CoreTool: Identificação e Análise de T*hreads* em Sistemas *Multicore*

Camila Yumi Koike, Eduardo Z. G. Max, Rodolpho Gheleri, Ricardo Santos

Faculdade de Computação (FACOM) – Universidade Federal do Mato Grosso do Sul (UFMS)
Caixa Postal 549 – 79070-900 – Campo Grande – MS – Brazil

{camilakoike,eduardo.max,rodolpho.gheleri}@gmail.com,
ricardo@facom.ufms.br

Abstract. *The availability of processing resources in current processors platforms along with the increase of software complexity reinforce the demands for tools on monitoring multithreaded applications. This paper presents CoreTool, a tool for threads behavior analysis in multicore systems. CoreTool monitors threads scheduling and execution providing useful information on cores utilization and threads load balancing (instructions per thread, threads scheduling on cores). CoreTool has been built on the top of the Pin dynamic binary instrumentation framework targeted to multithreaded applications. We have performed a set of experiments using Linux applications and programs from the Splash-2 benchmark. Our experiments were carried out on configurations of multicore processors with four (2 physical and 2 virtual) and eight (4 physical and 4 virtual) cores.*

Resumo. *A disponibilidade de recursos de processamento nos processadores atuais aliada à complexidade do software faz com que ferramentas automatizadas sejam cada vez mais importantes no processo de validação e avaliação de aplicações multithreaded. Este artigo apresenta o desenvolvimento de uma ferramenta para análise do comportamento de threads em sistemas multicore. Especificamente, a ferramenta proposta, denominada CoreTool, acompanha o escalonamento e execução das threads de uma aplicação e retorna informações precisas sobre a utilização dos núcleos de processamento assim como a execução de instruções por thread. CoreTool foi desenvolvida a partir da infraestrutura PIN para instrumentação binária dinâmica de aplicações multithreaded. Experimentos de validação e avaliação foram realizados com a ferramenta e aplicações Linux e do benchmark Splash-2. Os experimentos foram executados sobre duas configurações de processadores multicore com quatro e oito núcleos.*

1. Introdução

A medida que a complexidade do software aumenta, a instrumentação -- uma técnica para inserir código extra em aplicações para que seja possível observar seu comportamento -- está se tornando cada vez mais importante [Luk, et al 2005]. A instrumentação é essencial para análise de programas pois possibilita: caracterização e avaliação de desempenho, validação do software, detecção de *bugs* e pode ser realizada em vários estágios de desenvolvimento e execução: diretamente sobre o código fonte, em tempo de compilação, em tempo de execução ou em tempo de ligação. Para que isso seja possível, é necessário a utilização de ferramentas robustas para a execução dessas

tarefas.

Em se tratando de aplicações paralelas que exploram a multiciplidade de recursos de processamento existente no processador alvo, técnicas de instrumentação são ainda mais importantes pois possibilitam analisar o comportamento e o desempenho considerando a utilização dos recursos de processamento disponíveis. Neste cenário, um desafio existente está na dificuldade em instrumentar e obter informações detalhadas sobre o comportamento dessas aplicações. Existem ferramentas conhecidas para análise e monitoramento de aplicações *multithreaded.* Alguns exemplos dessas ferramentas são: *Microsoft® Process Monitor*, *Thread Status Monitor*, *Intel® VTune™*, *VisualVM*, *IBM® Thread and Monitor Dump Analyzer for Java*. Algumas características identificadas nessas ferramentas são: análise em programas específicos de linguagens (ênfase em programas Java), identificação de *hot spots* no código para melhoria de desempenho, monitoramento do desempenho por *thread* e validação de programas.

Uma lacuna observada nas ferramentas para análise de programas *multithreaded* refere-se ao detalhamento sobre o escalonamento das *threads* nos recursos de processamento. O conhecimento sobre o escalonamento pode apresentar informações relevantes para a melhoria do desempenho que extrapolam análises comumente utilizadas sobre o comportamento de *threads*. Como exemplo, a análise detalhada sobre o escalonamento e balanceamento de carga de *threads* pode revelar a necessidade de modificações no escalonador (utilização de novas políticas de escalonamento) ou no código do programa (aumento/redução do número de *threads*). Nos processadores atuais, em que a disponibilidade de processamento aparece na forma de núcleos heterogêneos (por exemplo: arquitetura ARM® BIG.little™ e similares, arquiteturas GP-GPU), analisar detalhadamente a utilização desses recursos pelas *threads* é de suma importância para otimizar a aplicação e/ou o sistema.

