its performance.*

Resumo. *Em aplicações industriais, são utilizados controladores programáveis, dispositivos desenvolvidos para tarefas de controle e para suportar o ambiente industrial hostil. Em algumas dessas aplicações, o tempo de resposta e o determinismo são essenciais, sendo necessário que os dispositivos utilizados possuam desempenho satisfatório. Os controladores da série Nexto são adequados para diversos processos, inclusive os com grande número de pontos e redundância. Contudo, o desempenho atual da série não é o suficiente para sistemas de motion control, ou outros sistemas que exijam resposta muito rápida. Este trabalho tem o objetivo de estudar a implementação dos controladores da série Nexto e determinar como otimizar seu desempenho.*

1. Introdução

A automação de tarefas outrora executadas manualmente é recorrente na indústria e causa grande aumento de produtividade. Esse processo tem origem em meados da década de 40, cunhado sob as necessidades da indústria automobilística americana. Por um longo tempo, a automação existiu em escalas bastante diminutas, utilizando-se principalmente de dispositivos mecânicos. Com o surgimento e disseminação da utilização de computadores, a flexibilidade adquirida possibilitou a automatização de praticamente qualquer tarefa. [Gupta e Arora 2009].

Uma das formas de se automatizar tarefas é através da utilização de relés, que são interruptores eletromecânicos, ligados entre si de forma a se obter o comportamento desejado. Contudo, um grande sistema de controle desenvolvido com relés apresentaria uma grande complexidade. Essa complexidade dificulta muito a depuração de problemas, a implementação de mudanças e a reutilização de partes do sistema. Foi nesse contexto

que surgiram os controladores lógico programáveis (CLPs), trazendo grandes melhorias para os problemas até então apresentados.

Um CLP é, basicamente, um dispositivo que captura informações do ambiente e produz uma reação, baseada na forma como foi programado. As informações do ambiente vêm de sensores, como sensores de temperatura ou pressão, por exemplo. Usando esses dados, o CLP produzirá uma saída de acordo com a lógica programada pelo usuário. Por exemplo, pode-se ativar um atuador que controle um ventilador uma vez que a temperatura suba além de um limiar aceitável.

Existem diversos tipos de aplicações para as quais os CLPs são adequados a desempenhar. Naturalmente, é necessário fazer um levantamento de requisitos específicos à aplicação antes de se avaliar qual equipamento é o mais apropriado utilizando-se alguma métrica. O CLP de mercado da série Nexto, disponível no laboratório do Instituto de Informática, em função de suas características, não possui o desempenho necessário para aplicações com requisitos de desempenho muito altos, considerando o tempo de ciclo como métrica.

Cada tipo de CLP é desenvolvido com o objetivo de se adequar a uma determinada gama de aplicações. A série Nexto é orientada para alta disponibilidade com soluções de redundância e suporte a grandes quantidades de pontos de entrada e saída. Assim, é possível dizer que a série Nexto apresenta um nicho de aplicação amplo, não focando especificamente em um tipo de aplicação. Outros equipamentos podem ter mais recursos de hardware, ou serem desenvolvidos para aplicações mais específicas, podendo ser mais otimizados para essas aplicações do que CLPs mais gerais. Um exemplo disso é mostrado na tabela 1.

Tabela 1. Comparativo de CLPs. (Fontes: [Série Nexto 2017], [Beckhoff Product Overview 2017], [Beckhoff HW Documentation 2017], [Motion Controllers 2017])

CLP	CLP de Mercado	Tempo de Ciclo	Quantidade de Pontos
Uso Geral	Nexto	Ordem de 5 ms	Grande
Uso geral com processador mais robusto	CX	Ordem de 3 ms	Grande
Motion Control	Q-Series	< 1 ms	Muito Pequena

Tendo em vista as características da série, o que se deseja é poder utilizar os controladores Nexto em sistemas de controle com requisitos muito fortes de desempenho. O objetivo deste trabalho é realizar um estudo sobre possíveis otimizações e melhorias que possibilitem o uso dos CLPs nesses sistemas. Para isso, serão apresentados testes que verifiquem o estado da situação atual no que diz respeito a desempenho, bem como uma análise sobre as possíveis causas e soluções.

Para contextualizar o leitor, na seção 2 são apresentados conceitos básicos sobre controladores programáveis. Em seguida, na seção 3, são apresentadas características, métricas e requisitos para sistemas de tempo real. A seção 4 enfatiza os objetivos do trabalho e descreve alguns trabalhos desenvolvidos na área de avaliação de desempenho.

O método proposto para avaliar o desempenho atual dos controladores está descrito na seção 5. Por fim, na seção 6, são apresentadas as conclusões do trabalho, bem como um planejamento de datas para o seu desenvolvimento.

