UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS CURITIBA

GERÊNCIA DE PESQUISA E PÓS-GRADUAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL - CPGEI

LUIZ FERNANDO COPETTI

MIMO – MONITOR HÍBRIDO PARA VERIFICAÇÃO DE RESTRIÇÕES TEMPORAIS EM SISTEMAS EMBARCADOS DE TEMPO REAL

DISSERTAÇÃO DE MESTRADO

CURITIBA
JUNHO -2008.

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

DISSERTAÇÃO
apresentada à UTFPR
para a obtenção do grau de

MESTRE EM CIÊNCIAS

por

LUIZ FERNANDO COPETTI

MIMO – MONITOR HÍBRIDO PARA VERIFICAÇÃO DE RESTRIÇÕES TEMPORAIS EM SISTEMAS EMBARCADOS DE TEMPO REAL.

Banca Examinadora:

Presidente e Orientador:

Prof. Dr. Volnei A. Pedroni	UTFPR

Examinadores:

Douglas P. B. Renaux	UTFPR
Fábio Schneider	UTFPR
Altair Olivo Santin	PUC-PR

Curitiba, junho de 2008.

LUIZ FERNANDO COPETTI

MIMO: MONITOR HÍBRIDO PARA VERIFICAÇÃO DE RESTRIÇÕES TEMPORAIS EM SISTEMAS EMBARCADOS DE TEMPO REAL.

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná, como requisito parcial para a obtenção do grau de "Mestre em Ciências" – Área de Concentração: Informática Industrial.

Orientador: Prof. Dr. Volnei Pedroni

Curitiba

2008

Ficha catalográfica elaborada pela Biblioteca da UTFPR – Campus Curitiba

C782m Copetti, Luiz Fernando
MIMO : monitor híbrido para verificação de restrições temporais em sistemas embarcados de tempo real / Luiz Fernando Copetti. – Curitiba : [s.n.], 2008.
x, 87 p. : il. ; 30 cm

Orientador: Volnei Pedroni
Dissertação (Mestrado) – Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. Área de Concentração Informática Industrial, Curitiba, 2008
Bibliografia: p. 84-87

1. Sistemas de computação. 2. Monitoração híbrida. 3. Processamento de dados em tempo real. 4. Informática industrial. I. Pedroni, Volnei A. (Volnei Antonio), orient. II. Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. Área de Concentração Informática Industrial. III. Título.

CDD 621.3

AGRADECIMENTOS

A Deus, a quem credito muito do sucesso alcançado neste trabalho e que nos fornece uma direção para quando pensamos estar perdidos.

Aos meus familiares, Ana Paula, companheira de todas as horas, minhas filhas Fernanda e Paloma que abriram mão de minha companhia e suporte. À minha mãe, Neusa, que me ensinou como sempre ir em frente apesar das dificuldades e a meu pai, Luiz Victório, por seus exemplos de retidão e correção de caráter.

Aos meus amigos e colegas do departamento de Eletrônica da UTFPR, com suas sugestões, revisões e muitas vezes auxílio em minhas atividades docentes. Agradeço também aos colegas da NSN pelo suporte dado e compreensão durante os momentos mais difíceis.

Ao colega e amigo João Cadamuro Júnior, incansável ajuda na busca pelos objetivos mútuos.

Ao meu orientador Volnei A. Pedroni e demais membros do CPGEI pela oportunidade de ter participado de um projeto de pesquisa que me acrescentou muito tanto em termos pessoais quanto profissionais.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BCET	*Best Case Execution Time* (tempo de execução no melhor caso)
DMA	*Direct Memory Access* (acesso direto à memória)
ERTS	*Embedded Real Time System* (Sistema embarcado de tempo real)
FPGA	*Field Programmable Gate Array* (*Gate array* reprogramável)
IP	*Intellectual Property*
ISR	*Interrupt Service Routine* (rotina de serviço de interrupção)
RAM	*Random Access Memory* (memória de acesso randômico)
RTOS	*Real Time Operating System* (sistema operacional de tempo real)
RTS	*Real Time System* (sistema em tempo real)
SUT	*System Under Test* (sistema sob teste)
TCET	Typical Case Execution Time (tempo de execução no caso típico)
UTFPR	Universidade Tecnológica Federal do Paraná
VHSIC	Very High Speed Integrated Circuit
WCET	Worst Case Execution Time (tempo de execução no pior caso)

