DCCE - DEPARTAMENTO DE CIÊNCIAS DE COMPUTAÇÃO E ESTATÍSTICA
IBILCE - INSTITUTO DE BIOCIÊNCIAS, LETRAS E CIÊNCIAS EXATAS
UNESP – UNIVERSIDADE ESTADUAL PAULISTA

MEDIÇÃO DE DESEMPENHO DE PROGRAMAS PARALELOS
- ESTENDENDO O PARADYN -

Davidson Rodrigo Boccardo
Henrique Jo Nakashima
José Nelson Falavinha Júnior
Lucas dos Santos Casagrande

Orientador: Prof. Dr. Aleardo Manacero Júnior

São José do Rio Preto, SP - Brasil
Janeiro de 2005

MEDIÇÃO DE DESEMPENHO DE PROGRAMAS PARALELOS - ESTENDENDO O PARADYN -

Davidson Rodrigo Boccardo
Henrique Jo Nakashima
José Nelson Falavinha Júnior
Lucas dos Santos Casagrande

Projeto Final de Curso submetido ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas (Ibilce) da Universidade Estadual Paulista Júlio de Mesquita Filho, como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado por:
Prof. Dr. Aleardo Manacero Jr.

(Presidente)

Prof[a]. Dra. Renata Spolon Lobato

Prof. Dr. Maurílio Boaventura

____

____

____

______________________________ ______________________________

Assinatura dos Orientados Assinatura do Orientador

São José do Rio Preto, SP - Brasil
Janeiro de 2005

DEDICATÓRIA

Aos nossos familiares.

AGRADECIMENTOS

A Deus, por ter nos conduzido até aqui e nos dado forças nas horas em que mais precisávamos.

Aos nossos familiares que sempre estiveram preocupados com nossa realização profissional e desenvolvimento intelectual, fornecendo todas as condições necessárias para tais sonhos.

Aos docentes deste instituto, com maior importância ao orientador Prof. Dr. Aleardo Manacero Júnior, por compartilharem no crescimento ético e intelectual.

Aos amigos que sempre estiveram presentes nos momentos difíceis incentivando e dando apoio necessário.

Aos pesquisadores da universidade de Winsconsin (EUA), responsáveis pelo *Paradyn*, por fornecerem apoio necessário nos momentos de dúvida.

A FAPESP – Fundação de Amparo à Pesquisa do Estado de São Paulo, pelo apoio financeiro para realização deste projeto.

A nós mesmos pela dedicação e empenho impostos no desenvolvimento do projeto, como também pela alegria e descontração de trabalharmos unidos.

RESUMO

A solução de muitos problemas demanda grande poder computacional, que pode ser alcançado com o emprego de programação paralela. No desenvolvimento de sistemas que resolvam tais problemas o desempenho é um fator importante, pois influencia na obtenção de resultados menos ou mais rapidamente, e num menor ou maior gasto com profissionais e máquinas. Com isso a utilização de ferramentas para medir e analisar tais sistemas é algo extremamente importante. Uma ferramenta de destaque na área de análise de desempenho é o *Paradyn*, que realiza instrumentação dinâmica (em tempo de execução) do código executável de uma aplicação.

Nesse projeto apresenta-se uma expansão das capacidades de análise e decisão oferecidas pelo *Paradyn*, permitindo o refinamento do bloco mínimo de análise para o nível dos blocos estruturais internos às funções (como os laços de repetição). A diminuição desse grão de medida significa uma análise mais precisa, determinando com um maior rigor os possíveis pontos a serem corrigidos em um programa paralelo. Ao longo do texto apresentam-se os problemas enfrentados no desenvolvimento desse projeto, as soluções encontradas, bem como os resultados obtidos.

ABSTRACT

The solution for some problems demands a large amount of computing power, which is available through parallel programming. In the development of such systems their performance is an important issue, since it influences the achieved processing speed and the costs with maintenance of personel and machines. Hence the application of measurement and analisys tools in these systems is extremely important. An important tool in the high performance area is Paradyn, which performs dynamic instrumentation over the application's executable code.

On this project it is presented an improvement in the Paradyn's analisys and decision capabilities through the reduction on its instrumentation granularity from function level to the Loop blocks inside functions. This reduction implies on a more meticulous analisys that results in more focused bottlenecks to be corrected. Throughout this text it is presented the problems that must be solved in this implementation, their solutions, as well as the results achieved with loop instrumentation.

ÍNDICE DE FIGURAS

FIGURA 3. 1 - ESTRUTURA BÁSICA DO *PARADYN* ____ 18

FIGURA 3. 2 - *PARADYN MAIN CONSOLE WINDOW* ____ 20

FIGURA 3. 3 - *PARADYN WHERE AXIS WINDOW* ____ 20

FIGURA 3. 4 - *PARADYN PERFORMANCE CONSULTANT WINDOW* ____ 20

FIGURA 3. 5 – *TIME HISTOGRAM WINDOW* ____ 21

FIGURA 3. 6 – *BARCHART WINDOW* ____ 21

FIGURA 3. 7 – *TABLE WINDOW* ____ 21

FIGURA 3. 8 - INTERFACE DE VISUALIZAÇÃO ____ 21

FIGURA 3. 9 – LAÇO SIMPLES ____ 24

FIGURA 3. 10 – LAÇOS ENTRELAÇADOS ____ 25

FIGURA 3. 11 – PSEUDOCÓDIGO DE *INSERELOOP* ____ 26

FIGURA 3. 12 – ENTRADAS E SAÍDAS ADICIONAIS ____ 29

FIGURA 3. 13 - PSEUDOCÓDIGO *DEFINSTPOINT* ____ 31

FIGURA 3. 14 - PSEUDOCÓDIGO DE *TIMERDRAGONSTART* E *TIMERDRAGONSTOP* ____ 33

FIGURA 3. 15 - INSTRUMENTAÇÃO ____ 33

FIGURA 3. 16 - PSEUDOCÓDIGO *INSTRUMLOOPS* ____ 37

FIGURA 3. 17 – RELOCAÇÃO DE CÓDIGO ____ 45

FIGURA 3. 18 – ESQUEMA DE CORES PADRÃO DO *PARADYN* ____ 46

FIGURA 3. 19 – FUNCIONAMENTO DA EXPANSÃO DE BLOCOS NA HIERARQUIA DE ÁRVORE ____ 47

FIGURA 3. 20 – ILUSTRAÇÃO DA INTERFACE PARA O NOVO MÓDULO TRATADO PELO PROJETO ____ 49

FIGURA 4.1 – CÓDIGO DA FUNÇÃO ____ 52

FIGURA 4. 2 – ESTRUTURA DE LAÇOS ____ 52

FIGURA 4. 3 – CÓDIGO *ASSEMBLY* ____ 53

FIGURA 4.4 – CÓDIGO FONTE DO PROGRAMA TESTADO ____ 54

FIGURA 4. 5 – CÓDIGO *ASSEMBLY* DA FUNÇÃO FILHO ____ 56

FIGURA 4.6 - CÓDIGO INSTRUMENTADO DA FUNÇÃO ____ 57

FIGURA 4.7 – CÓDIGO DA FUNÇÃO ____ 58

FIGURA 4. 8 – SAÍDA DO PROGRAMA COM *CLOCK()* ____ 59

ÍNDICE DE TABELAS

TABELA 4.1 – MEDIDAS DE TEMPO FORNECIDAS PELO *PARADYN* ________________ 59

TABELA 4.2 – COMPARATIVO ENTRE TEMPOS FORNECIDOS PELO *PARADYN* E *CLOCK()* ________________ 59

LISTA DE ABREVIATURAS E SIGLAS

COND - *condition*

DM - *Data Manager*

DISP - *displacement*

dis - *disassemble an object file*

ELF - *Executable and Linkable Format*

FAPESP - Fundação de Amparo à Pesquisa do Estado de São Paulo

IGEN - *Interface Generator*

imm22 - *22-bit immediate*

MPI - *Message Passing Interface*

NOP - *No Operation*

OP3 - *6-bit operation code*

PC - *Performance Consultant*

PPS - *Pacotes Por Segundo*

PVM - *Parallel Virtual Machine*

RD - *Destination Register*

RPC - *Remote Procedure Call*

RS1 - *Source 1 Register*

RS2 - *Source 2 Register*

simm13 - *signed 13-bit immediate*

SPARC - *Scalable Processor ARChitecture*

TCL - *Tool Control Language*

TK - *Tool Kit*

TPS - Transações Por Segundo

UI - *User Interface*

VM - *Visualization Manager*

ÍNDICE GERAL

RESUMO I

ABSTRACT II

ÍNDICE DE FIGURAS III

ÍNDICE DE TABELAS IV

LISTA DE ABREVIATURAS E SIGLAS V

CAPÍTULO 1 1

INTRODUÇÃO 1
1.1 Objetivos 2
1.2 Organização da monografia 2

CAPÍTULO 2 3

FUNDAMENTAÇÃO TEÓRICA 3
2.1 Análise de Desempenho 3
2.1.1 Técnicas baseadas em modelagem analítica 4
2.1.2 Técnicas baseadas em simulação 5
2.1.3 Técnicas baseadas em *benchmarking* 5
2.2 Medidas de Desempenho 6
2.2.1 Medidas dependentes de velocidade 7
2.2.2 Medidas independentes de velocidade 8
2.3 Estratégias de instrumentação 9
2.3.1 Classificação segundo a forma de medição e instrumentação 9
2.3.2 Classificação segundo os modelos dos dados gerados 11
2.4 Paradyn 13
2.5 Considerações finais 15

CAPÍTULO 3 16

DETALHAMENTO E DESENVOLVIMENTO DO PROJETO 16
3.1 Estrutura do Paradyn 17
Paradyn Front-end 18
Data Manager 19
User Interface 19
Performance Consultant 20
Visi Manager 20
Paradyn Daemon 22
3.2 Escolha do Dimensionamento do Grão de Medida 23

3.3 Identificação de laços ____ *24*
3.4 Identificação dos pontos de instrumentação ____ *29*
3.5 Correção de deslocamentos ____ *31*
3.6 Instrumentação ____ *32*
3.7 Integração do projeto ao Paradyn ____ *37*
3.7.1 Comunicação entre o *front-end* e o *daemon* ____ 38
3.7.2 Acesso à imagem da função ____ *42*
3.7.3 Memória Compartilhada ____ 42
3.7.4 Inclusão de funções em bibliotecas anexadas na aplicação pelo *Paradyn* ____ 43
3.7.5 Relocação de função ____ 44
3.8 Interface Gráfica ____ *46*
3.9 Considerações finais ____ *50*

CAPÍTULO 4 ____ **51**

TESTES E RESULTADOS ____ 51
4.1 Teste da identificação de laços ____ *51*
4.2 Teste da inserção do código de instrumentação ____ *54*
4.3 Teste dos tempos medidos ____ *58*
4.4 Considerações finais ____ *60*

CAPÍTULO 5 ____ **61**

CONCLUSÕES E TRABALHOS FUTUROS ____ 61
5.1 Conclusões ____ *61*
5.2 Trabalhos futuros ____ *62*

REFERÊNCIAS BIBLIOGRÁFICAS ____ **63**

CAPÍTULO 1

INTRODUÇÃO

Em qualquer sistema, computacional ou não, o desempenho é um fator determinante durante todo seu desenvolvimento, mas nem sempre o projetista se preocupa em fazer com que esse desempenho seja ótimo. Em sistemas computacionais, um melhor ou pior desempenho significa menos ou mais dinheiro gasto com manutenção de profissionais e máquinas ou resultados obtidos menos ou mais rapidamente. Desse modo, a busca pelo desempenho ótimo é crucial para sistemas que sejam muito utilizados ou que representem uma carga computacional muito elevada. Sistemas paralelos e/ou distribuídos, usualmente denominados sistemas de alto desempenho, pertencem a essa categoria, pois em geral são sistemas aplicados em problemas com grande carga computacional.

Dada a importância em se obter o melhor desempenho possível para um determinado sistema, é necessário que esse desempenho seja definido e medido de alguma forma. O padrão de referência de desempenho pode variar de sistema para sistema, envolvendo parâmetros como tempo, taxas de acerto em *caches*, etc., embora exista uma tendência forte de que o principal parâmetro deva ser tempo de execução [Sah 1996]. Para melhorar tais medidas normalmente são usadas ferramentas para medição e análise de desempenho, porém tais ferramentas têm seu uso limitado ao conjunto de aplicações para que foram projetadas.

Essa limitação implica, na realidade, na necessidade de seu usuário decidir *a priori* quais serão as medidas de desempenho de seu interesse e como elas poderão ser obtidas. Essa não é uma tarefa fácil dada a grande quantidade de medidas possíveis e de métodos para obtê-las. Uma técnica para resolver essa limitação, no caso de desempenho de programas

paralelos, é permitir a alteração dinâmica das medições durante o processo de análise, como faz a ferramenta *Paradyn* [Mil 1995].

1.1 Objetivos

Nesse projeto apresenta-se uma expansão das capacidades de análise e decisão oferecidas pelo *Paradyn*, permitindo o refinamento do bloco mínimo de análise para o nível dos blocos estruturais internos às funções (como os laços de repetição).

A atual versão dessa ferramenta não possibilita um alto grau de detalhes na determinação de gargalos de desempenho em programação paralela. Um dos fatores que colaboram para isso é o fato do *Paradyn* trabalhar com grãos do tamanho de funções, dificultando assim uma localização precisa dos reais responsáveis pelo consumo de tempo dentro da função apontada como gargalo. A diminuição desse grão de medida significaria uma análise mais precisa, a qual apontaria com um maior rigor os possíveis pontos a serem corrigidos em um programa paralelo.

1.2 Organização da monografia

No capítulo dois é descrita a fundamentação teórica necessária para a realização do projeto, focando-se a análise de desempenho de sistemas. As etapas de desenvolvimento do projeto, incluindo a especificação da própria codificação são descritas no capítulo três.

A seguir, no capítulo quatro são apresentados os testes e verificação da funcionalidade do trabalho realizado. Finalmente, o capítulo cinco expõe as conclusões e trabalhos que possam vir a ser realizados futuramente.

CAPÍTULO 2

FUNDAMENTAÇÃO TEÓRICA

No desenvolvimento do projeto foram estudadas técnicas para análise de desempenho, as quais são descritas na seção 2.1, com o objetivo de estudar as ferramentas existentes, compreendendo seu sistema e determinando o dimensionamento de seus componentes, de modo a identificar pontos para melhoria no desempenho.

A análise (ou avaliação) de desempenho é composta pela identificação de medidas que mostrem a eficiência de sistemas de *software*, com o objetivo de obter dados relevantes para o processo de avaliação do sistema, como também, pela identificação de estratégias de instrumentação. Esses tópicos são apresentados na seção 2.2.

As estratégias de instrumentação, descritas na seção 2.3, podem ser separadas quanto a forma de instrumentação e medição do programa, e ainda quanto aos modelos de dados gerados para análise. O estudo dessas estratégias de instrumentação permitiu definir qual delas seria a mais eficiente e eficaz para a interação com o *Paradyn*, descrito na seção 2.4, que é uma ferramenta para análise de desempenho baseada na instrumentação dinâmica das medidas a serem realizadas no programa. Em outras palavras, o programa a ser analisado é submetido, em tempo de execução, a modificações em seu código executável, permitindo que, durante sua execução, métricas diferentes sejam aplicadas.