2. Controladores Programáveis

CLPs são, em essência, iguais aos computadores de uso pessoal: recebem uma entrada, que é processada, e produzem uma saída, de acordo com sua programação. A grande diferença se dá na tarefa atribuída ao dispositivo. Computadores de uso pessoal são otimizados para realização de cálculos e tarefas de visualização. CLPs são otimizados para tarefas de controle, geralmente de tempo real, e para suportar ambientes industriais hostis. Isso inclui variações de temperatura, umidade, vibração mecânica, e ruído. [Bolton 2015]

2.1. Entradas e Saídas

A ideia principal de operação de um CLP é bastante simples: Existe um programa, ou mais especificamente, um sistema de controle que foi desenvolvido para reagir/tomar decisões baseado em informações coletadas do processo em questão. Essas informações chegam ao controlador através de suas entradas, que estão conectadas a sensores, equipamentos que convertem grandezas elétricas e mecânicas em sinais padrões para serem tratados pelos controladores. Similarmente, as decisões tomadas pelo sistema de controle chegam ao mundo exterior através das interfaces de saída. As interfaces de saída são ligadas a atuadores, componente que possui uma alimentação e recebe um sinal de controle, convertendo a energia em movimento.

É importante notar como a robustez do controlador é influenciada por todas as partes. Por exemplo, é extremamente indesejável que um surto de tensão em uma das entradas danifique componentes centrais, ou qualquer outro componente. De fato, é necessário que os controladores sejam meticulosamente planejados para evitar resultados desfavoráveis.

Tanto as entradas quanto as saídas podem ser de dois tipos: digitais ou analógicas. Sinais digitais possuem apenas dois níveis lógicos: nível alto e nível baixo. Sinais analógicos possuem uma quantidade discreta de estados, dependente da resolução do respectivo *hardware*, podendo ter diversos valores intermediários entre o nível lógico alto e o baixo [Bryan e Bryan 1997].

A decisão de quando deve ser utilizado cada tipo depende da aplicação em questão. Entradas e saídas digitais são adequadas para representar sinais discretos. Um interruptor, por exemplo, que pode estar ligado ou desligado, mas nunca em um estado intermediário. Entradas e saídas analógicas são adequadas para representar sinais contínuos. Uma tensão, por exemplo, cujo valor será discretizado para ser interpretado por aplicações desenvolvidas para rodar no CLP.

Uma outra possível aplicação para entradas analógicas é um controle de temperatura. Temperatura é uma grandeza contínua, que possui tantos valores quanto se desejar (ou quanto a resolução do *hardware* permitir). Para que a informação de temperatura possa ser utilizada pelo CLP, é preciso transformar a temperatura em um sinal que possa ser interpretado. Para isso, é usado um sensor de temperatura. Por exemplo, o sensor conectado a uma entrada analógica pode gerar um sinal de tensão, onde o valor corresponde

a uma determinada temperatura. O CLP poderá, então, realizar leituras a essa entrada, convertendo o valor lido para uma escala que seja conveniente ao programa de aplicação.

A leitura de entradas, execução do programa e escrita de saídas são feitas de forma cíclica no tempo. O ciclo do CLP é apresentado a seguir, na figura 1.

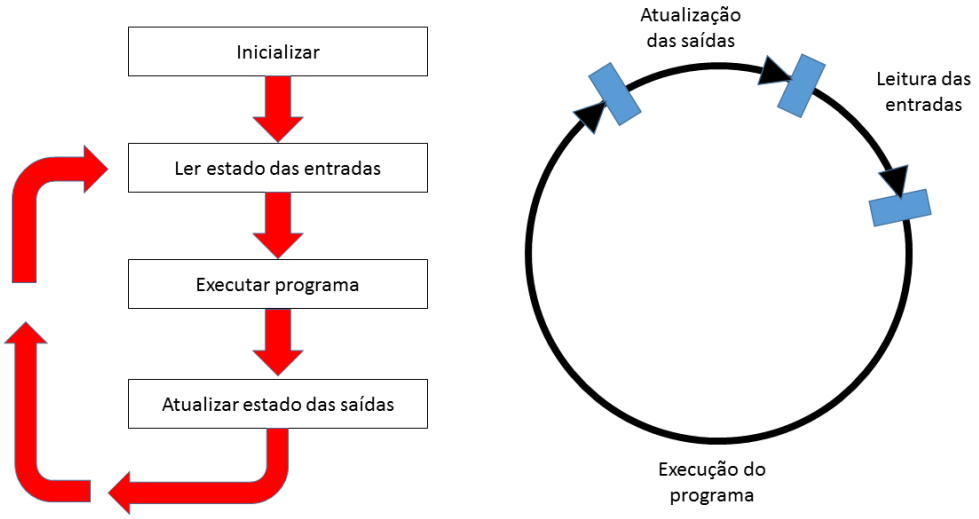


Figura 1. Ciclo do CLP (Fonte: Adaptado de [Bryan e Bryan 1997])

2.2. Tipos de CLPs

Usualmente, CLPs são classificados quanto ao seu design mecânico: os compactos e os modulares. Ambos os tipos apresentam as mesmas funcionalidades básicas.

