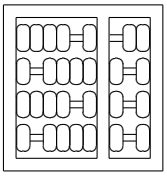

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Um sistema de ligação dinâmica independente de arquitetura baseado em ADL

Rafael Auler *Paulo Cesar Centoducatte*
Alexandro Baldassin

Technical Report - IC-09-43 - Relatório Técnico

November - 2009 - Novembro

Um sistema de ligação dinâmica independente de arquitetura baseado em ADL*

Rafael Auler Paulo Cesar Centoducatte Alexandro Baldassin

Resumo

Diferentes instruções são implementadas em processadores com conjunto de instruções específico para a aplicação (ASIPs - Application Specific Instruction-Set Processors), motivados pela necessidade de especializar o processador para uma determinada carga de software em uma plataforma embarcada. Para que o projeto seja concluído em tempo viável, é importante reduzir os esforços necessários para construir ferramentas de desenvolvimento de software e simuladores para a nova plataforma. Para isso, simuladores e demais ferramentas podem ser automaticamente sintetizados baseados na descrição da arquitetura do processador. Neste relatório técnico discutimos o projeto de um sistema completo de ligação dinâmica independente de arquitetura alvo, compreendendo a capacidade de geração automática de ligadores: um ligador de tempo de compilação e um carregador para ligação em tempo de execução. O sistema, entretanto, é dependente de arquivo objeto e adota o formato ELF. O carregador utilizado foi especificamente construído para este projeto de modo que não dependemos do carregador da biblioteca glibc. O objetivo principal é o fácil redirecionamento para aplicação em um processador alvo bem como às suas respectivas regras da interface binária da aplicação (ABI - Application Binary Interface). Tais informações dependentes do alvo são extraídas automaticamente de um modelo descrito em uma linguagem de descrição de arquiteturas (ADL - Architecture Description Language). O estudo foca a implementação na ADL ArchC, e testes de validação para a arquitetura ARM são apresentados, acompanhados de uma breve análise dos resultados experimentais para o simulador.

Abstract

Different instructions are implemented by Aplication Specific Instruction-Set Processors (ASIPs), motivated by the need of processor specialization to a known software load in an embedded platform. In order to meet time demands, it is critical to reduce necessary efforts to build software development tools and simulator for the platform under development. To address this problem, simulators and other tools can be automatically generated based on a processor architecture description. In this technical report we discuss the project of a complete architecture independent dynamic linking system: the linker for compile-time and loader for run-time. The system is object file dependent and relies on the flexible ELF. Our loader is specifically designed for this project and we don't depend upon glibc's loader. The main purpose is to be easily

*Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP. O projeto contou com suporte financeiro da Fundação de Amparo a Pesquisa do Estado de São Paulo (FAPESP), processo 08/53624-3.

retargetable for application in a target processor and respective Application Binary Interface (ABI) rules. These target specific information are extracted from an Architecture Description Language (ADL) model. Our study focuses in the ADL ArchC, and validation tests for the ARM architecture are presented.

1 Introdução

Decorrente da crescente integração dos dispositivos de hardware em um único chip, a indústria tem demonstrado uma grande aceitação de sistemas computacionais completos na forma de SoC(System on Chip). Esta tendência criou uma demanda por ferramentas sofisticadas para simulação do sistema completo em alto nível. Para abordar este desafio, uma alternativa surgiu com as ADLs (Architecture Description Language), capazes de descrever o projeto com elevado grau de abstração, mas flexível o suficiente para permitir o estudo do impacto de compromissos da arquitetura no projeto do software que irá executar nesta plataforma (*software/hardware co-design*).

Características importantes de ferramentas baseadas em ADLs não se restringem à capacidade de geração de simuladores comportamentais, mas englobam também a síntese de ferramentas para suporte à criação de software para a plataforma em desenvolvimento. Estas ferramentas compreendem desde montadores e ligadores até compiladores. Somente com este recurso o projetista conta de fato com um suporte completo ao estudo de comportamento em alto nível de seu processador integrado, pois um simulador é pouco útil se faltam meios para se criar softwares para serem simulados de maneira rápida. Rapidez é exatamente a ênfase buscada nas etapas iniciais de um projeto de plataforma, em que a equipe de hardware deve fornecer um modelo funcional de simuladores e ferramentas para que a equipe de software possa iniciar o seu desenvolvimento desde o início do projeto.

ArchC [2] engloba diferentes ferramentas que, de posse das informações contidas em um modelo ADL do processador, produzem simuladores (com ou sem precisão de ciclo), montadores, ligadores e depuradores. Os simuladores são produzidos com uma descrição SystemC do processador, o que significa que o módulo do processador pode ser integrado como parte de uma plataforma maior. A extensão do ArchC para geração automática de montadores e demais ferramentas binárias, denominada *acbingen* [3], enriqueceu a capacidade de descrição de um modelo arquitetural, com a possibilidade da descrição das sintaxes da linguagem de montagem específica do conjunto de instruções do modelo. Deste modo, uma ferramenta é capaz de extrair informações suficientes para produzir automaticamente um montador para a arquitetura alvo. As ferramentas binárias são produzidas com o redirecionamento do pacote GNU binutils [11], o que significa que a mesma interface usada pela suíte de programas binutils está disponível para o projetista gerar executáveis para a sua arquitetura.

Acbingen era capaz de gerar automaticamente ligadores simples para uma arquitetura alvo. Tais ligadores limitam-se a juntar trechos de código relocável produzido pelo montador produzindo em um executável final. Devido a sua limitação, o ligador é capaz de manipular apenas bibliotecas estáticas. Apesar de adequado para ambientes de software simples, não podemos nos limitar a usar bibliotecas estáticas para sistemas mais comple-

xos que executam vários processos, quando a quantidade de código duplicado (proveniente de bibliotecas comuns) é grande. É necessário recorrer a ligadores capazes de manipular bibliotecas compartilhadas para economizar o uso de memória primária do sistema. Para simular este caso, os simuladores também devem ser capazes de administrar o carregamento de bibliotecas compartilhadas solicitadas pelos executáveis.

Bibliotecas compartilhadas estáticas [10] apareceram como a primeira solução para o compartilhamento de trechos de código executável, ou módulos de uma biblioteca. Entretanto, o gerenciamento do espaço de endereços em sistemas operacionais que suportam este tipo de biblioteca é rígido e deve ser conhecido de antemão, dificultando o projeto e evolução dos sistemas de software. Também é necessário considerar que sistemas operacionais recentes *off the shelf*, isto é, projetados por terceiros e prontos para uso, não suportam esta abordagem.

O uso de bibliotecas dinâmicas, suportadas por uma unidade MMU (Memory Management Unit) para endereços virtuais, possibilita o uso flexível de recursos de memória de instruções, com carregamento de código sob demanda e compartilhado entre diferentes processos. Entretanto, a implementação de um ligador capaz de gerar tais bibliotecas não é trivial e adiciona um apreciável grau de complexidade ao ligador. Em particular, a síntese automática deste ligador para um alvo modelado com ArchC foi o motivo de estudo deste trabalho, que resulta de um esforço cuja contribuição foi estender o ArchC com este recurso e os simuladores gerados pelo ArchC com a capacidade de carregar bibliotecas dinâmicas da arquitetura alvo.

A partir de informações de arquitetura extraídas de modelos ArchC, para fazer a síntese automática de um ligador dinâmico, isto é, capaz de produzir bibliotecas dinâmicas, a estratégia adotada consistiu em redirecionar o ligador do pacote binutils, de maneira similar à ferramenta de síntese de montadores do ArchC. A suíte de ferramentas GNU binutils abrange diversos programas que manipulam código executável em vários formatos de arquivo objeto. Aproveita-se a modularidade do binutils para gerar um *backend* específico para o processador descrito com ArchC e desse modo produzir ferramentas redirecionadas para um alvo descrito em ArchC.

Uma importante biblioteca inclusa no pacote binutils, cuja responsabilidade é manipular arquivos objeto independentemente do formato e plataforma, é a libbfd [4]. Um backend para a libbfd consiste em uma implementação de um par formato-plataforma para atender diversas chamadas de manipulação de arquivo objeto, incluindo chamadas que realizam ligação. Estas últimas são usadas pelo ligador GNU ld que propomos redirecionar neste trabalho. Para isso, vamos nos concentrar em definir um backend para a libbfd baseado em um modelo ArchC e no formato ELF (Executable and Linking Format).

Para prover um sistema de ligação dinâmico completo, é necessário também adaptar os simuladores ArchC para a leitura de bibliotecas dinâmicas. Muito do trabalho de ligação dinâmica é deixado para o carregador das bibliotecas, que também é chamado de "ligador de tempo de execução", uma vez que sua funcionalidade é a mesma de um ligador, mas que atua toda vez que o programa é executado. Deve existir uma concordância entre o ligador de tempo de execução, que lê a biblioteca e a prepara para execução, e o ligador de tempo de compilação, que produziu a biblioteca. As convenções definidas na ABI (Application Binary Interface) de uma arquitetura definem esses detalhes, e neste trabalho procurou-se

criar uma ferramenta de síntese independente e flexível, funcional para diversas arquiteturas, aproveitando a infra-estrutura do projeto ArchC.

2 Trabalhos relacionados

O primeiro trabalho de interesse a se destacar é *Retargetable Binary Utilities* [1], que se insere no contexto de pesquisas para criar um conjunto de ferramentas binárias redirecionáveis para novas arquiteturas, especificadas através de ADLs. O foco, entretanto, é em ferramentas binárias básicas como montadores e ligadores, contando com suporte do pacote de software livre binutils, da mesma forma como acbingen.

Neste contexto, é explicitada a necessidade de determinadas informações serem disponibilizadas em uma descrição em ADLs. O projeto então se concentra em redirecionar o pacote livre binutils. O fato de que a complexidade de alguns mecanismos de ligação dinâmica é utilizada para justificar que ferramentas binárias básicas não são triviais.

Memory Footprint Reduction with Quasi-Static Shared Libraries in MMU-less Embedded Systems [9] se encaixa de maneira satisfatória no tema estudado, relacionando bibliotecas compartilhadas ao ambiente de sistemas embarcados. O objetivo é trazer os benefícios de reuso de espaço com a utilização de bibliotecas dinâmicas para sistemas simples, que não possuem MMU[1]. Para isso, uma técnica inovadora é desenvolvida e os pontos negativos do uso de bibliotecas dinâmicas, como a queda de desempenho devido a atribuição de símbolos em tempo de execução, são amenizados.

Um estudo de caso é elaborado para demonstrar os resultado obtidos. Para isso, um modem ADSL doméstico e seu sistema operacional são adaptados para utilizar compartilhamento de bibliotecas, e os resultados são promissores: redução de 35% no uso de memória *flash* e 10% no uso de memória RAM, alcançados com redução de desempenho de menos de 4%.

3 Convenções

Neste relatório, algumas convenções para a tradução do inglês para português foram adotadas, uma vez que material em português sobre este assunto é escasso e a linguagem usada normalmente utiliza os vocábulos diretamente do inglês. Especificamente, quando nos referimos a um montador, estamos falando sobre um *assembler*, da mesma forma que um ligador é um *linker*. *Object file relocations* foram naturalmente traduzidas para relocações.

Como o trabalho de um sistema de ligação dinâmica abrange desde o ligador (fase de compilação) até o carregador (em execução), que também executa tarefas de ligador, iremos utilizar o nome "ligador de tempo de compilação" (normalmente conhecido apenas como *linker*) para o primeiro e "carregador e ligador de tempo de execução" (conhecido como *runtime loader and linker*) para o segundo, para evitar confusão entre os dois.