2.1 Análise de Desempenho

A análise de desempenho consiste na execução de procedimentos, computacionais ou não, visando quantificar o comportamento de um determinado sistema, seja ele *hardware* ou *software*. As técnicas de análise

de desempenho podem ser classificadas em dois grupos: técnicas que estudam sistemas de *hardware* e técnicas que estudam sistemas de *software*.

As técnicas de avaliação de *software* medem o desempenho de programas com o objetivo de identificar funções, ou blocos de instruções, que representem gargalos de execução no sistema. As técnicas de avaliação de *hardware* são as que se preocupam em analisar o desempenho das máquinas em que são executados os programas paralelos.

Tanto para a avaliação de desempenho de *hardware*, como de *software*, pode-se dividir tais técnicas em três grupos, segundo a forma de obtenção dos resultados: técnicas baseadas em modelos analíticos, técnicas baseadas em simulação e técnicas baseadas em medição (*benchmarking*) [Jai 1991].

2.1.1 Técnicas baseadas em modelagem analítica

São aquelas baseadas em modelos analíticos, como por exemplo modelos de redes de Petri, em que o sistema é definido como uma seqüência de estados com transições atreladas a equações matemáticas.

Este tipo de técnica é amplamente utilizado nos períodos de elaboração e desenvolvimento de sistemas ou quando o alvo da avaliação (análise) não se encontra disponível. Possui baixo custo e velocidade de resposta variável, pois a velocidade está intimamente ligada à complexidade do modelo (quanto maior a complexidade, maior o tempo necessário para otimização e resolução das equações).

A precisão dos resultados fornecidos por este tipo de técnica depende do processo matemático envolvido na criação do modelo. Caso o equacionamento matemático e a otimização estejam corretos, os resultados fornecidos serão precisos. Um outro fator que interfere na precisão desses é a quantidade de simplificações e suposições (quanto maior o número de simplificações e suposições, menor é a precisão dos resultados).

2.1.2 Técnicas baseadas em simulação

Essas técnicas consistem no processo de modelagem matemática ou lógica do comportamento de sistemas, na tentativa de representar fielmente a realidade. Essas técnicas assemelham-se muito às técnicas analíticas, em que a diferença básica está na maneira como os resultados são obtidos: nas técnicas baseadas em simulação, substituem-se os sistemas de equações que definem o sistema por regras que determinem seu comportamento.

Um grande problema no uso de simulação é a dificuldade na obtenção de um modelo que seja fiel ao sistema real, e que produza resultados confiáveis. Esse problema pode ser corrigido através de refinamento do modelo a ser simulado (através de dados de *benchmarks*, por exemplo), e técnicas de validação dos dados de saída.

Em contrapartida, a simplicidade na elaboração do modelo de simulação, a flexibilidade e a possibilidade de tratamento de parâmetros difíceis de dimensionar analiticamente são fatores a favor da utilização deste tipo de técnica.

2.1.3 Técnicas baseadas em *benchmarking*

As técnicas de *benchmarking* são baseadas em medições reais dos sistemas em avaliação. Os dados são obtidos através da execução real dos programas (tanto na avaliação de sistemas de *hardware* quanto na avaliação de sistemas de *software*).

Estas técnicas apresentam, dentre todas, a menor complexidade na implementação e grande flexibilidade (desde que se tenha sistemas e programas disponíveis para teste). Por outro lado, elas dependem da disponibilidade da máquina na qual serão realizadas as medições, da sua velocidade e do número de repetições necessárias para obtenção de dados confiáveis, o que pode elevar seu custo operacional. Em determinadas

situações o uso de *benchmarking* torna-se proibitivo, pois a realização dos testes implica na multiplicação do *hardware* de teste (máquinas reais).

Muitas vezes, a criação de um programa *benchmark* esbarra em problemas que vão desde a quantificação da carga aplicada ao sistema até a escolha dos procedimentos de medição. Fazem parte desse conjunto os chamados *microbenchmarks* e os *benchmarks* proprietários (desenvolvidos por analistas para sistemas pré-determinados).

2.2 Medidas de Desempenho

Juntamente com a seleção da ferramenta ou técnica utilizada, a escolha da medida ou do conjunto de medidas de desempenho representa um fator chave durante o processo de avaliação. Uma escolha equivocada das medidas pode fazer com que os resultados sejam errôneos, não demonstrando o comportamento real do sistema.

Dentro do conjunto de métricas para avaliação de desempenho algumas se destacam como padrões, por serem utilizadas por grande parte dos analistas de desempenho ou por estarem inseridas em ferramentas de avaliação, enquanto outras são úteis por serem aplicadas a problemas específicos. Neste sentido, existe um consenso sobre a não existência de uma medida universal (que possa ser aplicável a todos os problemas), mas de medidas que são melhores ou pior adequadas aos problemas de avaliação.

Quando o sistema alvo de avaliação de desempenho é um sistema serial[1], um determinado conjunto de medidas pode ser utilizado na avaliação. Exemplos de medidas aplicadas a esses sistemas são: carga no sistema, porcentagem de falha de paginação, tempo de espera em operações de entrada e saída. Quando o alvo envolve sistemas paralelos, além das medidas para avaliação de sistemas seriais (devidamente modificadas)

[1] Sistemas de hardware baseado na máquina de Von Neumann e sistemas de software que possuam linha de execução seqüencial.

outras medidas devem ser empregadas, para atender características específicas, como por exemplo, memória compartilhada. Exemplos dessas medidas são escalabilidade e aceleração (*speedup*).

As medidas de desempenho podem ser divididas de várias maneiras. Neste trabalho decidiu-se tomar como parâmetro de classificação a dependência (ou não) das medidas em relação à velocidade de execução dos sistemas. Tal parâmetro foi escolhido pois o tempo de execução de um sistema é considerado um fator de grande valia [Sah 1996]. A partir desse parâmetro, podem-se classificar as medidas de desempenho em dependentes ou independentes de velocidade.

2.2.1 Medidas dependentes de velocidade

Medidas dependentes de velocidade são aquelas de alguma forma ligadas ao tempo de execução dos sistemas, sejam eles paralelos ou não. As duas medidas mais importantes, quando relacionados a sistemas paralelos, são o *speedup* e a eficiência.

Aceleração (*speedup*)

O fator que mais justifica a necessidade da utilização de sistemas paralelos é o possível ganho de velocidade (e conseqüente diminuição do tempo de execução) dos sistemas. Sendo assim, o *speedup* (ou aceleração), que representa o ganho de velocidade no processamento, torna-se a métrica de maior importância quando analisados desses sistemas.

A partir do conceito de ganho de velocidade, várias abordagens foram criadas para o cálculo do *speedup e* com isso outras medidas são consideradas significativas, com intuito de melhor traduzir o desempenho de sistemas paralelos. São elas eficiência, redundância, utilização e qualidade de paralelismo.

Eficiência

A eficiência é uma medida que busca apresentar quanto o ganho de velocidade (ou aceleração) no processamento está próximo ou não do ótimo. A eficiência pode variar de zero a cem por cento, mas dificilmente o valor máximo da eficiência é obtido, pois sempre ocorrem perdas que impedem tal resultado (sobrecarga na comunicação e de sincronismo, por exemplo).

Redundância e utilização

A redundância busca averiguar se o programa em questão foi paralelizado de forma adequada em relação ao *hardware* disponível. Já a utilização de um sistema indica a percentagem de recursos (processadores, memória, etc.) que permanecem ocupados durante a execução de um determinado programa paralelo.

Qualidade de paralelismo

A qualidade de paralelismo apresenta ao analista de desempenho uma medida mais completa sobre as interações do conjunto programa/máquina, ao estabelecer relações entre o quanto se pode aumentar a velocidade de processamento e quanto o programa está bem (ou mal) estruturado.

2.2.2 Medidas independentes de velocidade

Medidas independentes de velocidade compõem um conjunto de medidas que normalmente são utilizadas para quantificar o desempenho de subsistemas específicos ou de sistemas de *software*.

Quando o alvo da avaliação são sistemas de *hardware*, são utilizadas métricas que mostram a ocupação de registradores, falta de dados em *cache*, movimentação na memória (medidas que demonstram a quantidade de memória livre, por exemplo), além de outras como TPS (transações por

segundo) e PPS (pacotes por segundo), utilizadas na avaliação de comunicação entre computadores, entre outras.

Na avaliação de sistemas de *software* são utilizadas métricas que fornecem informações sobre a execução dos programas, ou seja, quanto determinado trecho do programa ocupa do tempo total de execução, número de ocorrências de determinadas funções, tempo gasto em comunicação, etc. Este conjunto torna-se vasto à medida que o número de aplicações aumenta, e algumas medidas podem assumir diferentes definições, por serem utilizadas na avaliação de desempenho de diferentes sistemas.

2.3 Estratégias de instrumentação

Duas estratégias de classificação podem ser definidas, conforme a forma de instrumentação e medição e quanto aos modelos de dados gerados para análise.

2.3.1 Classificação segundo a forma de medição e instrumentação

Pierce e Mudge [Pie 1994] apresentam uma classificação partindo do princípio que é possível obter dados sobre desempenho de sistemas de duas formas: através de monitoração (por *hardware* ou *software*) ou através de instrumentação de código (fonte, objeto ou executável) do programa, como descrito a seguir.

Monitoração por hardware

É feita pelo uso de analisadores lógicos acoplados diretamente ao processador ou ao barramento do sistema para obtenção dos dados desejados.

Embora possa obter medidas precisas, a monitoração não é adequada para análise de desempenho devido ao alto custo envolvido em sua implementação, à necessidade de um dispositivo especial para armazenamento dos dados e pela exigência de profissional especializado no *hardware* de monitoramento.

Monitoração por software

A forma mais simples de realizar monitoração por *software* consiste no uso das interrupções para gravação das informações sobre a execução dos programas. Este tipo de monitoração é desaconselhado por ser extremamente lento quando o número de interrupções e de referências à memória durante a execução é elevado.

Modificação do código fonte

Consiste na forma mais fácil de instrumentação, na qual comandos especiais são inseridos no código fonte do programa. Esses comandos farão as medições desejadas. Este tipo de modificação não é possível quando não há disponibilidade do código fonte ou quando o tamanho do código a ser modificado é expressivo ou em alguns casos, quando existe um grande número de referências às bibliotecas e rotinas de sistema.

Modificação do código objeto

A modificação é realizada inserindo os comandos para medições no código objeto. Pode ser realizada por um editor específico que reescreve o código na linguagem original (ou uma linguagem legível) e depois refaz a alocação do código e dos dados para torná-lo objeto.

Modificação do código executável

É realizada através da inserção de comandos já no código executável e é considerada como a forma mais complicada de inserção de código, pois

exige que o usuário possua um descompilador e um programa que realize adição do código.

2.3.2 Classificação segundo os modelos dos dados gerados

Com o objetivo de reduzir o impacto do código acrescido ao programa pelos métodos apresentados anteriormente, foram criadas várias formas de se implementar e controlar a invasão (código de instrumentação estranha ao programa). Daniel Reed [Ree 1994] apresenta uma divisão dos métodos de instrumentação a partir do tipo de informação resultante da medição e da forma como é realizada. O autor descreve quatro categorias de métodos de medição: *profiling*, contagem de eventos, medição de intervalos de tempo e extração de traços de eventos.

Profiling

Profiling é a técnica mais usada e cujo princípio de funcionamento é a obtenção de informações sobre quanto cada trecho do programa ocupa do tempo total de execução. Apesar de problemas com a precisão dos intervalos de amostragem para a determinação do uso de cada trecho e do tratamento de recursividade, ferramentas como o Gprof [Gra 1982], Dpat, Prof e Jprof [Rei 1994] entre outras, são utilizadas para medidas de desempenho de programas por *profiling*.

Contagem de eventos

Apesar de eliminar a necessidade de amostragem, é considerado um método muito mais invasivo, pois seu funcionamento está baseado na modificação do código para que sejam contadas as ocorrências de determinados eventos de interesse (chamadas de uma rotina, por exemplo). Isso significa um acréscimo no tempo de execução do programa e, também, das instruções que estiverem sendo "contadas". Essa carga computacional

adicionada e o excessivo volume de dados gerados acabam por fazer com que esse método não seja muito usado.

Medição de intervalos de tempos

Esse método é uma variação pouco usada de *profiling*, em que se trocam as amostragens do primeiro por chamadas para medir o relógio do sistema, inseridas no código do programa. Os problemas dessa técnica são: a precisão do relógio do sistema, que é atualizado numa baixa freqüência, e a possibilidade de se medir intervalos espúrios em sistemas com compartilhamento de tempo da CPU.

Extração de traços de eventos

Também conhecido como *event tracing*, é outro método bastante utilizado e presente em ferramentas como IDTrace [Pie 1994] e ALPES [Kit 1994], por exemplo. Fornece os resultados mais detalhados, pois se baseia em registros datados da ocorrência de cada evento no sistema. Infelizmente, paga-se um preço alto por essa precisão, pois o volume de dados é demasiadamente grande pela alta freqüência em que eventos ocorrem.

Das técnicas de instrumentação de código indicadas, tem-se que *profiling* e *event tracing* são as mais usadas nas ferramentas de análise de desempenho disponíveis. Existem diversas propostas para esse tipo de instrumentação, algumas já integrantes de sistemas comerciais, como Pixie [Mip 1989], criado por Earl Killian para sistemas MIPS, Prof e Gprof, disponíveis em sistemas UNIX. Várias outras têm sido propostas, principalmente fazendo a instrumentação no código executável, tais como: Nixie e Epoxie [Wal 1992], desenvolvidas pela Digital, Qp e Qpt [Lar 1992], desenvolvidas na Universidade de Wisconsin, Jprof, pela Mentor Graphics, Etnus Totalview, pela Etnus [Etn 2002], Rsim, na Universidade de Rice [Hug 2002], ou ainda o P3T+, do projeto Vienna [Fah 2000].

Todas essas ferramentas procedem a análise de modo estático, sobre o que se denomina dados *post-morten*, ou seja, dados que serão examinados quando o programa não estiver mais sendo executado. Uma abordagem diferente é fazer as medições diretamente no programa em execução, o que permite alterar os dados que interessam ao usuário automaticamente, sem a necessidade de se extrair um novo *profile*, por exemplo. Uma ferramenta nessa categoria é o *Paradyn*, proposto por Miller [Mil 1995].

Ferramentas como o *Paradyn* reduzem, em muito, a necessidade de conhecimento do usuário sobre análise de desempenho, uma vez que as medidas podem ser moldadas de maneira bastante flexível, com o programa em execução. Isso evita o trabalho do usuário em determinar, a *priori*, que pontos do programa devem ser analisados e de que forma os dados serão obtidos. Como essa tarefa é bastante complexa quando se fala de programas paralelo-distribuídos, que é a área de aplicação do *Paradyn*, pode-se perceber claramente suas vantagens.