RESUMO

Este trabalho apresenta o MIMO (*Minimal Invasive MOnitor*), um monitor híbrido usado em pesquisa acadêmica, mas com possibilidades de uso prático, para o teste de sistemas embarcados de tempo real. Monitores híbridos oferecem a acurácia e baixa intrusão típica de monitores baseados exclusivamente em *hardware* aliados à flexibilidade dos monitores construídos com base apenas em *software*. O monitor, em si, é um sistema embarcado de aquisição de dados, operando em tempo real, com seus próprios requisitos como eficiência e flexibilidade. Os princípios de concepção do MIMO, a estratégia para sua validação e o caso de teste são também apresentados. Um protótipo do MIMO foi construído e assim sua performance pode ser avaliada, com base na monitoração das atividades de um sistema embarcado de tempo real, multi-tarefa e preemptivo.

Palavras-chave : Monitoração híbrida, RTOS *Debug*, FPGA, *Software* embarcado

ABSTRACT

MIMO – Hybrid monitor for temporal restrictions verification in real-time embedded systems.

This work presents MIMO (Minimal Intrusion MOnitor), a hybrid monitor used in academic research, but also suitable for practical use, for testing embedded real-time systems. Hybrid monitors offer the accuracy and low intrusion of hardware-only monitors and the flexibility of software-only monitors. The monitor itself is an embedded real-time data acquisition system with its own requirements, including efficiency and flexibility. The design principles of MIMO, validation approach and tests case are also presented. A prototype of MIMO has been built and its performance could be evaluated, based on monitoring activities of a real-time, preemptive multitask, embedded system.

Keywords: Hybrid monitoring, RTOS Debug, FPGA, Embedded Software

1 INTRODUÇÃO

A face mais visível da computação é associada aos dispositivos normalmente utilizados em tarefas cotidianas e que são reconhecidos por nós pelo termo computador. Segundo Turley [Turley 03], a parcela de computadores embarcada nos mais variados dispositivos representa cerca de 98% do número de dispositivos fabricados. A demanda por tais sistemas, cada vez com maior capacidade de processamento e principalmente maior capacidade de execução de tarefas fez emergir o problema de verificação e/ou depuração de tais sistemas, em condições reais de uso. Em muitos casos, problemas de difícil reprodução só são passíveis de análise em tais condições.

Os sistemas embarcados e de tempo real têm ainda que respeitar restrições temporais. Não basta a apresentação dos resultados computacionais corretos em função dos dados de entrada, mas também que tais respostas estejam disponíveis em um tempo determinado após terem iniciado determinada atividade computacional. A verificação destas restrições requer um planejamento cuidadoso e um bom conjunto de métodos e ferramentas de suporte.

As informações necessárias para tal verificação são coletadas do sistema embarcado em questão por um ente denominado monitor. Este trabalho tem como objetivo apresentar uma ferramenta de monitoração de sistemas embarcados operando em tempo real, para validação de suas restrições temporais, de acordo com o método Dyretiva [Cadamuro 07]. Tal método foi objeto de Tese de Doutoramento no CPGEI.

A monitoração, assim como toda observação de um sistema impõe algum grau de interferência ao mesmo. Denomina-se de *probe-effect* tal interferência. Uma das dificuldades na observação das características dos sistemas embarcados é a mudança de comportamento dos mesmos quando da inclusão da monitoração. Esta interferência é tanto maior quanto maior for a parcela dos recursos do sistema subtraídos de sua função original para destinação à monitoração.

Como forma de minimização desta interferência, umas das propostas é a da permanência da monitoração mesmo no produto final, neste caso um sistema embarcado. O capítulo 2 detalha os tipos de interferência que a monitoração acarreta. Neste texto adotam-se os termos *target* e SUT (*System under test*) como sinônimos do sistema embarcado que é objeto de monitoração.

1.1 OBJETIVO

O objetivo é a demonstração de que o monitor proposto faça a coleta das evidências de execução de programas em sistemas embarcados de tempo real, impondo-lhes, para tanto, a menor sobrecarga possível. Tal restrição é especialmente importante onde os recursos computacionais são restritos. O termo “intrusão” é também usado para referir-se à sobrecarga. Esta não gera trabalho útil e deve, portanto, ser minimizada. Este objetivo é alcançado deixando-se o maior número possível de tarefas necessárias à monitoração para o monitor. A seção 2.3 discorre sobre os problemas decorrentes da adição da monitoração aos ERTS (*Embedded Real Time System*).