[1] Memory Management Unit

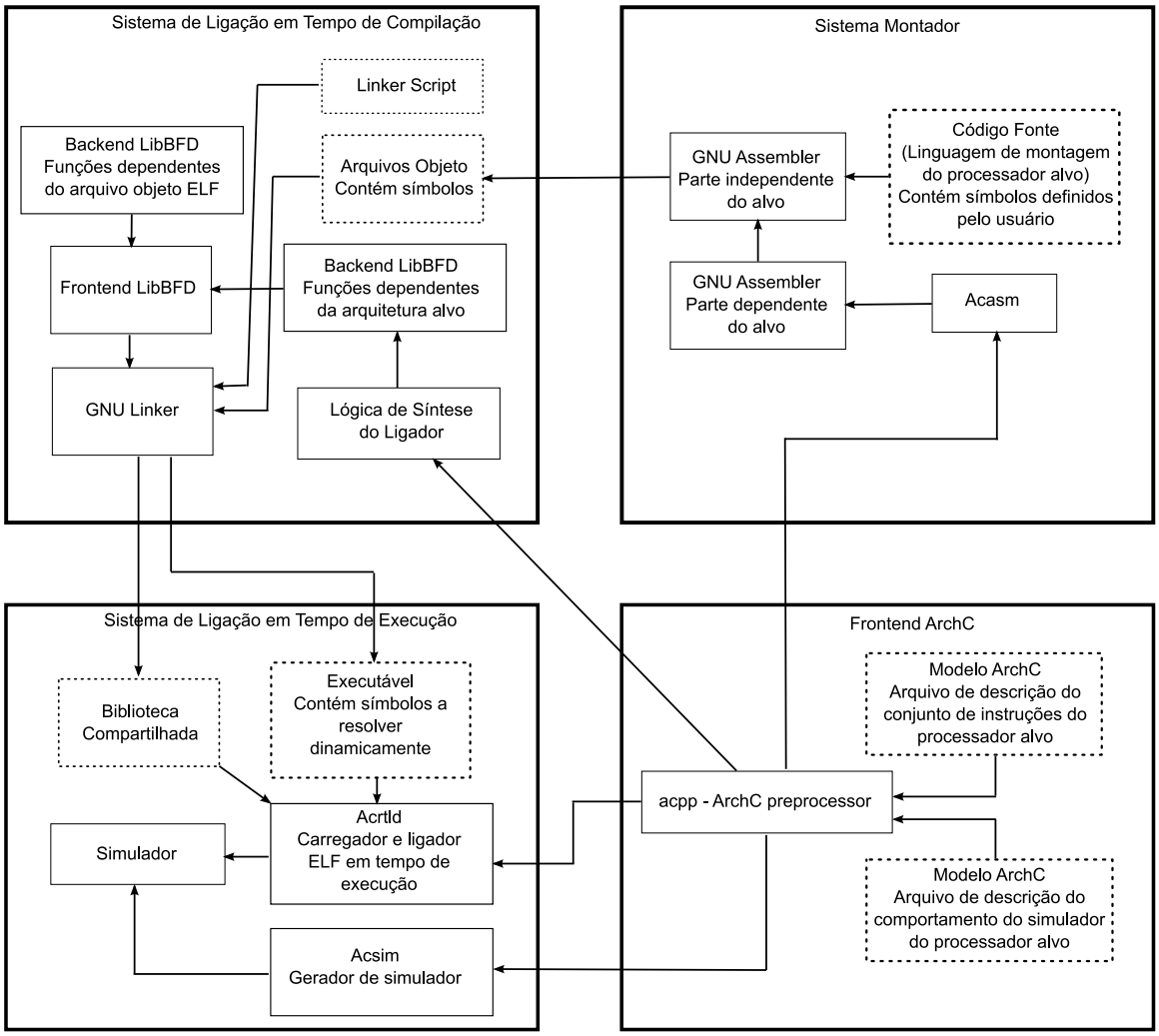

Figura 1: Diagrama da visão geral do sistema apresentado.

4 Visão geral

O trabalho de implementação do projeto pode ser dividido em dois subsistemas distintos, que se integram no que chamamos de sistema de ligação dinâmica. Os dois subsistemas também se comunicam com o projeto ArchC para extrair informações dependentes de arquitetura. O sistema de ligação em tempo de compilação, em tempo de execução e o projeto ArchC aparecem no diagrama da figura 1. Para compreender o processo completo, foi incluído também o sistema montador, que também extrai informações dependentes de arquitetura de um modelo ArchC. Sua função é *redirecionar* o GNU assembler [6], isto é, fazer com que esse montador produza código para a arquitetura descrita com ArchC.

No diagrama, caixas tracejadas representam os produtos e entradas dos processos. Estes últimos são representados em caixas de traço contínuo. O trabalho apresentado aqui

abrange a implementação da "Lógica de síntese do ligador", localizada no sistema de ligação em tempo de compilação, e do "`acrtld`" (*ArchC runtime loader*), localizado no sistema de ligação em tempo de execução. O primeiro sistema, a partir de um *linker script* e arquivos objeto oriundos do sistema montador, produz bibliotecas compartilhadas e executáveis, que podem ou não depender de bibliotecas compartilhadas para sua execução. O segundo sistema carrega executáveis e bibliotecas compartilhadas para o funcionamento do simulador, gerado por *acsim* [2], que irá interpretar as instruções do processador alvo (aquele descrito no modelo interpretado pelo frontend ArchC) carregadas na memória.

O objetivo final deste trabalho é observado com a produção dos itens "biblioteca compartilhada"(síntese), "executável"(síntese) e "simulador"(extensão), explicitados no diagrama.

5 Ligador de tempo de compilação

A construção das duas fases do sistema de ligação dinâmica foram realizadas utilizando, como exemplo, um modelo ArchC de entrada, a descrição da arquitetura ARM. É importante lembrar, entretanto, que o objetivo final é criar um sistema independente, capaz de criar ligadores para qualquer arquitetura devidamente descrita com ArchC utilizando as extensões de linguagem propostas[2].

5.1 Conceitos

O objetivo desta seção é interrelacionar os temas de relocações (as anotações para um ligador fazer consertos no código quando uma seção é reposicionada no arquivo final), bibliotecas dinâmicas e ligadores, explicando o funcionamento do ligador sintetizado por este trabalho. Relocações são tão importantes para um ligador quanto instruções para um montador, e podem ser classificadas em diferentes categorias. Para nosso estudo, interessa-nos a divisão entre estáticas e dinâmicas.

As relocações estáticas recebem este nome pois são criadas pelo montador no momento de gerar um arquivo objeto relocável, e devem ser liquidadas no momento em que um arquivo final (executável) é gerado. Já as relocações dinâmicas aparecem apenas em bibliotecas dinâmicas ou arquivos executáveis ligados a um objeto dinâmico. As relocações dinâmicas são normalmente mais simples do que as estáticas, uma vez que os objetos em questão já foram processados pelo ligador de tempo de compilação, e apenas referências a funções dinâmicas ou dados globais (dependentes de posição) precisam ser resolvidos no momento da execução, que é o momento onde a posição das seções de um objeto dinâmico é determinada na memória.

Para criar um ligador de tempo de compilação capaz de gerar uma biblioteca dinâmica, então, é imprescindível definirmos novas relocações para o ligador do ArchC (isto é, gerado pelo ArchC). Contudo, conforme discutido, este número de relocações é reduzido e não é relacionado à codificação de nenhuma instrução específica. Isto facilita nossa implementação,

[2]As extensões da linguagem não são modificações na gramática em si, mas apenas a adição de arquivos de descrição que contêm informações sobre ligação dinâmica dependentes da ABI do processador.

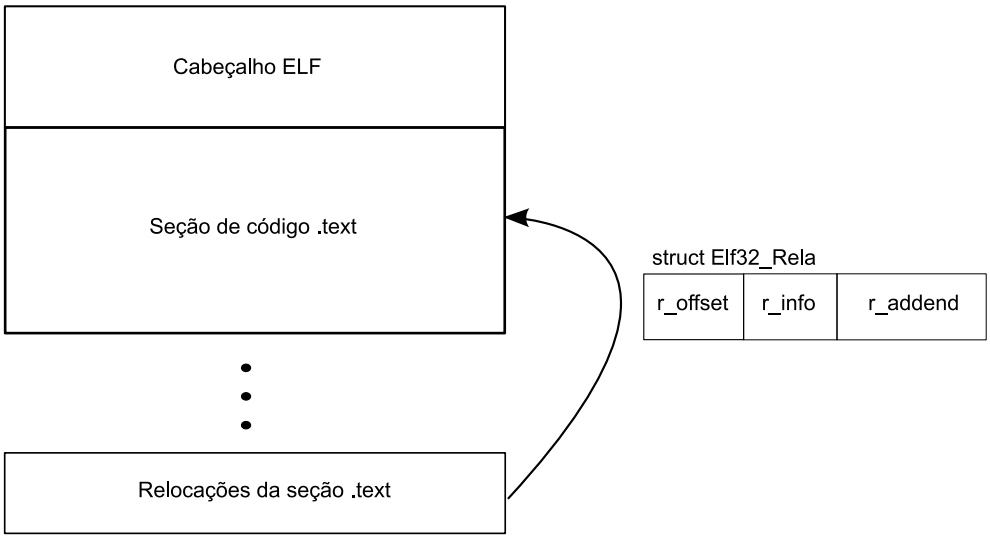

Figura 2: Relocações em um arquivo objeto ELF.

pois quem irá interpretar e resolver as ligações dinâmicas é o carregador do sistema operacional. Em nosso caso, é o simulador do ArchC e, portanto, seria dificultoso fazer com que o simulador dependesse de relocações específicas de arquitetura, uma vez que atualmente o código gerado para o simulador não leva em conta as informações necessárias para criação das ferramentas binárias. Portanto, nossas relocações dinâmicas são independentes de arquitetura e isto encaixa bem em nossas necessidades. Ao se estudar a ABI de diferentes arquiteturas, como a do ARM, MIPS, SPARC, PowerPC e Intel, também se observa estratégia similar. Isto acontece porque as relocações que necessitam de conhecimento de uma codificação de instrução específica são estáticas, ao passo que as relocações dinâmicas atuam apenas em ponteiros (portanto, praticamente independentes de arquitetura, sendo que a única característica dependente de ABI é o tamanho do ponteiro) que precisam ter seu destino resolvido. Utilizando o jargão da linguagem C, estes ponteiros podem ser ponteiros de função (isto é, contêm o endereço de memória de uma instrução que dá início a uma função) ou ponteiros para dados (mais comuns).

O modo como um arquivo objeto ELF é organizado em seções e relocações para estas seções é esboçado na figura 2. A relocação, na figura, é representada como uma estrutura de nome `Elf32_Rela`. Este é o nome como usado no arquivo `elf.h`, que pode ser incluído por qualquer programa em C no ambiente GNU/Linux padrão com os caminhos de inclusão configurados corretamente. O campo `r_offset` possui 32 bits e contém o endereço virtual (e não do arquivo) do alvo, isto é, da instrução que deve sofrer o conserto, ou relocação. O campo `r_info` possui o mesmo tamanho e armazena duas informações: o símbolo referenciado cujo endereço é desconhecido até o momento, motivo da criação da relocação (é armazenado na forma de um índice para um elemento da tabela de símbolos, contida no arquivo em uma seção especial) e o *tipo* da relocação. O código de tipo ou código da relocação é dependente da ABI da arquitetura e complica um pouco o projeto de nosso li-

gador independente de alvo (a solução é discutida na seção 6.5 e envolve uma ferramenta de conversão para um espaço canônico de códigos, quando necessário). O último campo possui 16 bits e armazena o adendo, ou deslocamento, que é aplicado em relação ao endereço virtual do símbolo de referência. Este adendo muitas vezes é omitido com a utilização de uma estrutura `Elf32_Rel`. Nesta outra abordagem, o adendo é armazenado diretamente no local onde a relocação deve ser aplicada (como uma espécie de valor "pré-computado"), mas não é utilizada em nosso projeto[3].

O fato de relocações dinâmicas atuarem em ponteiros não é por acaso. O código independente de posição (PIC - Position Independent Code), utilizado em bibliotecas dinâmicas, garante esta característica, ao fazer com que todas as referências a dados globais ganhem um nível de indireção (é necessário dereferenciar um ponteiro antes de acessar o dado). Note que o compilador deve ser configurado para gerar este tipo especial de código. Com este artifício, o ligador pode colocar este trecho de código em qualquer posição (determinada pelo carregador e ligador de tempo de execução) sem que seja necessário "consertos" (os chamados *patches*) neste código, apenas nos ponteiros. Todavia, há exceções. Considere o seguinte trecho de código C, que tem por objetivo ler 1 byte da entrada padrão e armazená-lo na variável *buffer*:

```
1. extern int errno;
2.
3. int main() {
4.   char buffer;
5.   if (read(1, &buffer, 1) != 1) {
6.     printf("Erro de código %d.\n", errno);
7.   }
8. }
```

Não há motivos para gerar código PIC aqui, uma vez que definimos a função *main*, que é utilizada como ponto de entrada para executáveis, e não para bibliotecas dinâmicas. Sendo assim, o compilador irá gerar uma instrução que acessa diretamente a variável *errno*. Mas como a posição de memória onde esta variável global reside é desconhecida, o montador irá gerar um arquivo objeto com um pedido de relocação para esta instrução, com referência para o símbolo *errno*. Em uma situação padrão, este executável irá ser ligado, nos ambientes GNU/Linux com gcc [13], à biblioteca glibc [7] dinâmica, que define a variável global, ou símbolo, *errno*. Aqui temos um problema: o ligador de tempo de compilação, mesmo analisando a biblioteca, não sabe exatamente a posição de *errno*, pois a biblioteca dinâmica pode ser carregada em qualquer endereço.