Os chamados "compactos"contêm todos os componentes que formam um CLP (alimentação, processador, memória e unidades de entrada/saída) integrados em uma única entidade. Esse tipo geralmente é utilizado para sistemas de menor porte e que necessitem de menos recursos.

CLPs do tipo modular possuem os mesmos componentes que os compactos, mas cada um deles é uma unidade separada, chamada de módulo. Os módulos são conectados a um bastidor, permitindo que se escolha exatamente quais módulos são adequados para o sistema de controle a ser desenvolvido.

Sistemas que necessitam de uma maior quantidade de recursos geralmente acabam por usar CLPs do tipo modular. Em um CLP modular, é muito mais fácil expandir a quantidade de entradas e saídas, ou a quantidade de memória, simplesmente adicionando-se novos módulos ao bastidor.

Na figura 2, um exemplo de CLP modular, disponível no laboratório, utilizando os seguintes componentes: bastidor, fonte externa, CPU, módulo de saída digital e dois módulos de entrada digital.

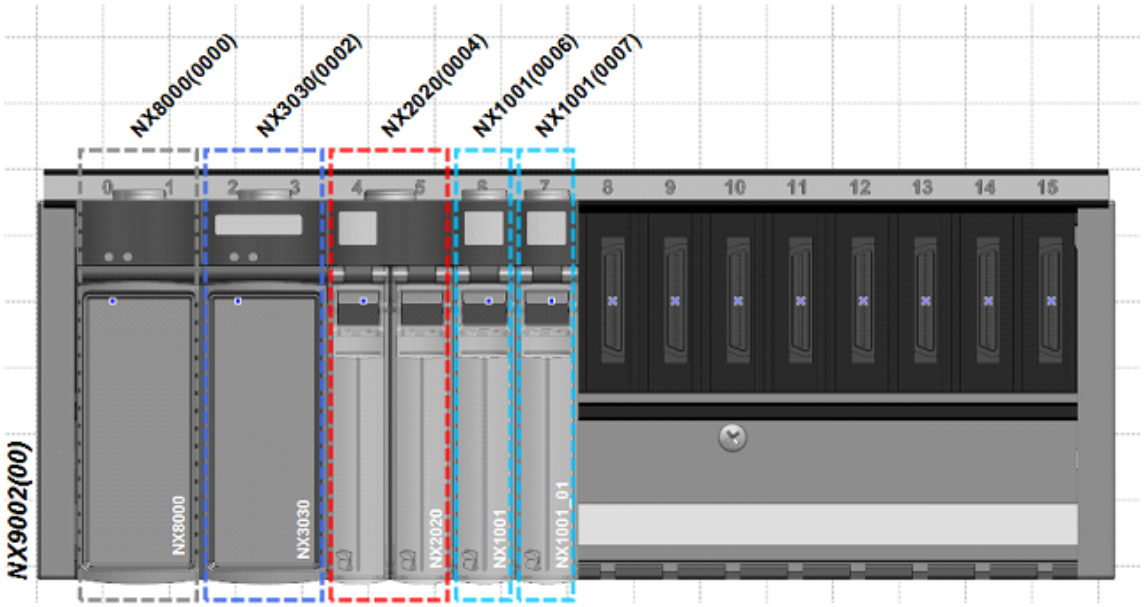


Figura 2. Exemplo de disposição física dos módulos em um bastidor

2.3. Norma IEC 61131-3

A norma IEC 61131-3 é vista como um conjunto de orientações para programação de CLP. Fabricantes de CLPs precisam provar em quais partes o padrão foi cumprido ou não. Atualmente, essa norma é aceita pela maioria desses fabricantes. Várias vantagens são obtidas com o seu uso, e.g., reusabilidade de código, portabilidade, diminuição na curva de aprendizagem dos usuários [John e Tiegelkamp 2010].

Dentre as definições contidas na norma, está a das linguagens de programação adequadas para a programação de CLPs. As seguintes linguagens estão definidas:

- *Ladder diagrams* (LAD): Descrição visual de circuito de automação por relé. Por isso, é de fácil compreensão por equipes de manutenção que estejam acostumadas a lidar com diagramas de relé. Além disso, pelo mesmo motivo, é de fácil utilização para a elaboração de lógicas de intertravamento.
- *Instruction list* (IL): Linguagem textual de baixo nível, onde cada linha representa uma operação simples. Pode ser considerado como uma versão textual da linguagem Ladder [Bolton 2015]. Geralmente utilizado para programas de pequenos, bastante diretos e simples.
- *Sequential function charts* (SFC): Linguagem gráfica, de fácil visualização. É conveniente para controles de atividades sequenciais, onde seja necessário ter a noção de máquina de estados.
- *Structured text* (ST): Esta linguagem possui grande semelhança com Pascal. Programas são escritos como uma série de sentenças separadas por um ponto e vírgula. Através de sentenças e subrotinas predefinidas, as variáveis do programa podem ser usadas ou terem seus valores trocados. Variáveis são, neste caso, valores definidos internamente, ou valores associados a entradas ou saídas.
- *Function block diagrams* (FBD): Linguagem gráfica onde o programa é descrito através de diversos blocos funcionais interligados. Um bloco funcional é um ob-

jeto que, quando executado, produz uma ou mais saídas. Também facilita a reutilização de código devido a sua modularidade.

