Melhorando o desempenho do ELUS através de especialização de *templates*

Rita de Cássia Cazu Soldi, Giovani Gracioli e Antônio Augusto Fröhlich

[1]Laboratório de Integração de Software e Hardware (LISHA)
Universidade Federal de Santa Catarina (UFSC)
88040900 – Florianópolis, SC – Brasil

{rita,giovani,guto}@lisha.ufsc.br

Abstract. *Dynamic reconfiguration of software in embedded systems with severe resource constraints is a task that requires a performance optimization, since the reconfiguration process competes with the software for the limited system resources. ELUS is an operating system infrastructure that performs this activity effectively and safely, however, the use of resources can still be improved. This article presents a study of ELUS and proposes a performance improvement using templates specialization.*

Resumo. *Reconfiguração dinâmica de software em sistemas embarcados com severas restrições de recursos é uma tarefa que exige otimização de desempenho, visto que o processo de reconfiguração compete com o software pelos limitados recursos do sistema. O ELUS é uma infraestrutura de sistema operacional que realiza esta atividade de maneira eficaz e segura, porém, a utilização dos recursos ainda pode ser aprimorada. Este artigo apresenta um estudo do ELUS e propõe uma melhoria de desempenho por meio do uso de especialização de templates.*

1. Introdução

Reconfiguração dinâmica é a capacidade de atualizar o *software* de um determinado sistema sem perder o seu estado de execução e pode ser usada para diversas finalidades como realizar atualizações, implementar algoritmos para substituir componentes do sistema, monitoramento dinâmico, apoio a otimizações específicas da aplicação, ajuda para que componentes do sistema possam ser recebidos da rede e instalados em tempo de execução, entre outros.

Este tipo de reprogramação é extremamente importante no contexto de sistemas embarcados para que se possa adaptar os dispositivos às mudanças que ocorrem no ambiente, modificar o seu conjunto de tarefas e até aprimorar o uso de seus recursos. Quando a necessidade é aplicar a reconfiguração dinâmica em sistemas com severas restrições (ex. processamento, memória, energia) torna-se um desafio, pois, apesar dos mecanismos de atualização competirem pelos já limitados recursos do sistema, eles não devem influenciar no funcionamento do mesmo.

O ELUS (*Epos Live Update System*) [Gracioli and Fröhlich 2009] é uma infraestrutura de sistema operacional para a reconfiguração dinâmica que difere das outras infraestruturas pelo baixo consumo de memória, alta configurabilidade, simplicidade e transparência para as aplicações. Se comparado a trabalhos equivalentes, o ELUS consome

menos memória, possui um menor tempo de invocação de métodos e menor tempo de reconfiguração. Apesar dos resultados favoráveis, foi identificado que é possível melhorá-los. O ELUS foi implementado em C++ utilizando o *framework* de componentes meta-programado do EPOS [Fröhlich 2001]. O framework utiliza templates e com isso há uma replicação de código sempre que um novo componente é marcado como reconfigurável. Para contonar esse problema, este artigo propõe o uso da técnica de especialização de templates para diminuir a replicação de código dentro do *framework*. Com o uso dessa técnica, foi possível melhorar o consumo de memória sem perder desempenho em outras caracterícias, como o tempo de invocação de métodos e tempo de reconfiguração dos componentes.

A estrutura deste artigo apresenta-se da seguinte forma. A seção 2 detalha o ELUS quanto à sua organização e algumas das suas limitações. Na seção 3 é detalhada a modelagem para a melhoria de desempenho do processo de reconfiguração dinâmica para a infraestrutura ELUS. Os resultados obtidos com as modificações são apresentados na seção 4. A seção 5 apresenta os trabalhos relacionados e, finalmente, a seção 6 conclui o trabalho.

2. Epos Live Update System (ELUS)

O ELUS é uma infraestrutura de reconfiguração dinâmica que pode ser utilizada em sistemas embarcados com recursos limitados. Construída dentro do *framework* de componentes do EPOS (*Embedded Parallel Operating System*) [Fröhlich 2001] em torno do aspecto de invocação remota, esta arquitetura permite que um componente seja selecionado como reconfigurável ou não em tempo de compilação. As principais características que a difere dos outros trabalhos relacionados são a configurabilidade, o consumo de memória reduzido, a simplicidade e a transparência para as aplicações e reconfiguração.