Esta situação é resolvida com a introdução de um tipo especial de relocação, *copy relocation*, ou relocação de cópia. O ligador de tempo de compilação irá resolver a relocação estática em nossa função *main* como se *errno* (o símbolo referenciado) fosse uma variável global definida no próprio executável, digamos, na posição `0x1000`. Em seguida, uma relocação de cópia, que é um tipo especial de relocação dinâmica, é emitida para o endereço `0x1000` com referência para *errno*. O carregador e ligador de tempo de execução resolve

[3] É utilizada em diversas ABIs, como a do ARM, Intel e MIPS

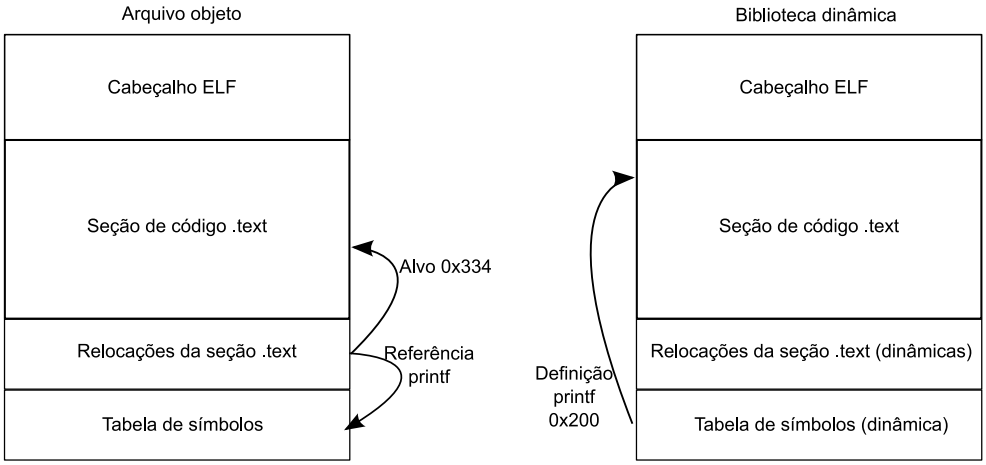

Figura 3: Ligando um objeto contra uma biblioteca dinâmica.

esta relocação especial *copiando*[4] a variável que pertence à biblioteca dinâmica (em endereço arbitrário) para o endereço `0x1000`. Assim, o programa irá acessar a posição correta de memória como se a variável fosse sua, ao passo que a biblioteca dinâmica, por usar referenciamento indireto (devido ao código PIC), terá seus ponteiros ajustados para acessar a variável *errno* no endereço `0x1000`.

5.1.1 Símbolos

Quando o ligador de tempo de compilação está resolvendo as relocações de um objeto, isto é, analisando o símbolo de referência, pesquisando sua localização nas bibliotecas incluídas e alterando o alvo da relocação com o endereço do símbolo, é importante distinguir se o símbolo de referência é definido em um objeto (ou biblioteca) estático ou dinâmico. No caso estático, o objeto que define o símbolo é inteiramente incluso no executável final. No caso dinâmico, temos dois subcasos: no primeiro, o símbolo é um dado global, e no segundo, o símbolo é uma função. Para o caso do dado global, já vimos que ocorre uma relocação de cópia quando o código não é PIC. Caso contrário, o código já estará preparado para acessar o dado global através de um ponteiro, que será simplesmente corrigido com o endereço do dado global (relocação `GLOB_DAT`) pelo ligador de tempo de execução.

Para o caso do símbolo ser uma função, a diferença é mais acentuada entre a ligação estática e a dinâmica. Como exemplo, considere a figura 3, que ilustra o objeto produzido pela compilação do código C do exemplo da subseção anterior no momento em que o ligador de tempo de compilação resolve seus símbolos com a `glibc`, biblioteca dinâmica que possui

[4]O termo mais adequado aqui é *mover*, uma vez que a localização antiga do dado é abandonada. Mover objetos é um processo não usual para a resolução de relocações. Normalmente, tudo que é feito é a atribuição de endereços a determinadas localizações. Note ainda que relocação de cópia é um recurso bastante limitado. Por exemplo, uma biblioteca dinâmica não deve solicitar essa relocação (apesar de ser possível), uma vez que se o executável também solicitar, o dado terá endereço indeterminado.

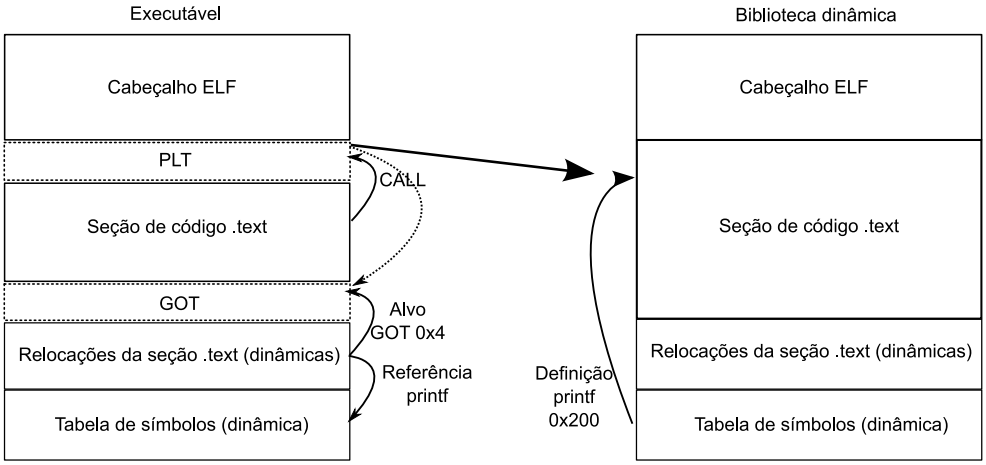

Figura 4: Ligação concretizada de um objeto contra uma biblioteca dinâmica.

a definição da função chamada. A instrução de chamada de função `CALL` está na posição `0x334` da seção de código do objeto. Existe uma relocação estática para esta posição, criada pelo montador e de código específico da arquitetura, solicitando que esta instrução seja preenchida com o endereço de um símbolo indefinido de nome `printf`. Já no arquivo da biblioteca dinâmica lida pelo ligador, o símbolo `printf` está definido na posição `0x200`. Entretanto, esta posição é em relação ao endereço base da biblioteca, que é desconhecido neste momento. A solução adotado pelo ligador está ilustrada na figura 4 e envolve duas seções novas: `GOT` (*global offset table*) e `PLT` (*procedure linkage table*). Estas seções aparecem ilustradas como retângulos tracejados, indicando que foram inseridas pelo ligador. Todavia, note que as seções de relocação e tabela de símbolos também são novas.

O conceito da primeira nova seção, `.got`, já foi ilustrado sem que seu nome fosse citado. Trata-se do local onde os ponteiros são buscados pelas referências indiretas para dados globais, e onde *todas* as relocações dinâmicas são aplicadas, inclusive relocações com referência para funções. Isto ocorre porque as seções de código devem ficar inalteradas (viabilizar o compartilhamento interprocesso de páginas de memória somente leitura). A segunda nova seção, `.plt`, é uma tabela com código. Cada posição específica carrega um ponteiro armazenado na `.got` e pula para este endereço. Assim, na figura 4, o executável final, dependente da biblioteca dinâmica, é gerado. Uma relocação dinâmica (a ser resolvida pelo ligador de tempo de execução) é emitida com alvo para a posição `0x4` da `GOT`, cujo símbolo de referência é `printf`. Quando o programa for executado e processado pelo carregador e ligador de tempo de execução, as relocações dinâmicas serão resolvidas, de modo que a posição `0x4` da `GOT` agora contém o endereço da função `printf`, `0x200` somado ao endereço base de carregamento da biblioteca dinâmica glibc. A instrução `CALL` original, cujo objetivo era acessar a função `printf`, foi alterada para acessar uma entrada da `PLT` que, por sua vez, irá carregar o endereço contido em `GOT + 0x4` (flecha tracejada) e pular para a posição onde a função `printf` foi finalmente carregada (flecha de traço contínuo, de tamanho maior).

Neste processo descrito, é importante notar que a tabela `PLT` é dependente do proces-

sador alvo, pois é escrita diretamente com instruções deste processador. Para resolver esse problema, a linguagem ArchC foi estendida para que o projetista do modelo possa descrever como as entradas `PLT` de sua arquitetura específica devem ser. Esta descrição é feita em um arquivo separado, que contém outras informações necessárias para a síntese do ligador de tempo de compilação. Outras informações incluem o tamanho dos dados envolvidos (ponteiros), bem como a definição de código C que irá indicar como o ligador deve proceder para alterar a entrada `PLT` descrita (que é genérica) para acessar uma posição específica da `GOT`. Este processo é, na verdade, idêntico ao de aplicar uma relocação, e o projeto ArchC já previa meios para descrições de funções de relocação para o ligador estático tradicional.

5.1.2 Versionamento de símbolos

Em bibliotecas dinâmicas ELF, há ainda o conceito de versão de um símbolo. Trata-se de um sufixo aplicado ao nome do símbolo usado para possibilitar a evolução das bibliotecas sem que haja conflito com programas que foram ligados com versões anteriores da biblioteca. Tais programas possivelmente dependem de comportamentos das funções da biblioteca que podem ter sido alterados nas versões subsequentes.

Para que uma biblioteca resolva o problema de compatibilidade com programas ligados a versões anteriores, ao invés de precisar mudar o nome de uma função ou variável cujo comportamento (no caso de funções) ou organização (no caso de variáveis) foram alterados, basta definir uma nova *string de versão*. Note que isto também soluciona o problema de executáveis ligados a uma versão recente da biblioteca, mas que são carregados em um ambiente com uma versão antiga da biblioteca. O carregador e ligador de tempo de execução deve procurar pela versão correta do símbolo importado pelo executável. Por convenção, um símbolo ELF versionado é representado por seu nome, dois caracteres `@` e a string de versão. O programa em C do exemplo das subseções anteriores provavelmente realizaria a importação do símbolo `printf@@GLIBC2.0`, pois glibc é uma biblioteca dinâmica versionada. O ligador de tempo de compilação sintetizado e o carregador e ligador de tempo de execução deste projeto oferecem suporte a versionamento de símbolos.

5.1.3 Criação de biblioteca dinâmica

Os passos que o ligador de tempo de compilação deve seguir para gerar uma biblioteca dinâmica ELF são:

1. Código deve ser PIC (ou seja, compilador corretamente configurado para gerar este tipo de código).
2. Criar GOT para dados globais ou importados.
3. Criar PLT para acessar funções globais ou importadas.
4. Criar tabela de símbolos e sua respectiva tabela de strings, bem como sua tabela de espalhamento (*hashtable*).
5. Escolher nome da biblioteca e interpretador (carregador).

Vale lembrar que no terceiro item é citada a necessidade de uma tabela de espalhamento, que ainda não foi comentado neste relatório. Trata-se de uma estrutura de dados para acesso rápido a um símbolo pelo seu nome. Isto é necessário quando um símbolo está sendo resolvido, situação na qual toda a informação disponível é o nome do símbolo. No exemplo anterior, o símbolo `printf` teve que ser buscado na lista de símbolos da biblioteca dinâmica glibc. Para isso, uma função de computação de resumo (*hash*) pré-definida pelo padrão ELF foi aplicada na cadeia de caracteres que define o nome do símbolo, de modo que uma posição na *hashtable* foi inferida, alcançando o símbolo `printf`, descobrindo que se trata de uma função e o seu endereço. Este processo é necessário pois é realizado toda vez que o executável é carregado, de modo que é importante que seja o mais rápido possível para minimizar as desvantagens do sistema de ligação dinâmica (degradação do desempenho).

Outras informações sobre um símbolo podem ser inclusas na tabela. Um símbolo exportado pode ser *weak* (jargão ELF), caso em que sua definição será utilizada apenas se outro símbolo com o mesmo nome não existir. Se um símbolo importado for *weak*, a ausência da definição deste símbolo em uma biblioteca não é considerada um erro (um exemplo comum é a importação do símbolo `gmon_start` com a propriedade *weak*, que irá acionar o *profiler* GNU gprof para analisar o desempenho do aplicativo apenas se a biblioteca *gprof* estiver carregada). Ainda, um símbolo possui tipo, fato que já foi utilizado nos exemplos anteriores (para descobrir que `printf` é função). Os mais utilizados são função e objeto. É responsabilidade do montador caracterizar o tipo correto dos símbolos que são informados em seu arquivo objeto produzido.

5.1.4 A biblioteca dinâmica ELF descrita por suas seções

Descreveremos a seguir as seções de uma biblioteca dinâmica ELF importantes para as atividades de carregamento e ligação. A lista a seguir relaciona cada nome de seção, seu tipo e sua descrição. Note que todo nome de seção ELF é prefixado por um ponto.