2.4 *Paradyn*

Paradyn é uma ferramenta utilizada na análise de desempenho de programas paralelos e distribuídos de grande escala (programas que possuem tempo de execução elevado, chegando a executar por várias horas ou dias, mesmo quando executado em máquinas com número elevado de nós de processamento). *Paradyn* realiza a instrumentação de código executável da aplicação dinamicamente, buscando extrair traços dos eventos sem que o usuário precise alterar o código fonte do programa ou necessite de um compilador especial. A instrumentação e a análise de desempenho são realizadas durante a execução do programa aplicação, em tempo real, disponibilizando informações ao usuário assim que elas são obtidas. Existe também a possibilidade da análise dos arquivos gerados após a execução do programa (análise *post-morten*), embora isso não seja objetivo da ferramenta.

A instrumentação é feita através da criação de trampolins, em que chamadas de função são substituídas por desvios incondicionais para medições do *Paradyn*. Como essas substituições ocorrem sem a alteração do restante do código, torna-se possível manter o programa em execução mesmo durante a operacionalização dessas alterações, viabilizando portanto sua instrumentação dinâmica.

A ferramenta decide quais dados serão coletados enquanto o programa está sendo executado, e através da biblioteca de instrumentação dinâmica, cujo nome é *DyninstAPI*, adiciona-se pequenas porções de instrumentação no código executável da aplicação para procurar possíveis pontos de degradação de desempenho (como gargalos de comunicação ou entrada/saída). A cada ponto de degradação encontrado a ferramenta adiciona uma quantidade maior de instrumentação para pesquisar o possível problema.

O *Paradyn* usa os conceitos de *metric-focus grid* (grade orientada a medida) e *time-histogram* (histograma de tempo) para selecionar, analisar e apresentar dados sobre desempenho. A grade é um conjunto de dois vetores, um que armazena uma lista de medidas de desempenho (como tempo de CPU, taxa de mensagem, taxa de utilização de entrada/saída), e outro que contêm uma lista de componentes de programa (processos, disco, canais de comunicação e instâncias de barreira). O produto dos dois vetores produz uma matriz com cada medida listada para cada componente do programa, e o usuário pode selecionar um desses pares ordenados para observar seu comportamento.

Em sua versão atual o *Paradyn* é capaz de identificar gargalos de execução em programas paralelos com uma granulação média (funções). Os trabalhos mais recentes em seu desenvolvimento estão restritos a pesquisas sobre técnicas mais eficientes para a identificação dos gargalos de execução, tais como a introdução de grafos de chamadas de função [Cai 2000], uso de dados históricos [Kar 1999], ou ainda por amostragem da

pilha de chamadas [Rot 2002]. Outros trabalhos em andamento envolvem o porte da ferramenta para outros ambientes, como o AIX, em adição ao Solaris, Linux e IRIX.

2.5 Considerações finais

Este capítulo tratou das principais técnicas para análise de desempenho e algumas estratégias de instrumentação estudadas. Com isso obteve-se a fundamentação teórica necessária para o desenvolvimento do projeto, e conseguiu-se definir a estratégia de instrumentação mais eficiente e eficaz para a interação com o *Paradyn*.

A descrição dos principais módulos desenvolvidos, assim como o detalhamento do trabalho realizado são apresentados no próximo capítulo, no qual se detalha ainda as principais estruturas do *Paradyn*.

CAPÍTULO 3

DETALHAMENTO E DESENVOLVIMENTO DO PROJETO

A atual versão do *Paradyn* (com implementação para a arquitetura SPARC) não permite um alto grau de detalhes na determinação dos pontos de degradação de desempenho em programação paralela. Um dos fatores que colaboram para isso é o fato desta ferramenta trabalhar com uma granulação média (funções), dificultando assim uma localização exata dos reais responsáveis pelo consumo de tempo dentro da função apontada como gargalo. A diminuição desse grão de medida significaria uma análise mais precisa, a qual apontaria com maior rigor os possíveis pontos a serem corrigidos para melhorar o desempenho em um programa paralelo.

Para tanto, propõe-se uma expansão das capacidades de análise e decisão oferecidas pelo *Paradyn*, permitindo o refinamento do tamanho de grão de medida (bloco mínimo analisado) para o nível dos blocos internos às funções (laços de execução).

Este capítulo apresenta uma descrição sucinta das principais estruturas do *Paradyn,* bem como o detalhamento dos principais módulos desenvolvidos no projeto. Assim sendo, o capítulo se encontra organizado em oito seções. Na seção 3.1 é apresentada uma descrição da estrutura do *Paradyn,* de um modo geral. A partir desta seção seguem-se os principais módulos implementados neste projeto. Na seção 3.2 é descrito o processo de escolha do grão de medida a ser utilizado. Na seção 3.3, apresenta-se um estudo sobre os possíveis padrões de laços encontrados, bem como explicita o processo de identificação e gerenciamento dos mesmos. Na seção 3.4 é abordada uma descrição sobre pontos de instrumentação, e de como os

mesmo são tratados. Na seção 3.5, apresenta-se o modo de correção dos deslocamentos utilizados em algumas instruções. Na seção 3.6 é explicado processo de instrumentação de laços, enquanto que na seção 3.7 descreve-se a integração do projeto desenvolvido com o *Paradyn*. E finalmente, a seção 3.8 apresenta a modelagem de uma possível interface gráfica para esse novo módulo desenvolvido.

3.1 Estrutura do *Paradyn*

O *Paradyn* é composto por vários subsistemas (figura 3.1), dentre os quais se destacam *Paradyn* e *Paradynd* [Par 2003A, Par 2003B]. *Paradyn* nada mais é que o *front-end*, o qual coordenará todo o processo de coleta de dados e análise de desempenho da aplicação. O *Paradynd* (*back-end* da ferramenta) é o processo *daemon* responsável pela efetivação da instrumentação e medições em todas as máquinas definidas para avaliação, realizando basicamente as funções listadas a seguir:

- começar e controlar a execução dos processos da aplicação;
- ler a tabela de símbolos da aplicação;
- ler a imagem binária da aplicação com a finalidade de encontrar os pontos de instrumentação;
- avaliar métricas, gerar código, e inserir instrumentação na aplicação;
- periodicamente coletar dados de desempenho vindos da aplicação e redirecionar tais valores para os módulos do *front-end*.

Paradyn Front-end

O *Paradyn front-end* é um sistema *multi-threads* composto basicamente por: *Data Manager* (*DM*), *Performance Consultant* (*PC*), *User Interface* (*UI*) e *Visualization Manager* (*VM*).

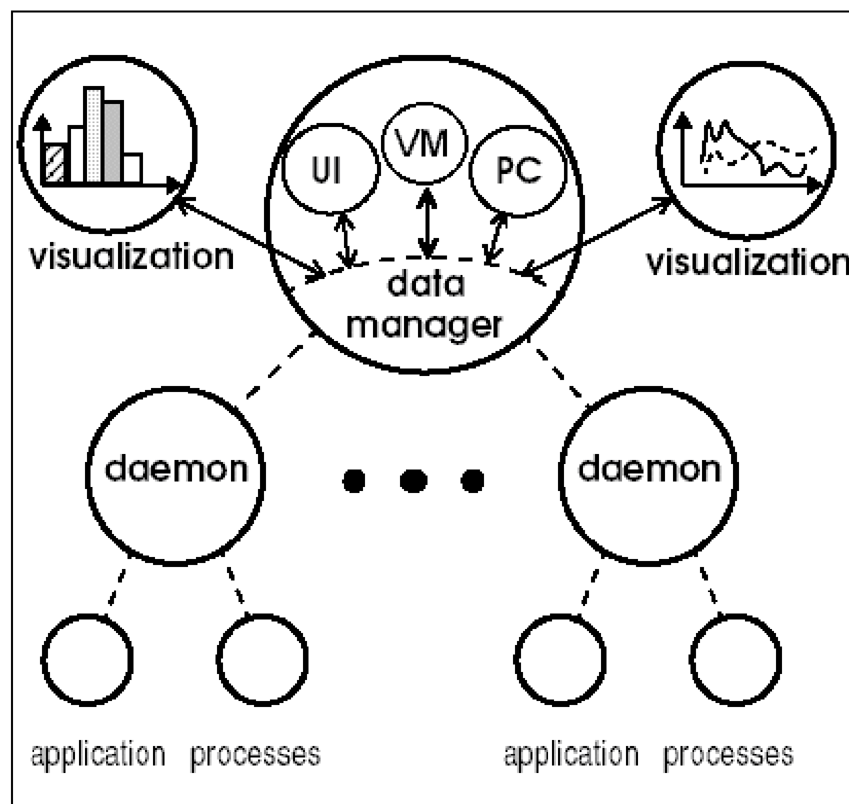


Figura 3. 1 - Estrutura básica do *Paradyn*

Quando um processo *Paradyn* é executado, cada um desses *threads* é criado e iniciado. O *thread* responsável por interagir com o usuário será o *User Interface*, ao qual o usuário passa informações como o caminho do aplicativo a ser executado e diversas especificações do processo de análise , além dos tipos de medidas a serem realizadas. Este *thread* envia estas informações para o *Data Manager*, que cria um canal de comunicação com o *thread Performance Consultant*, o qual especifica aos processos *daemons*,

por intermédio do *Data Manager*, os tipos de métricas e o foco da medição, entre outros parâmetros necessários para a coleta de dados. Os vários processos de visualização, dos dados medidos, são gerenciados pelo *Visi Manager*, que cuida da comunicação entre cada um destes processos com o *Data Manager*.

Data Manager

Tem um papel central dentro da estrutura do *Paradyn*, gerenciando requisições dos outros *threads* para coleta de dados de desempenho, recebendo dados dos *daemons* e entregando-os aos *threads* solicitantes, além de manter informações sobre as métricas e hierarquias de recursos da aplicação em execução.

Requisições de coleta de dados feitas por um *thread* usam o procedimento público *dataManager::enableDataRequest*, que dentre os parâmetros passados estão:

- **par *metric/focus* –** tipo de métrica e o foco a serem habilitados;
- ***perfStreamHandle* do *thread* requisitante -** um objeto provido pelo *Data Manager* que abstrai um canal de comunicação, pelo qual os dados são transmitidos do(s) *daemon*(s) aos *threads* solicitantes.

User Interface

Responsável basicamente por receber comandos dos usuários e gerenciar as janelas gráficas (como *Paradyn Main Console Window*, *Where Axis*, e *Performance Consultant*, ilustradas respectivamente nas figuras 3.2, 3.3 e 3.4).

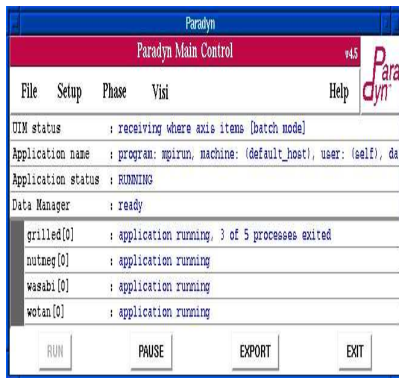


Figura 3. 2 - ***Paradyn Main Console Window***

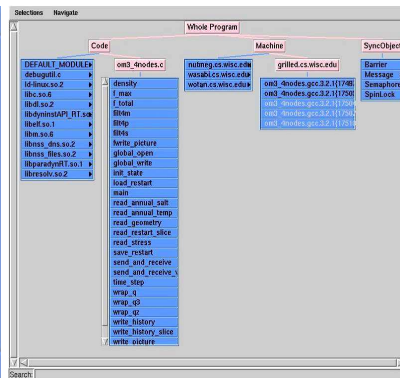


Figura 3. 3 - ***Paradyn Where Axis Window***

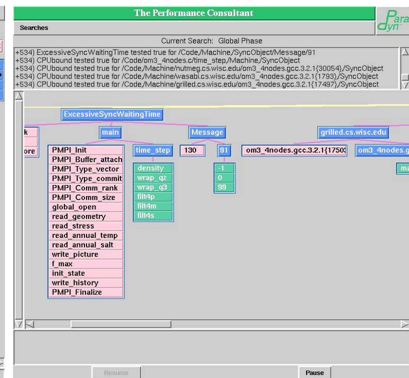


Figura 3. 4 - ***Paradyn Performance Consultant Window***

A manipulação de todas as janelas gráficas é feita por meio de pacotes *TCL/TK*. Simultaneamente, o *thread* de interface com o usuário aguarda eventos gráficos gerados por estes pacotes, mensagens vindas do *Data Manager*, além de outros eventos, como os gerados pelo teclado e/ou pelo mouse.

Performance Consultant

Coordena o processo de busca automática por gargalos de desempenho. Para tanto, interage com o *Data Manager*, ao qual habilita (desabilita) pares de métricas/focos e troca informações sobre recursos. A partir dessa interação requisita e recebe os dados das instrumentações efetuadas pelos *daemons*. Esses dados passam por alguns filtros (como *PCfilter, PCmetricInst* e *experiment*), e alguns testes de desempenho são efetuados sobre os mesmos a fim de definir e guiar a busca por funções que realmente sejam gargalos. Outra interação do *PC thread* se dá com o *User Interface*, a fim de controlar o conteúdo da janela *Performance Consultant.*

Visi Manager

Entre suas principais funções estão as de gerenciar diversos processos de visualização (como *Time Histogram Visi, Bar Chart Visi* e *Table Visi*, janelas gráficas ilustradas respectivamente pelas figuras 3.5, 3.6 e 3.7) e intermediar a comunicação entre estes processos e o *DM*.

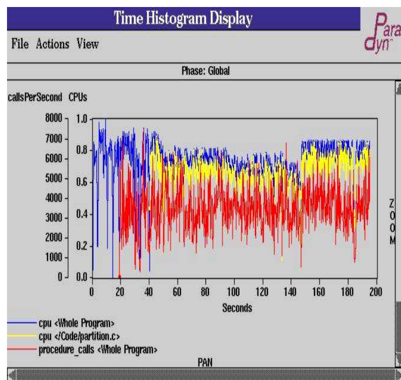


Figura 3. 5 – *Time Histogram Window*

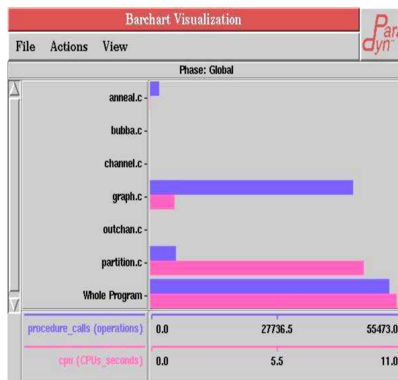


Figura 3. 6 – *Barchart Window*

Figura 3. 7 – *Table Window*

Quando uma nova requisição de visualização é solicitada, um *visi thread* é criado, e a partir dele um processo externo de visualização é iniciado (utilizando pacotes *TCL/TK*). Assim uma interface entre este processo e o *Paradyn* é estabelecida, permitindo que as visualizações solicitem/recebam dados de desempenho dos outros *threads*, e com isso atualizem suas informações a respeito das medições. A figura 3.8 ilustra este módulo que controla os processos de visualização.