2.1. Arquitetura

A arquitetura do ELUS é apresentada na Figura 1. O *framework* de componentes original do EPOS foi estendido para suportar também reconfiguração dinâmica. A invocação de um método de um componente passa pelo PROXY que envia uma mensagem ao AGENT. O valor de retorno depois da execução do método é enviado de volta ao cliente através de uma mensagem. Um nível de indireção é criado entre o cliente e o método real, tornando o AGENT o único membro da estrutura ciente da posição do componente na memória do sistema. A OS BOX no AGENT controla o acesso aos métodos do componente através de um semáforo para cada componente, chamando os métodos do AGENT através de uma tabela. O AGENT quando recebe uma invocação de método, envia o pedido para o ADAPTER que chama o método real do componente através da tabela de métodos virtuais (VTABLE) do componente.

O ELUS TRANSPORT PROTOCOL (ETP) é um protocolo para o recebimento de mensagens de reconfiguração e toda atualização ocorre a partir de uma requisição neste formato. O *framework* permite a atualização de componentes, da aplicação e do próprio *framework* de reconfiguração. Conforme pode ser observado na Figura 2, para atualizações referentes a um componente, o ELUS tenta alcançar um estado em que nenhuma *thread* esteja invocando métodos deste componente, ou seja, o estado quiescente. Em seguida é verificada a necessidade de alocar um espaço novo para o código, necessário

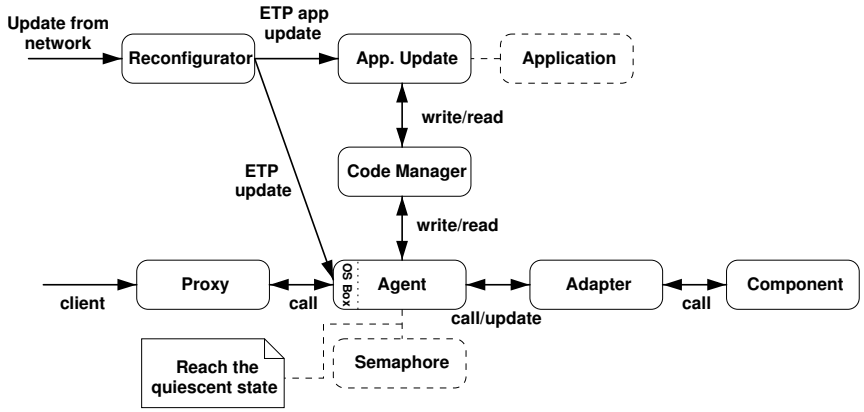

Figura 1. Visão geral da arquitetura do ELUS [Gracioli and Fröhlich 2010].

apenas quando o código novo é maior que o antigo. Na sequência é feita a atualização da tabela de métodos virtuais e é verificado se há necessidade da atualização do componente no *framework*.

Para a atualização de componentes do *framework* é necessário um pouco mais de cautela, pois o `Agent` - mais especificamente o método `trapAgent` - é o ponto de entrada para o serviço de reconfiguração e o seu endereço deve ser conhecido previamente para que o novo código do componente possa ser compilado corretamente. Então para ter certeza de que o endereço do `trapAgent` não será modificado ele não é uma função passível de reconfiguração.

Na reconfiguração da aplicação, a partir do pedido de atualização se inicia o processo de desabilitar as interrupções para atingir o estado quiescente da aplicação. O tamanho do código recebido é comparado com o código antigo para saber se ocorre uma simples atualização do endereço (se o novo código for menor ou igual ao antigo), caso contrário, é necessário alocar um novo espaço de memória antes de atualizar o endereço. Após a atualização da aplicação o sistema deve ser reiniciado para que suas as novas atividades possam ser realizadas.

2.2. Limitações

Devido à estrutura do ELUS, cada componente selecionado como configurável gera um sobre custo em termos de memória, porque além da implementação real do método pelo próprio componente, ainda se faz necessário incluir os métodos do componente no *framework*. Após a geração do sistema final, o código dos métodos do *framework* é replicado para cada componente. Por exemplo, cada componente selecionado como reconfigurável irá ter o código do método *update* replicado. A configuração de consumo mínimo de memória para adicionar o componente no *framework* seria conter apenas os métodos criar, destruir, atualizar e um método sem parâmetros e retornar o valor e mesmo assim haveria o sobre custo de cerca de 1.6KB de código e 26 *bytes* de dados.