1.2 JUSTIFICATIVA E RELEVÂNCIA

As atividades desenvolvidas no CPGEI, área de sistemas de tempo real, compreendem desde métodos para estimação estática de tempos de execução em sistemas embarcados – PERF, até métodos dinâmicos – DYRETIVA. Este último dispõe de uma série de ferramentas de suporte, entre elas um monitor com baixa intrusão.

Em linhas gerais o método DYRETIVA compreende uma metodologia tanto para para o estabelecimento de restrições temporais para um sistema embarcado de tempo real bem como para a verificação destas restrições em condições reais de uso do referido sistema embarcado. O DYRETIVA é abordado em maiores detalhes no capítulo 7.

Não foi encontrado um monitor, nem no âmbito acadêmico, nem disponível comercialmente que satisfizesse as necessidades de baixa intrusão e de flexibilidade impostas pelo método DYRETIVA. Propôs-se então o monitor MIMO.

1.3 OBJETO

O problema:

Como monitorar sistemas embarcados de tempo real com um baixo nível de intrusão, pequeno consumo de recursos do *target* e eliminação do *probe-effect*? O objeto desta dissertação é a proposição de um monitor para ser usado durante os estágios de verificação e validação da fase de testes de um sistema embarcado de tempo real. Além destas fases, o monitor poderá ser usado durante o uso normal do *target*, pois o mesmo continuará gerando as evidências de monitoração. Complementarmente, o sistema de monitoração deve ser compatível com as seguintes premissas:

- Minimizar a intrusão causada pelo sistema de monitoração.
- Ser independente da arquitetura do processador usado no *target.*
- Portabilidade entre diferentes processadores embarcados.
- Compatibilidade com todas as formas de instrumentação do código do usuário, descritas pelo método DYRETIVA.
- Não ser afetado pelo *probe-effect.* Mais precisamente, que o *target* não sofra tal efeito.
- Ser tão simples quanto possível, já que objetiva ser usado em sistemas embarcados de baixo custo e que possibilite a monitoração simultânea de vários *targets*, dado que cada *target* demanda um monitor em separado.

1.4 CONTRIBUIÇÕES

As contribuições deste trabalho são as seguintes:

a. Medição dos tempos de execução de programas em sistemas embarcados com a intrusão possível.
b. Coleta de evidências de execução de programas em ERTS para auxílio na localização de faltas nestes sistemas.

1.5 ESTRUTURA DA DISSERTAÇÃO

O Capítulo 2 mostra a fundamentação teórica sobre temas relevantes para a construção do monitor, objeto desta dissertação: a) conceituação dos ERTS; b) tipos e estratégias de monitoração; c) VHDL, FPGAs e método usado para desenvolvimento do monitor.

Os diversos monitores usados como comparação com o monitor proposto são descritos no Capítulo 3. As bases da arquitetura do sistema de monitoração, o *hardware* do monitor, seu *software* de aplicação e a interface com o *host* são apresentados no capítulo 4.

A validação do monitor com os casos de teste são tratados no Capítulo 5. Como fechamento são apresentados no Capítulo 6 as contribuições deste trabalho, publicação decorrente e propostas de trabalhos futuros.

Sendo este trabalho parte da proposta do método Dyretiva, o Capítulo 7 mostra sucintamente o método Dyretiva e um de seus componentes, utilizado em conjunto com o MIMO, a ferramenta Softscope. Para melhor situar o leitor são também apresentadas as

linhas gerais de um método estático de estimação de tempos de execução em sistemas de tempo real – PERF.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 INTRODUÇÃO

São abordados neste capítulo temas relevantes para a apresentação do monitor no Capítulo 4, quais sejam: ERTS; Validação de especificações temporais de ERTS; Monitoração, com uma introdução à terminologia e principais técnicas utilizadas comumente; especificidades da monitoração em sistemas embarcados de tempo real e a base utilizada para confecção do Monitor – FPGA.