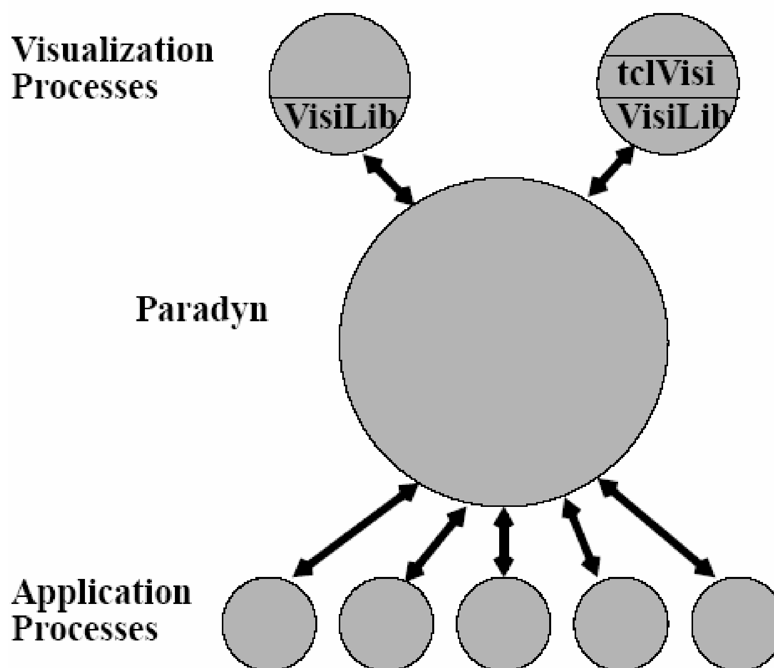


Figura 3. 8 - Interface de visualização

Paradyn Daemon

O *Paradynd* (*back-end* da ferramenta) é o processo *daemon* responsável pela efetivação da instrumentação e medições em todas as máquinas definidas para avaliação. Em cada uma dessas máquinas haverá um *daemon* gerenciando a execução de um programa paralelo. A comunicação entre os *daemons* e o *front-end* é feita por meio de chamadas tipo RPC, geradas automaticamente pelo *IGEN* (seção 3.7.1 - *IGEN*).

Os *daemons* são inicializados pelo *Paradyn front-end*, o qual passa como argumento o tipo de *daemon* (PVM, MPI, etc.), o nome da máquina onde o *front-end* está executando e o endereço do *socket* para conexão. Por esta conexão estabelecida com o *front-end* é que os *daemons* recebem e servem as requisições de instrumentação, além de realizarem notificações como a de criação de novos recursos (como processos criados durante a execução do programa).

A fim de manipular os processos de uma aplicação sem se preocupar com a arquitetura da máquina utilizada, o *Paradyn daemon* utiliza a classe *process*. Desta forma, apenas alguns métodos desta classe possuem uma implementação específica para cada tipo de plataforma.

O espaço de memória utilizado pelos *daemons* para armazenar o código de instrumentação é denominado *inferior heap.* Para manipular tal bloco de memória existe a classe *inferiorHeap*, utilizada no momento de realocação de funções.

O *Paradyn daemon* lê o código objeto do programa, analisando sua tabela de símbolos. Para a realização de tarefas relacionadas à identificação dos pontos de instrumentação, bem como a própria inserção da mesma, o *daemon* utiliza várias classes, principalmente *pd_Function, Image* e *Object*. Esta última é responsável pela manipulação do executável (utilizando a biblioteca *ELF*) e gerenciando processo de leitura da tabela de símbolos.

Posteriormente essas informações irão alimentar a classe *image*, que representa o código executável do programa. Ela utiliza as informações coletadas da tabela de símbolos para identificar as funções (representadas pela classe *pd_Function*), e assim localizar possíveis pontos sujeitos a instrumentação. Estes pontos são representados pela classe *instPoints*, e pelo método *findInstPoints* (*pd_Function*), com os quais definem-se os endereços de tais pontos, as instruções a serem relocadas e outras informações relevantes.

A classe i*mage* utiliza um objeto do tipo *dictionary_hash* (classe que implementa uma tabela *hash*), que armazena todos os objetos *pd_Function*. Sempre que um *Daemon* encontra uma nova chamada de função ainda não instrumentada, ele notifica ao *Performance Consultant*, que por sua vez toma a decisão da necessidade ou não de novas instrumentações. Assim que esse processo de coleta de dados e identificação de gargalos de execução esgota-se, o *Performance Consultant* reporta todas as informações necessárias ao *Data Manager*, o qual as destina ao *Visi Manager*, responsável pela apresentação ao usuário das informações analisadas.

3.2 Escolha do Dimensionamento do Grão de Medida

A escolha de um melhor dimensionamento do grão de medida teve por base o exame da forma de medição aplicada ao atual grão mínimo adotado pelo *Paradyn*, e como esta pode ser estendida para grãos menores. Isso permitiu o entendimento exato do menor elemento de medição e quais implicações da diminuição de seu tamanho.

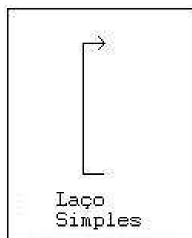
Atualmente, as medições são feitas sobre as funções de um programa. Uma função pode ser definida como um conjunto de blocos estruturais e instruções isoladas que formam um corpo indivisível para a análise do *Paradyn*. Nesse contexto, os blocos estruturais são conjuntos de instruções cuja execução é dependente de uma condição, como *while*, *for* e *if*.

Os possíveis tamanhos de grãos para instrumentação incluem funções, como já é feito, blocos estruturais e instruções isoladas. Percebe-se claramente que a instrumentação de instruções isoladas não é recomendável, pois como na execução do programa o processador percorre uma única vez tais instruções, então o custo de medição não seria compensado, pois para efetuar uma medida precisa-se de novas instruções.

Portanto, o dimensionamento escolhido para realizar medições neste projeto é baseado em blocos estruturais de repetição (laços como *for, while*), pois são nesses blocos que se gasta um tempo maior de execução. Os demais blocos estruturais (envolvendo testes e caminhos de decisão) não interessam também por se caracterizarem como especializações de instruções isoladas (agora com caminhos alternativos).

3.3 Identificação de laços

A identificação dos laços é uma etapa fundamental, pois eles serão os blocos internos às funções que serão instrumentados. Um laço é caracterizado pela existência de uma instrução de salto para algum endereço anterior ao da própria instrução. Assim, para encontrar os laços de uma função é necessário analisar todo o seu código e encontrar todas as instruções de salto para trás. Porém, ao identificar um laço, podemos encontrar os seguintes casos:



Laço Simples – o corpo do laço é delimitado pelo endereço da instrução de salto e o endereço de destino do salto.

Figura 3. 9 – Laço Simples

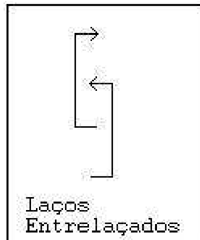


Figura 3. 10 – Laços entrelaçados

Laços Entrelaçados – ocorre quando um laço não está totalmente contido dentro do outro, mas há uma intersecção entre eles. Isso pode ocorrer entre vários laços ao mesmo tempo. Neste caso, todos os laços que possuem esta característica são tratados como laços irmãos. Os laços filhos comuns aos dois laços são tratados como filhos apenas para o primeiro laço a ser identificado como pai. Não há problemas nesta abordagem, pois todos os laços serão analisados.

Para se realizar a análise dos laços de uma função, obviamente é necessário ter acesso a suas instruções. Inicialmente, para que pudessem ser realizados os testes necessários no novo módulo completamente separado do *Paradyn*, foram utilizados o comando *dis* (comando *Unix* capaz de gerar o código *assembly* de um programa) e algumas funções auxiliares criadas para obter as instruções e endereços nos formatos desejados. Porém, ao fazer a integração com o *Paradyn* (seção 3.7) isso não é mais necessário, já que existem classes e métodos para a manipulação da imagem do programa (seção 3.7.2).

Ao identificar os laços, estes são armazenados em uma estrutura denominada *Loop*. Esta estrutura implementa uma árvore binária e armazena informações sobre o laço, como seus endereços de início e fim. A escolha deste tipo de estrutura se deu devido à sua organização hierárquica (pois os laços também são organizados hierarquicamente dentro do programa) e à grande facilidade em se localizar a posição correta de um determinado laço dentro desta hierarquia

Para realizar a inserção de um laço na árvore, foi implementada a função *insereLoop*. Esta é uma função recursiva que percorre a árvore para localizar a posição onde o laço será inserido, de acordo com o seu endereço. O pseudocódigo desta função está ilustrado a seguir:

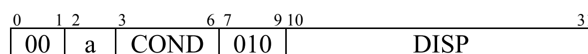
```
insereLoop(Loop no, Address start, Address end){
    se (no==null){
      insere como raiz da árvore;
    }

     se((start >= no.start) && (end < no.end)){
       //é filho de no
       se(no.son==null)
         insere como filho de no;
       else
         insereLoop(no.son, start, end);
     }else{
       //é irmão de no
       se(no.brother==null)
         insere como irmão de no;
       else
         insereLoop(no.brother, start, end);
     }
}
```

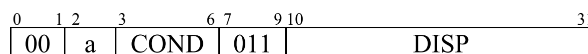
Figura 3. 11 – Pseudocódigo de *insereLoop*

Para realizar a identificação das instruções de salto, foi necessário um estudo sobre as instruções de máquina da arquitetura SPARC [Pau 1994, Wea 1993] a fim de encontrar padrões que pudessem identificar os diversos tipos de saltos existentes. Nesta arquitetura, as instruções possuem 32 *bits*. As instruções de salto, em particular, podem ser identificadas pelos seguintes formatos:

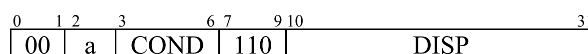
BRANCH ON INTEGER CONDITION CODES



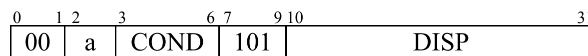
BRANCH ON INTEGER REGISTER WITH PREDICTION



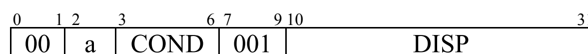
BRANCH ON FLOATING-POINT CONDITION CODES



BRANCH ON FLOATING-POINT CONDITION CODES WITH PREDICTION



BRANCH ON INTEGER CONDITION CODES WITH PREDICTION



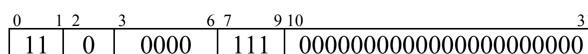
Como se pode notar, as instruções de salto possuem os seguintes campos variáveis:

- ***a*** **-** utilizado para anular a instrução seguinte;
- ***COND*** **-** codifica o tipo específico de salto;
- ***DISP*** **-** contém o valor do deslocamento que será feito em relação ao endereço atual da própria instrução.

Os demais *bits* (constantes) são utilizados para identificar uma determinada instrução como uma instrução de salto.

Para verificar se uma instrução é de salto, foi implementada a função *isBranch*. Esta função executa o algoritmo descrito nos próximos parágrafos:

Aplica-se uma máscara sobre o código de máquina da instrução, de forma a considerar apenas os *bits* constantes que identificam a instrução como de salto, zerando os demais *bits*. De acordo com os formatos das instruções de salto, teremos uma máscara da seguinte forma:



Para utilizar a máscara, basta utilizar o operador ***e lógico***: `Instrução & MÁSCARA`. Com isso, os *bits* desnecessários para a identificação da instrução são descartados.

Verifica-se se os *bits* do valor resultante codificam uma instrução de salto. Para isso, criamos constantes com o formato das instruções de salto para que esta verificação possa ser realizada com apenas uma comparação.

É aplicada uma máscara sobre o campo *COND* para obter o tipo específico do salto:

0 1	2	3 6	7 9	10 31
00	0	1111	000	0000000000000000000000

Para obter o endereço de destino do salto, implementou-se a função *jumpAddr*. Esta função obtém o valor de *DISP* da instrução de salto utilizando uma máscara e realiza a seguinte operação:

```
(DISP << 2) + endereço_atual
```

Como dito anteriormente, as instruções, nesta arquitetura, possuem 32 *bits*, ou seja, 4 bytes. Por isso, o valor do deslocamento sempre será um valor múltiplo de 4. Como os dois últimos *bits* de um número múltiplo de 4 são constantes (sempre 00), o valor do deslocamento na instrução de salto é codificado sem esses últimos *bits*. Para obter o valor do deslocamento real, é necessário acrescentar dois *bits* 0 à direita do valor de *DISP* fazendo a operação de deslocamento de *bits*. Em outras palavras, pode-se dizer que *DISP*=deslocamento/4. Para se obter o endereço de destino do salto, basta somar esse deslocamento ao endereço atual.

3.4 Identificação dos pontos de instrumentação

No módulo desenvolvido, os laços identificados serão instrumentados em suas extremidades, ou seja, serão inseridos: um código para iniciar as medições de desempenho ao entrar no laço, e outro código para finalizar as medições ao sair. Porém, é possível que existam saltos que alterem o fluxo de execução do programa de forma que este entre (ou saia) de um laço sem que seja executada a sua instrumentação. Isto é ilustrado na figura abaixo:

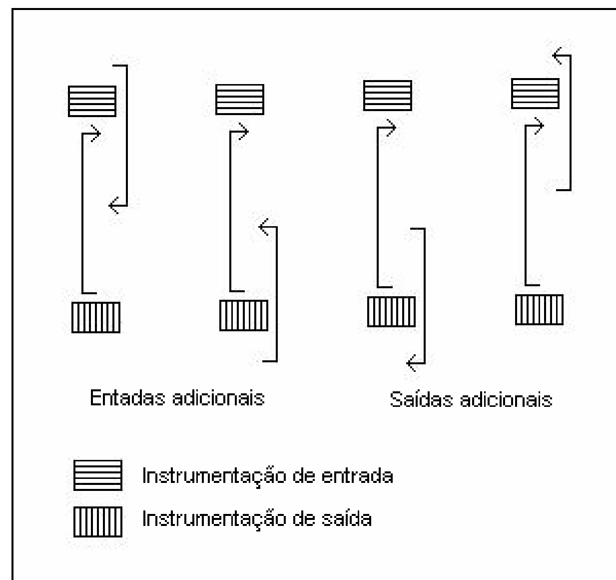


Figura 3. 12 – Entradas e saídas adicionais

As entradas (ou saídas) adicionais também devem ser consideradas como pontos de instrumentação do laço e, com isso, torna-se necessário armazenar todos os saltos para frente (ou para trás) em alguma estrutura, para que seja possível verificar se serão pontos de instrumentação posteriormente. Para armazenar esses saltos foi implementada uma classe chamada *fjumpList* que nada mais é do que uma lista dinâmica encadeada. Após fazer a identificação de todos os laços e saltos para frente (ou para

trás), utiliza-se essas duas estruturas (*Loop* e *fjumpList*) para determinar os pontos de instrumentação de um laço.

Além dos saltos, existe um outro caso em que a execução do programa pode sair do corpo do laço instrumentado, que são as chamadas de funções. Assim, deverá existir um ponto de instrumentação antes de chamadas de funções que estão dentro do laço, para parar a medição de tempo ao entrar em outra função, e também logo após, para continuar a medição quando a execução retornar ao corpo do laço.