Outra limitação é o tempo de invocação dos métodos, pois ocorre uma indireção entre a aplicação e a invocação do método real, atribuindo ao *framework* uma série de responsabilidades que acabam aumentando o tempo de invocação. Dentre as atribuições do *framework* estão o armazenamento e controle sobre os objetos reconfiguráveis criados pela aplicação, incluindo alcançar o estado quiescente do componente. Todas estas

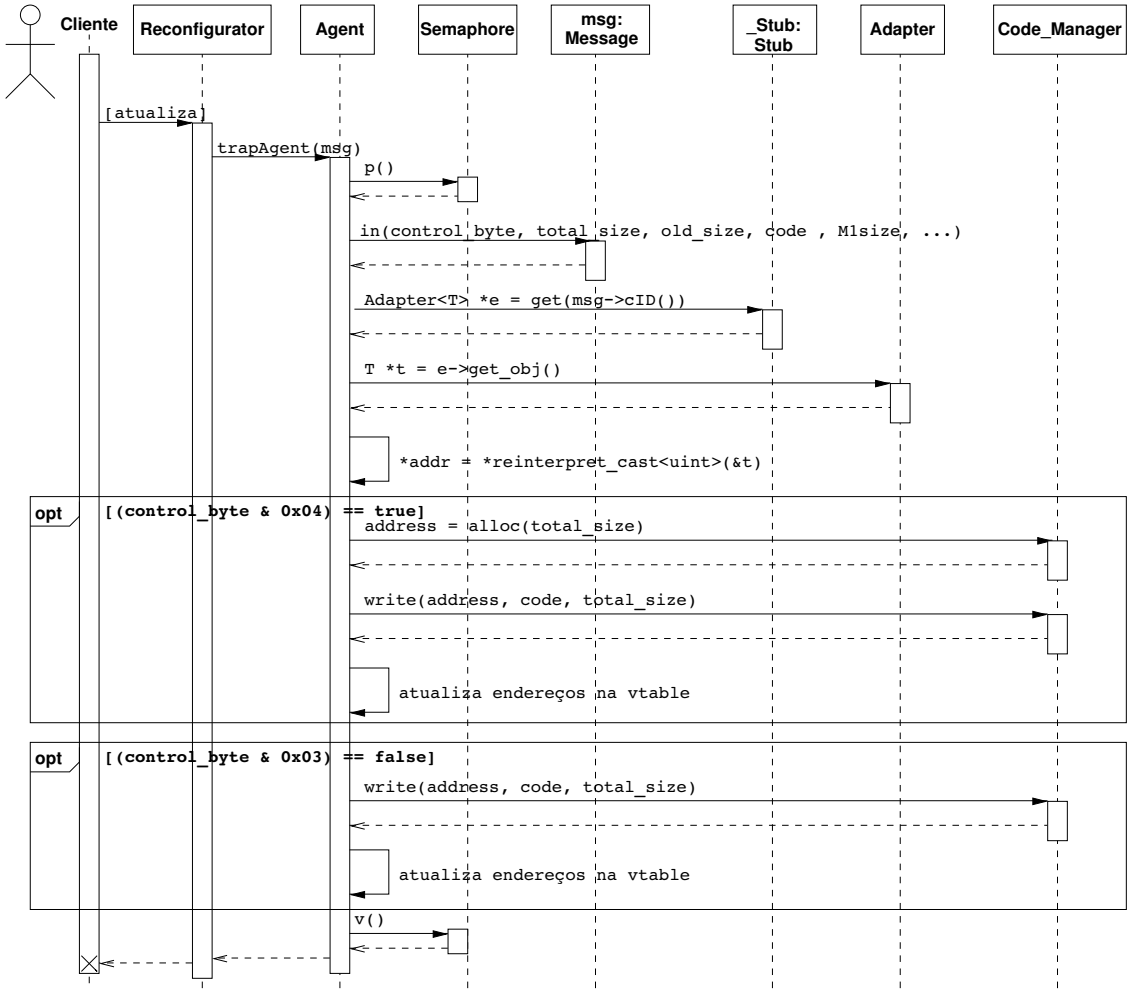

Figura 2. Resumo da sequência de atividades realizadas pelo ELUS em uma reconfiguração de componentes.[Gracioli and Fröhlich 2009]

responsabilidades fazem com que o tempo de invocação de métodos utilizando o *framework* seja cerca de 10 vezes mais lento do que a invocação normal ou através de uma VTABLE. Mesmo com esse sobrecusto, o ELUS apresenta um melhor desempenho que os trabalhos relacionados, possuindo o tempo de invocação pelo menos 7 vezes mais rápido, o tempo de reconfiguração com cerca de 197 ciclos menor e consumindo menos de 33% da memória utilizada por qualquer um dos trabalhos relacionados [Gracioli and Fröhlich 2010].

3. Proposta de modelagem

Analisando o código do *framework* da infraestrutura ELUS foi possível observar uma estrutura metaprogramada utilizando *templates*, conforme pode ser observado na Figura 3.