A partir desta motivação, este trabalho apresenta a ferramenta *CoreTool* com o objetivo de analisar o comportamento dinâmico de aplicações *multithreaded*. Especificamente, *CoreTool* detalha o escalonamento das *threads* de uma aplicação identificando os núcleos de processamento utilizados ao longo da execução, o tempo de escalonamento e a distribuição de instruções executadas por *thread* em cada núcleo. A ferramenta proposta neste trabalho foi desenvolvida a partir da infraestrutura Pin [Reddi, et al 2004] para instrumentação binária dinâmica. *CoreTool* realiza a instrumentação do código da aplicação em tempo de execução de forma que não é necessário qualquer alteração sobre o código fonte original.

O texto do artigo está organizado conforme segue: a Seção 2 apresenta a infraestrutura de instrumentação binária dinâmica Pin; a Seção 3 apresenta e discute características de outras ferramentas baseadas em Pin para análise de programas *multithreaded*; o detalhamento sobre o desenvolvimento da ferramenta *CoreTool* é descrito na Seção 4; os experimentos, resultados e discussões são apresentados na Seção 5; a Seção 6 apresenta as conclusões e extensões para trabalhos futuros.

2. Ferramenta Pin

Pin [Reddi et al 2004], [Luk et al 2005], [Pin 2012] é um sistema de instrumentação binária dinâmica que permite a criação de ferramentas para análise de programas. Pin é similar ao *ATOM toolkit* [Reddi, et al 2004] e enfatiza as seguintes características: fácil usabilidade, portabilidade, transparência, eficiência e robustez; além de oferecer bibliotecas específicas para instrumentação de programas. Os usuários de Pin

desenvolvem ferramentas customizadas - denominadas *plugins pintools* – a partir dessas bibliotecas.

A *API* de Pin foi projetado para ser independente de arquitetura e permite que as *pintools* sejam compatíveis com diversos conjuntos de instruções (IA-32, Intel64, IA-64, ARM) e sistemas operacionais (Linux, Windows, MacOS, Android) [Pin 2012]. O motor de instrumentação Pin permite a uma *pintool* inserir chamadas de funções em qualquer parte do código mantendo a consistência semântica do programa (funções da *API* Pin salvam e restauram os valores dos registradores da aplicação). Em Pin, os recursos de instrumentação utilizam uma cache de códigos a fim de amenizar o impacto da instrumentação sobre o desempenho [Wallace, et al 2007].

A infraestrutura de software Pin consiste em uma máquina virtual, cache de código e uma *API* de instrumentação, que são invocados pela *pintool*. A máquina virtual consiste em um compilador *just-in-time* (JIT), um emulador e um *dispatcher*. A Figura 1 apresenta a organização desses componentes e a interação entre o software Pin, a aplicação alvo e uma *pintool*. Após a execução, a máquina virtual coordena esses componentes para que a aplicação seja executada e a *pintool* possa obter as informações que requisitou.

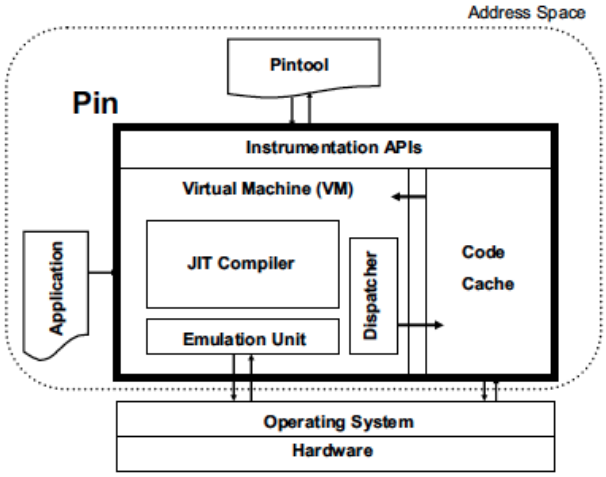

Figura 1. Arquitetura *Pin* [Luk, et al 2005].

Pin, *pintool* e a aplicação executam no mesmo espaço de endereços. O usuário invoca todos os três através da linha de comandos em tempo de execução. O software Pin então usa o *process trace* para conseguir o controle da aplicação e capturar o contexto do processador. Depois, ele carrega e inicializa a *pintool* do usuário. Por último, *Pin* interpreta ou compila o código da aplicação com instrumentação intercalada como especificado pela *pintool* de forma que seja transparente para a execução da aplicação [Wallace, et al 2007].

3. Ferramentas Pin para Análise de Programas Paralelos e *Multithreaded*

Pin tem sido amplamente utilizada para análise de programas paralelos [Bach et al 2010]. Apresenta-se a seguir algumas ferramentas e *pintools* que utilizam a

infraestrutura Pin para extrair informações sobre o comportamento de programas que executam desde processadores *multicore* até ambientes de *clusters* de computadores.