As chamadas de funções podem ser facilmente identificadas através das instruções de *CALL*, que seguem o seguinte padrão:

INSTRUÇÃO *CALL*

A instrução de *CALL* é feita utilizando-se um deslocamento em relação ao atual endereço da instrução. Este deslocamento é armazenado no campo *DISP30* da instrução. Assim como nas instruções de *BRANCH*, o deslocamento real é armazenado na instrução sem os dois últimos *bits*. De acordo com o esquema, pode-se verificar que é apenas necessário verificar os dois primeiros *bits* para fazer a identificação deste tipo de instrução.

Para obter uma proporção de tempo que o laço ocupa na função de forma precisa, também foram adicionadas instrumentações para a função. Estas instrumentações são inseridas nas instruções *CALL*, como no caso dos laços, e também no início e retorno da função.

Todas as posições de memória são mapeadas em um vetor que será usado para marcar os pontos de instrumentação. Além disso, existem outros vetores que serão usados para realização da correção de deslocamentos dos *BRANCHs*. Isto é explicado em detalhes na próxima seção. O processo de identificação dos pontos de instrumentação foi implementado na função

defInstPoint, da classe *loopAnalyzer*. A figura 3.13 ilustra o pseudocódigo desta função.

```
defInstPoint(loop){

  marca extremidades do loop como pontos de instrumentação de entrada
e saida;

  para cada branch b {
    se b.origem esta fora do loop e b.destino esta dentro{
      //b é uma entrada de loop;
      marca ponto de instrumentação de entrada de loop;
    }

    se b.origem esta dentro do loop e b.destino esta fora{
      //b é uma saída de loop;
      marca ponto de instrumentação de saída de loop;
    }
  }

  marca inicio como ponto de instrumentação de entrada de função;
  para cada call c {
    //Marca o endereço da instrução de call como saída
    marca ponto de instrumentação de saida de função;
    se c esta dentro de loop
      marca ponto de instrumentação de saida de loop;

    //Marca o endereço da instrução de call +2 como entrada
    marca ponto de instrumentação de entrada de função;
    se c esta dentro de loop
      marca ponto de instrumentação de entrada de loop;
  }
}
```

Figura 3. 13 - Pseudocódigo *defInstPoint*

3.5 Correção de deslocamentos

Ao inserir instruções para instrumentação de um laço, torna-se necessário realizar uma correção nos deslocamentos de algumas instruções de salto da função. Isto ocorre caso existam instrumentações no intervalo do salto, sendo necessário assim incrementar o deslocamento da instrução para

que o salto seja feito para a mesma instrução que faria originalmente. Porém, no caso do salto ser realizado a partir do interior do laço, com um endereço de destino externo ao espaço do bloco de repetição, seu deslocamento deve ser corrigido de forma que seu destino seja uma instrumentação de saída.

Para realizar essa correção, foram criados vetores que guardam a quantidade total de instrumentações existentes a partir do começo da função até uma determinada posição. Por exemplo, o vetor *qStdIn[n]* guarda a quantidade total de instrumentações de entrada até a posição da n-ésima instrução. Assim, para determinar a quantidade destas instrumentações em um intervalo [a,b], basta fazer uma operação: *qStdIn[b]-qStdIn[a-1]*. Multiplicando-se a quantidade de instrumentações existentes no intervalo de uma instrução de *BRANCH* pelo tamanho da instrumentação, encontra-se o valor que deverá ser acrescentado ao deslocamento para corrigí-lo.

A instrução de *CALL* também apresenta este tipo de problema, pois também funciona através de deslocamento. Porém, a solução é mais simples porque o endereço lógico de destino está em um lugar fixo, isto é, não é alterado devido à inserção das instrumentações. Para obter o endereço lógico da função que será chamada pela instrução de *CALL*, basta somar o deslocamento (obtido multiplicando-se *DISP30*4*) ao endereço da instrução. A diferença entre o endereço lógico da função e o novo endereço lógico da instrução de *CALL* será o novo deslocamento.

3.6 Instrumentação

Instrumentações são códigos adicionados ao programa que será analisado, para que seja possível iniciar ou parar as medições de desempenho. Basicamente, estes códigos farão a chamada para função *timerdragonstart* para iniciar as medições, ou *timerdragonstop* para pará-las. O pseudocódigo destas funções está ilustrado na figura 3.14.

```
timerdragonstart(timer &t){
      se (t.isRunning) return;
      t.isRunning=1;
      t.lastStart = tempo_atual;
}

timerdragonstop(timer &t){
      se (!t.isRunning) return;
      t.isRunning=0;
      t.totalTime += tempo_atual-t.lastStart;
}
```

Figura 3. 14 - Pseudocódigo de *timerdragonstart* e *timerdragonstop*

Estas duas funções são adicionadas ao espaço da aplicação através de bibliotecas anexadas à aplicação (seção 3.7.4). A estrutura *timer* é utilizada para guardar o tempo medido, e é alocada em um espaço de memória compartilhada (seção 3.7.3) para que o *Paradynd* seja capaz de fazer amostragens desses valores. Ao executar a função *timerdragonstart*, o tempo atual é armazenado no campo *lastStart* da estrutura *timer*. Assim, ao executar a função *timerdragonstop*, é possível calcular o tempo que se passou neste intervalo. O tempo total é acumulado no campo *totalTime*.

O código básico da instrumentação é composto pelas instruções mostradas na figura 3.15:

```
1. NOP
2. SAVE
3. SETHI timerAddr>>10, %o0
4. OR   timerAddr & 0x000003ff, %o0 ,%o0
5. CALL timerdragonstart ou timerdragonstop
6. NOP
7. RESTORE
```

Figura 3. 15 - Instrumentação

Na arquitetura SPARC, a aplicação tem acesso a um conjunto de registradores composto por:

- **%o0 - %o7** – registradores de saída. São utilizados para passar argumentos para funções que serão chamadas utilizando-se a instrução *CALL*;
- **%i0 - i7** – registradores de entrada. São utilizados para receber argumentos;
- **%l0 - %l7** – registradores locais;
- **%g0 - %g7** – registradores globais.

Para passar o argumento para a função *startdragontimer*, ou *stopdragontimer*, é necessário atribuir ao registrador **%o0** o valor do endereço do timer. Porém, caso a instrumentação seja inserida antes de uma instrução de *CALL*, é possível que existam instruções anteriores que atribuem um valor a este registrador para ser utilizado como argumento da instrução de *CALL*. Para que este valor não seja perdido, é utilizada a instrução *SAVE* (figura 3.15, linha 2) antes da inserção do argumento no registrador **%o0**, para que possamos recuperá-lo depois (figura 3.15, linha 7). As instruções *SAVE* e *RESTORE* possuem os seguintes formatos:

INSTRUÇÃO *SAVE/RESTORE*

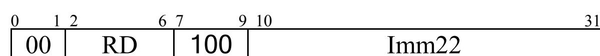
0 1	2 6	7 12	13 17	18	26	27 31
10	RD	OP3	RS1	0	-	RS2

0 1	2 6	7 12	13 17	18	31
10	RD	OP3	RS1	1	SIMM13

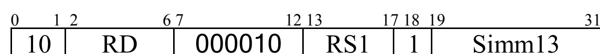
Existem estes dois formatos para as instruções de *SAVE* e *RESTORE*. O campo *OP3* define se a instrução é *SAVE* (com a constante 111100) ou *RESTORE* (111101). A soma dos valores dos registradores *RS1* e *RS2* (ou a constante *SIMM13*) é atribuída ao registrador *RD*.

O argumento a ser passado para as funções *startdragontimer* e *stopdragontimer* é o endereço do timer, que possui 32 *bits*. Porém, não é possível atribuir um valor de 32 *bits* a um registrador utilizando-se uma única instrução. Para isso, são utilizadas duas instruções:

INSTRUÇÃO SETHI



INSTRUÇÃO OR



A instrução *SETHI* é utilizada para atribuir os primeiros 22 *bits* do endereço do *timer* ao registrador **%o0** (figura 3.15, linha 3). No campo *RD* deve ser colocado o código do registrador de destino e, em *Imm22*, o valor de 22 *bits*. Os 10 *bits* restantes são completados através de uma instrução *OR* entre o registrador **%o0** e o valor dos 10 últimos *bits* do endereço, atribuindo-se o valor ao registrador **%o0** (figura 3.15, linha 4). O campo *RS1* é o código do registrador que será o primeiro operando, o campo *Simm13* contém uma constante de 13 *bits* que será o segundo operando e o campo *RD* contém o código do registrador de destino.

Após atribuir o argumento, utiliza-se uma instrução de *CALL* para uma das funções (figura 3.15, linha 5). Logo em seguida, é necessária a presença de uma instrução de *NOP*, que garante que a próxima instrução a ser executada será a da função chamada. Essa técnica é utilizada a fim de reduzir os efeitos de atraso na transferência de controle, causados no processador (com arquitetura *pipeline* [Man 1993]) devido à transferência de controle por parte de *CALL* e *BRANCH* (que alteram os registradores **%pc** e **%npc**). Isto, pois em uma arquitetura *pipeline*, como SPARC, simultaneamente a execução da instrução de *CALL* é feita a operação de

fetch, que busca a próxima instrução a ser executada. Com isso, essa próxima instrução seria executada antes que a transferência de controle para o alvo da chamada ocorra.

Existem seis tipos de instrumentação:

- ***STD_IN* -** instrumentação inserida no começo do intervalo de um laço e após instruções de *CALL*. Tem a função de iniciar a medição de tempo do laço;
- ***STD_OUT* -** instrumentação inserida no final do intervalo de um laço e antes de instruções de *CALL*. Tem a função de parar a medição de tempo do laço;
- ***FUNC_IN* -** instrumentação inserida no início da função. Muito semelhante à instrumentação *STD_IN*, mas faz uso de uma estrutura *timer* diferente. Tem a função de iniciar a medição de tempo da função;
- ***FUNC_OUT* -** instrumentação inserida no final da função. Muito semelhante à instrumentação *STD_OUT*, mas faz uso de uma estrutura *timer* diferente. Tem a função de parar a medição de tempo da função;
- ***OTHER_IN* -** instrumentação inserida antes de saltos que são entradas adicionais ao laço. Semelhante à instrumentação STD_IN, mas possui uma proteção que impede que a instrumentação seja executada se o salto não for executado. Esta proteção é implementada com uma instrução de salto com a condição inversa à condição do salto que entra no laço;
- ***OTHER_OUT* -** instrumentação inserida no destino de saltos que são saídas adicionais ao laço. Semelhante à instrumentação STD_OUT, mas possui uma proteção que impede que a instrumentação seja

executada caso o salto tenha sido executado. Esta proteção é implementada com uma instrução de salto incondicional.

A geração do código instrumentado é feito pela função *instrumLoops* (pseudocódigo ilustrado na figura 3.16), que também realiza a correção dos deslocamentos dos saltos, no caso de existirem instrumentações em seu intervalo. O novo código é armazenado em um buffer, para que posteriormente possa ser copiado para o novo espaço alocado para a função.

```
instrumLoops(buf){
  posicao_atual=0; //marca a posicao no buffer

  para cada instrução{

    para cada instrumentação a ser inserida nesta posição{
      insere a intrumentaçao em buf[posicao_atual];
      posicao_atual=posicao_atual + tamanho da instrumentação;
    }

    se instrução é branch ou call{
      faz correção do deslocamento;
      buf[posicao_atual] = instrução corrigida;
    }senão{
      buf[posicao_atual] = instrução;
    }
    posicao_atual=posicao_atual+1;
  }
}
```

Figura 3. 16 - Pseudocódigo *instrumLoops*

3.7 Integração do projeto ao *Paradyn*

Para integrar o novo módulo de medição de laços de repetição foram utilizadas várias funções, classes, métodos e tipos implementados pelo *Paradyn*, adaptando-os às necessidades do projeto.

Com isso buscou-se alcançar maior coesão e compatibilidade com a ferramenta a ser estendida, tratando-se basicamente os seguintes aspectos:

- **comunicação entre o *front-end* e o daemon -** no *front-end* é decidido quais os locais que serão instrumentados, mas a instrumentação é feita pelo daemon;

- **acesso à imagem da função -** utilização de estruturas já existentes no *Paradyn* para acessar o código do programa;

- **memória compartilhada** – os dados medidos na aplicação são armazenados em uma região de memória compartilhada entre o processo e o *daemon*;

- **inclusão de funções em bibliotecas anexadas à aplicação pelo Paradyn** – funções de medição, externas a função original, são carregadas ao espaço da aplicação em tempo de execução;

- **relocação de funções** – insere-se o código instrumentado em uma nova região de memória alocada ao processo em execução.

3.7.1 Comunicação entre o *front-end* e o *daemon*

Para realizar a comunicação entre o *front-end* e o *daemon* foram utilizadas as interfaces já existentes, adicionando-se novas funcionalidades. As principais alterações no código do *Paradyn* estão descritas a seguir:

- **Interface *DataManager***(arquivos *src/core/Paradyn/h/dataManager.I* e *src/core/Paradyn/src/DMthread/DMpublic.C*): esta interface é utilizada para a comunicação com o *Data Manager*. Criou-se a função *instLoops* para que o *Performance Consultant* possa passar ao *Data Manager* a função que deverá ser instrumentada. O *Performance Consultant* decide instrumentar os laços de uma função no momento em que se identifica que ela é um gargalo. Ao ser chamada, esta função encontra todos os *daemons* utilizando a estrutura *Paradyndaemon::allDaemons*, e para cada um utiliza a interface *dynRPC* para requisitar a instrumentação e, ao final da

mesma, receber os dados medidos. Criaram-se também algumas funções que retornam ao *Performance Consultant* os dados solicitados por cada *daemon*.

- **Interface *dynRPC*** (arquivos *src/core/Paradyn/h/dyninstRPC.I* e *src/core/Paradynd/src/dynrpc.C*): esta interface é utilizada para a comunicação com o *daemon*. Adicionou-se a função *instLoop*, responsável pelo gerenciamento de instrumentação, bem como algumas funções que permitem ao *Data Manager* receber os dados coletados pelos *daemons*.

IGEN - INTERFACE GENERATOR

IGEN é um utilitário já existente no *Paradyn* que permite a criação de interfaces remotas de forma automática. Estas interfaces permitem a execução de chamadas como RPC (*Remote Procedure Call*), em que os sistemas finais (cliente e servidor) podem ser tanto processos como *threads* (como *DMthread* e *PCthread*, no caso do *Paradyn*).

Assim, algumas funções podem ser usadas como se fossem locais, através dessas interfaces remotas geradas pelo *IGEN*. Para isso é necessário declarar tais funções como membros da interface em um arquivo **<*interfaceX*>.I**. A partir deste arquivo, o *IGEN* gera vários arquivos fontes e *headers,* dentre os quais estão o com o código para interface com o cliente (**<interfaceX>.CLNT.C**) e o com o código para o interface com o servidor (**<interfaceX>.SRVR.C**).