1. `.hash` - Tipo `HASH` - Contém a estrutura de tabela de espalhamento descrita na seção 5.1.3, usada para buscas do índice de um elemento da tabela dinâmica de símbolos rapidamente. Sua integridade é essencial para a correta execução do carregador e ligador de tempo de execução.

2. `.dynsym` - Tipo `DYNSYM` - Contém a tabela dinâmica de símbolos, com todas as informações que um símbolo possui. Nomes, de uma maneira geral em um arquivo ELF, são descritos por um índice para a tabela de strings (neste caso, a tabela dinâmica de strings).

3. `.dynstr` - Tipo `DYNSTR` - A tabela dinâmica de strings armazena todos os nomes de símbolos, versões de símbolos, bibliotecas importadas e o próprio nome da biblioteca. Se alguma estrutura ELF precisa fazer referência a um nome, um índice para esta tabela é usado.

4. `.gnu.version` - Tipo `VERSYM` - Conforme discutido na seção 5.1.2, contém informações relativas a versionamento de símbolos. Esta seção especificamente contém uma tabela

com o mesmo número de elementos da tabela de símbolos. Cada posição nesta tabela apresenta a versão do símbolo de índice correspondente na tabela de símbolos. Se o símbolo não possui versão, o código 0 é usado para representar um símbolo local, ao passo que o código 1 é usado para representar um símbolo global. Caso contrário, um índice da tabela do próximo item é usado.

5. `.gnu.version_d` - Tipo `VERDEF` - Contém as versões dos símbolos definidos nesta biblioteca.

6. `.gnu.version_r` - Tipo `VERNEED` - Semelhante à tabela anterior, mas utilizada em arquivos que importam símbolos versionados. Contém uma tabela de versões necessárias para os símbolos importados.

7. `.rel.dyn` - Tipo `REL` - Apresenta as relocações dinâmicas, conforme discutido na seção 5.1. Também pode se chamar `.rela.dyn`, caso em que o tipo é `RELA`. Este é o caso em que a estrutura que representa uma relocação contém um campo a mais, o do adendo. O seu uso ou não é especificado pela ABI do processador.

8. `.rel.plt` - Tipo `REL` - Esta seção é opcional (depende da ABI específica), mas normalmente é usada. Contém relocações específicas que irão ajustar endereços de funções importadas, acessadas pelo mecanismo descrito na seção 5.1 que utiliza um nível de indireção introduzido pela seção `PLT`. O fato das relocações relativas à PLT estarem segregadas ajuda na implementação do mecanismo de *lazy-binding*.

9. `.plt` - Tipo `PROGBITS` - O tipo indica que esta seção já é carregada como código executável. Contém as instruções de salto, caracterizando a tabela PLT, que acessam funções definidas em objetos dinâmicos. As instruções buscam o endereço da função acessando a tabela `GOT`, preenchida com os endereços corretos graças às seções `.rel.dyn` ou `.rel.plt`, que solicitam relocações dinâmicas com referência para as funções utilizadas. Para cada função, haverá uma entrada nesta seção. Cada entrada ocupa aproximadamente o espaço necessário para 4 instruções do processador alvo. Em nosso sistema de ligação, o projetista do modelo do processador em ArchC precisa descrever manualmente a PLT.

10. `.text` - Tipo `PROGBITS` - A tradicional[5] seção com a porção de código executável deste objeto gerada pelo montador.

11. `.data` - Tipo `PROGBITS` - Contém dados globais do programa, criada pelo montador. Normalmente, aqui já se dá início a uma transição de páginas em que as seções são carregadas. Mais especificamente, os objetos ELF são carregados em segmentos, de forma que há um mapeamento de seções para segmentos. As seções a partir de `.data`, incluindo esta, são carregadas em segmentos leitura e escrita, enquanto as anteriores somente leitura. Esta abordagem viabiliza o compartilhamento de páginas. Mesmo nas páginas com suporte a leitura e escrita, o compartilhamento é feito com o mapeamento COW (*copy-on-write*).

[5]Este nome de seção para representar o código em linguagem de máquina é usado desde o formato a.out [10].

12. `.dynamic` - Tipo `DYNAMIC` - Esta seção contém a tabela mais importante para o processo de ligação dinâmica. O endereço desta tabela deve ser fornecido ao carregador, que, através dela, terá acesso a todas as informações necessárias para carregar e ligar dinamicamente a biblioteca. Sua estrutura é discutida com mais detalhes na seção 5.1.5.

13. `.got` - Tipo `PROGBITS` - Contém uma série de ponteiros para diversos dados cujo endereço só é conhecido no momento em que as bibliotecas são carregadas (tempo de execução). O carregador e ligador de tempo de execução irá atuar resolvendo endereços de símbolos e preenchendo esta tabela GOT, orientado por meio das relocações dinâmicas da seção `.rel.dyn` e `.rel.plt`. Note que esta última não contém relocações para a tabela PLT, mas sim para a tabela GOT. O que ocorre é que a tabela PLT é composta de diversas entradas somente leitura, com código em linguagem de máquina que irá acessar seu respectivo ponteiro na tabela GOT.

14. `.bss` - Tipo `NOBITS` - Outra seção criada pelo montador que também existe em qualquer executável estático. Não possui conteúdo (por isso recebe a atribuição de tipo `NOBITS`), mas indica a presença de dados globais não inicializados. Na prática, o carregador apenas irá levar em consideração o tamanho da seção, que indica a quantidade de memória a ser reservada para esses dados globais.

5.1.5 A tabela ELF DYNAMIC

Como apresentado na seção 5.1.4, a tabela `DYNAMIC`, inserida na seção `.dynamic` de um arquivo objeto (dinâmico) ELF, possui dados necessários para o funcionamento do carregador e ligador de tempo de execução. Esta tabela é organizada como um vetor de pares marcador(*tag*)-dado. O marcador identifica os diversos tipos de entrada, que são expostos na lista a seguir. Apenas os tipos mais comuns são listados.

- `NULL` - Marca o fim do vetor (deve ser a última entrada).
- `NEEDED` - Contém um índice para a tabela dinâmica de strings, cujo nome indica uma biblioteca dinâmica necessária para a resolução dos símbolos importados.
- `HASH` - Indica o endereço onde está carregada a tabela de espalhamento discutida na seção 5.1.3.
- `STRTAB` - Indica o endereço onde está carregada a tabela dinâmica de strings.
- `SYMTAB` - Indica o endereço onde está carregada a tabela dinâmica de símbolos.
- `STRSZ` - Tamanho, em bytes, da tabela dinâmica de strings.
- `SYMENT` - Tamanho, em bytes, de uma entrada da tabela de símbolos.
- `PLTGOT` - Normalmente indica o endereço onde está carregada a tabela GOT, mas o valor depende da ABI do processador.

- `PLTRELSZ` - Tamanho, em bytes, de uma entrada da lista de relocações dinâmicas.
- `PLTREL` - Indica o tipo da relocação dinâmica (REL ou RELA), conforme discutido na seção 5.1.
- `JMPREL` - Indica o endereço das relocações dinâmicas relacionadas a uma entrada da tabela PLT.
- `REL` - Indica o endereço das relocações dinâmicas.
- `RELSZ` - Tamanho, em bytes, da lista de relocações dinâmicas.
- `RELENT` - Tamanho, em bytes, de uma entrada da lista de relocações dinâmicas.
- `VERNEED` - Indica o endereço da tabela de versão de símbolos importados, conforme discutido na seção 5.1.4. O análogo para símbolos definidos é `VERDEF`.
- `VERNEEDNUM` - Número de elementos da tabela de versão de símbolos importados. O análogo para símbolos definidos é `VERDEFNUM`.
- `VERSYM` - Indica o endereço da tabela de versão de símbolos, conforme discutido na seção 5.1.4.

5.2 Gerador de montadores Acbingen

Para viabilizar o sistema de ligação dinâmica, é necessário não só criar o sintetizador de ligador de tempo de compilação (tarefa realizada com a geração de um backend libbfd específico para a arquitetura descrita com ArchC), mas também modificar o montador sintetizado por `acbingen`. Isto não é necessário para objetos comuns, que irão apenas utilizar uma biblioteca dinâmica, mas para a geração de uma biblioteca dinâmica. Como comentado, o código precisa ser independente de posição. O código PIC gerado pelo compilador contém algumas anotações nos símbolos utilizados para sinalizar ao montador que estes símbolos não são ordinários, mas fazem parte de uma biblioteca dinâmica e irão requerer tratamento especial pelo ligador. Sendo assim, o montador deve reconhecer tais indicações nos símbolos.

Especificamente, deve ser adicionado um *parser* para verificar pela presença das strings "(got)", "(gotoff)" e "(plt)" após qualquer símbolo, e então criar uma relocação do tipo apropriado. Isto implica atuar no parser de qualquer operando do tipo `%expr` [6] em ArchC, bem como em operandos de diretivas como `.word`, que normalmente são usadas para indicar que na posição da diretiva deverá ser inserido o endereço de um símbolo, cujo nome aparece após a diretiva.

O `acbingen` aborda o problema dos códigos de relocação atribuindo automaticamente um código para cada instrução descrita no modelo. Assim, uma instrução `add` que precise de relocação em um de seus operandos terá um código de relocação, ao passo que outra instrução `sub` terá outro. É preciso então gerar códigos adicionais, para indicar que tais operandos estão referenciando símbolos especiais, dinâmicos (residentes na `PLT` ou `GOT`).

[6] Usado pelo projetista para indicar que o operando de uma determinada instrução de linguagem de montagem é uma expressão, que por sua vez pode conter símbolos.

Note que o ligador sintetizado é claramente incompatível com outros ligadores da mesma arquitetura, uma vez que não existe mecanismo para informar especificamente qual o código de relocação que a ABI da arquitetura alvo definiu para uma determinada relocação. Este problema foi superado com a criação de uma ferramenta específica capaz de converter tais códigos, de posse de um mapa de conversão. A criação desta ferramenta é discutida na seção 6.5.

5.3 Gerador de backend libbfd

Conforme o diagrama da figura 1 ilustra, o nosso sintetizador de ligador de tempo de compilação consiste na verdade em um gerador de *backend* para a biblioteca BFD (Binary File Descriptor) [4]. O código do backend foi produzido neste trabalho, de modo que alguns trechos dependentes do processador alvo foram sinalizados com *macros*. Tais anotações são substituídas pelo gerador pelas informações específicas do processador ou arquitetura alvo, como, por exemplo, uma estrutura descrevendo como a `PLT` deve ser preenchida.

5.3.1 Libbfd

A `libbfd` foi criada, no âmbito do projeto de software livre GNU, para prover uma camada de abstração com capacidade para manipular uma variedade de pares "*arquivo objeto - arquitetura alvo*". Atualmente, é parte do projeto GNU binutils. As ferramentas binutils constituem os principais usuários da biblioteca, uma vez que a idéia central é prover um conjunto de programas versados em mais de um tipo de arquivo objeto, provendo completude para a cadeia de ferramentas de compilação gcc.

Um projeto que utilize a biblioteca BFD irá fazê-lo realizando referências a uma estrutura chamada `bfd` e acessando dados desta. O modelo conceitual estabelece a estrutura `bfd` como uma espécie de interface abstrata que irá se situar entre o aplicativo e as camadas de backend, que de fato realizam o trabalho de manipulação do arquivo objeto do tipo escolhido. Isso garante o nível de abstração pretendido com a criação da biblioteca, necessário para isolar detalhes de cada formato de arquivo objeto. Contudo, um compromisso importante ocorre com esta decisão de projeto. Muitas vezes os aplicativos (como montadores e ligadores) não desejam abstração, uma vez que precisam gerenciar diretamente as particularidades de cada formato de objeto. Na tentativa de ganhar flexibilidade perdida com o uso da abstração, a biblioteca BFD fornece meios para que aplicações acessem mecanismos específicos de formato, o que torna o projeto da biblioteca, nesses casos, confuso e inconsistente.

As estruturas que representam as principais abstrações utilizadas na biblioteca são `bfd`, como já dito, que representa um arquivo aberto pela biblioteca (pertencente a uma de 3 categorias, `object`, `archive` ou `core dump`), `section`, que contém os dados das seções nas quais um arquivo objeto é dividido e de fato o conteúdo do arquivo objeto, `symbol`, que representa um nome associado a uma localização mas não se restringe a isso e finalmente `reloc`, que se manifesta junto a uma seção e indica uma referência a algum símbolo que deve sofrer ajustes.

Uma `bfd` do tipo `archive` contém uma estrutura do tipo lista ligada envolvendo um conjunto de `bfd`s, correspondendo a cada arquivo objeto contido em uma biblioteca. `Sections` são armazenadas também em uma estrutura do tipo lista ligada, cujo ponteiro para o nó cabe a está associado a uma bfd. `Symbol`s são agrupados em um vetor, assim como `reloc`s.