Para fins de testes de desempenho, será utilizada a linguagem ST. A escolha é justificada pela facilidade de programação da linguagem, além de que grande parte das bibliotecas internas do CLP foram implementadas em ST.

2.4. Execução: Tipos de Tarefas

Uma tarefa é um elemento de controle de execução capaz de invocar a execução de um conjunto de unidades de organização de programas (POUs), que incluem programas e blocos funcionais. A execução pode ser de forma periódica ou na ocorrência de eventos [International Electrotechnical Commission and others 1993]. Dessa forma, existem três tipos de tarefas:

- Tarefa Cíclica (*Cyclic Task*): Executada em intervalos regulares de tempo. Este tipo de tarefa irá interromper outras tarefas de prioridade mais baixa uma vez que tenha se esgotado o intervalo de tempo.
- Tarefa por Evento (*Event Task*): Executada, apenas uma vez, na ocorrência da borda de subida da variável booleana especificada.
- Tarefa Contínua (*Continuous Task*): Tarefa com execução contínua. O período de execução é definido pelo tempo de processamento gasto pela tarefa em um determinado momento. Isso significa que o período de execução é variável, e será mais longo caso a tarefa necessite de mais tempo de processamento.

3. Sistemas de Tempo Real e Determinismo

Um sistema de tempo real é definido como uma atividade ou sistema de processamento que precisa responder a entradas geradas externamente dentro de um período finito e especificado [Field-Richards 1983] (conforme citado em [Burns e Wellings 2001]). A dimensão deste tempo depende do tipo de aplicação. Por exemplo, um sistema de controle de um míssil provavelmente terá um intervalo menor do que um sistema de montagem de automóveis.

No contexto de sistemas de tempo real, determinismo significa que o comportamento do sistema seja previsível em termos de valores e tempo [Kopetz 2011]. Em um sistema de frenagem, por exemplo, não é suficiente garantir que a ação de frenagem vai ser executada em algum momento posterior. Neste caso, seria necessário uma restrição mais forte, como definir que a ação de frenagem ocorrerá 2 ms após o seu acionamento.

Assim, relacionando esses conceitos com os tipos de tarefas anteriormente apresentados, é possível perceber que tarefas contínuas não são adequadas para aplicações que necessitem de determinismo. Uma vez que não existe um tempo definido entre cada execução, não é possível saber antecipadamente em que momentos a tarefa executará.

3.1. *Jitter de Atualização de I/O*

Jitter é definido como a diferença entre o menor e o maior valor de atraso. Neste caso, mais especificamente, esse atraso é relativo ao momento de atualização de I/O. O ciclo de atualização de I/O foi explicado na seção 2.1. Analisando-se a figura 1, podemos notar um inerente problema: a atualização das saídas ocorre após a execução do programa do

usuário. Ou seja, o determinismo da atualização das saídas ficará comprometido, já que dependerá do tempo de execução da lógica de usuário.

É possível visualizar esse problema com o auxílio da figura 3. Supondo uma tarefa cíclica, i.e., que possui um intervalo entre duas ativações consecutivas I_{tarefa} predefinido, e supondo que não haja *jitter* na execução da tarefa (sem qualquer perda de generalidade); idealmente, assim como o tempo entre duas execuções da tarefa, o tempo entre duas escritas consecutivas deveria ser igual ao intervalo de ativação da tarefa. Contudo, sendo a atualização das saídas realizada imediatamente após a execução da lógica de usuário, haverá um *jitter* de saídas J_o tal que:

$$0 \leq J_o < 2I_{tarefa}$$

No caso específico onde o final da atualização das saídas coincidir no mesmo ponto de