2.2 ERTS

O primeiro tópico abordado conceitua brevemente os sistemas embarcados de tempo real (ERTS). Pode-se considerar um sistema de tempo real todo aquele que deve responder com alguma atividade a estímulos externos em um tempo finito. Burns [Burns 97], ao discorrer sobre tal afirmação, considera que ela englobaria praticamente todos os sistemas computacionais. A diferenciação, segundo ele, decorreria de quão crítica seria a possibilidade de uma resposta não vir no tempo adequado. O restante desta seção é baseado na introdução feita a ERTS por [Burns 97].

Nesta linha de raciocínio, um computador usado para editoração de textos não seria um sistema de tempo real. O computador deve responder às teclas digitadas com o eco das mesmas em uma tela e outras ações. Caso o eco não aconteça, nenhuma catástrofe advirá do fato. Adicionalmente não há nenhuma especificação de tempos de resposta, nem por parte do fornecedor do *hardware*, nem do sistema operacional nem tampouco do *software*, um editor de textos no caso.

Considerando-se a definição acima, há uma subdivisão importante nos ERTS: os ditos *hard real-time systems* e os assim chamados *soft real-time systems*. Nos primeiros há o imperativo de que todas as restrições temporais sejam cumpridas ao passo que no segundo grupo há a tolerância para eventuais perdas de *deadlines*. Este termo é usado na literatura específica para designar o prazo de entrega de determinada computação.

A Figura 2-1 ilustra uma possível aplicação a ser controlada por um sistema embarcado de tempo real e a Figura 2-2 mostra os componentes típicos de tal sistema. O relógio de tempo real do sistema (*real time clock*) fornece ao computador elementos para que este controle adequadamente a transição entre seus estados internos. Uma execução sem

problemas de um ERTS seria, então, uma contínua transição entre estados válidos. Cada estado representa o conjunto de valores tanto do *hardware* quanto do *software*.

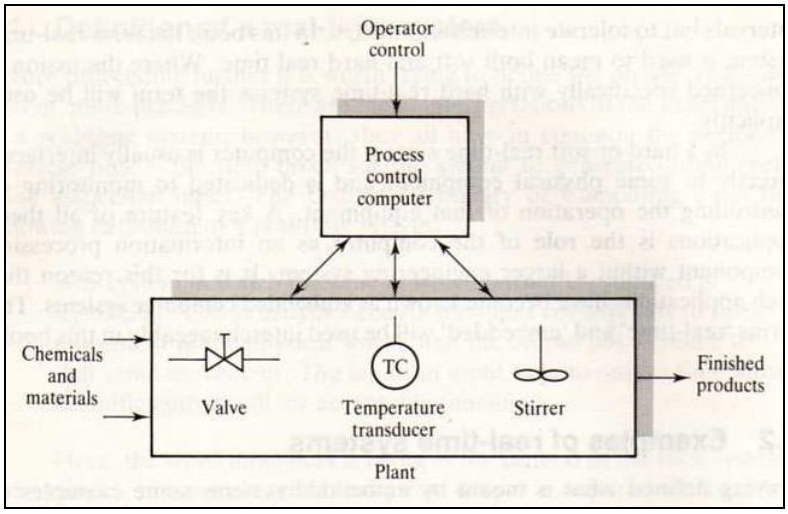

Figura 2-1 – Sistema controlado por um ERTS
Fonte: [Burns 07]