Os elementos compõem o `framework` e são utilizados no processo de reconfiguração que a infraestrutura provê são: `Handle`, `Stub`, `Proxy`, `Agent`, `Scenario` e `ScenarioHash`. Segue abaixo uma pequena descrição das suas respectivas funções.

- `Handle` - Quando o suporte ao *framework* e à reconfiguração dinâmica estão

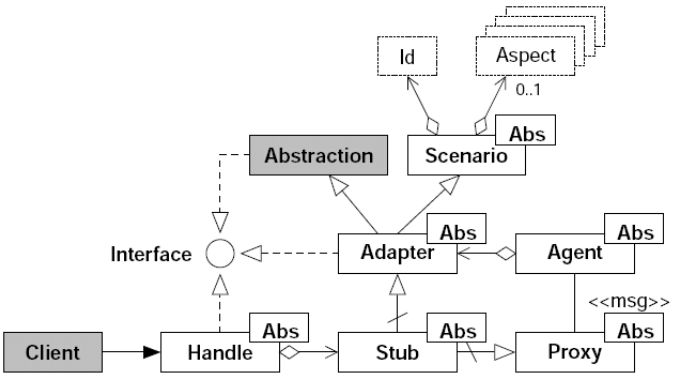

Figura 3. Interação entre os elementos do *framework* metaprogramado [Gracioli and Fröhlich 2009].

habilitados, o componente é exportado como um parâmetro para o `Handle`, que possui a mesma interface (operações) que o componente, mas com a invocação dos métodos encaminhadas para o `Stub`. O `Handle` é responsável por verificar se um componente está ou não selecionado como reconfigurável.

- `Stub` - Responsável por criar um `Proxy` ou um `Adapter` conforme os parâmetros recebidos na sua criação via Handle. Então se a reconfiguração do componente está habilitada, Handle irá criar um `Stub` que contém todos os métodos do `Proxy`, permitindo assim a comunicação do `Proxy` com o `Agent`, surgindo aqui um nível de indireção entre as chamadas de métodos da aplicação e os componentes reais do sistema.
- `Proxy` - Responsável por efetuar os métodos do componente. Quando recebe um pedido do `Handle`, ele utiliza a classe `Message` para passar o `ID` do componente e a solicitação desejada para o `Agent`.
- `Agent` - Possui duas funções : a invocação de métodos e a reconfiguração dos componentes e do próprio *framework*. Caso seja uma invocação de métodos, a `Message` será enviada pelo `Proxy` então ele chama o `Adapter` e reconfigura o componente através do método do `update`. Senão a `Message` foi enviada pela *thread* `Reconfigurator`, e o `Agent` deve fazer a reconfiguração do próprio *framework*.
- `Scenario` - responsável por aplicar os aspectos selecionados para cada componente recebido pelo metaprograma. Também armazena os objetos criados pelo `Agent` em tempo de execução.
- `ScenarioHash` - Funciona como uma abstração das operações de controle, armazenamento, busca e recuperação de objetos através de uma tabela *hash*.

Os elementos auxiliares são: `Message`, `Adapter`, `Application_Update`, `Reconfigurator` e `Code_Manager`.

- `Message` - Ponto comum entre o `Proxy` e o `Agent`. Oferece uma interface para anexar e recuprar informações úteis para a reconfiguração, como por exemplo tamanho do códito, tipo de reconfiguração, retorno de um método, entre outros.
- `Adapter` - Funciona como um *wrapper* e realiza a adaptação do componente que foi recebido como parâmetro através dos métodos `leave` e `enter` do `Scenario` antes e depois da invocação do método real.

- `ApplicationUpdate` - É um componente criado exclusivamente para dar suporte à atualização da aplicação. Sua tarefa principal é desabilitar interrupções do sistema para conseguir atingir o estado quiescente.
- `Reconfigurator` - É uma `thread` criada na inicialização do sistema e é responsável por receber o pedido de reconfiguração, construir `Message` e enviar um pedido de reconfiguração para o `Agent`
- `Code_Manager` - Responsável por utilizar a estrutura fornecida pelo mediador de *hardware* para gerenciar alocação e liberação de espaços na memória.

Apesar de todos os elementos serem importantes para o processo de reconfiguração, o comportamento padrão para implementação de *templates* utilizando C++ é replicar o código para cada função recebida como argumento. Desta forma, os elementos que interagem diretamente com o componente atualizável são replicados para cada argumento de tipo de componente recebido. Uma boa alternativa para evitar a replicação de código é extrair os techos comuns a todas as funções *template* para uma especialização para ponteiros *void*.