Todo programa usuário de libbfd deve iniciar seu uso com a chamada a função `bfd_init()`. Em seguida, um arquivo de interesse deve ser aberto e mapeado em memória (através da estrutura de dados bfd) com uma função `bfd_openr()` para leitura ou `bfd_openw()` se o objetivo é escrita. Note que o fluxo de operação com a libbfd não permite que o usuário leia e grave no mesmo arquivo. Caso modificações sejam necessárias, o arquivo deve ser lido, processado e o resultado salvo em um arquivo diferente. De fato, esta restrição no modo de operação é sempre respeitada pelos ligadores.

Atualmente, a função `bfd_init()` não contém código e a convenção de chamada para inicialização da biblioteca é uma decisão de projeto da biblioteca para um possível uso futuro. As chamadas para abertura de arquivo traduzem-se em chamadas a funções de manipulação de arquivo pela biblioteca de entrada e saída padrão do C, como `fopen()`, bem como preenchimento dos campos da estrutura `bfd` com valores padrões. Nenhum dado ainda é lido. O usuário deve fazer uso de `bfd_check_format()` ou `bfd_check_format_matches()`, que então irão verificar o formato do arquivo e acionar o backend responsável para preenchimento das estruturas `bfd`, `section`, `symbol` e `reloc`, bem como todos os campos relacionados. Com a `libbfd`, ainda é possível definir a abertura de um arquivo com um formato genérico, padrão. Então a chamada a `bfd_check_format()` ou similar irá resultar no uso de um algoritmo para seleção de backend (cada backend é consultado e pelo menos um deve reconhecer o arquivo, de modo que então este formato é guardado e as estruturas `bfd` preenchidas de acordo).

A partir deste ponto, a estrutura `bfd` deverá contar com o campo `xvec` (que denota um vetor alvo, um grande conjunto de informações sobre o formato do arquivo objeto aberto e o backend responsável pela manipulação direta deste) preenchido corretamente. Este vetor contém inúmeros ponteiros para funções do backend que responde por funções deste formato específico, determinado conforme mostrado anteriormente. A maioria das funções bfd que operam o arquivo irão finalmente se traduzir diretamente em uma macro `BFD_SEND`. Tal macro sempre recebe um ponteiro para uma instância da estrutura bfd como argumento. Deverá então dereferenciar o ponteiro em questão, acessar seu campo `xvec` e por sua vez acessar o ponteiro para função solicitado (também como argumento), que aponta então para um ponto de entrada no backend (encarregado de atender a solicitação).

Para cada conceito chave da abstração utilizada pela libbfd (seções, símbolos, relocações) existe um conjunto de funções utilizadas para sua manipulação. A tarefa do backend, nestes casos, é extrair os dados do formato de arquivo objeto específico e traduzi-los para a forma canônica de representação de estruturas da biblioteca bfd, armazenada em memória, independente de formato. Com isto, a biblioteca permite a leitura de um ou mais arquivos objeto, bem como a produção de um arquivo executável final, cada um em um formato diferente. Suponha, por exemplo, um objeto `a.out` [10] ligado a uma biblioteca de objetos `COFF` [10] no esforço de produzir um executável `ELF`.

Entretanto, é importante atentar ao fato de que cada formato de arquivo objeto armazena um conjunto de informações próprio. Normalmente, este conjunto de informação é

bastante similar entre diferentes formatos (sobreposição), uma vez que o objetivo é o mesmo (prover meios para que um ligador alcance seus objetivos de, em uma visão geral, gerência de trechos de programa para construção de um programa final, completo). Contudo, sempre existem particularidades como informações disponíveis apenas em um formato, ou restrições específicas de um formato. Podemos enxergar então a representação canônica bfd como um outro tipo de formato, mais abrangente, mas que ainda assim não constitui uma união de todos os formatos e pode descartar informações no processo de leitura de um arquivo objeto de formato específico. Assim, um arquivo objeto final produzido terá as informações que aparecem na intersecção entre os formatos de entrada, o formato canônico bfd e o formato de saída. Isto é, se algum dos formatos deste processo não armazenar determinado tipo de informação de um arquivo de entrada, esta informação será perdida.

Uma ilustração deste conceito é fácil de enxergar quando se usa um formato de saída bastante limitado, como `a.out`. Este formato só admite 3 seções, com os nomes ".text", ".data" e ".bss". Se no processo de ligação, utilizando diferentes formatos de entrada, uma seção que não se encaixe em uma destas três for criada, esta informação será perdida.

5.3.2 Backends

Os backends são livres para implementar suas funções conforme sua conveniência. Por exemplo, muitos backends adiam a leitura do conteúdo efetivo do arquivo objeto (o conteúdo das seções) até que uma função do tipo `bfd_get_section_contents()` seja chamada.

5.3.3 Atividade de ligação

Um ponto interessante sobre a `libbfd` é sua independência para as atividades de ligação. Isto significa que um programa que atue como um ligador, como GNU ld, limita-se apenas a processar uma linguagem de *linker script* e determinar os parâmetros de ligação. A mescla de seções, busca por resolver símbolos indefinidos, gerenciamento de espaço e qualquer outra atividade relacionada a ligação é responsabilidade da `libbfd` e é oferecida ao programa usuário em 3 etapas. A primeira é a criação de uma tabela de espalhamento de símbolos utilizada pelas demais funções. A segunda adiciona de fato os símbolos de um objeto a esta tabela de espalhamento. A terceira realiza de fato a ligação. As rotinas podem ser auxiliadas com uma estrutura do tipo `bfd_link_info` que contém orientações e parâmetros para a ligação.

Antes de chamar `bfd_final_link()`, o usuário (o programa que utiliza a libbfd) deve preparar as seções a serem ligadas. Para isso, a estrutura `bfd_link_info`, em seu campo `input_bfds`, deve indicar todos os arquivos objetos utilizados como entrada neste processo. A lista é continuada utilizando o campo `link_next` da estrutura `bfd`. Ainda, o arquivo objeto de saída (que futuramente irá conter todas as entradas ligadas) deverá ser um `bfd` já inicializado com suas seções a serem criadas. Cada seção deve conter um ponteiro para o nó cabeça de uma lista ligada de estruturas `link_order`. Tal estrutura possui diversos tipos, sendo que o mais importante deles é `bfd_indirect_link_order`. Este tipo aponta para uma seção de um arquivo objeto de entrada que deverá ser incluída nesta seção do arquivo objeto de saída. Outros tipos também podem indicar relocações novas a serem inseridas na

seção, bem como indicações de como o conteúdo desta seção de saída será criado em função do conteúdo das seções dos arquivos de entrada. Quando `bfd_close()` for chamado pelo usuário, o ligador bfd ainda irá chamar a função de backend `write_object_contents()` para criar o arquivo final em disco.

As seções pertencentes a um `bfd` de entrada (determinado por `input_bfds` em `bfd_link_info`) possuem os campos `output_section` e `output_offset` preenchidos, relacionando com seu destino ao final da ligação. Tais informações são frequentemente usadas para cálculo de endereços, no momento da aplicação de uma relocação, em conjunto com o campo `vma` (endereço virtual) da seção de saída (sendo criada) que deve indicar seu endereço de carregamento.

5.3.4 Características do backend sintetizado

Para determinar se uma relocação em um objeto regular (não dinâmico) contra um objeto dinâmico deve usar PLT, o algoritmo atual consulta se o símbolo é `STT_FUNC` (tipo de símbolo ELF que caracteriza uma função). Backends específicos, entretanto, podem inferir se o alvo é uma função apenas pelo tipo de relocação, sabendo se a instrução é, por exemplo, *jump/call* (infere-se que o símbolo referenciado é uma função) ou *load* (infere-se que o símbolo referenciado é um dado). Portanto, o correto funcionamento do ligador sintetizado, no momento, depende de que os símbolos sejam corretamente definidos como `STT_FUNC` quando oportuno (isto depende do compilador, que normalmente emite diretivas especiais informando que um símbolo indica um procedimento ou função).

Para fins de uma estimativa, ainda que grosseira, do trabalho realizado, o arquivo principal de backend em código C possui aproximadamente 2000 linhas de código. As funções do backend ELF definidas são as mesmas que as de diversos outros backends (como MIPS, Intel e ARM), indicando que o projeto é capaz de sintetizar um ligador com as mesmas funcionalidades, e são listadas na tabela 1.

A organização de um backend libbfd é composta por dois módulos, a saber, um responsável pela manipulação do arquivo objeto e outro pela manipulação de atividades específicas de um processador alvo, respeitando o protocolo de ABI definido. Existe ainda um conjunto genérico de funções, independente de arquivo objeto e processador alvo, geralmente marcadas com a palavra "`generic`" em seu nome. Em nosso caso, o formato de arquivo objeto ELF é utilizado e, portanto, funções do backend ELF são inclusas. Estas funções redefinem o comportamento de boa parte das funções genéricas libbfd, no esforço de produzir ligadores que aproveitem ao máximo os recursos ELF. Ao backend específico de processador, aquele sintetizado por nossa ferramenta, restam as tarefas de relocação de seções (uma vez que cada ABI de processador define como interpretar os códigos de relocação) e criação de seções extras (somando-as ao conjunto de seções padrões já criadas pelo backend ELF). Também é possível definir diversas funções gancho, chamadas em pontos chave do processo de ligação executado pelo backend ELF, para que o backend específico de processador tenha a oportunidade de manipular dados e customizar o algoritmo de ligação de modo a concordar com suas exigências.

Muito da manipulação de símbolos e seções necessária para a criação de bibliotecas dinâmicas e executáveis ligados a bibliotecas dinâmicas é também deixado ao backend es-

	Nome de função canônico backend bfd	**Nome de função particular do backend sintetizado**
1.	elf_backend_gc_mark_hook	elf_archc_gc_mark_hook
2.	elf_backend_gc_sweep_hook	elf_archc_gc_sweep_hook
3.	elf_backend_relocate_section	elf_archc_relocate_section
4.	elf_backend_create_dynamic_sections	elf_archc_create_dynamic_sections
5.	elf_backend_finish_dynamic_symbol	elf_archc_finish_dynamic_symbol
6.	elf_backend_finish_dynamic_sections	elf_archc_finish_dynamic_sections
7.	elf_backend_size_dynamic_sections	elf_archc_size_dynamic_sections
8.	elf_backend_adjust_dynamic_symbol	elf_archc_adjust_dynamic_symbol
9.	elf_backend_check_relocs	elf_archc_check_relocs
10.	elf_backend_always_size_sections	elf_archc_always_size_sections
11.	elf_backend_reloc_type_class	elf_archc_reloc_type_class
12.	elf_backend_copy_indirect_symbol	elf_archc_copy_indirect_symbol
13.	elf_backend_additional_program_headers	elf_archc_additional_program_headers

Tabela 1: Funções implementadas pelo backend bfd sintetizado

pecífico de processador. É imperativo que as relocações sejam analisadas (função 9 da tabela 1) para verificar a necessidade de criação das seções `.got` e `.plt`. Por sua vez, a criação destas seções é realizada por chamada à função 4. Uma série de outras funções ajudam no processo de determinação do tamanho das seções de auxílio à ligação dinâmica (função 7), aplicam relocações em seções levando em consideração símbolos com referência dinâmica (função 3), isto é, que na verdade não serão inclusas no arquivo final, como no caso estático, e auxiliam o decorrer deste processo (funções 5, 6 e 8) até que o arquivo objeto final (seja biblioteca ou executável) seja corretamente produzido. Note que este backend é que irá alocar entradas nas tabelas GOT e PLT, bem como gerar relocações dinâmicas para que o carregador e ligador de tempo de execução possa resolver.

6 Carregador e ligador de tempo de execução

Nesta seção iremos discutir a implementação da peça do sistema de ligação dinâmica que atua mais próxima do simulador. Em contraste, o módulo anterior foi implementado junto ao `acbingen`, que é encarregado de gerar ferramentas binárias. A real necessidade da implementação de um carregador e ligador dinâmico é discutível, pois poderia ser substituído pelo próprio carregador `ld.so` fornecido com a glibc para a arquitetura alvo (algumas modificações seriam necessárias no simulador, a saber, carregar o módulo dinâmico inicial `ld.so` e transferir um vetor de informações pela pilha). A decisão da criação deste módulo, contudo, apoiou-se no fato de que apenas alguns processadores são suportados pela GNU libc, ficando evidente que uma nova arquitetura descrita com ArchC ficaria sem o sistema de ligação dinâmica.

Ainda, o projeto de carregador e ligador desenvolvido não foi em nenhum aspecto baseado no já existente *runtime loader* `ld.so` da biblioteca glibc, consistindo em uma abordagem original ao problema, largamente independente de arquitetura alvo (alguns aspectos, como

endianess e tamanho de palavra, são obtidos do pré-processador ArchC) e que teve sua funcionalidade verificada com testes mostrados na seção 7 para o processador ARM.