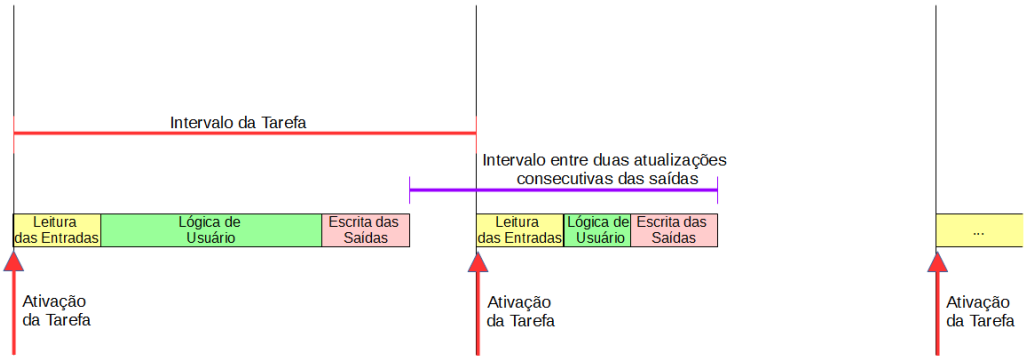


Figura 3. Execução hipotética de uma tarefa com atualização de entradas e saídas

tempo dentro do intervalo de ativação da tarefa, não haverá *jitter*. Contudo, salvo esse caso, em todos os outros existirá essa variação no tempo. Considerando o pior caso, onde em um ciclo tanto os tempos de atualização de I/O quanto o tempo de execução de lógica de usuário são mínimos no primeiro ciclo, e no ciclo seguinte esses tempos são máximos, teremos um *jitter* que não excede duas vezes o tempo do intervalo de ativação da tarefa. Além disso, mesmo esses tempos sendo mínimos, eles existirão (mesmo que arbitrariamente pequenos), e isso justifica o porquê o valor do *jitter* de atualização estar contido no intervalo $[0, 2I_{tarefa})$. Isso pode ser visualizado na figura 4.

Nos casos apresentados nas figuras 3 e 4, os comandos de leitura de entradas e escrita de saídas são enviados em momentos separados. É possível agrupá-los e transmití-los juntos, restando apenas decidir qual o melhor momento para isso: na ativação da tarefa, ou após a execução da lógica de usuário. Ambas as opções apresentam vantagens e desvantagens, conforme analisado a seguir:

- Envio dos comandos após a execução da lógica de usuário: Neste caso, o determinismo de atualização de entradas e saídas será afetado devido a variação no tempo de execução da lógica de usuário. Além disso, na primeira execução, as entradas não estarão com seus valores atualizados. Este método é mais adequado para quando o intervalo de ativação da tarefa é grande, pois suas saídas serão escritas logo após o término da execução do programa, independentemente do tempo de intervalo.

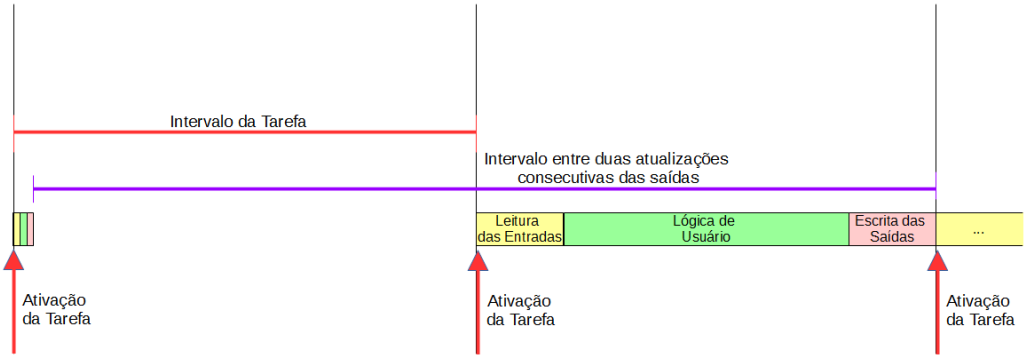


Figura 4. Exemplo de pior caso de *jitter* na atualização das saídas

- Envio dos comandos no momento de ativação da tarefa: A atualização das saídas ocorrerá na mesma cadência de ativação da tarefa, ou seja, manterá seu determinismo. Além disso, as entradas serão lidas antes do início da execução da lógica de usuário, o que significa que seus valores estarão atualizadas desde a primeira execução. A desvantagem aparece no momento em que o intervalo de ativação da tarefa é bastante grande. Ao final da execução do programa, embora os valores das saídas já tenham sido calculados, somente na próxima ativação da tarefa é que esses valores serão de fato escritos.

3.2. *Atualização de I/O*

A comunicação entre os módulos decorrente da atualização de entradas e saídas é feita através de um protocolo de comunicação. No equipamento disponível, o protocolo usado é o EtherCAT, um protocolo de comunicação industrial de tempo real e alta performance baseado em Ethernet.

O EtherCAT funciona em uma configuração mestre/escravo, onde o mestre envia comandos aos nodos escravos, que podem escrever e/ou ler os dados. Através desse mecanismo, a atualização de I/O acontece ciclicamente no tempo. O objetivo é maximizar a utilização do canal de comunicação e minimizar os tempos de resposta.

Os comandos são inseridos no segmento de dados de um telegrama EtherCAT. Um telegrama é composto por um ou mais endereços de escravos, dados e alguns bits de controle. Esses telegramas são inseridos como *payload* em um *frame* Ethernet. Cada um desses *frames* pode conter diversos telegramas EtherCAT.