3.1. Especializações de *Templates* em C++

Mecanismos de *templates* providos pelo C++ são uma maneira de adicionar à classes e funções conceitos genéricos, isto é, permitem criar um código reutilizável onde os tipos sejam passados como parâmetro.

Por padrão, as classes *template* possuem uma única implementação para qualquer argumento recebido e quando deseja-se dar um tratamento mais refinado à um determinado tipo de argumento, é necessário realizar descrições alternativas do *template*. Estas descrições, também conhecidas como especializações, são escolhidas pelo compilador com base nos argumentos fornecidos para o `template`.

O ponteiro *void* é um ponteiro que pode apontar para qualquer tipo de objeto e pode ser explicitamente convertido para qualquer outro tipo. Sendo assim, se for realizada uma especialização de *template* para ponteiros *void*, ela poderia implementar as funções comuns a todos os tipos de argumentos sem que haja replicação de código.

Como exemplo desta ideia são apresentados na Figura 4 (classe `Vetor` genérica) e na Figura 5 (utilizando uma base para os demais argumentos).

```
1  template <class T> class Vector{
2     T* v;
3     int sz;
4  public:
5     Vector();
6     Vector(int);
7
8     T& elem (int i) {return v[i]; }
9     T& operator [] (int i);
10
11    void swap(Vector &);
12
13    // ...
14 };
```

Figura 4. Classe `Vector` generalizada [Stroustrup 1997].

```
1  template <class T> class Vector <T*> : private
       Vector <void*>{
2  public:
3    typedef Vector <void*> Base;
4
5    Vector() : Base() {}
6    explicit Vector(int) : Base(i) {}
7
8    T& elem (int i) {return static_cast<T*&>(Base::
         elem(i)); }
9    T& operator [] (int i) {return static_cast<T*&>(
         Base::operator[](i)); }
10
11   // ...
12 };
```

Figura 5. Classe `Vector` com base especializada para para ponteiros *void* [Stroustrup 1997].

Podemos observar que na Figura 5 não há mais a implementação de alguns métodos dentro da classe genérica de `Vector` e, consequentemente, ao invés de uma replicação haverá apenas uma chamada de método.

Existem vários trechos do *framework* que podem ser modificados de acordo com esta proposta. A Figura 6 mostra um trecho do método `update` da classe `Agent`. Nela podemos observar que em grande parte do código não há nenhuma referência ao argumento `T` do *template* e por isso é uma das classes em que há mais replicação de código.

```
static void update (Message* msg) {
  unsigned char mID, ctr_byte;
  unsigned int size, t_size, code, old_size;
  unsigned int n_addr = 0;
  unsigned int *addr = 0;
  Message *msg_extra;
  msg->in(ctr_byte, mID, t_size, code, msg_extra);
  unsigned char *new_code = (unsigned char *) code;
                Código que utiliza o argumento "class T"
  unsigned int *p = (unsigned int *)(*addr);
  char zero = 0;
  unsigned int displacement = 0;
  unsigned int dispatcher_size = 0;
  unsigned int total_msgs = (ctr_byte & 0xF0) >> 4;
  switch(ctr_byte) {
      ...
  }
}
```

```
unsigned int *keys = _Stub::get_keys();
Adapter<T> *e;
T * t;
for (unsigned int i = 0; i < Adapter<T>::N_OBJS; i++) {
    if(keys[i] == 0) {
        continue;
    } else {
        e = _Stub::get(keys[i]);
        t = e->get_obj();
        if(T::TYPE == msg->id().type()) {
            addr = *reinterpret_cast<unsigned int**>(&t);
        }
        break;
    }
}
```

Figura 6. Trecho do método `update` da classe `Agent`.

Considerando as modificações que devem ser realizadas no modelo atual do ELUS, a Figura 7 apresenta o novo diagrama de classes. Tanto o `Agent<void*>` quanto o `Scenario<void*>` são especializações utilizadas como base para diminuir a replicação código existente. Estas classes realizam os mesmos métodos, porém o retorno agora é um ponteiro `void` e deve ser reinterpretado pelo tipo `T` do *template* através das classes `Agent<T>` e `Scenário<T>`.