6.1 Conceitos

Antes de iniciar a discussão sobre a implementação do carregador e ligador de tempo de execução, esta seção resgata alguns conceitos importantes. Pode-se delinear as características gerais das biblioteca dinâmicas em ELF como a seguir:

- Utilizam código PIC para permitir que sejam carregados em qualquer endereço sem alteração, permitindo compartilhamento;
- Seção especial GOT criada para permitir o código PIC;
- Uso de PLT pelos clientes de bibliotecas dinâmicas, para permitir *lazy-binding* e o mecanismo de chamadas de funções em bibliotecas dinâmicas;
- Seção (*dynsym*) contendo uma tabela de símbolos contendo os símbolos importados e exportados;
- Seções (*dynstr* e *hash*) que contêm os nomes para consulta aos símbolos;
- Seção (*dynamic*) contendo ponteiros que localizam informações cruciais para a execução do carregador e ligador de tempo de execução;
- Seção (*interp*) preenchida com o endereço do carregador e ligador de tempo de execução a ser utilizado.

Os passos para carregar um programa que utiliza bibliotecas ligadas dinamicamente são apresentados a seguir. Consiste em uma visão geral do algoritmo utilizado por um carregador ELF ordinário como `ld.so` da glibc, e possui algumas diferenças em relação àquele implementado em nosso trabalho que serão comentadas a seguir.

1. Mapear as páginas do programa como normalmente.
2. Mapear as páginas do ligador dinâmico (uma biblioteca compartilhada em si) em um endereço conveniente.
3. Passa a execução ao ligador dinâmico junto com um vetor de informações auxiliares. Este vetor inclui:
   - Informações da localização do program header, tamanho de cada entrada no cabeçalho e número, descrevendo os segmentos do arquivo do programa carregado na memória.
   - Endereço de início do programa carregado.
   - Endereço base de onde o ligador dinâmico foi mapeado.

4. O carregador localiza sua GOT, cuja primeira entrada aponta para o segmento DYNAMIC.

5. Através deste segmento, localiza-se as informações para relocação, e o ligador resolve as entradas e referências a rotinas externas.

6. É criada uma lista de símbolos conjunta do programa e do ligador, a qual serão adicionados símbolos a medida que mais bibliotecas forem sendo carregadas.

7. O cabeçalho do programa contém um ponteiro direto para seu segmento DYNAMIC, que por sua vez possui um ponteiro para a tabela de strings com entradas de marcadores NEEDED.

8. Com os nomes das bibliotecas necessárias conhecidos, estas são procuradas no sistema na seguinte ordem:

   - Um caminho descrito no próprio segmento `DYNAMIC`
   - Na variável de ambiente `LD_LIBRARY_PATH`
   - Em caminhos contidos no arquivo `/etc/ld.so.conf`
   - Em `/usr/lib`

9. Para cada biblioteca carregada, sua tabela GOT é preenchida e relocações para os dados são realizadas.

10. Se existem seções .init, estas são executadas (apenas para as bibliotecas, o código do programa é responsável por executar seu código de inicialização).

11. É transferido controle ao ponto de entrada[7] do programa.

O código do ligador dinâmico fica mapeado na memória, pois ele ainda pode ser chamado em tempo de execução por funções como `dlopen()` ou `dlsym()` para carregar novas bibliotecas ou resolver um símbolo, respectivamente. No caso de nosso carregador e ligador de tempo de execução, estas funcionalidades não são oferecidas, pois ele não é carregado como um aplicativo do processador alvo, mas atua junto ao simulador, carregando e resolvendo todas as relocações imediatamente (não faz *lazy-binding*). Como o objetivo do simulador ArchC, em geral, não é carregar sistemas completos, mas sim apenas um aplicativo, o simulador assume muitas tarefas que seriam do sistema operacional, como atender a chamadas de sistema. Sendo assim, esta abordagem para o sistema de ligação dinâmica se assemelha ao modo do sistema Windows, em que o próprio sistema operacional resolve as relocações [10], não utilizando código em espaço de usuário como se propõe no ELF e a idéia de um interpretador (geralmente `ld.so` da glibc) que é mapeado no processo.

Os caminhos de procura por bibliotecas, no caso do nosso carregador e ligador de tempo de execução implementado, não são obtidos da mesma forma como exposto no algoritmo ordinário. Se o fizéssemos, estaríamos carregando bibliotecas dinâmicas do computador

[7] Entry point

hospedeiro que está executando o simulador ArchC, e não aquelas produzidas para o processador simulado. Para abordar este aspecto, foi definida uma variável de ambiente `AC_LIBRARY_PATH` que deve conter os caminhos de busca para as bibliotecas.

As funções de inicialização e finalização das seções `.init` e `.fini`, respectivamente, são executadas com o auxílio de uma chamada de sistema especial atendida pelo simulador ArchC. Como o nosso carregador não tem acesso direto ao fluxo de instruções executadas pelo simulador, o endereço da instrução inicial a ser executada é alterado para uma função de inicialização requisitada. Esta função irá retornar utilizando as convenções da ABI do processador simulado. Por esse motivo, o endereço de retorno é configurado para um endereço especial, atendido por uma chamada de sistema[8] que irá executar as próximas funções de inicialização, até que o próprio aplicativo final seja chamado e a simulação prossiga. Estratégia análoga é usada para as funções de finalização.

Finalmente, não existe passagem de vetor de informações no caso implementado por este projeto, uma vez que foi implementado como um módulo do simulador, já contendo as informações do vetor para a atividade de carregamento e ligação.

6.2 ArchC runtime loader

Como este módulo do projeto foi produzido em linguagem orientada a objetos (recurso não disponível no módulo anterior pois a libbfd é em C), podemos apresentar um diagrama de classes UML na figura 5 expondo todas as classes utilizadas neste módulo, batizado pelo acrônimo `acrtld` (ArchC runtime loader). O diagrama de classes foi gerado automaticamente a partir do cabeçalho das classes, e omite os atributos e assinatura completa dos métodos (apresentando apenas o nome) por motivos de espaço. O objetivo é demonstrar a relação entre classes e a funcionalidade de cada uma.

A lista a seguir discrimina cada classe separadamente, explicando sua funcionalidade.

- `ac_rtld`: Trata-se do *façade* do módulo, abstraindo o objetivo geral do carregador e ligador dinâmico em funções como `loadnlink_all()`, que inicia o algoritmo de carregamento semelhante ao descrito na seção 6.1.

- `ac_rtld_config`: Módulo capaz de ler um mapa de conversão de códigos de relocação e realizar a conversão em tempo se execução. Crucial para a capacidade de carregar bibliotecas produzidas por outros ligadores, que possuem códigos de relocação definidos pela ABI do processador e até então desconhecidos pelo carregador genérico.

- `memmap`: Gerencia um mapa com informações de quais regiões da memória estão ocupadas (com bibliotecas carregadas). Seu propósito inicial era de ajudar na escolha de endereços para carregar as bibliotecas, mas foi estendido para auxiliar no atendimento a chamadas de sistema `mmap()`, conforme será descrito na seção 6.4.

- `symbolwrapper`: Ao invés de ler um símbolo diretamente do arquivo ELF, os módulos instanciam esta classe para implementar uma abordagem independente de arquitetura. Isto ocorre porque nesta classe conversões de *endianess* são realizadas.

[8] O motivo de um endereço especial ser atendido como uma chamada de sistema oferecida pelo simulador é exposto na seção 6.4 e corresponde à abordagem antiga, conforme será discutido.

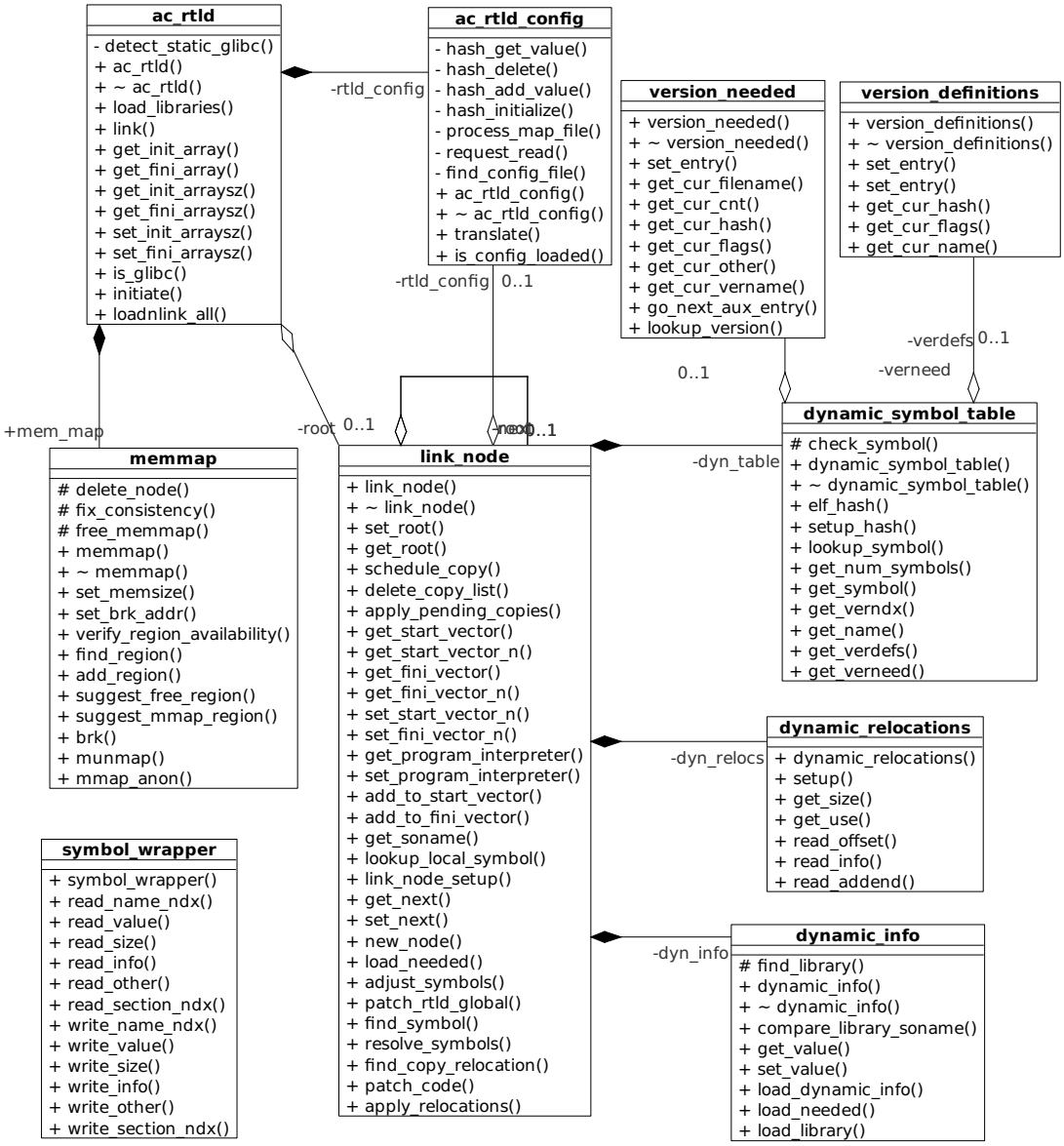

Figura 5: Diagrama de classes UML do projeto do carregador e ligador de tempo de execução ArchC runtime loader.

- `link_node`: A principal classe do projeto, semelhante em abstração à `bfd` da libbfd para ligadores de tempo de compilação. Representa um arquivo ELF aberto (executável ou biblioteca dinâmica) e contém todas as informações que podem ser ex-

traídas deste arquivo, com o auxílio de outras classes agregadas aqui. Trata-se de um "nó" pois pode ser ligado em lista, de tal modo que um ponteiro acessa o próximo arquivo ELF aberto. Em geral, o executável é alcançado com um ponteiro `root`, como um nó cabeça, e as bibliotecas dinâmicas requisitadas por este executável são adicionadas na lista ligada pelo ponteiro `next`. Aqui ficam os métodos de resolução de relocações e ligação, chamados na ordem `link_node_setup()`, `adjust_symbols()`, `resolve_symbols()` e `apply_relocations()`.

- `dynamic_info`: Responsável pela leitura e manipulação da seção ELF `.dynamic`, uma tabela com todas as informações necessárias para a ligação. Nesta tabela, ao ler marcadores `NEEDED`, que indicam dependência de biblioteca dinâmica, esta classe também se responsabiliza por instanciar novos nós `link_node` e carregar as bibliotecas.

- `dynamic_relocations`: Manipulação de estruturas ELF que contém informações sobre as relocações a serem resolvidas pelo carregador.

- `dynamic_symbol_table`: Manipula e acessa a tabela de símbolos dinâmica, implementando a tabela de espalhamento discutida na seção 5.1.3 e o algoritmo de busca por símbolos (Para cada entrada na tabela de espalhamento, compara o nome e versão procurados utilizando o algoritmo 1).