***Intel® Parallel Inpector™*:** analisa a execução dos programas *multithreaded* para encontrar erros de memória e de *threads*. Utiliza Pin para instrumentar o programa em execução e coletar as informações necessárias para detectar erros.

***Intel® Parallel Amplifier™*:** realiza três tipos de análises:

Hotspots: identifica partes do programa que podem ser paralelizadas visando melhoria de desempenho;

Concurrency: indica a utilização das *CPUs* pelos componentes *(funções)* de um programa;

Lock e *wait*: informa o tempo de espera devido aos *locks* de programas *multithreads.* Utiliza Pin para instrumentar a aplicação para coleta de dados para a análise de *locks* e *waits*.

Intel® Trace Analyzer and Collector™: fornece informações para otimização do desempenho de aplicações em ambientes de *clusters*, pois identifica gargalos desempenho com a comunicação via *Message Passing Interface* (MPI). Utiliza Pin para instrumentar as chamadas à biblioteca MPI.

As ferramentas *Intel® Parallel Inpector™*, *Intel® Parallel Amplifier™* e *Intel® Trace Analyzer and Collector™* fazem parte do pacote *Intel® Parallel Studio™*.

***CMP$IM [Jallel et al 2008]*:** utiliza Pin para coletar o endereço de memória dos programas paralelos e *multithreaded*. Essa ferramenta informa taxa de *miss*, reuso, compartilhamento e coerência de *cache.*

***PinPlay [Patil 2010]*:** é um conjunto de ferramentas, baseadas em Pin, para a captura em nível de usuário e repetição determinística de programas *multithreaded*. O programa instrumentado executa sob o controle uma ferramenta Pin para captura de informações (*logging*) de chamadas do sistema e dependências de memória entre *threads*. Uma outra *pintool* executa o *log* registrado das chamadas do sistema, reproduzindo exatamente a execução gravada.

***Prospector [Kim 2010]*:** atua sobre programas seriais identificando paralelismo em laços. Realiza *profiling* sobre a execução dos laços de um programa (número de iterações, número de instruções executadas dentro do laço). Também detecta dependências de dados *loop-carried*, que devem ser preservadas durante o processo de paralelização.

Intel® Software Development Emulator (SDE): é um emulador funcional em nível de usuário para novas instruções no conjunto de instruções da Intel64 construída sobre Pin. Suporta emulação e depuração de programas *multithreaded* que utiliza a extensão dos conjuntos de instruções da Intel AVX, AES, e SSE4.

LIKWID [Treibig et al 2010]: é um conjunto de ferramentas por linha de comando disponível em ambiente *Linux* que suporta processadores baseados em arquitetura Intel x86 e que não necessita de componentes adicionais de software, funcionando como uma camada acima da aplicação e sistema operacional. O *LIKWID* possui um ferramenta chamada *likwid-pin*, permitindo realizar instrumentação binária para obter informações referente ao uso de *threads* em aplicações *multi-threaded*, suportando *threads* baseadas no modelo *POSIX* e híbridos "*MPI+threads*".

4. Ferramenta *CoreTool*

A implementação da ferramenta *CoreTool* foi desenvolvida na linguagem C++ e foi baseada nas APIs oferecidas pela biblioteca Pin (*sched.h)* para identificação do escalonamento inicial de *threads* sobre os *cores.* No projeto de *Coretool* novas características foram implementadas como o tempo do escalonamento da *thread* sobre um *core,* a identificação das instruções (a qual *thread* pertence) executadas em cada *core*, a sequência de escalonamento de uma *thread* sobre o conjunto de *cores*. Adicionalmente, novas classes foram desenvolvidas para manipular estruturas de dados que armazenam os resultados de tempo, escalonamento e quantidade de instruções das *threads*.

O fluxo de execução da ferramenta *CoreTool* consiste na execução de três funções (Figura 2) para registro das *threads*, escalonamento das *threads* nos núcleos de processamento e a impressão dos resultados, nessa ordem: *Thread_Trace*, *Thread_Start* e *Fini*. *Thread_Trace* é a função responsável pelo registro do traço de execução das *threads*. Nessa função também registra-se o número de instruções em cada bloco básico de cada *thread* (função CountBBL). *Thread_Start* é a função que registra o escalonamento das *threads* nos núcleos e o tempo desse escalonamento. *Fini* é a função que cria o relatório de resultados das *threads* e retorna para o usuário as informações das estruturas de dados de *thread*, *cores,* instruções (função SumInst) e tempo de escalonamento.