Um *frame* enviado pelo mestre é repassado apenas para o primeiro nodo escravo, que o repassa ao próximo, e assim sucessivamente. A leitura ou inserção de dados se dá neste momento, quando o *frame* está sendo transmitido, ou seja, não ocorre uma pausa na transmissão: os dados são lidos/inseridos *on the fly*. Deste modo, uma vez que o *frame* chegue no último nodo escravo, este o enviará de volta ao mestre, utilizando-se da comunicação *full-duplex* do Ethernet. Assim, o tempo de processamento de cada escravo é praticamente desconsiderável, fazendo com que essa solução apresente o determinismo desejado.

3.3. Tempo de Ciclo

O tempo de ciclo é definido como o tempo entre a ativação da tarefa até o final da execução da mesma. Isso inclui o tempo gasto pela atualização de I/O e o tempo de execução da lógica do usuário.

Os CLPs da série Nexto apresentam como tempo de intervalo mínimo 5 ms. Tanto aplicações como o sistema de frenagem mencionado na seção 3, quanto aplicações de *motion control*, necessitam de tarefas que executem mais rapidamente. Atualmente, não é possível o uso desses CLPs nessas aplicações devido às características da série Nexto.

4. Objetivo e Trabalhos Relacionados

Sabendo-se que existem diversos componentes do sistema que podem otimizados em relação ao seu desempenho, o que se deseja é identificar qual deles é o gargalo. Naturalmente, melhorias podem ser feitas em qualquer um deles, mas deseja-se ter o maior impacto possível. Identificou-se os seguintes assuntos para serem estudados:

- *Driver* de comunicação EtherCAT: o *driver* responsável pela comunicação relacionada a atualização de I/O. Lógicas de intertravamento, implementações ineficientes, entre outros fatores podem degradar o desempenho, especialmente em casos onde ocorra muita comunicação.
- Interrupção da execução por tarefas mais prioritárias: a tarefa de usuário não possui prioridade máxima, ou seja, existem diversas tarefas que podem interrompê-la durante a execução, diminuindo o desempenho. Podem existir problemas de escalonamento.
- Interface de rede: possíveis problemas de gerenciamento dos *frames* EtherCAT que contêm os comandos de atualização de I/O. Possível compartilhamento de recursos com a interface de rede que realiza a comunicação com o *software* de programação.

Este trabalho consiste em uma avaliação de desempenho de uma implementação única, utilizada em equipamentos específicos. Por este motivo, não existem outros trabalhos especificamente relacionados com os mesmos dispositivos. Contudo, existem outros trabalhos de avaliação de desempenho que utilizam diferentes ambientes, ferramentas e equipamentos, ou mesmo trabalhos de cunho teórico.

Em [Jarp 2002], o autor propõe identificar gargalos em programas através do uso de *performance counters*. Com esse mecanismo de monitoração, são obtidas informações úteis sobre o programa em execução, como: quantidade de ciclos de processamento, número de instruções, ciclos de *stall* (causados por eventos como *cache miss*, predição de desvios incorreta, entre outros) e atividade dos diversos níveis de cache. Essas informações por si só não conduzem a um melhor desempenho de execução, mas são muito úteis para ajudar na identificação de gargalos.

Em [Jain 1990], são descritos critérios para a seleção de parâmetros para serem testados a fim de se identificar o gargalo do sistema. Parâmetros que não dizem respeito ao componente cujo desempenho se deseja melhorar podem ser omitidos, simplificando a análise. Também é descrito o uso de monitores, que são ferramentas utilizadas para observar atividades de uma determinada parte do sistema. Um monitor observa o desempenho do sistema, coleta dados, forma estatísticas, analisa e exibe os dados.

5. Método Proposto

Sintetizando o que foi dito nas seções anteriores, temos como principais requisitos de desempenho:

- Pouco *jitter* de atualização de I/O;
- Tempo de ciclo muito rápido;

Para averiguar se os CLPs estão de acordo com os requisitos, será utilizada uma abordagem que consiste em experimentos na forma de *workloads* sintéticos. Para a programação dos experimentos, será utilizada ST, uma das linguagens IEC, pois esta é utilizada em aplicações reais. Assim, serão realizados dois testes:

- Teste para verificar a existência de *jitter* na atualização de I/O;
- Teste para verificar se o tempo mínimo de ciclo é adequado;

No primeiro teste, a ideia proposta consiste em criar uma tarefa cíclica com um intervalo de ativação bastante grande para a tarefa. A tarefa não executará nenhum código, exceto em um ciclo, onde ela executará um código bastante custoso. Este código ocupará a maior parte do tempo de intervalo da tarefa. Dessa forma, caso o momento de envio dos *frames* dependa do tempo de execução da tarefa, será possível perceber a variação na cadência de atualização de I/O. Para o teste, será utilizada uma saída digital, que terá seu valor alterado e atualizado a cada ciclo da tarefa.

Para a avaliação da existência de *jitter* na saída, será usado um osciloscópio para capturar a forma de onda do sinal correspondente à saída digital em teste. Caso o *jitter* de fato exista, haverá um deslocamento no eixo do tempo da mesma ordem de grandeza do tempo de ativação da tarefa.