- `version_needed` e `version_definitions`: Agregados à classe da tabela de símbolos, expressam informações para versionamento de símbolos, conforme discutido na seção 5.1.2, um conceito importante e crucial para o funcionamento de bibliotecas dinâmicas modernas, como a glibc.

O carregador e ligador de tempo de execução inicia após carregado o executável inicial. Suas bibliotecas dinâmicas requisitadas são lidas e carregadas em seguida. Os endereços dos símbolos e relocações são todos corrigidos, incrementando-os com o endereço base onde suas respectivas bibliotecas foram carregadas. A localização dos símbolos importados é determinada. Para isso, o nome e versão do símbolo é pesquisado em cada biblioteca carregada (procurando sua definição). A tabela de espalhamento é utilizada e, para cada símbolo ELF candidato, o algoritmo 1 é aplicado para determinar se o candidato é aceito ou recusado. Neste algoritmo, note que as funções `Obter` apenas extraem informações disponíveis na estrutura de símbolo ELF. Existe ainda a condição *weak*, em que o símbolo ELF é fracamente atribuído. Neste caso, a atribuição se consolida apenas se não houver mais nenhuma outra definição para este nome e versão procurados. Para o candidato aceito, seu endereço é informado ao módulo que o importou, de modo que todas as relocações agora estão prontas para serem aplicadas (uma vez conhecidos os endereços de todos os símbolos utilizados). Após aplicadas as relocações, o carregador e ligador encerra sua tarefa.

6.3 Modificações no decodificador do simulador

Um módulo importante do simulador é o decodificador de instruções, que, de posse do mapa de formatos do conjunto de instruções, irá determinar qual é a instrução indicada pelo *program counter*. O decodificador deve buscar bytes da memória de instruções quantas

```
     Entrada: simbolo, nome e versao. simbolo é um objeto resgatado da tabela do arquivo objeto ELF, nome
              é o nome de símbolo que se procura e versao, quando não está indefinido, é o nome da versão
              solicitada para o símbolo
     Saída: Comparando simbolo com nome e versao, julga recusado, aceito ou weak, no caso de ser uma
            definição fraca
 1.1 se ObterValor(simbolo) igual a 0 ou ObterSecao(simbolo) igual a UNDEF então
 1.2 |   retorna recusado;
 1.3 fim
 1.4 se ObterTipo(simbolo) diferente de NOTYPE, OBJECT, FUNC, COMMON então
 1.5 |   retorna recusado;
 1.6 fim
 1.7 se ObterVisibilidade(simbolo) diferente de DEFAULT então
 1.8 |   retorna recusado;
 1.9 fim
1.10 se ObterNome(simbolo) diferente de nome então
1.11 |   retorna recusado;
1.12 fim
1.13 se versao está definido então
1.14 |   se ObterVersao(simbolo) não existe então
1.15 |   |   retorna recusado;
1.16 |   senão
1.17 |   |   se ObterVersao(simbolo) diferente de versao então
1.18 |   |   |   retorna recusado;
1.19 |   |   fim
1.20 |   fim
1.21 senão
1.22 |   se ObterVersao(simbolo) existe então
1.23 |   |   se ObterVersao(simbolo) diferente de LOCAL e GLOBAL então
1.24 |   |   |   retorna weak;
1.25 |   |   fim
1.26 |   fim
1.27 fim
1.28 se ObterBind(simbolo) igual a WEAK então
1.29 |   retorna weak;
1.30 fim
1.31 retorna aceito;
```

Algoritmo 1: Comparação para casamento de símbolo com um nome e versão específicos

```
1. // Formato entendido pelo acbingen (ordem natural, esquerda para direita):
2. ac_format Type_DPI1 =  "%cond:4 %op:3 %func1:4 %s:1 %rn:4 %rd:4 %subop:8
%rm:4";
3. // Formato entendido pelo pelo decodificador antigo (da direita para
esquerda):
4. ac_format Type_DPI1 = "%rm:4 %subop:8 %rd:4 %rn:4 %s:1 %func1:4 %op:3
%cond:4";
```

Figura 6: Exemplo de maneiras conflitantes de se expressar o formato little-endian entre duas ferramentas importantes do Projeto ArchC

vezes for necessário para determinar a instrução. A organização dos bytes da arquitetura (little-endian ou big-endian) é crucial neste momento. Algumas arquiteturas de processador são little-endian, mas o formato de suas instruções é definido, da esquerda para a direita, na ordem em que os bytes são buscados pelo decodificador, como a Intel x86. Já o ARM little-endian especifica um formato que não corresponde à organização real dos bytes das instruções na memória. Este formato deve ser invertido byte a byte, como uma palavra de 32 bits little-endian. Para o decodificador, o mais razoável é que o conjunto de instruções seja descrito na ordem em que os bytes aparecem, como acontece com a arquitetura Intel x86. Contudo, o decodificador do processador ARM espera instruções de tamanho fixo (4 bytes por vez). Dessa forma, o fato de que as instruções são invertidas como dados na memória não gera grandes desafios ao decodificador em hardware do processador ARM (basta buscar 4 bytes e invertê-los). O gerador de simuladores `acsim` apresentava uma maneira confusa de interpretar o formato das instruções para modelos little-endian (em que se encaixa o modelo ARM utilizado como referência para validar este projeto). A saber, o formato das instruções little-endian eram interpretados de forma a invertê-lo, como na arquitetura ARM. Entretanto, os campos precisavam ser descritos de trás para frente. Isto não possui nenhum significado especial, mas foi feito desta maneira apenas por razões do funcionamento interno do decodificador de instruções.

Infelizmente, o gerador de ferramentas binárias `acbingen` (que foi estendido com este projeto para síntese de ligadores capazes de gerar bibliotecas dinâmicas) e o gerador de simuladores `acsim` (onde o ArchC runtime loader foi implementado) discordavam neste aspecto, nos modelos little-endian. O `acbingen` entendia as instruções da maneira usual, mas o decodificador de instruções do simulador gerado por `acsim` só funcionava em modelos little-endian se os formatos fossem descritos da direita para esquerda (veja figura 6.3 para ilustrar o conceito).

De fato, o primeiro byte da instrução, após sofrer a inversão (byte a byte, e não campo a campo) comum em dados little-endian, contém o campo `cond` e não o campo `rm`. É importante notar que a inversão campo a campo requisitada pelo decodificador não tem relação com a inversão byte a byte característica de conversões entre endianess[9]. Além do

[9]Como explicitado, o único motivo do decodificador exigir inversão campo a campo era o de facilitar o seu funcionamento interno.

mais, seguindo a maneira invertida de `acsim`, se o projetista quisesse alternar o modelo de little-endian para big-endian, era necessário que ele invertesse a notação manualmente.

Levando estes aspectos em consideração, um estudo completo do decodificador foi realizado. As melhorias implementadas fizeram com que o decodificador não mais dependesse do nome do campo específico (evitando inclusive comparações entre strings em tempo de execução do simulador, o que causa penalidade de desempenho), mas apenas de um código. Desta forma, o problema foi resolvido com um simples código que inverte automaticamente a ordem dos campos em casos de modelos little-endian, deixando a notação igual para `acsim` e `acbingen` (ordem natural, esquerda para a direita).

Após esta modificação, ao contrário do esperado, nenhum ganho de desempenho foi observado (devido à eliminação de comparações de strings no processo de decodificação, substituindo por comparações por inteiros). Este comportamento pode ser explicado pelo fato de que o processo de decodificação é executado apenas no momento em que a instrução é buscada pela primeira vez. Uma vez decodificada, a solução é armazenada em um mecanismo de *cache*, tornando a simulação mais veloz e independente da velocidade do decodificador. Todavia, a modificação, apesar de não melhorar o desempenho, foi crucial para fazer com que `acsim` e `acbingen` concordassem quanto aos modelos ArchC.

6.4 Atendimento de chamadas de sistema

Os simuladores gerados pelo projeto ArchC contavam com uma abordagem diferente para tratamento de chamadas de sistema (necessário para executar programas aplicativos de uma arquitetura alvo que esperam um sistema operacional linux a seu serviço), definidas no módulo `ac_syscall` [5]. Esta abordagem estava intimamente ligada ao uso da biblioteca de funções padrão C `newlib` [12]. O processo consistia em atribuir um endereço fixo em determinada posição de memória reservada (endereços `0x000` a `0x100`) para cada chamada de sistema suportada pelo simulador (18 ao todo). Para fazer com que `newlib` "pulasse" para tais endereços toda vez que precisasse de uma chamada de sistema, uma biblioteca especial foi criada, `libac_sysc`, para ser compilada junto com `newlib`. Esta biblioteca define símbolos como `open` e `close`, que normalmente definem *wrappers* para chamadas de sistema. Estes símbolos eram definidos como funções que executavam uma instrução de salto para o endereço correspondente à chamada de syscall tratada pelo simulador (módulo `ac_syscall`).

Estes mecanismos dificultavam a validação completa do carregador ArchC runtime loader. A proposta original consistia em executar o benchmark Mibench [8] com todos seus programas ligados com bibliotecas dinâmicas. Para isto, seria usado uma biblioteca padrão C dinâmica. Contudo, `newlib` não pode ser produzida como biblioteca dinâmica, por não ter sido projetada para tal. A alternativa foi compilar os programas do benchmark Mibench com a glibc. Entretanto, seria muito custoso realizar o trabalho de portar a biblioteca glibc para utilizar as (poucas) funções disponíveis em `libac_sysc`, com grande risco de fracasso pelo fato de que `libac_sysc` não contempla todas as chamadas de sistemas necessárias para o funcionamento correto de glibc.

Levando em consideração estes problemas, uma nova abordagem para atendimento de chamadas de sistema foi elaborada no simulador ArchC, complementar à antiga (as duas po-

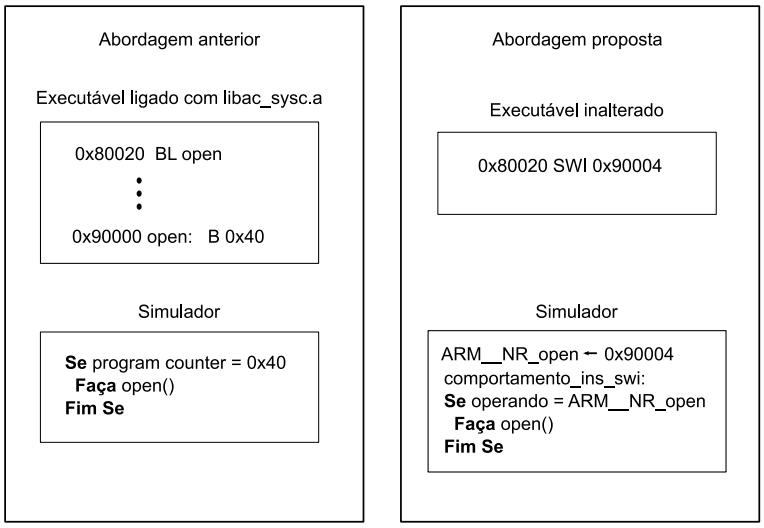

Figura 7: Esquematização comparando as duas abordagens para tratamento de chamadas de sistema no simulador ArchC.

dem funcionar juntas para prover compatibilidade com modelos antigos). Esta abordagem, mais natural nos simuladores, não necessita de nenhuma alteração na biblioteca C (glibc ou `newlib`). As chamadas de sistema são atendidas pela instrução de `trap` do processador alvo. Isto não exige grande esforço do projetista do modelo, uma vez que sua tarefa é apenas a de chamar a função genérica responsável por responder as chamadas de sistema. Ainda, é necessário que o projetista especifique completamente os códigos de chamadas de sistema suportados, pois estes variam entre cada arquitetura (algumas vezes entre processadores que implementam a mesma arquitetura), conforme definido pela ABI.

A figura 7 apresenta uma esquematização comparando as duas abordagens discutidas aqui. A primeira se refere à abordagem já implementada no simulador, em que não era possível simulador executáveis originais, mas apenas ligados com uma versão modificada de `newlib` que ligava-se à `libac_sysc`. Neste exemplo, um programa aplicativo possui uma instrução no endereço `0x80020` que faz uma chamada de sistema para `open`. Na primeira abordagem, a chamada é tratada como uma função, cuja implementação consiste em realizar um salto para um endereço reservado pelo simulador (e que portanto não pode ser utilizado pelo aplicativo, que deve tomar o cuidado de ter sua seção de código gerada acima do endereço `0x100`). As instruções em linguagem de montagem são da arquitetura ARM, onde `BL` é uma chamada a função, `B` um salto e `SWI` a instrução de *interrupção por software* que, segundo a ABI ARM, deve ser usada para implementar a transição entre código usuário e código privilegiado (do sistema operacional). Um pequeno trecho do algoritmo executado no simulador é também mostrado no diagrama da primeira abordagem, mostrando que uma comparação é realizada para verificar se a posição do *program counter* se refere a um endereço reservado. Na segunda abordagem, o executável está inalterado (da mesma maneira como seria executado em um sistema operacional Linux para ARM). Desta vez, o compor-

tamento da instrução `SWI`, no modelo ARM, é implementado, cujo código é executado no simulador quando esta instrução é encontrada. Na prática, o projetista não precisa comparar o operando para descobrir qual a chamada de sistema, pois esta tarefa é realizada por uma função independente de processador implementada diretamente no simulador ArchC. A tarefa do usuário é informar apenas o operando.