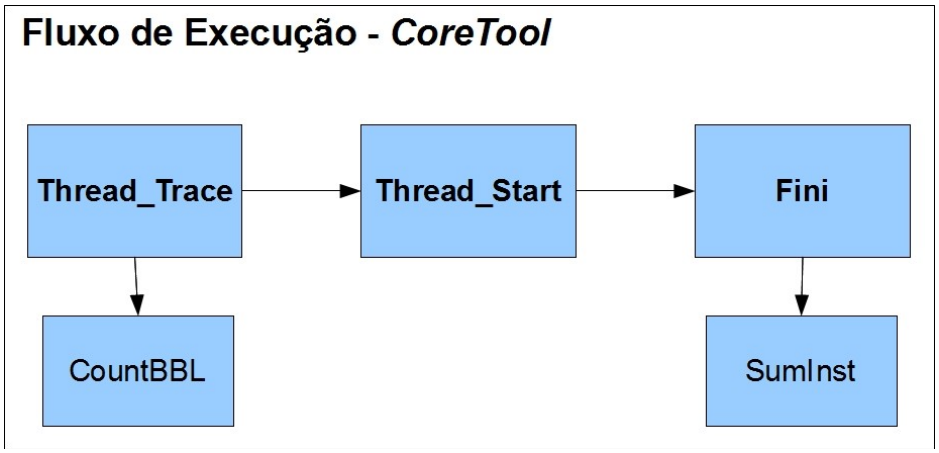

Figura 2. Fluxo de execução da ferramenta CoreTool.

O único parâmetro passado como argumento para *CoreTool* é a aplicação onde será feita a instrumentação. Os resultados da instrumentação são apresentados no arquivo "*report.txt*" a partir dos seguintes parâmetros de linha de comando:

%pin -t CoreTool.so -- <programa>

A Figura 3 apresenta um exemplo da instrumentação realizada sobre o programa *thunderbird* em um processador com oito núcleos de processamento. O relatório com informações sobre as *threads* possui quatro seções principais:

1. *Thread Origin*: lista em qual *core* cada *thread* foi iniciada e fornece informação da *thread* que gerou a *thread* atual.
2. *Core Analysis*: contém a quantidade e a porcentagem de instruções executadas em cada *core* do processador e o tempo total que durou a instrumentação.

3. *Thread Details*: quantidade de instruções executadas pela *thread* em cada *core* do sistema.
4. *Thread Schedule*: informa o tempo do escalonamento da *thread* nos *cores*.

```
================================================
This application is instrumented by CoreTool
================= Thread Origin ==========================
Thread num 0 started in CPU 0 Get PID: 12533 Parent Thread: 0
Thread num 1 started in CPU 6 Get PID: 12533 Parent Thread: 12533
Thread num 2 started in CPU 7 Get PID: 12533 Parent Thread: 12541
Thread num 3 started in CPU 5 Get PID: 12533 Parent Thread: 12533
Thread num 4 started in CPU 5 Get PID: 12533 Parent Thread: 12533
Thread num 5 started in CPU 5 Get PID: 12533 Parent Thread: 12533
Thread num 6 started in CPU 5 Get PID: 12533 Parent Thread: 12533
Thread num 7 started in CPU 4 Get PID: 12533 Parent Thread: 12533
Thread num 8 started in CPU 5 Get PID: 12533 Parent Thread: 12533
Thread num 9 started in CPU 6 Get PID: 12533 Parent Thread: 12533
                                    ...
================= Core Analysis ==========================
CoreTool analysis results:
Number of CPU core's: 8
Number of instructions executed: 3282183721
Number of basic blocks executed: 824118108
Number of threads created: 40
Time elapsed: 228.076 seconds

Number of Instructions executed in CPU 0: 1119219143(34.0998%)
Number of Instructions executed in CPU 1: 555651429(16.9293%)
Number of Instructions executed in CPU 2: 530371736(16.1591%)
Number of Instructions executed in CPU 3: 905855503(27.5992%)
Number of Instructions executed in CPU 4: 33122296(1.00915%)
Number of Instructions executed in CPU 5: 17612592(0.536612%)
Number of Instructions executed in CPU 6: 116407275(3.54664%)
Number of Instructions executed in CPU 7: 3943747(0.120156%)
Total = 3282183721
```

```
================= Thread Details ==========================
Thread 0
 CPU: 0, ins: 1039252763
 CPU: 1, ins: 419702681
 CPU: 2, ins: 476747894
 CPU: 3, ins: 840611769
 CPU: 4, ins: 26850537
 CPU: 5, ins: 14527897
 CPU: 6, ins: 114613062
 CPU: 7, ins: 1683
Total instructions thread 0: 2932308286

Thread 1
 CPU: 0, ins: 417449
 CPU: 1, ins: 92548
 CPU: 2, ins: 194960
 CPU: 3, ins: 576269
 CPU: 4, ins: 0
 CPU: 5, ins: 0
 CPU: 6, ins: 1456127
 CPU: 7, ins: 0
Total instructions thread 1: 2737353

Thread 2
 CPU: 0, ins: 4056871
 CPU: 1, ins: 650868
 CPU: 2, ins: 2141307
 CPU: 3, ins: 150333
 CPU: 4, ins: 0
 CPU: 5, ins: 384209
 CPU: 6, ins: 0
 CPU: 7, ins: 47851
Total instructions thread 2: 7431439
...
================= Thread Schedule ==========================
...
Thread: 23
CPU: 6, Scheduling time: 191.06s

Thread: 24
CPU: 6, Scheduling time: 191.09s

Thread: 25
CPU: 5, Scheduling time: 192.64s
CPU: 3, Scheduling time: 192.65s
CPU: 0, Scheduling time: 193.18s
CPU: 1, Scheduling time: 209.24s

Thread: 26
CPU: 4, Scheduling time: 212.93s

Thread: 27
CPU: 5, Scheduling time: 232.08s
...
```