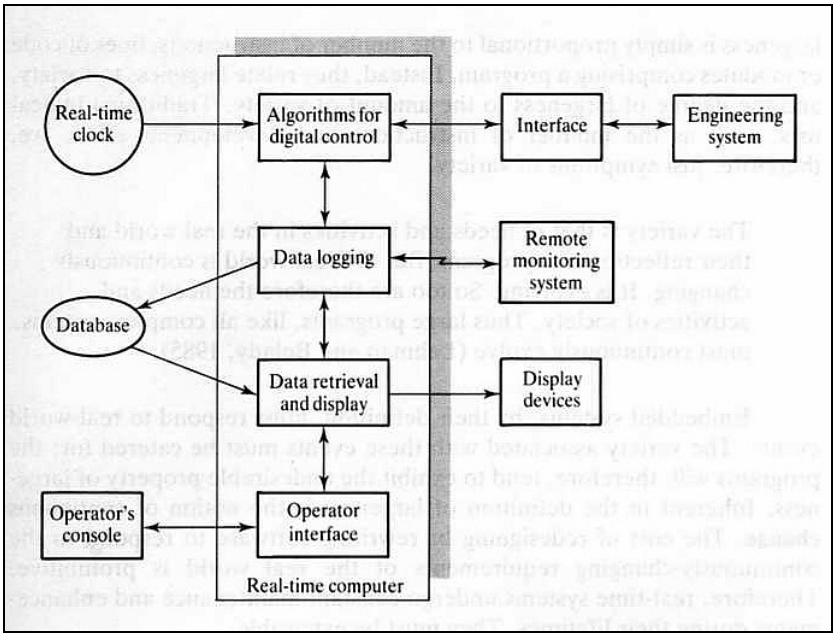

Figura 2-2 – Exemplo de ERTS
Fonte: [Burns 07]

Uma **falta** em determinado elemento de software e/ou de hardware pode levar a uma transição indevida entre **estados válidos** ou até mesmo a transição para um estado **inválido**. Tal fato é denominado **erro**. Caso haja manifestação externa ao sistema deste erro, diz-se

que houve uma **falha**. Perdas de *deadlines* constituem faltas. Componente de hardware deteriorados e *bugs* de *software* são outros exemplos de faltas. A seção 7.1.3 descreve o modelo de faltas do método Dyretiva. Tal modelo permite a criação de outras ferramentas que permitem verificar a ocorrência de faltas a partir de informações coletadas a partir do funcionamento de um ERTS qualquer. A seção 2.3 trata da coleta de tais informações.

Os ERTS de que trata esta dissertação são constituídos por processadores com apenas um núcleo. As atividades concorrentes são denominadas de *tasks* e simulam a existência de processamento paralelo pela execução de cada *task* por um determinado período de tempo. O elemento central que faz tal comutação é denominado de escalonador.

Os primeiros sistemas deste tipo forneciam uma quantidade de tempo igual para todas as tarefas. Tal abordagem se mostrou ineficiente pois algumas tarefas gastavam menos tempo do que lhes havia sido reservado e a outras faltava tempo. Os ERTS têm evoluído no sentido de tornar dinâmica a alocação de tempos para as diversas tarefas. Se, para um ERTS, é possível que o escalonador forneça tempo suficiente para todas as tarefas, em todos os casos de utilização, diz-se que o ERTS é escalonável.

2.3 VALIDAÇÃO DE ESPECIFICAÇÕES TEMPORAIS EM ERTS

A validação de um ERTS com relação às suas especificações temporais é o processo de contrapô-las com os respectivos tempos obtidos na execução real, em estimativas ou em simulação. Se há um determinado *deadline x* para a execução de um determinado trabalho *y* e a especificação temporal *w* impõe que $x < Xmax$, então a validação da restrição temporal envolve a verificação se *x* é menor que *Xmax* para todas as condições de execução do trabalho *y*.

Considerando-se que a especificação temporal de um determinado sistema seja conhecida, o problema passa ser a determinação dos tempos de execução das várias tarefas em um ERTS. Tais tempos podem ser obtidos por simulação, estimação e medição. Os dois primeiros são denominados métodos estáticos e o terceiro, dinâmico.

Os métodos estáticos baseiam-se na análise do código objeto dos programas executados no ERTS. Tal análise pressupõe conhecimento detalhado do tempo de execução de cada instrução e ainda a ordem com que tais instruções serão executadas. Como não há uma ordem pré-determinada de execução destas instruções, as várias possibilidades devem ser verificadas. Define-se como WCET (*worst case execution time*) o pior caso de execução,

ou seja o mais longo. Uma consideração sobre o WCET é feita por Góes [Góes 01]. Ele afirma que o WCET estimado deve ser sempre maior ou igual ao medido.

Os métodos estáticos precisam, ainda, modelar o hardware onde o código será executado. Nos processadores atuais, uma série de recursos é utilizada para aceleração da execução dos programas, tais como *pipelines* e uso de memórias *cache*. Além destes fatores, características inerentes aos modernos processadores tais como refresh de memórias dinâmicas, DMA e interrupções alteram significativamente a ordem de execução das instruções. Linhares [Linhares 01] entre outros propõem modelos para o hardware que permitam estimar a influência de tais fatores. Um trabalho de longo prazo tem sido desenvolvido no CPGEI [Kawamura 99][Renaux 99] [Renaux 02] para a estimação dos tempos de execução em ERTS baseados nos microprocessadores da família 8086. Denomina-se PERF e é melhor detalhado na seção 7.3.