Gerenciamento de instrumentação

O *Paradyn* possibilita o gerenciamento de processos por meio da classe *processMgr* e da função *getProcMgr()*. Através destes percorre-se todos os processos de um *daemon*, armazenando em um vetor (*allProcs*) ponteiros para cada um deles. Com eles é possível obter informações e

manipular recursos dos processos, como *pids*, *status*, representação de sua imagem e inclusive manipular seu espaço compartilhado de memória.

Todos os processos de um *daemon* possuem alguns dados em comum, como os endereços lógicos das funções de instrumentação (armazenados na tabela de símbolos) e estruturas que: armazenam instruções (e seus respectivos endereços), laços, saltos e pontos de instrumentação de uma função. Assim, tais inicializações são necessárias apenas para o primeiro processo.

A fim de simplificar o trabalho de instrumentação, optou-se por realizá-lo a um laço de cada vez. Assim, percorre-se a árvore que contém todos os laços da função, alocando-se uma região de memória para estender o espaço pertencente ao processo atual, a qual receberá o código da função a ser relocada. Aloca-se ainda uma posição de memória compartilhada, entre o processo em execução e seu *daemon*, para armazenar amostras de valores de tempo e do contador do número de execuções.

Em seguida instrumenta-se o laço analisado (por meio do método *instrumLoops*, pertencente a classe *loopAnalyzer*), e escreve-se o código gerado na região destinada à função a ser relocada.

Durante o processo de geração de códigos de instrumentação para o laço mapearam-se também os endereços onde as funções de medição (como a de tempo e do contador) serão chamadas. Em tais posições são geradas instruções de *CALL* (utilizando-se a função *generateCALL*) para as posições em que as funções de medição estão armazenadas. Outra instrução de *CALL* necessária é a que desvia o fluxo de execução, da função original para a função relocada, gerada no início da função.

Gerenciamento da coleta de dados medidos

Durante o processo de medição de um laço é preciso aguardar um tempo mínimo para se conseguir amostras. Juntamente a esse tempo utiliza-

se também um contador para controlar o número de execuções da função, e com isso garantir um número mínimo de amostras. Assim, implementou-se um critério de parada das medições necessárias.

Durante o processo de coleta de dados medidos é preciso realizar ainda um controle de concorrência, a fim de garantir a consistência destes dados. Um possível cenário que justifique tal necessidade seria o de, a função instrumentada ter seu tempo de execução iniciado, assim como o laço, e o contador que sinaliza tal medição ser incrementado. Só que antes do final da execução da função, e assim antes do tempo da função ser finalizado, o *daemon* coleta os dados armazenados na memória compartilhada. Com isso o contador indicaria a conclusão de uma medição, quando muito provavelmente os valores do laço, e principalmente da função estariam incompletos, inviabilizando uma análise correta dos dados coletados.

Diante desse cenário adotou-se a seguinte estratégia. Quando a condição de parada for cumprida, não se realiza a coleta dos dados do laço atual. Tal coleta é adiada até o momento em que o contador da instrumentação do próximo laço a ser analisado for alterado (caso o mesmo exista). Ao final do tempo de observação da medição desse próximo laço, com o fato da instrumentação ser realizada para um laço por vez, e em espaços de relocação distintos, tem-se uma garantia mínima da conclusão das medições para o laço anterior. A fim de realizar a coleta para o último laço, repete-se o trabalho de coleta para o mesmo, tratando-o como se fosse um anterior.

Os dados coletados pelo *daemon*, a partir da memória compartilhada (seção 3.7.3) são armazenados na estrutura que representa o laço analisado, para posteriormente serem enviados ao *Performance Consultant*, e ali serem devidamente tratados e amostrados.

3.7.2 Acesso à imagem da função

Para se realizar a análise dos laços de uma função, obviamente é necessário ter acesso a suas instruções. Isto é feito por meio de classes e métodos para a manipulação do executável (imagem) do programa, já implementados pelo *Paradyn*. As principais classes utilizadas são:

- **classe *pd_process* -** representa o processo. Com esta classe é possível obter sua imagem utilizando a função *getImage*. Também é possível localizar a instância da classe *function_base* correspondente à função que se deseja instrumentar;
- **classe *image* -** representa a imagem do programa. Com esta classe é possível obter a instrução localizada em um determinado endereço de memória utilizando-se a função *get_instruction*;
- **classe *function_base* -** representa uma função do programa. Com esta classe é possível obter o endereço inicial da função e seu tamanho utilizando-se os métodos *getAddress* e *size*, respectivamente. Com isso, é possível percorrer seu espaço na memória e obter cada uma de suas instruções;
- **classe *object*** – responsável pela manipulação do executável (utilizando a biblioteca ELF), gerenciando processo de leitura da tabela de símbolos. Posteriormente tais informações irão alimentar a classe *image*.

3.7.3 Memória Compartilhada

Após a aplicação executar a função relocada (armazenada em seu espaço de memória), e coletar todos os dados necessários, é preciso enviá-los ao *daemon*, para posterior análise dos mesmos. No entanto, o espaço de memória dele é distinto ao da aplicação.

A fim de tratar tal situação adotou-se como estratégia alocar uma região de memória compartilhada entre a aplicação e o *daemon*, na qual se realiza o depósito/coleta de dados medidos. Para isso aproveitou-se o método *getSharedMemMgr* (da classe *pd_process*) que gerencia uma posição de memória compartilhada, e a classe *shmMgr* que manipula tal posição, ambos implementados pelo *Paradyn*.

Criou-se assim um ponteiro do tipo *shmMgr* para receber um gerenciador de memória do processo. Por meio dele alocou-se um tamanho de memória suficiente para armazenar os dados medidos, e atribuiu-se tal posição à variável *baseAddrInDaemon*, que permite ao *daemon* acessá-la. O acesso a essa região comum, a partir da aplicação, é feito por meio do endereço armazenado na variável *baseAddrInApplic*. Tal endereço é obtido pelo método *getAddressInApplic* (*shmMgr*), que permite que este endereço seja mapeado a partir do espaço lógico de endereçamento da aplicação.

3.7.4 Inclusão de funções em bibliotecas anexadas na aplicação pelo *Paradyn*

No código de instrumentação é preciso inserir algumas chamadas às funções que realizam as medições necessárias sobre a função a ser analisada, e seus laços. No entanto, como estas funções adicionais não faziam parte do código original, seus endereços também não constariam na tabela de símbolos da aplicação.

Uma solução para tal entrave é realizar a inclusão destas funções nas bibliotecas que o *Paradyn* anexa à aplicação. Para isso estendeu-se um recurso implementado pelo *Paradyn*. No caso, declarou-se a função desejada no arquivo *RTetc-solaris.C*, o qual será utilizado para carregar as funções de instrumentação no espaço de memória da aplicação. Para gerar a instrução de *CALL* da aplicação instrumentada para a função de medição é necessário ter acesso ao endereço da função em questão, a fim de calcular o deslocamento daquela instrução. Isto é realizado por meio da função

findInternalSymbol (classe *pd_process*), que busca na tabela de símbolos da aplicação tal endereço.

3.7.5 Relocação de função

O processo de instrumentação trata da inserção de código em locais pré-determinados (pontos de instrumentação). No entanto essa inserção tornar-se-ia inviável se fosse feita diretamente no código original da função a ser analisada. Isto se deve por duas razões. Como o código encontra-se armazenado de forma contínua, a inserção de novas instruções de códigos, em posições arbitrárias, demandaria um trabalho custoso de deslocamento e acerto do código original, que seria duplicado pelo fato de ser necessário desfazer tal instrumentação ao final das medições. Outra razão que inviabiliza tal metodologia é o fato da instrumentação ser dinâmica, podendo-se haver inconsistência durante o fluxo de execução do programa e a realização da instrumentação.

Assim, uma maneira mais segura, eficiente e menos intrusiva de se realizar tal instrumentação é através da relocação da função a ser analisada, em que se transfere o código da função já instrumentada para outra região de memória previamente alocada ao processo.

Para a relocação do código foram utilizados métodos da classe *process*, já existente no *Paradyn*. O método *inferiorMalloc* aloca a um determinado processo um novo espaço de memória com tamanho suficiente para armazenar a função com as devidas instrumentações já inseridas.

Em seguida é preciso inserir o código instrumentado no espaço de memória já reservado para tais instruções. Para isso utiliza-se o método *writeDataSpace* (classe *process*), que escreve em um dado endereço uma quantidade continua de bytes fornecidos como argumento.

A esse processo de relocação estão relacionadas algumas outras atividades, descritas a seguir:

- quando a relocação é feita, as instruções de chamadas a outras funções (*CALL*) devem ser recalculadas. Isto pelo fato do código de tal instrução ser baseado no deslocamento entre seu endereço de origem e seu endereço de destino. Com isso esse deslocamento precisa ser corrigido (seção 3.5), pelo fato da instrução de *CALL* estar armazenada em um novo endereço de origem;

- após inserir o código instrumentado na região de relocação, é necessário alterar o fluxo de execução da função, de modo que quando a aplicação realize uma chamada pra tal função ela seja redirecionada para o código relocado. Isto é feito por meio de uma instrução de salto incondicional para esta nova área de memória. Tal instrução é gerada pela função *generateBranchOrCALL*, e inserida no início da função original. A figura 3.17 ilustra tal atividade;

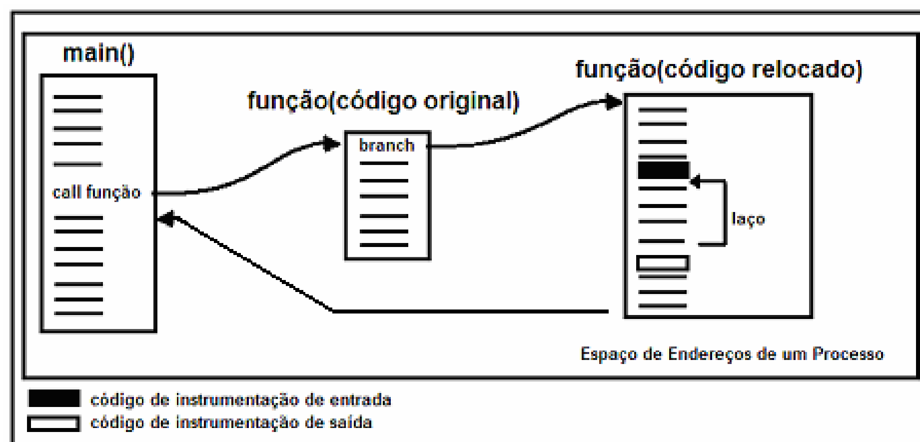


Figura 3. 17 – Relocação de código

- ao final das medições de um dado laço é necessário liberar a região de memória alocada para a função instrumentada. Isto é feito pelo método *inferiorFree* (classe *process*);

- depois de instrumentar e analisar todos os laços de uma função é preciso recuperar suas instruções iniciais. Para isso realiza-se um

novo acesso à imagem do processo, recuperando-se assim tais instruções e realizando o trabalho de retirada das instrumentações.

3.8 Interface Gráfica

Com os módulos básicos de instrumentação e medição de laços implementados modelou-se ainda uma possível interface que pode ser implementada para amostrar tais resultados.

A escolha da ferramenta a ser utilizada para a construção dos componentes de visualização foi norteada levando-se em conta sua eficiência para construção e execução. Outro fator preponderante para tal decisão foi a busca pela manutenção da compatibilidade com a interface já existente no *Paradyn*. Tendo isso em vista, TCL/TK foi a ferramenta escolhida para desempenhar tal tarefa.

Escolheu-se uma hierarquia de árvore como forma de representação dos grãos de medidas. Tal visualização é apresentada em uma nova janela, visando maior harmonia e organização, já que o número de laços possíveis de se encontrar em uma dada função pode ser grande. Nesta árvore estarão representados os diversos blocos de laços identificados dentro da função, cada um com uma cor específica que represente seu estado de análise. O padrão de cores será o mesmo utilizado na interface do *Paradyn*:

Figura 3. 18 – Esquema de cores padrão do *Paradyn*

Esta janela será aberta quando o usuário der um duplo clique sobre a função identificada como gargalo. Nesta janela a árvore será representada dentro de um *scroll-frame*, para o caso de a função apresentar um número de blocos superior ao máximo visível no espaço da janela.

Os blocos que compõem a árvore serão dispostos na mesma seqüência em que ocorrem na função, a fim de manter uma relação lógica. E sempre que um dado laço contiver outros em seu interior, um novo ramo hierárquico partirá deste ponto da árvore, representando tais laços contidos por este bloco (denominados "filhos").

Quando um bloco representar um gargalo, e ao mesmo tempo contiver algum "filho", um sinal mais, dentro de um quadrado, aparecerá no ramo que se liga a este bloco, significando a possibilidade de expansão para um outro nível hierárquico da árvore de laços (o qual representará todos os laços "filhos" deste bloco).

Sempre quando houver dois blocos seqüenciais dentro de um mesmo nível hierárquico, a expansão do bloco de cima fará com que o subseqüente seja deslocado para baixo, garantindo uma melhor visualização. O esquema a seguir ilustra tal situação:

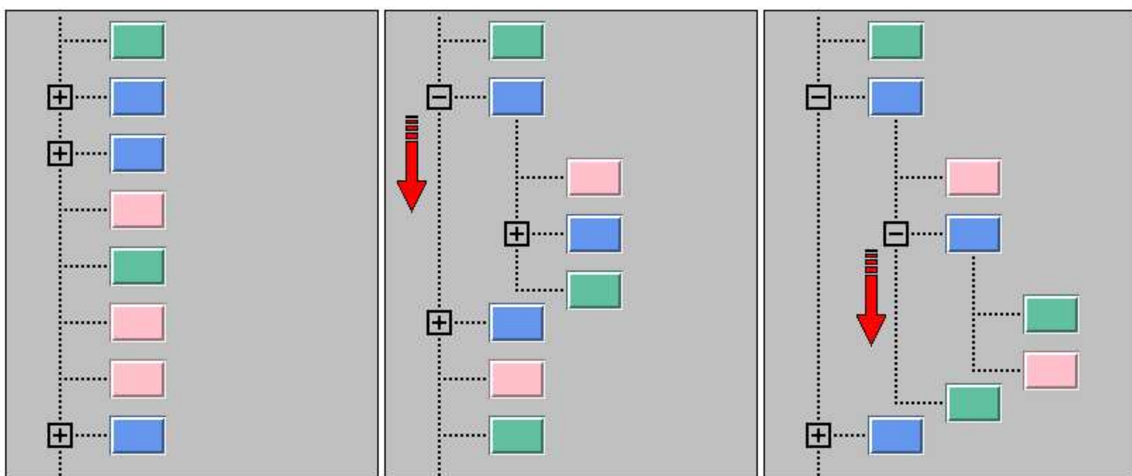


Figura 3. 19 – Funcionamento da expansão de blocos na hierarquia de árvore

Com esse módulo bem definido e devidamente implementado, e conhecendo os principais pontos e parâmetros de interface já existente no paradyn poder-se-á anexar esse novo módulo de interface ao *Visi Manager* e ao *User Interface*.