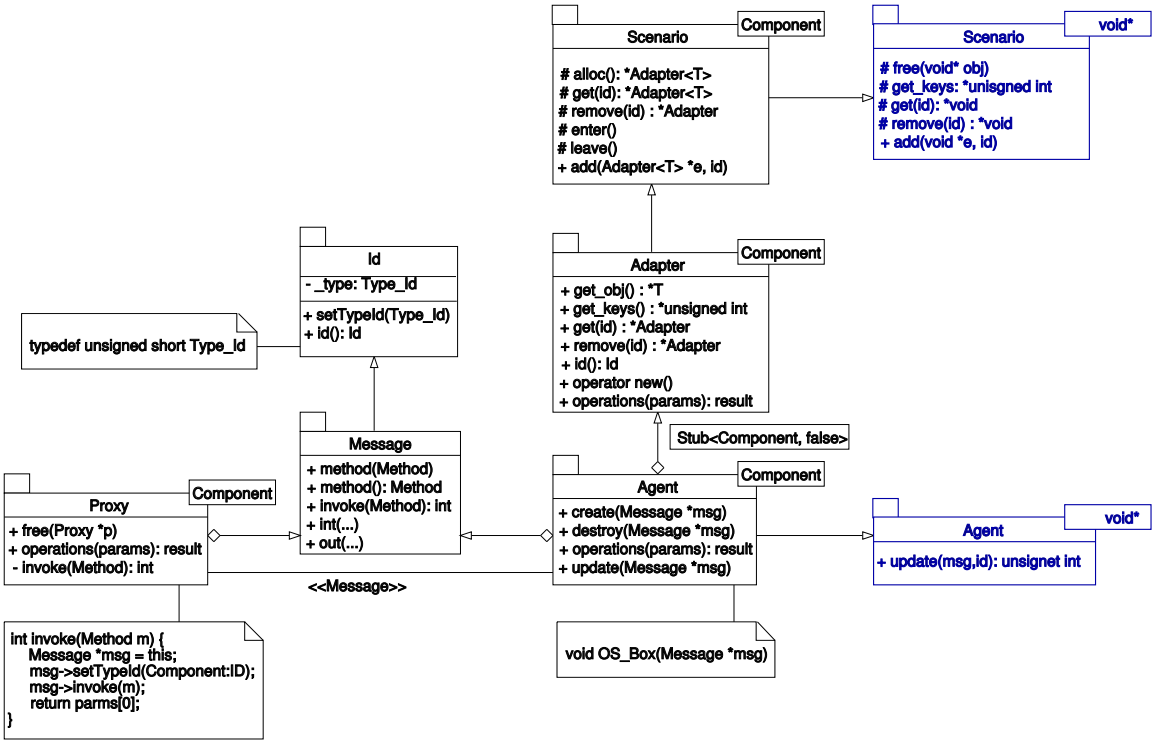

Figura 7. Diagrama da classes envolvidas no processo de reconfiguração.

4. Resultados

A avaliação do ELUS modificado foi realizada na plataforma Mica2 [1] e o código gerado pelo compilador GNU g++ versão 4.0.2. Os testes realizados visam medir o consumo de memória, desempenho de invocação de métodos e tempo de reconfiguração. Ainda, na medição de consumo de memória foi utilizada a ferramenta GNU `objdump` na versão 2.16.1 para analisar as imagens do sistema geradas pelo compilador e o tempo gasto na reconfiguração foi medido em ciclos do processador segundo o manual do microcontrolador.

4.1. Consumo de memória

Esta avaliação mensurou o sobrecusto adicionado ao sistema quando um componente é marcado como reconfigurável. A Tabela 1 apresenta os resultados comparando o ELUS original com a versão modificada. A configuração mínima para o ELUS original é de cerca de 1.6 KB de memória, sendo que com a modificação realizada por este trabalho são necessários apenas 664 bytes para representar o mesmo componente.

Para verificar qual parte do *framework* foi a responsável por este ganho, foram realizadas medições mais específicas, desta vez considerando os métodos `create`, `destroy`, `update` e mais os quatro tipos possíveis de métodos (com parâmetro, sem parâmetro, com valor de retorno e sem valor de retorno), conforme pode também ser observado na Tabela 1.

[1] Microcontrolador Atmel Atmega128 de 8Mhz, 4KB de memória RAM, 128KB de memória flash, 4KB de EEPROM

Tabela 1. Consumo de memória dos métodos individuais em ambas versões do ELUS.

Método do Framework	ELUS original Tam. seção (bytes)		ELUS modificado Tam. seção (bytes)		Direfença
	.text	.data	.text	.data	(%)
Create	180	0	178	0	1,11
Destory	138	0	136	0	1,45
Método sem parâmetro e valor de retorno	94	0	90	0	4,25
Método com um parâmetro e sem valor de retorno	98	0	94	0	4,08
Método sem parâmetro e com valor de retorno	112	0	104	0	7,14
Método com um parâmetro e valor de retorno	126	0	122	0	3,17
Update	1250	0	260	0	79,20
Dispatcher	0	2 *(# métodos)	0	2*(# métodos)	0
Semaphore	0	18	0	18	0
Tamanho mínimo	1662	26	664	26	59,12

A Tabela mostra que o método `Update` é o maior responsável por esta diminuição. Este método se encontra na classe `Agent` que é a classe que continha uma grande parte de código que não utilizava informação de tipo, mas mesmo assim era replicada para cada componente marcado como reconfigurável.