Para que o modelamento de ERTSs seja possível há algumas condições impostas ao sistema sob análise. São citadas por Puschner e Koza [Puschner 89] e se relacionam com características de arquitetura do ESTR que deficultam ou impossibilitam a estimativa, *a priori*, dos tempos de execução de programas. São elas:

- Ausência de rotinas recursivas
- Inexistência de chamadas a funções através de ponteiros para funções
- Utilização de desvios em desacordo com as estruturas de controle de laços das linguagens de programação tais como o *goto*.

Os métodos dinâmicos medem os tempos de execução, ao invés de estimá-los. Burns [Burns 97] chama a atenção que os métodos dinâmicos complementam os métodos estáticos. Este últimos são de preciosa valia para os primeiros estudos e dimensionamentos iniciais de um ERTS. O método dinâmico para o qual o monitor MIMO é destinado é mostrado na seção 7.1.

2.4 MONITORAÇÃO

Em um tutorial sobre monitoração, Schroeder [Schroeder 95] cita vários termos relacionados à monitoração de um programa. São eles: ação, evento, sensor, monitor, instrumentação e intrusão.

Define-se ação como qualquer atividade relacionada ao programa. A execução de instruções ou de rotinas de tratamento de interrupção, chamadas de função ou chamadas de sistema são exemplos de ações do programa cuja duração é não nula.

A definição de evento é a de um fato instantâneo, ou seja, sem duração associada. Um sensor pode ser definido como uma entidade que observa de forma quantitativa parte da aplicação que se deseja analisar. Uma vez disparado, o sensor produz um evento. Se tal disparo ocorre devido à mudança de valor da entidade observada, diz-se que houve um *trace* da entidade observada. O termo rastro também é utilizado neste texto com o mesmo significado. Pode-se também, requisitar ao sensor que reporte o valor instantâneo da entidade observada, o que se denomina de *sampling*.

Pode-se dizer que os eventos são as indicações de atuação dos sensores. Como exemplos de eventos pode-se citar desde uma requisição de interrupção, o começo da execução de uma função ou troca de mensagens entre tarefas em um sistema multitarefa.

Um monitor é um componente de *software* ou de *hardware*, cujo objetivo é capturar eventos, associá-los a uma ou mais marcações e armazená-los para posterior análise. Uma das possíveis marcações é o *timestamp*, necessário para monitorar as limitações de tempo do sistema sob análise. O conjunto de todos os eventos capturados durante a monitoração é chamado de *program trace*.

Instruções de instrumentação, instruções de monitoração ou apenas instrumentação são inseridas no programa, para atuarem como sensores. Como estão localizadas junto aos pontos que se deseja monitorar, terminam por desencadear eventos para o monitor. É possível inserir instruções de monitoração no código do usuário, no sistema operacional ou usar um método alternativo de concentrar toda a instrumentação em uma tarefa do sistema.

Intrusão é definida como o nível de interferência imposto pelo sistema de monitoração à aplicação que está sendo monitorada. A intrusão é o uso de recursos do sistema sob análise para atividades não relacionadas com o seu propósito principal. Recursos tipicamente usados pelo sistema de monitoração são ciclos de CPU, espaço em memória e canais de comunicação. Pode-se também utilizar invasão em lugar de intrusão.

Na década de 1960, monitorar um programa era sinônimo de executá-lo sob a supervisão de um depurador de código, conceito que precisou evoluir para acomodar outras formas de acompanhar a execução de um programa. A classificação de monitores feita por Schroeder [Schroeder 95] dividiu-os em sete categorias, conforme o seu objetivo: confiabilidade, melhoria de desempenho, correção, segurança, controle, teste e depuração e avaliação de desempenho. As características de cada categoria são apresentadas a seguir.

- **Confiabilidade:** é a monitoração feita com o objetivo de verificar que o sistema se mantenha dentro das especificações.