A figura 3.20 ilustra um esboço desse módulo:

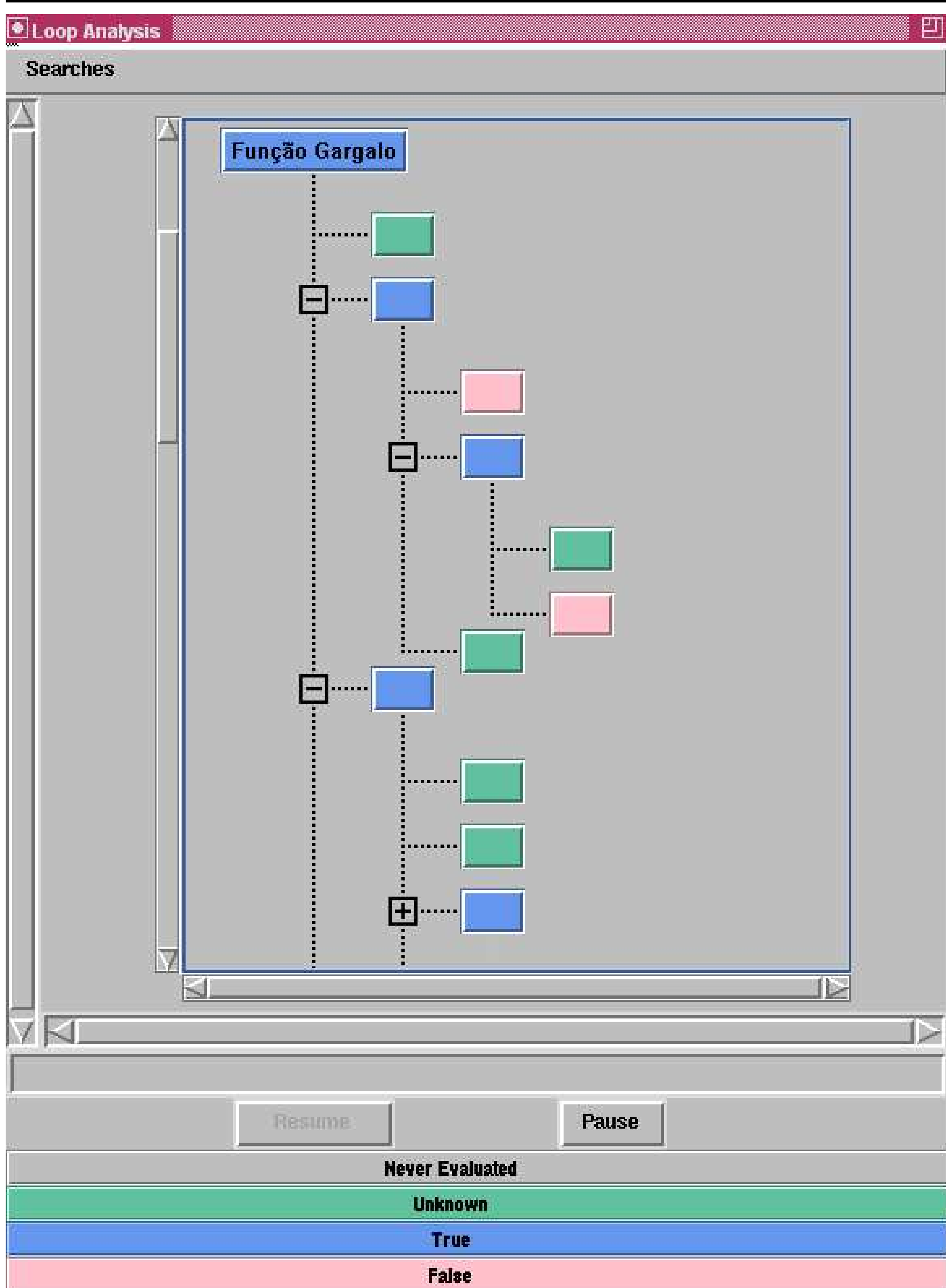


Figura 3. 20 – Ilustração da interface para o novo módulo tratado pelo projeto

3.9 Considerações finais

Este capítulo apresentou de forma sucinta as principais estruturas do *Paradyn*, a fim de fornecer um conhecimento preliminar do suporte básico oferecido pela ferramenta, e das funcionalidades a serem estendidas.

Tratou ainda das principais etapas e metodologias seguidas para o desenvolvimento do projeto, descrevendo vários módulos e funcionalidades implementadas para a obtenção de um protótipo que atendesse as principais especificações do projeto. Alguns testes realizados sobre este protótipo, e os resultados obtidos com os mesmos encontram-se descritos no próximo capítulo.

CAPÍTULO 4

TESTES E RESULTADOS

Neste capítulo serão descritos alguns testes feitos usando a ferramenta *Paradyn* com o novo módulo identificador e analisador de laços. Para efetuar esses testes foi utilizada a arquitetura *Sun SPARC*, com sistema operacional *Solaris 5.8*, ambiente em que este projeto foi desenvolvido.

Os testes aqui descritos abrangem a verificação da identificação de laços, a análise de código a fim de verificar a exatidão da inserção do código de instrumentação e a comparação dos tempos medidos entre o método de instrumentação dinâmica e a inserção de instruções de medição de tempo no código fonte.

4.1 - Teste da identificação de laços

Para fazer a verificação do módulo de identificação de laços, implementou-se a função *testLoop*, que percorre a árvore mostrando seus nós e o endereço lógico de suas extremidades.

A figura 4.1 ilustra uma função utilizada para testes, contendo diversos laços irmãos e filhos:

```
void funcao(){
  //do nothing!
  int k;
  int j;

  for(int i=0; i<100; i++){
    j=0;
    do{
      for(int w=0; w<50;w++);
      for(int w=0; w<50;w++){
        for(int t=0; t<200; t++);
      }
      j++;
    }while(j<i);
  }

  k=0;
  while(k<500000){
    k++;
  }
}
```

Figura 4.1 – Código da função

A saída exibida pela função *testLoop* foi:

```
Loop 10730 10808
   Loop 1074c 107f4
      Loop 10750 10774
      Loop 10780 107d4
         Loop 1079c 107c0
Loop 10814 10840
```

Figura 4. 2 – Estrutura de laços

O código *assembly* é representado na figura 4.3:

```
1078c:  9d e3 bf 78   SAVE     %sp, -136, %sp
10790:  c0 27 bf e4   ST       %g0, [%fp - 28]
10794:  c2 07 bf e4   LD       [%fp - 28], %g1
10798:  80 a0 60 63   CMP      %g1, 99
1079c:  04 80 00 04   BLE      0x107ac
107a0:  01 00 00 00   NOP
107a4:  10 80 00 34   BA       0x10874
107a8:  01 00 00 00   NOP
107ac:  c0 27 bf e8   ST       %g0, [%fp - 24]
107b0:  c0 27 bf e0   ST       %g0, [%fp - 32]
107b4:  c2 07 bf e0   LD       [%fp - 32], %g1
107b8:  80 a0 60 31   CMP      %g1, 49
107bc:  04 80 00 04   BLE      0x107cc
107c0:  01 00 00 00   NOP
107c4:  10 80 00 07   BA       0x107e0
107c8:  01 00 00 00   NOP
107cc:  c2 07 bf e0   LD       [%fp - 32], %g1
107d0:  82 00 60 01   ADD      %g1, 1, %g1
107d4:  c2 27 bf e0   ST       %g1, [%fp - 32]
107d8:  10 bf ff f7   BA       0x107b4
107dc:  01 00 00 00   NOP
107e0:  c0 27 bf e0   ST       %g0, [%fp - 32]
107e4:  c2 07 bf e0   LD       [%fp - 32], %g1
107e8:  80 a0 60 31   CMP      %g1, 49
107ec:  04 80 00 04   BLE      0x107fc
107f0:  01 00 00 00   NOP
107f4:  10 80 00 13   BA       0x10840
107f8:  01 00 00 00   NOP
107fc:  c0 27 bf dc   ST       %g0, [%fp - 36]
10800:  c2 07 bf dc   LD       [%fp - 36], %g1
10804:  80 a0 60 c7   CMP      %g1, 199
10808:  04 80 00 04   BLE      0x10818
1080c:  01 00 00 00   NOP
10810:  10 80 00 07   BA       0x1082c
10814:  01 00 00 00   NOP
10818:  c2 07 bf dc   LD       [%fp - 36], %g1
1081c:  82 00 60 01   ADD      %g1, 1, %g1
10820:  c2 27 bf dc   ST       %g1, [%fp - 36]
10824:  10 bf ff f7   BA       0x10800
10828:  01 00 00 00   NOP
1082c:  c2 07 bf e0   LD       [%fp - 32], %g1
10830:  82 00 60 01   ADD      %g1, 1, %g1
10834:  c2 27 bf e0   ST       %g1, [%fp - 32]
10838:  10 bf ff eb   BA       0x107e4
1083c:  01 00 00 00   NOP
10840:  c2 07 bf e8   LD       [%fp - 24], %g1
10844:  82 00 60 01   ADD      %g1, 1, %g1
10848:  c2 27 bf e8   ST       %g1, [%fp - 24]
1084c:  fa 07 bf e8   LD       [%fp - 24], %i5
10850:  c2 07 bf e4   LD       [%fp - 28], %g1
10854:  80 a7 40 01   CMP      %i5, %g1
10858:  06 bf ff d6   BL       0x107b0
1085c:  01 00 00 00   NOP
10860:  c2 07 bf e4   LD       [%fp - 28], %g1
10864:  82 00 60 01   ADD      %g1, 1, %g1
10868:  c2 27 bf e4   ST       %g1, [%fp - 28]
1086c:  10 bf ff ca   BA       0x10794
10870:  01 00 00 00   NOP
10874:  c0 27 bf ec   ST       %g0, [%fp - 20]
10878:  c2 07 bf ec   LD       [%fp - 20], %g1
1087c:  3b 00 01 e8   SETHI    %hi(0x7a000), %i5
10880:  ba 17 61 1f   OR       %i5, 0x11f, %i5 ! 0x7a11f
10884:  80 a0 40 1d   CMP      %g1, %i5
10888:  04 80 00 04   BLE      0x10898
1088c:  01 00 00 00   NOP
10890:  10 80 00 07   BA       0x108ac
10894:  01 00 00 00   NOP
10898:  c2 07 bf ec   LD       [%fp - 20], %g1
1089c:  82 00 60 01   ADD      %g1, 1, %g1
108a0:  c2 27 bf ec   ST       %g1, [%fp - 20]
108a4:  10 bf ff f5   BA       0x10878
108a8:  01 00 00 00   NOP
108ac:  81 c7 e0 08   RET
108b0:  81 e8 00 00   RESTORE
108b4:  81 c3 e0 08   JMP      %o7 + 8
108b8:  ae 03 c0 17   ADD      %o7, %l7, %l7
```

Figura 4. 3 – Código *Assembly*

4.2 - Teste da inserção do código de instrumentação

Neste teste teve-se como intuito a verificação da inserção do código de instrumentação no código executável em tempo de execução, ou seja, dinamicamente. Para isso gerou-se o arquivo *identLoops*, que exibe a imagem de uma função do programa, juntamente com as representações das instruções de instrumentação para medição de um determinado laço, como também as instruções de *CALL* e *BRANCH* com seus deslocamentos corrigidos. A figura 4.4 apresenta o programa que possui a função **filho( ),** a qual foi analisada e será utilizada para demonstrar os testes realizados.

```
int main(){
    int valor;
    while(true){
      printf("Entrando filho\n");
      valor = filho(1);
     printf("saindo filho\n");
   }
}

int filho(int w){
  double var;
  printf("comeco filho\n");

  for (int k = 0; k < 100000000; k ++)
      var = 2^k;

  int a = 3;
  neto(a);

  for(int n = 0; n < (100000000/2); n ++)
     var=2^n;

  printf ("fim filho\n");
  return 2;
}

void neto(int x ){
   int a = x;
   a = 10;
   return;
}
```

Figura 4.4 – Código fonte do programa testado

Apesar do programa acima não ter funcionalidade alguma, cabe exatamente para demonstração de acertos de *BRANCHs, CALLs* e verificação dos locais de instrumentação.

As instruções do código objeto da função **filho( )** foram obtidas através do comando *dis* e são exibidas na figura 4.5.

Os laços encontrados pelo módulo identificador foram dois laços irmãos**:**

- ***Loop* 107bc 10800**
- ***Loop* 10820 10864**

O arquivo *testeInstrumentação* gerado para a instrumentação do segundo laço (***Loop 10820 10864***), está ilustrado pela figura 4.6.

A partir desse arquivo pode-se confirmar que a inserção do código de instrumentação e a modificação das instruções de *CALL* e *BRANCH* estão corretas.

```
107a0:  9d e3 bf 80   SAVE    %sp, -128, %sp
107a4:  f0 27 a0 44   ST      %i0, [%fp + 68]
107a8:  03 00 00 42   SETHI   %hi(0x10800), %g1
107ac:  90 10 61 68   OR      %g1, 0x168, %o0 ! 0x10968
107b0:  40 00 40 9d   CALL    Printf
107b4:  01 00 00 00   NOP
107b8:  c0 27 bf e4   ST      %g0, [%fp - 28]
107bc:  c2 07 bf e4   LD      [%fp - 28], %g1
107c0:  1b 01 7d 78   SETHI   %hi(0x5f5e000), %o5
107c4:  9a 13 60 ff   OR      %o5, 0xff, %o5  ! 0x5f5e0ff
107c8:  80 a0 40 0d   CMP     %g1, %o5
107cc:  04 80 00 04   BLE     0x107dc
107d0:  01 00 00 00   NOP
107d4:  10 80 00 0d   BA      0x10808
107d8:  01 00 00 00   NOP
107dc:  c2 07 bf e4   LD      [%fp - 28], %g1
107e0:  82 18 60 02   XOR     %g1, 2, %g1
107e4:  c2 27 bf f0   ST      %g1, [%fp - 16]
107e8:  d5 07 bf f0   LD      [%fp - 16], %f10
107ec:  91 a0 19 0a   FITOD   %f10, %f8
107f0:  d1 3f bf e8   STD     %f8, [%fp - 24]
107f4:  c2 07 bf e4   LD      [%fp - 28], %g1
107f8:  82 00 60 01   ADD     %g1, 1, %g1
107fc:  c2 27 bf e4   ST      %g1, [%fp - 28]
10800:  10 bf ff ef   BA      0x107bc
10804:  01 00 00 00   NOP
10808:  82 10 20 03   MOV     3, %g1
1080c:  c2 27 bf e4   ST      %g1, [%fp - 28]
10810:  d0 07 bf e4   LD      [%fp - 28], %o0
10814:  40 00 00 1e   CALL    _Z4netoi
10818:  01 00 00 00   NOP
1081c:  c0 27 bf e0   ST      %g0, [%fp - 32]
10820:  c2 07 bf e0   LD      [%fp - 32], %g1
10824:  1b 00 be bc   SETHI   %hi(0x2faf000), %o5
10828:  9a 13 60 7f   OR      %o5, 0x7f, %o5  ! 0x2faf07f
1082c:  80 a0 40 0d   CMP     %g1, %o5
10830:  04 80 00 04   BLE     0x10840
10834:  01 00 00 00   NOP
10838:  10 80 00 0d   BA      0x1086c
1083c:  01 00 00 00   NOP
10840:  c2 07 bf e0   LD      [%fp - 32], %g1
10844:  82 18 60 02   XOR     %g1, 2, %g1
10848:  c2 27 bf f0   ST      %g1, [%fp - 16]
1084c:  d5 07 bf f0   LD      [%fp - 16], %f10
10850:  91 a0 19 0a   FITOD   %f10, %f8
10854:  d1 3f bf e8   STD     %f8, [%fp – 24]
10858:  c2 07 bf e0   LD      [%fp - 32], %g1
1085c:  82 00 60 01   ADD     %g1, 1, %g1
10860:  c2 27 bf e0   ST      %g1, [%fp – 32]
10864:  10 bf ff ef   BA      0x10820
10868:  01 00 00 00   NOP
1086c:  03 00 00 42   SETHI   %hi(0x10800), %g1
10870:  90 10 61 78   OR      %g1, 0x178, %o0 ! 0x10978
10874:  40 00 40 6c   CALL    Printf
10878:  01 00 00 00   NOP
1087c:  82 10 20 01   MOV     1, %g1
10880:  b0 10 00 01   MOV     %g1, %i0
10884:  81 c7 e0 08   RET
10888:  81 e8 00 00   RESTORE
```