Figura 3. Relatório da execução e escalonamento das *threads* do programa *Thunderbird*.

CoreTool é uma ferramenta *freeware* e *open-source* e a versão atual está disponível para *download* a partir do endereço http://lscad.facom.ufms.br/wiki/index.php/Downloads.

5. Experimentos e Resultados

Os experimentos para validação e avaliação da ferramenta *CoreTool* utilizaram duas configurações de processadores com diferentes quantidades de *cores* (Tabela 1) e dois conjuntos de aplicações (Tabela 2): aplicativos do sistema operacional Linux e programas *multithreaded* do benchmark *The Modified SPLASH-2* [Splash-2 2007]. A Tabela 2 também apresenta a quantidade de *threads* geradas, para cada programa, em cada uma das configurações de processadores utilizadas. Os programas *firefox* e *thunderbird* foram executados a partir da inicialização e abertura de um *site* de busca (página inicial).

Caracteríticas	Configuração 1	Configuração 2
Processador	Intel® Core™ i7-3610QM CPU @ 2.3GHz x 8	Intel® Core™ i5-3317U CPU @ 1.70GHz × 4
Memória	16GB DDR3/1600	6GB DDR3/1600
Sistema Operacional	Ubuntu 12.04 LTS 64bit	Ubuntu 12.04 LTS 64bit

Tabela 1. Configurações dos processadores utilizados nos experimentos.

Conjuntos	Programas	Nº *Threads* - Configuração 1	Nº *Threads* - Configuração 2
Linux	Firefox v. 30.0	55	58
	Thunderbird v. 24.6.0	41	52
Splash-2	*Cholesky*	8	4
	FFT	8	4
	FFM	8	4
	LU	8	4
	Ocean	8	4
	Radiosity	8	4
	Radix	8	4
	RayTrace	8	4
	Volrend	8	4
	Water-Nsq	1	1
	Water-Sp	1	1

Tabela 2. Conjuntos de programas utilizados nos experimentos.

Para fins de comparação e validação do escalonamento de *threads* em *cores*, a partir de binários de programas, utilizou-se a ferramenta de monitoramento de processos e *threads htop*, disponível em sistemas operacionais Linux. O *htop* monitora processos e *threads* em tempo de execução. A validação de *CoreTool* foi realizada comparando a seção *Thread Origin* do relatório gerado para cada programa com o *status* de execução retornado por *htop* para o mesmo programa. A Figura 4 demonstra a execução de partes do código do programa *firefox* e o seu monitoramento nas duas ferramentas: do lado esquerdo o *CoreTool* e do lado direito, o *htop*. Ressalta-se que *CoreTool* informa o identificador do processo que gerou cada *thread* (*PID*) enquanto que *htop* informa, na coluna *PID*, o identificador do processo e da *thread* TID.