- **Melhoria de desempenho**: em sistemas configuráveis dinamicamente, a monitoração fornece subsídios para a reconfiguração do mesmo, visando melhor performance.
- **Correção**: é a monitoração de uma aplicação para certificar-se de sua consistência com uma especificação formal. Pode ser utilizada para detectar erros em tempo de execução ou apenas como técnica de verificação.
- **Segurança**: é a monitoração que se preocupa em detectar violações de segurança, tais como utilização por um usuário ilegal ou tentativa de acesso a recursos do sistema sem os devidos direitos de acesso.
- **Controle**: são os casos em que o sistema de monitoração faz parte da plataforma alvo, ou seja, é um componente necessário para prover funcionalidade computacional.
- **Teste e depuração**: Monitores que extraem dados da aplicação para fins de validação lógica, ou seja, tem objetivos semelhantes aos depuradores de código.
- **Avaliação de desempenho**: é a monitoração voltada a extrair dados do SUT para uso na avaliação do desempenho do sistema.

É importante notar que um sistema de monitoração pode contemplar mais de uma das categorias apresentadas acima.

Além destas, está sendo definida neste trabalho uma oitava categoria, chamada de permanência, cuja característica principal é de o monitor ser parte integrante do produto final, ainda que a monitoração não esteja sendo utilizada. Em muitos casos, a instrumentação é utilizada apenas durante a fase de testes, sendo removida posteriormente, antes da entrega aos clientes. Esta remoção, notadamente no caso de ERTS, causa uma perturbação no tempo relativo entre os eventos do sistema, alterando as seqüências de execução e fazendo com que o produto final seja diferente do produto testado. Podem, neste caso, ser introduzidas falhas que não existiriam sem a monitoração. Tal fato é conhecido como *probe effect* [Thane 00].

São citadas, também, por Stunkel [Stunkel 91] três tipos de distorção que a monitoração pode causar: padrão de execução, distorção do tempo de espera e distorção da ordem de acesso. A primeira refere-se à mudança relativa entre os tempos relativos a eventos determinados quando considerado o caso do sistema sem instrumentação e com instrumentação. Na Figura 2-3 há um exemplo com duas tarefas concorrentes A e B. Antes da instrumentação a tarefa A provocava a execução de dois eventos nos instantes i e i + 2. Na tarefa B, eventos em t = i + 1 e t = i + 3 seguiam respectivamente os eventos ocorridos

em t = i e t = i + 2. Com a inserção da instrumentação, houve um aumento da diferença de tempo entre os eventos consecutivos para uma mesma tarefa. Este aumento, porém, foi desigual. Note-se que o segundo evento associado à tarefa B, em t = i + 2 ocorre antes daquele relativo à tarefa A, em t = i + 3.

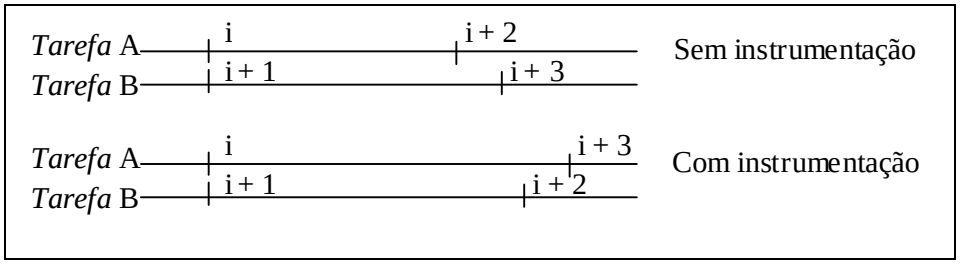

Figura 2-3 – Execution Pattern Distortion
Baseado em [Stunkel 91]

A segunda, distorção do tempo de espera, ocorre quando a instrumentação provoca diferenças no tempo em que as tarefas ficam esperando por recursos indisponíveis ou por algum evento de sincronização. Na Figura 2-4, supondo que o evento desencadeador do bloqueio das tarefas A e B seja o segundo evento, nota-se que a diferença entre os casos com e sem instrumentação é de ordem de bloqueio das tarefas. Na figura utilizou-se o termo *idle* para refererir-se ao estado bloqueado.