Para a monitoração do sistema, será utilizado um *workload* sintético. O seu código é composto por: um laço que é repetido um grande número de vezes, levando um tempo pouco menor do que o intervalo da tarefa. Esse laço é executado uma vez a cada 6 ciclos da tarefa. Nos outros ciclos, não existe nenhuma carga computacional intensiva. O programa foi escrito em ST, e seu código fonte é apresentado na figura 5.

O intervalo de ativação da tarefa escolhido para o teste foi o máximo permitido pela ferramenta de programação, 100 ms. A justificativa é que deseja-se minimizar a possível influência de fatores externos na tarefa, como interrupção por tarefas mais prioritárias, por exemplo. Nos ciclos onde a tarefa não executa o laço, a tarefa permanecerá a maior parte do tempo entre ativações sem realizar qualquer processamento, diminuindo possíveis interrupções.

No segundo teste, para avaliar o valor do tempo de ciclo mínimo, será criada uma tarefa com tempo de ativação em 5 ms, e amostrado diversas vezes o seu tempo de ciclo. Esse tempo, mesmo para uma tarefa que não execute nenhum código de usuário, não será nulo, uma vez que ainda se faz necessário a atualização de I/O, além de outras funções como diagnósticos, por exemplo. Dessa forma, se o tempo de ciclo da tarefa for próximo do tempo limite inferior definido, estará justificado o limite, pois a tarefa não pode ter um tempo de ciclo maior do que seu intervalo de ativação.

5.1. Resultados Preliminares

O teste de *jitter* produziu o resultado mostrado na figura 6.

```
PROGRAM UserPrg
VAR
        out : BOOL;
        load : INT := 0;
        count : DINT := 0;
        temp : INT := 0;
END_VAR

IF (load = 5) THEN
        FOR count := 0 TO 3160000 DO
                temp := temp+1;
        END_FOR
        load := 0;
END_IF
out := NOT out;
load := load + 1;
```

Figura 5. Programa implementado para avaliação do *jitter* de atualização de I/O

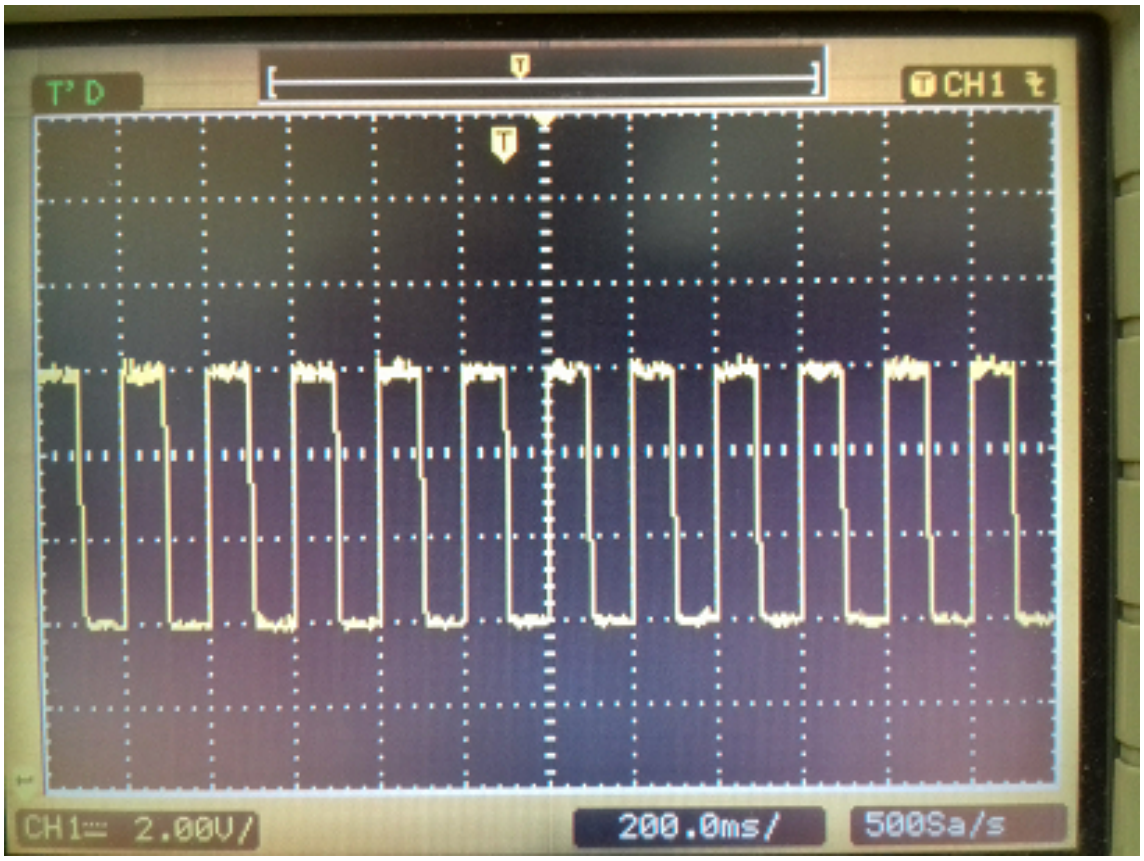


Figura 6. Forma de onda da saída digital

A saída digital teve seu valor alterado a cada 100 ms, o que é condizente com o intervalo definido. Além disso, mesmo nos ciclos onde o laço foi executado, a mudança na

saída permaneceu corretamente cadenciada, não havendo indícios de variação no tempo de atualização. Portanto, pode-se concluir que o controlador cumpre com o requisito de *jitter*.