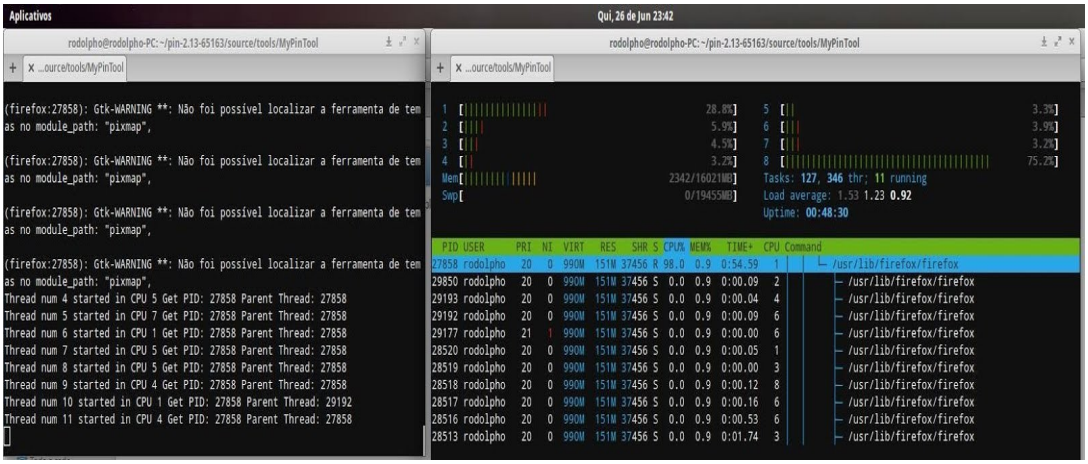

Figura 4. *CoreTool* (esquerda) e htop (direita).

A partir da execução dos experimentos com a ferramenta *CoreTool* e da análise de distribuição das instruções em cada *thread*, apresenta-se resultados sobre a porcentagem de instruções executadas em cada núcleo de processamento (Tabelas 3 e 4), a sobrecarga no tempo de execução dos programas utilizando a ferramenta Pin e *CoreTool* (Tabelas 5 e 6) e a distribuição de *threads* sobre os núcleos de processamento disponíveis na configuração 1 (Figura 5).

Os resultados apresentados nas Tabelas 3 e 4 foram obtidos a partir da Equação 1 que, basicamente, acumula as instruções executadas em cada *thread* (0<*j*<*TotalThreads)* que executou sobre o núcleo (*0*<*i*<*TotalNúcleos*).

%Instruções_Núcleo_i=(∑Instruções_Thread_j_em_i) / Total_Instruções_Programa (Equação 1)

As Tabelas 3 e 4 revelam que mesmo utilizando recursos de processamento e *threads* homogêneas, a utilização desses recursos não é, necessariamente, homogênea. Como exemplo, analisando a distribuição das instruções executadas no programa *Firefox* (Tabela 3) e comparando com a Figura 5, percebe-se, claramente, que os quatro primeiros núcleos foram mais utilizados pelas *threads.* De um total de 55 *threads* distintas geradas pelo programa *firefox* na plataforma de hardware da configuração 1, o *Core 0* foi utilizado por 36 *threads* distintas enquanto que o *Core 7* foi utilizado apenas por 19 *threads.* Resultado similar acontece com o programa *Thunderbird* ao comparar a utilização do *Core 1* com o *Core 4*. A análise mais detalhada sobre a utilização dos recursos pode revelar que melhores estratégias de escalonamento podem ser adotadas uma vez que nem sempre há relacionamento entre a carga de execução de uma aplicação com o número de *threads* executadas. No aplicativo *firefox*, o *Core* 6 recebeu menos *threads* do que os *cores* 4 e 5, mas executou mais instruções do que esses *cores*. O mesmo comportamento pode ser observado no aplicativo *thunderbird* ao analisar o escalonamento sobre os *cores* 6 e 7. A maioria dos programas do *benchmark Splash-2*, mesmo com pequenas variações de carga (instruções executadas) sobre os *cores*, demonstram homogeneidade na utilização dos recursos.

Embora não apresentado neste artigo, destaca-se que os mesmos experimentos foram executados sobre 9 programas (*blackcholes, bodytrack, canneal, dedup, fluidanimate, freqmine, streamcluster, swaptions, x264*) do benchmark *Parsec* [Biena et al 2008] cuja distribuição e balanceamento de carga das *threads* apresentaram comportamentos similares aos obtidos com *Splash-2*.

Programas	***Core 0***	***Core 1***	***Core 2***	***Core 3***	***Core 4***	***Core 5***	***Core 6***	***Core 7***
Firefox	15,98	24,58	20,87	20,75	3,73	4,71	7,52	1,86
Thunderbird	9,94	27,82	16,67	20,96	0,34	0,19	14,4	9,68
Cholesky	8,06	8,61	8,32	8,6	8,11	8,44	41,67	8,16
FFT	6,5	51,37	6,43	6,64	9,41	6,14	6,72	6,75
FFM	12,02	12,48	12,15	12,48	12,82	12,36	12,54	13,11
LU	11,57	11,65	11,29	11,53	11,59	11,54	19,33	11,47
Ocean	11,97	12,57	12,29	12,83	12,3	12,77	12,66	12,57
Radiosity	11,64	12,1	12,15	12,23	11,8	14,8	13,04	12,18
Radix	12,41	12,15	12,52	12,15	12,63	12,48	13,12	12,49
RayTrace	12,13	12,15	12,05	11,87	12,23	15,34	12,28	11,92
Volrend	9,43	9,28	9,94	9,9	31,03	10,11	10,71	9,56
Water-Nsq	0	0	0	0	0	0	0	100
Water-Sp	0	0	0	0	0	100	0	0

Tabela 3. Porcentagem de instruções executadas por *core* de cada programa na configuração 1.