4.2. Tempos de invocação de método

O tempo de invocação de métodos é avaliado em termos de quantidade de ciclos para realizar uma chamada de método e analisar se houve algum sobre custo neste processo. É uma avaliação importante porque o *framework* deve realizar uma série de atividades antes e depois da invocação de métodos. A Tabela 2 apresenta o tempo de invocação de métodos do ELUS original, modificado e os compara com a chamada de um método normal e através da tabela de métodos virtuais. Percebe-se claramente que há uma perda de desempenho causada pelo nível de indireção criado entre a aplicação e a invocação do método real através do *framework* metaprogramado.

Apesar de ser uma das partes substituídas para uma tentativa de melhora de desempenho, o resultado numérico das duas versões foi igual. Estudando o código gerado (e.g, *assembly*) para ambas versões pode-se concluir que houve diferença das instruções, mas após a recontagem do número de ciclos utilizados para a invocação de método, este número permaneceu sem alterações.

4.3. Tempo de Reconfiguração

A comparação dos tempos de reconfiguração foi medida em dois cenários: (i) o novo código do componente é menor ou igual ao antigo, sendo assim a atualização pode utilizar a mesma posição do componente e (ii) o novo código de um componente é maior que o

Tabela 2. Comparação dos tempos de invocação de método entre uma invocação normal, através da vtable, do ELUS e do ELUS modificado

Invocação	Normal	Vtable	ELUS	ELUS Modificado
Método sem parâmetro e sem valor de retorno	4	13	135	135
Método com parâmetro e sem valor de retorno	7	15	139	139
Método sem parâmetro e com valor de retorno	6	14	152	152
Método com parâmetro e com valor de retorno	8	16	161	161

antigo, em que se faz necessário a alocação de uma nova posição para o componente. Para ser uma comparação justa em relação aos trabalhos relacionados, esse teste não considerou o tempo de recebimento de dados pela rede e nem o tempo de escrita dos dados na memória de programa. O resultado do teste pode ser observado na Tabela 3.

Tabela 3. Comparação dos tempos de reconfiguração para um componente no ELUS e do ELUS modificado, medido em ciclos

Atualização	ELUS original	ELUS modificado
Mesma posição	362	368
Nova posição	414	397

Examinando o tempo de reconfiguração em relação ao ELUS original e os resultados obtidos pelas modificações é possível observar um aumento no tempo da reconfiguração quando o componente é colocado na mesma posição e uma diminuição do tempo em relação à atualização em uma nova posição. Embora este resultado pareça controverso, pode ser facilmente entendido se o código *assembly* gerado tanto pelo ELUS original quanto pelo ELUS modificado forem analisados. Devido à utilização da especialização para ponteiros *void* o código modificado inclui instruções de cálculo e retorno de um valor pela classe `Base`, que servirá para verificar se é necessário ou não a atualização do endereço da tabela virtual em cada objeto.

Já a diminuição do tempo da reconfiguração para uma nova posição pode ser associada à quantidade e à ordem das instruções do código modificado, pois a classe `Base` é responsável por todas as operações, com exceção da atualização do endereço da tabela virtual em cada objeto. O mesmo não ocorre com o ELUS antigo, que realiza este processo através da classe dependente de *template* e precisa realizar verificações relativas ao tipo.

5. Trabalhos relacionados

Contiki [Dunkels et al. 2004] tem seu *kernel* baseado em eventos com *multithreads* preemptivas para cada um dos processos, sendo que cada um deles possui uma interface *stub*, responsável por redirecionar as chamadas para a interface que possui ponteiros para as implementações das funções relativas ao serviço oferecido. Quando recebe uma mensagem de reconfiguração, o *kernel* disponibiliza uma interface para a passagem de ponteiros

entre as duas versões e, no caso de uma remoção de serviço, é enviado um evento no qual o serviço remove-se automaticamente do sistema. Contiki não possui o melhor desempenho para a utilização dos recursos, pois apenas para fornecer à reconfiguração dinâmica já é necessário cerca de 6KB de memória, além disso sua chamada de métodos possui vários ciclos de verificações que devem ser realizadas antes e depois da invocação.