Figura 4. 5 – Código *assembly* da função filho

```
>>107a0    9de3bf80
>>107a4    CONTADOR
>>107b4    INSTRUMENTACAO DE ENTRADA DE FUNÇÃO
>>107d0    f027a044
>>107d4    3000042
>>107d8    90106168
>>107dc    INSTRUMENTACAO DE SAIDA DE FUNÇÃO
>>107f8    4030ebb3 - INSTRUCAO DE CALL
>>107fc    1000000
>>10800    INSTRUMENTACAO DE ENTRADA DE FUNÇÃO
>>1081c    c027bfe4
>>10820    c207bfe4
>>10824    1b017d78
>>10828    9a1360ff
>>1082c    80a0400d
>>10830    4800004 SALTO PRA FRENTE 10840
>>10834    1000000
>>10838    1080000d SALTO PRA FRENTE 1086c
>>1083c    1000000
>>10840    c207bfe4
>>10844    82186002
>>10848    c227bff0
>>1084c    d507bff0
>>10850    91a0190a
>>10854    d13fbfe8
>>10858    c207bfe4
>>1085c    82006001
>>10860    c227bfe4
>>10864    10bfffef SALTO PRA TRAS 10820
>>10868    1000000
>>1086c    82102003
>>10870    c227bfe4
>>10874    d007bfe4
>>10878    INSTRUMENTACAO DE SAIDA DE FUNÇÃO
>>10894    4030ab26 - INSTRUCAO DE CALL
>>10898    1000000
>>1089c    INSTRUMENTACAO DE ENTRADA DE FUNÇÃO
>>108b8    c027bfe0
>>108bc    INSTRUMENTACAO DE ENTRADA DE LAÇO
>>108d8    c207bfe0
>>108dc    1b00bebc
>>108e0    9a13607f
>>108e4    80a0400d
>>108e8    4800004 SALTO PRA FRENTE 108f8
>>108ec    1000000
>>108f0    1080000d SALTO PRA FRENTE 10924
>>108f4    1000000
>>108f8    c207bfe0
>>108fc    82186002
>>10900    c227bff0
>>10904    d507bff0
>>10908    91a0190a
>>1090c    d13fbfe8
>>10910    c207bfe0
>>10914    82006001
>>10918    c227bfe0
>>1091c    10bfffef SALTO PRA TRAS 108d8
>>10920    1000000
>>10924    INSTRUMENTACAO DE SAIDA DE LAÇO
>>10940    3000042
>>10944    90106178
>>10948    INSTRUMENTACAO DE SAIDA DE FUNÇÃO
>>10964    4030eb58 - INSTRUCAO DE CALL
>>10968    1000000
>>1096c    INSTRUMENTACAO DE ENTRADA DE FUNÇÃO
>>10988    82102001
>>1098c    b0100001
>>10990    INSTRUMENTACAO DE SAIDA DE FUNÇÃO
>>109ac    81c7e008
>>109b0    81e80000
```

Figura 4.6 - Código instrumentado da função

4.3 - Teste dos tempos medidos

Para fazer a verificação dos tempos medidos com o novo módulo identificador de laços, colocou-se instruções de *clock()* no programa em análise. Com a impressão dos tempos medidos por estas instruções é possível comparar os dados medidos pelo *Paradyn*.

Segue na figura 4.7 o código de uma função utilizada para testes, contendo dois laços irmãos e um laço filho:

```
int funcao(){
  int a=0;
  // Primeiro for
  for(int j=0; j<100; j++){
    a++;
    //for aninhado
    for(int k=0; k<3000000; k++){
      a--;
    }
  }
  // Segundo for
  for(int w=0; w<500000; w++){
    a++;
  }
  return 1;
}
```

Figura 4.7 – Código da função

Após a inserção de instruções de *clock* ao redor de cada laço, foi obtido os tempos fornecidos pela figura 4.8.

Tempo da função:	9740000.000000 µs
Tempo do primeiro *for*:	9720000.000000 µs
Tempo do segundo *for*:	20000.000000 µs
Tempo do *for* aninhado:	9640000.000000 µs

Figura 4. 8 – Saída do programa com *clock()*

Os tempos medidos pela ferramenta são exibidos na tabela 4.1:

Loop	Tempo do *Loop*	Tempo da Função	Percentagem de Uso (%)	Classificação
Primeiro *for*	9.667192 s	9.683207 s	99.83	Gargalo
Segundo *for*	0.016026 s	9.603596 s	0.17	Não-gargalo
for aninhado	9.727158 s	9.744907 s	99.82	Gargalo

Tabela 4.1 – Medidas de tempo fornecidas pelo *Paradyn*

Note que os tempos de funções medidos tanto para o primeiro *for* quanto para o segundo são praticamente os mesmos. Comparando-se os dados anteriores verificou-se a eficiência da ferramenta para medição dos tempos dos *loops*, pois são praticamente os mesmos dados. Por exemplo, como apresentado na tabela 4.2:

Loop	Tempo do *Loop*	Tempo por *clock( )*	Diferença
Primeiro *for*	9.667192 s	9.720000 s	0,052808
Segundo *for*	0.016026 s	0.020000 s	0,003974
for aninhado	9.727158 s	9.640000 s	0,087158

Tabela 4.2 – Comparativo entre tempos fornecidos pelo *Paradyn* e *clock()*

Outro fato importante é a classificação de um determinado laço analisado como sendo gargalo. Na atual implementação utiliza-se uma taxa de 20% para relacionar o tempo máximo que um laço pode ocupar do tempo total de sua função. Se essa taxa for ultrapassada o laço é considerado gargalo. Senão, ele recebe a classificação de não-gargalo.

4.4 Considerações finais

Este capítulo descreveu alguns testes realizados com o intuito de validarem os principais módulos desenvolvidos no projeto, como o que identifica laços, o que realiza instrumentação e o responsável pela medição de tempo.

CAPÍTULO 5

CONCLUSÕES E TRABALHOS FUTUROS

5.1 Conclusões

Neste trabalho expandiu-se a capacidade do *Paradyn* de analisar desempenho de programas computacionais para a arquitetura *SPARC*. Apesar de atualmente a ferramenta obter uma boa precisão na análise de programas através da instrumentação do grão do tamanho de funções, o refinamento para blocos de repetição foi essencial para determinar e focar com maior rigor o gargalo do programa analisado.

Devido aos avanços tecnológicos e a demanda crescente de programas distribuídos que necessitem de grandes volumes de dados e um poder de processamento elevado, uma ferramenta com esta precisão pode proporcionar informações significativas sobre as estruturas do programa, permitindo uma reestruturação do código do programa nos pontos considerados gargalos pela ferramenta. E isso, conseqüentemente resultaria num código mais efetivo e menos dispendioso.

Além do desenvolvimento da ferramenta, o trabalho proporcionou um grande desenvolvimento intelectual, acrescentando conhecimentos advindos de várias áreas, focando principalmente a área de sistemas distribuídos e paralelos. Este acréscimo foi considerável, tanto no aprendizado teórico quanto prático, pois a implementação exigiu um árduo estudo da codificação da ferramenta, o que demandou análise de código de programação C++, como também estudo da própria arquitetura na qual o projeto foi realizado.

Enfim, os resultados obtidos com o uso da ferramenta modificada pelo projeto mostraram de forma evidente a eficiência da implementação

realizada. A análise refinada de um programa pode fornecer resultados importantes afim de ajudar na melhoria de sistemas de alto desempenho, nos quais simples mudanças podem fornecer bons ganhos de velocidade.

5.2 Trabalhos futuros

Mesmo com o grande desenvolvimento acrescido à ferramenta, novas melhorias ainda podem ser realizadas. Entre elas, pode-se citar:

- identificação e implementação de possíveis técnicas para otimização de gargalos localizados durante a análise de programas, em que se buscará por técnicas de otimização (semelhantes às de otimização de código em compiladores), para que blocos identificados como gargalos possam ser melhorados e, com isso, deixem de ser gargalos;
- implementação de uma função no daemon para o cálculo do tempo gasto com instrumentação, afim de descontar este valor do tempo medido de cada laço;
- otimização das funções de medição de tempo;
- instrumentação dos blocos de repetição da função principal (*main*), pois sua realização foi impedida devido ao fato de instrumentar o código dinamicamente. A função *main* é executada somente no início do programa e, para que análise pudesse ser feita seria necessário inserir a instrumentação antes do início da execução;
- modificação da interface atual do *Paradyn*, para a visualização precisa dos blocos de repetição com as respectivas classificações baseadas nas medições feitas.

REFERÊNCIAS BIBLIOGRÁFICAS

[Cai 2000] Cain, H.W., Miller, B.P. and Wylie, B.J.N.; "A *CALL*graph-based search strategy for automated performance diagnosis", em *Euro-Par 200*, Munich, 2000. Também disponível em *ftp://grilled.cs.wisc.edu/technical_papers/CALLgraphpc.pdf*

[Etn 2002] Etnus; "Etnus Totalview 4.1 multiprocess debugger, disponível em *http://www.etnus.com*, 2002.

[Fah 2000] Fahringer, T. et alii; "Evaluation of P3T+: A performance estimator for distributed and parallel applications", em *Proc. Of 14th International Parallel and Distributed Processing Symposium (IPDPS'00),* pp 229-234, Cancun, 2000.

[Gra 1982] Graham, S.L., Kessler, P.B. and McKusick, M.K.; "Gprof: a *CALL* graph execution profiler"; *ACM Sigplan Notices,* vol.17, n.6, p. 120-126, 1982.

[Hug 2002] Hughes, C. Et alii; "Rsim: simulating shared-memory multiprocessors with ILP processors", *Computer*, vol. 35, n.2, pp 40-49, 2002.

[Jai 1991] Jain. R. (1991). The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. John Wiley & Sons, 2nd edition.

[Kar 1999] Karavanic, K.L. and Miller, B.P.; "Improving online performance diagnosis by tthe use of historical performance data", em *Annals of SC'99*, Portland, 1999. Também disponível em *ftp://grilled.cs.wisc.edu/technical_papers/diagnosis.pdf*

[Kit 1994] Kitajima, J.P. and Plateau, B.; "Modelling parallel program behaviour in ALPES"; *Information and Software Technology,* vol.36, n.7, p. 457-464, 1994.

[Lar 1992] Larus, J.R. and Ball, T.; "Rewriting executable files to measure program behavior"; relatório técnica tr1083 do Depto. de Ciência da Computação da Universidade de Wisconsin (EUA), disponível por ftp anônimo em ftp.cs.wisc.edu, no arquivo endereçado por pub/tech-reports/reports/92/tr1083.ps.Z, 1992.

[Man 1993] Mano, M. M.; "Computer System Architecture", 3nd ed. Englewood Cliffs, NJ: Prentice Hall, p. 310-319, 1993.

[Mil 1995] Miller, B.P., et alii; "The *Paradyn* parallel performance measurement tool", *IEEE Computer,* vol.28, n.11, p. 37-46, 1995.

[Mip 1989] MIPS Computer Systems; "RISCompiler and C programmer's guide"; MIPS Copmuter Systems, Sunnyvale, CA, EUA, 1989.

[Par 2003A] Developer; "Paradyn Developer's Guide", manual escrito pelo grupo de pesquisa do Paradyn, da Universidade de Winsconsin(EUA), disponível por ftp anônimo em ftp.cs.wisc.edu, no arquivo endereçado por paradyn/releases/release4.1/doc/developerGuide.pdf, Release 4.0, 2003.

[Par 2003B] User; "Paradyn User's Guide", manual escrito pelo grupo de pesquisa do Paradyn, da Universidade de Winsconsin(EUA), disponível por ftp anônimo em ftp.cs.wisc.edu, no arquivo endereçado por paradyn/releases/release4.1/doc/userGuide.pdf, Release 4.0, 2003.

[Pau 1994] Paul, R. P.; "Sparc Architecture, Assembly Language Programming, & C", Prentice Hall, 1994.

[Pie 1994] Pierce, J. And Mudge, T.; "IDTrace – A tracing tool for i486 simulation"; relatório técnico CSE-TR-203-94 do Depto. de Eng. Elétrica e Ciência da Computação da Universidade de Michigan (EUA), disponível por

ftp anônimo em ftp.eecs.umich.edu, no arquivo endereçado por techreports/cse/1994/CSE-TR-203-94.ps.Z, 1994.

[Ree 1994] Reed, D.A.; "Experimental analysis of parallel systems: techniques and open problems"; em LNCS 794, *Proc. Of the 7th int 7 Conference on Computer Performance Evaluation: Modelling Techniques and Tool,* p. 25-51, Viena, 1994.

[Rei 1994] Reiser, J.F. and Skudlarek, J.P.; "Program profiling problems and a solution via machine language rewriting"; *ACM Sigplan Notices,* vol.29, n.1, p. 37-45, 1994.

[Rot 2002] Roth, P.C. and Miller, B.P.; "Deep Start: a hybrid strategy for automated performance problem searches", em *Euro-Par 2002*, Paderborn, 2002. Também disponível em *ftp://grilled.cs.wisc.edu/technical_papers/deepstart.pdf*.

[Sah 1996] Sahni, S. and Thanvantri, V.; "Performance metrics: keeping the focus on runtime"; *IEEE Parallel and Distributed Technology,* vol. 4, n.1, p. 43-56., 1996.

[Wal 1992] Wall, D.W.; "Systems for late code modification"; em *Code Generation – Concepts, Tools, Techniques;* R. Giegerich, S.L. Graham (editores), p.275-293, Spring-Verlag, 1992.

[Wea 1994] Weaver, D. L., and Germond T.; "The *Sparc* Architecture Manual", Version 9, Englewood Cliffs, NJ: Prentice Hall, 1994