Programas	***Core 0***	***Core 1***	***Core 2***	***Core 3***
Firefox	31,72	20,36	31,55	16,35
Thunderbird	40,8	21,5	24,48	13,2
Cholesky	15,92	18,1	15,4	50,56
FFT	65,19	10,44	10,86	13,5
FFM	25,45	27,4	23,52	23,61
LU	23,52	24,41	31,52	20,54
Ocean	27,78	30,03	21,63	20,54
Radiosity	27,81	26,76	22,41	23
Radix	27,59	26,37	16,44	29,57
RayTrace	25,1	26,05	22,86	25,96
Volrend	42,07	19,2	20,44	18,27
Water-Nsq	0	0	0	100
Water-Sp	0	100	0	0

Tabela 4. Porcentagem de instruções executadas por *core* de cada programa na configuração 2.

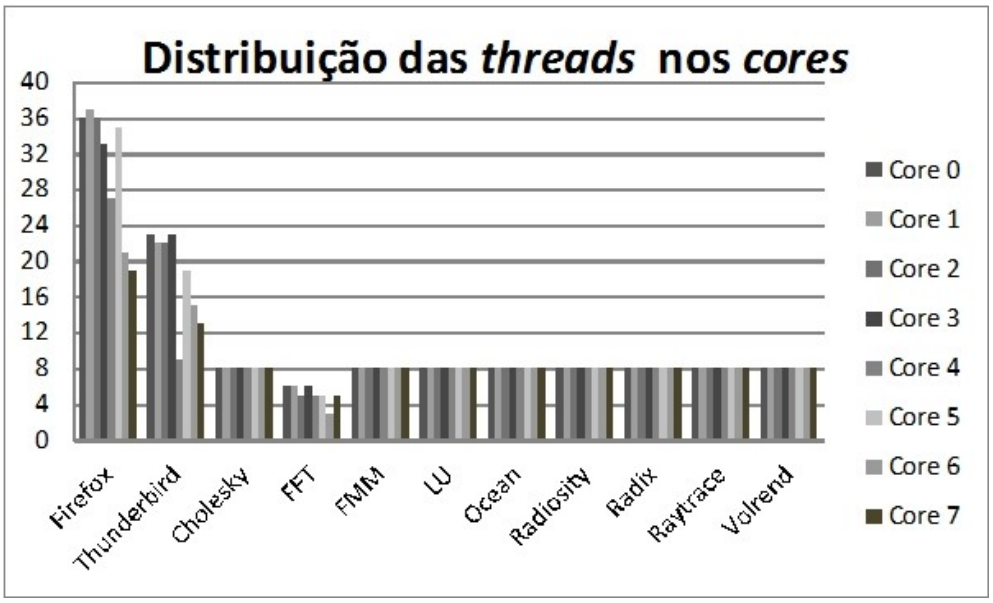

Figura 5. Distribuição das *threads* na configuração 1.

Os resultados (em segundos) apresentados nas Tabelas 5 e 6 foram obtidos a partir de 10 execuções para cada combinação do programa (Sem Pin, Com Pin e Com Pin+*CoreTool*). Observa-se um acréscimo significativo no tempo de execução dos programas com a utilização de Pin+*CoreTool*. Embora seja esperado que a utilização de uma infraestrutura de instrumentação dinâmica aumente o tempo de execução dos programas, parte considerável dessa sobrecarga deve-se à utilização de estruturas de dados do tipo lista encadeada para manipulação das *threads* sob instrumentação. A substituição dessas estruturas por outras com menor tempo de acesso ou mesmo com tempo de acesso constante (como estruturas do tipo árvore ou *hashing)* deve minimizar essas sobrecargas.

Programas	**Sem *Pin***	**Com *Pin***	**Com *Pin* e *CoreTool***
Firefox	2,3	84,14	150,11
Thunderbird	3,64	79,07	257
Cholesky	0,07	0,78	11,66
FFT	0,004	0,5	0,58
FFM	0,11	1,79	19,41
LU	0,05	0,63	25,93
Ocean	0,1	2,47	14,99
Radiosity	0,03	1,55	12,09
Radix	0,02	0,55	2,51
RayTrace	0,29	3,21	526,12
Volrend	0,12	1,12	8,08
Water-Nsq	0,03	0,36	0,94
Water-Sp	0,002	0,34	0,54

Tabela 5. Tempo de execução (em segundos) por programa na configuração 1.