SOS [Han et al. 2005] é composto por módulos atualizáveis em tempo de execução e programados de forma cooperativa, de tal modo que conseguem enviar mensagens e se comunicar com o *kernel* através de uma tabela com *jumps*. Para a atualização dos módulos existe um protocolo de distribuição que faz o *download* dos dados, também aloca memória para o novo módulo, além de ajustar os ponteiros na tabela e dar inicio ao processo através da mensagem `init`. Quando a atualização consiste em uma remoção de módulo, o *kernel* envia uma mensagem `final` para que o módulo libere todos os recursos que está utilizando e informe sua remoção aos módulos que o utilizam. Observando a utilização de recursos este sistema operacional utiliza cerca de 23KB e ainda possui uma grande perda de desempenho para a invocação de métodos, acrescentando mais de 850 ciclos no processo.

O sistema operacional RETOS pretende contornar de forma eficiente limitações de recursos e ainda atingir os seguintes objetivos: fornecer uma interface para programação *multithread*, ter uma abstração orientada à rede de sensores sem fio (RSSF), ser um sistema resiliente e possuir um *kernel* extensível através de reconfiguração dinâmica [Cha et al. 2007]. O RETOS fornece um mecanismo que utiliza relocação dinâmica de memória e ligação em tempo de execução para a reconfiguração dos módulos. Desta maneira, é capaz de gerar metadados a partir de variáveis e funções em tempo de compilação que são armazenados para posteriormente substituir os endereços utilizados pelo módulo quando o mesmo é carregado pelo sistema. Embora utilizar metadados seja uma boa alternativa para a atualização, existe um sobre custo associado à quantidade extra de dados enviados pela rede, o que resulta em uma maior utilização dos recursos energia e memória.

Já MOS (*MANTIS sensor OS*) [Bhatti et al. 2005] é um sistema *multithread* que possui suporte à reprogramação dinâmica e acesso remoto via *shell*. MOS consegue economizar energia através de uma implementação diferenciada do escalonador de eventos, em que é possível identificar quando as *threads* estão ativas e controlar o consumo de energia do microcontrolador. A reconfiguração dinâmica é implementada através de uma chamada para uma biblioteca do *kernel* do MOS e ela pode ser realizada em diversas granularidades, variando desde uma variável dentro de *thread* até o sistema operacional inteiro. O baixo consumo de recursos é uma característica forte no MOS, um exemplo é o tamanho total necessário para armazenar suas estruturas, pois, somando o tamanho do *kernel*, do escalonador de enventos e da rede são utilizados menos de 500 *bytes* de RAM e 14KB de memória de programa.

6. Considerações finais

Neste artigo apresentou-se um estudo do *framework* do ELUS e uma proposta de melhoria da desempenho da reconfiguração dinâmica realizada pela infraestrutura. A nova abordagem foi avaliada em termos de consumo de memória, tempo de invocação de métodos e tempo de reconfiguração. A especialização através de ponteiros *void* apresentou resultado positivo para o consumo de memória, pois foi possível reduzir cerca de 60% do

consumo e atualmente o tamanho mínimo para um novo componente do sistema passa a ser 664 KB. Já o tempo de invocação de métodos não sofreu alterações, mas ainda é positivo se comparado aos trabalhos relacionados. Por fim, o tempo de reconfiguração de componentes diminuiu cerca de 17 ciclos para um componente em uma nova posição, entretanto, as modificações também foram responsáveis pelo aumento em 6 ciclos no tempo de atualização de um componente na mesma posição.

Dentre as implementações futuras estão a medição do consumo de energia, que também é um recurso muito importante no domínio de sistemas embarcados e o aumento da reusabilidade do código.

Referências

Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, A., and Han, R. (2005). Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks and Applications*, 10(4):563–579.

Cha, H., Choi, S., Jung, I., Kim, H., Shin, H., Yoo, J., and Yoon, C. (2007). RETOS: resilient, expandable, and threaded operating system for wireless sensor networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*, page 157. ACM.

Dunkels, A. et al. (2004). Contiki-a lightweight and flexible operating system for tiny networked sensors.

Fröhlich, A. A. (2001). *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin.

Gracioli, G. and Fröhlich, A. (2009). ELUS: a Dynamic Software Reconfiguration Infrastructure for Embedded Systems.

Gracioli, G. and Fröhlich, A. (2010). ELUS: A dynamic software reconfiguration infrastructure for embedded systems. In *Telecommunications (ICT), 2010 IEEE 17th International Conference on*, pages 981–988. IEEE.

Han, C., Kumar, R., Shea, R., Kohler, E., and Srivastava, M. (2005). A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176. ACM.

Stroustrup, B. (1997). *C++ Programming Language, The (3rd Edition)*. Addison-Wesley Professional.