São aproximadamente 30 chamadas de sistema suportadas que, juntas, foram capazes de prover o funcionamento completo de todos os programas do benchmark Mibench ligados com a glibc estática (para atestar o funcionamento desta nova abordagem de chamadas de sistema) e com a glibc dinâmica (para avaliar o carregador ArchC runtime loader do sistema de ligação dinâmica). Ainda, a chamada de sistema `mmap` com o parâmetro para mapeamento anônimo, muito usado para alocar memória, teve a oportunidade de integração com o gerenciador de memória `memmap` de `acrtld`, de forma a verificar se o endereço de memória requisitado para alocação está disponível, levando em consideração todas as regiões ocupadas por bibliotecas carregadas dinamicamente.

6.5 Ferramenta de conversão de código de relocação

Da mesma forma como códigos de chamadas de sistema Linux são definidos por uma ABI e portanto dependentes do processador alvo, códigos de relocações são específicos e também definidos por um acordo do tipo ABI para um determinado conjunto de instruções.

Visando buscar uma solução para esta incompatibilidade entre ligadores sintetizados automaticamente pelo ArchC e ligadores originais que obedecem à ABI, foi projetada uma ferramenta que modifica os códigos de relocação em um arquivo objeto ELF. O objetivo é ler um mapa simples que traduz os códigos de relocação e aplicar esta tradução no arquivo alvo.

Esta ferramenta pode ser usada para converter permanentemente um arquivo objeto, executável ou biblioteca. Todavia, seu código foi usado também no projeto do *ArchC runtime loader*, de forma que é possível uma conversão em memória, para fins de simulação, mas que não altera o arquivo original. O objetivo é prover a capacidade de simulação com executáveis e bibliotecas originais para uma arquitetura, sem modificá-los.

A ferramenta foi escrita em código C, resultando em aproximadamente 1000 linhas.

7 Resultados

Para realizar a etapa de validação do sistema de ligação dinâmica, o plano inicial era o de usar as ferramentas conforme disposto do diagrama da figura 1, isto é, utilizando o ligador de tempo de compilação para gerar código (executáveis e bibliotecas dinâmicas) e utilizando o carregador e ligador de tempo de execução para carregar corretamente o código. Ainda, este processo seria feito com código-fonte do benchmark Mibench, em diferentes modelos ArchC (diferentes processadores). A complexidade do projeto e as dificuldades técnicas não previstas (como a necessidade de alterar o decodificador e o sistema de chamadas de sistema, entre diversas outras na elaboração do backend libbfd), entretanto, fizeram com que o tempo previsto de 1 ano para o projeto não fosse suficiente para validar o projeto conforme planejado.

Programa	Entradas pequenas		Entradas grandes	
	# instruções	MIPS	# instruções	MIPS
Quick Sort	34.946.894	10,58	128.544.142	8,93
Susan (Corners)	1.458.659	8,53	24.040.456	9,42
Susan (Edges)	2.941.524	6,30	74.635.179	9,59
Susan (Smoothing)	19.607.777	12,25	270.139.639	11,43
Basic Math	295.293.692	7,83	3.084.424.516	8,49
Bit Count	43.659.272	10,83	653.664.671	11,11
CRC32	65.753.584	9,55	1.278.150.006	9,84
ADPCM Coder	28.491.221	9,85	565.641.848	8,98
ADPCM Decoder	21.597.849	8,85	424.035.884	8,65
FFT	123.362.089	8,60	1.545.321.883	8,79
FFT Inv	216.291.416	8,87	1.394.091.519	8,64
GSM Decoder	14.540.897	9,50	791.551.892	9,80
Dijkstra	53.148.886	9,82	244.912.043	9,85
Patricia	79.208.076	5,32	488.577.410	5,29
SHA	14.203.153	11,18	147.836.401	10,80
LAME MP3 Encoder	2.162.444.271	9,44	26.658.678.088	10,04

Tabela 2: Resultados experimentais do simulador ARM executando código ligado com glibc estático.

O ligador de tempo de compilação foi o módulo em que houve maior carência de um processo de validação mais completo (mas que deverá ser realizado antes do código ser publicado em uma versão do projeto ArchC). Em particular, seu funcionamento foi assegurado apenas com pequenos programas de teste, criados com este propósito, no momento de desenvolvimento. O método de validação consistiu em comparar as bibliotecas geradas por um ligador ARM e pelo ligador sintetizado para o modelo ARM, que produziram saídas idênticas, diferentes apenas no código de relocação. Para que o ligador de tempo de execução fosse testado com a ligação de grandes programas, seria necessário sua integração, junto ao montador sintetizado, a um pacote de *cross-compiler* (compilador que roda em uma plataforma diferente daquela que irá produzir código) gcc e glibc. Esta integração não é trivial e consiste em um projeto que pode levar tempo considerável para a conclusão. Isto já foi feito para o *cross-compiler* gcc ARM e newlib para ArchC, e o motivo da dificuldade técnica é que no processo de geração da biblioteca padrão C (newlib, no caso), parte do código em linguagem de montagem é feito por humanos (não compiladores), que costumam utilizar diversas idiossincrasias dependentes de arquitetura que fazem com que o montador genérico sintetizado pelo ArchC falhe.

Para comprovar o funcionamento do carregador e ligador de tempo de execução (ArchC runtime loader), os programas Mibench foram compilados com a glibc, conforme discutido na seção 6.4, em versão estática e dinâmica. A saída produzida por cada programa foi comparada com um modelo de referência ("golden model") para assegurar que a execução dos programas está correta. A tabela 2 apresenta o número de instruções executadas pelo simulador para alguns programas selecionados que compõem o benchmark Mibench, bem como a velocidade da execução em milhões de instruções por segundo (MIPS). Estes programas estão ligados com glibc de maneira estática. Finalmente, a tabela 3 apresenta os mesmos dados para programas ligados com a glibc dinâmica, com o sistema de carregamento e ligação de tempo de execução em funcionamento.

Programa	Entradas pequenas		Entradas grandes	
	# instruções	MIPS	# instruções	MIPS
Quick Sort	36.484.522	9,91	132.122.771	8,96
Susan (Corners)	1.469.704	7,34	24.051.554	9,32
Susan (Edges)	2.957.142	9,85	74.836.512	9,49
Susan (Smoothing)	19.620.449	10,72	270.153.157	11,29
Basic Math	300.064.701	7,74	3.175.910.276	7,97
Bit Count	43.658.393	11,22	653.663.882	11,26
CRC32	93.138.162	9,35	1.810.523.030	9,08
ADPCM Coder	28.496.952	9,25	565.799.031	9,63
ADPCM Decoder	21.603.580	10,04	424.193.067	9,31
FFT	124.890.445	8,81	1.560.843.885	8,26
FFT Inv	218.688.089	8,80	1.406.845.274	8,96
GSM Decoder	14.970.178	9,72	814.912.612	10,44
Dijkstra	54.285.442	9,82	247.900.888	10,15
Patricia	83.597.966	5,46	514.558.217	5,45
SHA	14.223.630	11,47	148.051.817	11,79
LAME MP3 Encoder	2.165.948.958	9,37	26.696.180.850	9,25

Tabela 3: Resultados experimentais do simulador ARM executando código ligado com glibc dinâmica.

Nestas tabelas já é possível notar alguns efeitos da ligação dinâmica. Primeiramente, todos os programas tiveram o número de instruções aumentado. Isto ocorre devido ao referenciamento indireto dos dados globais no código das bibliotecas dinâmicas, bem como adição de um nível de indireção na chamada a funções definidas em bibliotecas dinâmicas. O código das bibliotecas dinâmicas também é independente de posição, que afeta também o número de instruções pela necessidade de dedicar um registrador para armazenar o endereço da tabela `GOT`. Também é importante notar como o sistema de ligação dinâmica afeta de maneira desigual os programas. Programas de cálculo intensivo, como ADPCM, GSM e principalmente o codificador de MP3 Lame tiveram todos um pequeno aumento relativo no número de instruções. Isto ocorreu pelo fato de que o tempo gasto na execução foi majoritariamente em funções internas, de cálculo, ligadas estaticamente. Poucas chamadas à biblioteca dinâmica glibc foram realizadas (a necessidade de leitura e escrita é bem menor do que a necessidade de processamento interno). No outro extremo, CRC32 aumentou seu número de instruções em aproximadamente 40%. Este programa faz muitas chamadas à funções da biblioteca glibc para leitura da entrada, tendo seu desempenho afetado por estas funções estarem em biblioteca dinâmica.

8 Conclusão e trabalhos futuros

Neste relatório técnico, o projeto e desenvolvimento de um sistema completo de ligação dinâmica independente de arquitetura específica foi apresentado. Este sistema utiliza o projeto ArchC para extrair informações dependentes do processador alvo, viabilizando a síntese automática de ligadores que permitem o processo de ligação dinâmica. Para a simulação, o sistema também engloba o carregador e ligador de tempo de execução, cuja função é realizar o correto carregamento do executável e todas suas bibliotecas importa-

das para que a simulação inicie. O trabalho permitiu um estudo teórico e implementação prática profundos sobre arquivos objeto e bibliotecas ELF, bem como mecanismos de ligação dinâmica do sistema operacional GNU/Linux.

Os resultados experimentais apresentaram fortes evidências do correto funcionamento do carregador dinâmico para o processador ARM. Entretanto, um processo de validação mais completo deverá ser feito futuramente, uma vez que não foi completado a tempo. É importante ressaltar que todas as modificações realizadas no projeto ArchC apresentadas neste relatório estarão disponíveis como código livre, no formato de uma nova versão do ArchC, no sítio oficial[10].

Este trabalho consistiu na continuação do projeto de síntese automática de ferramentas binárias para modelos ArchC, em que montadores e ligadores simples já estavam disponíveis [3]. Desta maneira, um projeto futuro consiste na geração de backend para compiladores a partir de modelos ArchC, que irá consolidar a síntese de um conjunto completo de ferramentas para desenvolvimento de software em sistemas embarcados de processadores customizados.

Referências

[1] Maghsoud Abbaspour and Jianwen Zhu. Retargetable binary utilities. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 331–336, New York, NY, USA, 2002. ACM.

[2] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The ArchC architecture description language and tools. *International Journal of Parallel Programming*, 33(5):453–484, October 2005.

[3] Alexandro Baldassin, Paulo Centoducatte, Sandro Rigo, Daniel Casarotto, Luiz C. V. Santos, Max Schultz, and Olinto Furtado. An open-source binary utility generator. *ACM Trans. Des. Autom. Electron. Syst.*, 13(2):1–17, 2008.

[4] Steve Chamberlain. Libbfd: The binary file descriptor library. cygnus support, bfd version 3.0 edition. In *in FSF binutils distribution; Copyright Free Software Foundation*, 1991.

[5] Universidade Estadual, De Campinas, Marcus Bartholomeu, Sandro Rigo, Rodolfo Azevedo, and Guido Araujo. Emulating operating system calls in retargetable isa simulators, 2003.

[6] D. Elsner et al. *Using as, the GNU assembler*. Free Software Foundation, Inc., March 1993. version 2.15.

[7] Free Software Foundation, Inc. *The GNU C Library manual*, 2009. version 2.9.

[10] www.archc.org

[8] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th Annual IEEEWorkshop on Workload Characterization*, pages 3–14, December 2001.

[9] Jaesoo Lee, Jiyong Park, and Seongsoo Hong. Memory footprint reduction with quasi-static shared libraries in mmu-less embedded systems. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 24–36, Washington, DC, USA, 2006. IEEE Computer Society.

[10] John R. Levine. *Linkers and Loaders (The Morgan Kaufmann Series in Software Engineering and Programming)*. Morgan Kaufmann, January 2000.

[11] Roland H. Pesch and Jeffrey M. Osier. *The GNU binary utilities*. Free Software Foundation, Inc., May 1993. version 2.15.

[12] Red Hat, Inc. *Red Hat newlib C library documentation*, 2008. version 1.17.

[13] Richard Stallman. *Using the GNU compiler collection*. Free Software Foundation, Inc., May 2004. For GCC version 3.4.3.