Programas	**Sem *Pin***	**Com *Pin***	**Com *Pin* e *CoreTool***
Firefox	2,72	86,01	667,8
Thunderbird	4,47	81,03	472,4
Cholesky	0,11	0,85	21,52
FFT	0,004	0,7	1,15
FFM	0,24	1,86	178,55
LU	0,05	0,75	51,67
Ocean	0,1	2,54	9,21
Radiosity	0,16	1,67	22,92
Radix	0,03	0,69	2,87
RayTrace	0,86	3,33	859,06
Volrend	0,08	1,2	11,78
Water-Nsq	0,13	0,4	0,88
Water-Sp	0,08	0,38	0,9

Tabela 6. Tempo de execução (em segundos) por programa na configuração 2.

6. Conclusões e Trabalhos Futuros

Este artigo apresentou o desenvolvimento da ferramenta para identificação e análise do comportamento de *threads* em plataformas de hardware compostas por vários núcleos de processamento. A ferramenta proposta, denominada *CoreTool*, é uma *pintool* que executa no mesmo espaço de endereço da programa sob análise e retorna informações, em tempo de execução, sobre o mapeamento das *threads* nos núcleos de processamento assim como a execução de instruções sobre essas *threads* e núcleos.

As funcionalidades projetadas e implementadas em *CoreTool* possibilitam que a ferramenta seja útil para usuários com diferentes perfis. Desenvolvedores de aplicações *multithreaded* podem utilizar a ferramenta para verificar a utilização de recursos de processamento pelas *threads* da aplicação objetivando a melhoria do desempenho. Projetistas de sistemas podem adotar *CoreTool* para analisar a eficiência do escalonador de *threads* visando balancear a utilização dos recursos e minimizar o tempo de resposta do sistema.

Experimentos foram realizados utilizando aplicações *multithreaded* de dois conjuntos de programas: aplicativos linux e programas do *benchmark Splash-2*. Os resultados possibilitaram observar que nem sempre a quantidade de *threads* executadas é uma métrica determinante para a sobrecarga de um núcleo de processamento.

Como trabalhos futuros, pretende-se desenvolver novas estruturas de dados a fim de minimizar a sobrecarga de *CoreTool* sobre o desempenho da aplicação *multithreaded.* Além disso, vislumbra-se adicionar outras funcionalidades referente à análise de *threads* como tempo de espera devido a condições de disputa e espera por dados de dispositivos de entrada/saída.

Agradecimentos

Os autores agradecem as agências de fomento de pesquisa CAPES, CNPq e Fundect-MS e à UFMS pelo suporte e financiamento de vários projetos de pesquisa desenvolvidos no Laboratório de Sistemas Computacionais de Alto Desempenho (LSCAD) da Faculdade de Computação da UFMS.

Referências

Bach, M., Charney, M., Cohn, R., Demikhovsky, E., Devor, T., Hazelwood, K., Jaleel, A., Luk, C. -K., Lyons, G., Patil, H. Tal, A. (2010) “Analyzing Parallel Programs with Pin”, IEEE Computer Society.

Biena, C., Kumar, S., Singh, J. P., Li, K. (2008). “The PARSEC Benchmark Suite: Characterization ad Architectural Implications”. In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 72-81, ACM.

Jaleel, A., Cohn, R. S., Luk, C. K., Jacob, B. (2008). “CMP$im: A Pin-Based On-the-Fly Multi-Core Cache Simulator”. In Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA, pp. 28-36.

Kim, M., Kim, H., & Luk, C. K. (2010). “Prospector: A Dynamic Data-Dependence Profiler to Help Parallel Programming”. In Proceedings of the USENIX Workshop on Hot Topics in Parallelism.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi,V. J., Azelwood, K. (2005). “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 190–200, ACM.

Patil, H., Pereira, C., Stallcup, M., Lueck, G., Cownie, J. (2010). “Pinplay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs”. In Proceedings of the 8th annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 2-11, ACM.

Pin. (2012). Pin 2.13 User Guide. Disponível em: https://software.intel.com/en-us/articles/pintool. Acesso em: 23/05/2014

Reddi, V., Settle, A. M., Connors, D. A., Cohn, R. S. (2004) “Pin: A Binary Instrumentation Tool for Computer Architecture Research and Education.” In Proceedings of the Workshop on Computer Architecture Education, June.

SPLASH-2 (2007). The Modified SPLASH-2. Disponível em: http://www.capsl.udel.edu/splash/. Acesso em 16/07/2014.

Treibig, J., Hager, G., Wellein, G. (2010). “ LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments”. In Proceedings of the 39th International Conference on Parallel Processing Workshops, pp. 207-216.

Wallace, S. and Hazelwood, K. (2007). “Superpin: Parallelizing Dynamic Instrumentation for Real-Time Performance”. In Proceedings of the International Symposium on Code Generation and Optimization, pp. 209–220, IEEE Computer Society.