O *workload* foi executado e foram coletadas 150000 amostras. O resultado produzido é mostrado na tabela 2

Tabela 2. Resultado do teste de tempo de ciclo.

Tempo de Ciclo Máximo (μs)	Tempo de Ciclo Mínimo (μs)	Tempo de Ciclo Médio (μs)
2686	1759	1918

Considerando que o tempo de ciclo máximo chegou a aproximadamente 2.6 ms, é justificável que o intervalo de ativação mínimo da tarefa seja 5 ms. É preciso manter uma certa fatia de tempo para a execução do programa do usuário. Com esta configuração, considerando o pior caso, tem-se aproximadamente metade do tempo de intervalo para execução de lógica. Seria vantajoso se a parcela de tempo gasta com tarefas diversas que não o programa de usuário fosse bem menos considerável.

6. Conclusões e Cronograma

Ao longo deste artigo, foi feita uma revisão sobre assuntos relevantes ao estudo. Assim, foi consolidada a proposta, com a execução de testes e produção de resultados preliminares. Por fim, foi elaborada a proposta de método para execução no Trabalho de Graduação II (TG-II).

Tendo o objetivo definido e tendo apresentado os conceitos importantes, podemos seguir para a próxima etapa. As atividades previstas foram divididas em seis etapas principais: estudo das implementações atuais, experimentação, implementação de soluções, análise dos resultados, escrita da monografia e apresentação.

Tabela 3. Tabela com o plano de atividades para o Trabalho de Graduação 2.

	Jul	Ago	Set	Out	Nov	Dez
Estudo	X	X				
Experimentação	X	X				
Implementação		X	X			
Análise			X	X		
Monografia			X	X	X	
Apresentação						X

Referências

3S-Smart Software Solutions GmbH (2012). *CODESYS Control V3 Manual*. 3S-Smart Software Solutions GmbH.

Beckhoff Automation GmbH Co. KG (2016). Ethercat system documentation.

Beckhoff HW Documentation (2017). BECKHOFF CX9020 Hardware documentation. `https://infosys.beckhoff.com/content/1033/cx9020_hw/9007202790037387.html`. Acessado em 29/06/2017.

Beckhoff Product Overview (2017). BECKHOFF Fieldbus Components: Product Overview. https://infosys.beckhoff.com/english.php?content=../content/1033/tcsample_crestron/html/tckb_crestron_performance.htm. Acessado em 27/06/2017.

Beckmann, G. (2004). Ethercat communication specification, version 1.0. *EtherCAT technology group*.

Bolton, W. (2015). *Programmable logic controllers*. Newnes.

Bryan, L. A. e Bryan, E. A. (1997). *Programmable controllers: theory and implementation*. Industrial Text Company.

Burns, A. e Wellings, A. J. (2001). *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education.

Erickson, K. T. (1996). Programmable logic controllers. *IEEE potentials*, 15(1):14–17.

Field-Richards, H. S. (1983). Real-time languages-design and development: Sj young-john wiley (1982)£ 29.50, pp 352. *Microprocessors and Microsystems*, 7(4):184.

Gupta, A. e Arora, S. (2009). *Industrial automation and robotics*. Laxmi Publications.

International Electrotechnical Commission and others (1993). Iec 61131-3. *Programmable Controllers-Part*, 3.

Jain, R. (1990). *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons.

Jarp, S. (2002). A methodology for using the itanium 2 performance counters for bottleneck analysis. Technical report, Technical report, HP Labs.

John, K.-H. e Tiegelkamp, M. (2010). *IEC 61131-3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids*. Springer Science Business Media.

Kopetz, H. (2011). *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media.

Motion Controllers (2017). Motion controllers. http://dl.mitsubishielectric.com/dl/fa/document/catalog/ssc/l03014/qmotion.pdf. Acessado em 09/06/2017.

OMRON Automation Pvt Ltd (2016). Ethercat communication manual.

Série Nexto (2017). Série Nexto - Altus. www.altus.com.br/ftp/Public/Portugues/Produtos/Nexto/00DocSerie/ManuaiseApostilas/MU214000.pdf. Acessado em 29/06/2017.

Vosough, S. e Vosough, A. (2011). Plc and its applications. *International journal of multidisciplinary sciences and engineering*, 2(8).