Chapter 1

Prática: Análise de binários e sistemas assistida por *hardware*

Marcus Botacin[1], Paulo Lício de Geus[2], and André Grégio[1]

[1]Universidade Federal do Paraná (UFPR)
[2]Universidade Estadual de Campinas (UNICAMP)

1.1. Introdução

As atividades práticas deste curso estão divididas em duas categorias: i) análise exploratória de código; e ii) exercícios com o GDB. A primeira consiste em analisar o código de diversas soluções de código aberto que implementam recursos com o suporte de *hardware* para verificar, no campo prático, a implementação dos conceitos vistos na teoria. Este tipo de atividade visa cobrir a lacuna existente para a real implementação de recursos de *hardware*, que exige recursos e tempo superiores aos disponíveis neste minicurso. A segunda classe de tarefas, por sua vez, consiste em permitir aos alunos experimentar um dos recursos de *hardware* (contadores de *performance*. Escolhemos os monitores de *branch* para esta tarefa visto que seu uso é o mais simples dentre as soluções apresentadas, o que torna sua experimentação factível no tempo de um minicurso. Para experimentá-los, contaremos com o suporte do *Linux perf*, uma solução de *profiling* integrada ao *kernel*.

1.2. Análise exploratória de código

As etapas abaixo foram desenvolvidas de modo a guiar a audiência pelo código fonte das soluções, visando apresentar a interação destas com os recursos de *hardware* apresentados na parte teórica.

1. **Obtenha o código do *Xen***: `git clone https://github.com/xen-project/xen.git`
   (a) **Navegue até as definições de *hardware***: `xen/xen/arch/x86/hvm`
      i. **Observe o suporte da AMD**: `svm`
      ii. **Observe o suporte da Intel**: `vmx`
   (b) **Considere o suporte da Intel**: `xen/xen/arch/x86/hvm/vmx/`

        i. **Observe a definição das estruturas de controle**: `vmcs.c`

2. **Obtenha o código do *Coreboot***: `git clone https://github.com/coreboot/coreboot.git`
    (a) **Navegue até o código SMM**: `coreboot/src/cpu/x86/smm`
        i. **Observe o tratamento de interrupções**: `smihandler.c`
3. **Obtenha o código do *perf***: `git clone https://github.com/torvalds/linux.git`
    (a) **Navegue pelo código do *perf* em *kernel***: `linux/arch/x86/events/intel`
        i. **Observe a definição das estruturas de *hardware***: `bts.c`
    (b) **Navegue pelo código do *perf* em *userland***: `linux/tools/perf`
4. **Obtenha o código do *BranchMonitor***: `git clone https://github.com/marcusbotacin/BranchMonitoringProject.git`
    (a) **Navegue até**: `BranchMonitoringProject/BranchMonitor.NMI/src/`
        i. **Observe o registro da *callback* NMI**: `BranchMonitor.c`
    (b) **Navegue até**: `BranchMonitoringProject/BranchMonitor.NMI/src/BTS`
        i. **Observe a implementação da rotina NMI**: `interrupt.c`

1.3. Extraindo *branches* com o *perf*

Inicialmente, extrairemos informações dos *branches* tomados durante a execução de diferentes aplicações. Em seguida, reconstruiremos informações de mais alto nível a partir destas.

1. **Estatísticas Gerais**: `perf stat ./app`
2. **Traçando um binário**: `perf record -b ./app`
3. **Visualizando os dados coletados**: `perf report`
4. **Filtrando por tipo de dado**: `perf record -j any_call,any_ret ./app`
5. **Filtrando pelo escopo da captura**: `perf record -j any_call,u ./app`
6. **Identificando regiões frequentes**: `perf report -sort symbol_from,symbol_to -stdio`
7. **Identificando caminhos frequentes**: `perf report -branch-history -stdio`
8. **Utilizando Processor Tracer**: `perf record -e intel_pt,u ./app`