O terceiro caso, distorção da ordem de acesso, é devido à diferença de ordem de execução de determinado evento, como nos casos anteriores, porém com a especificidade de se tratar de recursos compartilhados entre processadores em um sistema multi-processado. Pode determinar que determinado processador obtenha o acesso a um barramento compartilhado antes de outro quando considerado o caso sem instrumentação e em ordem inversa quando a instrumentação estiver incluída.

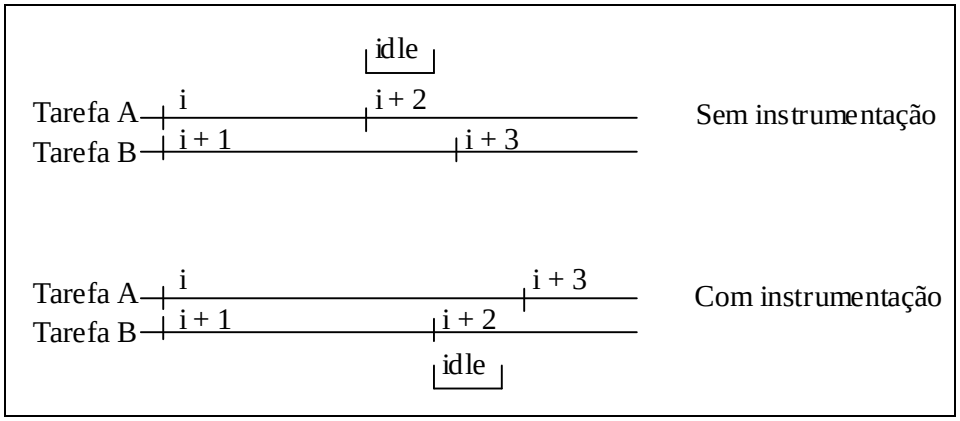

Figura 2-4 – Wait-time Distortion
Baseado em [Stunkel 91]

Após a explanação sobre os problemas que um ERTS sofre quando é ora utilizado com a instrumentação, ora sem, menciona-se uma vantagem adicional da monitoração persistente. É a possibilidade de observar o sistema durante todo o seu ciclo de vida, se necessário. Por outro lado, existe a desvantagem que parte dos recursos do sistema será sempre consumida pela monitoração. Por isso, é importante que a intrusão causada pela instrumentação em monitores persistentes seja minimizada. Esta minimização depende da eficiência do sistema de monitoração, bem como da quantidade de instrumentação inserida durante o processo de desenvolvimento.

A seguir são apresentadas as três estratégias de monitoração comumente utilizadas: exclusivamente por hardware, exclusivamente por software e híbrida, como atesta Mahrenholz [Mahrenholz 01].

2.4.1 MONITORAÇÃO EXCLUSIVAMENTE POR *HARDWARE*

Neste caso os sensores são os próprios barramentos externos do processador. (endereços, dados e controle). Sempre que houver mudanças nos valores dos sinais de tais entidades, informações de monitoração serão armazenadas. Como tais mudanças são conseqüências diretas da execução dos programas, termina-se por monitorar indiretamente as atividades dos programas.

A localização dos sensores é específica para cada sistema. A informação obtida dos sensores (barramentos) deve ser correlacionada de alguma maneira com o código-fonte do programa para prover algum *feedback* útil ao usuário. A implementação de monitores por *hardware* é considerada altamente complexa, devido à amostragem dos barramentos e a correlação entre acessos de memória e o código fonte não ser trivial. Além disso, a evolução dos processadores em geral, e no caso específico, dos embarcados, com vários níveis de *pipeline*, execução especulativa e fora de ordem, predição de desvios, MMU (*Memory Management Unit*), *caches* de dados e de instruções está criando níveis crescentes de complexidade para a criação de monitores por *hardware.*

Outra desvantagem associada com este tipo de monitor é o difícil porte de um monitor por *hardware* de uma arquitetura para outra, ou mesmo entre dois processadores com as mesmas características de *core.* Isto se deve ao estreito relacionamento entre a arquitetura dos barramentos do processador e o monitor.

A principal vantagem dos monitores por *hardware* é a sua inerente característica de completa não intrusão do ponto de vista do *software*. Outra vantagem é a capacidade de observação de eventos não visíveis pelo *software*, como a ativação de uma linha de interrupção ou o uso de um barramento por um controlador de DMA (*Direct Memory Access*).

Em resumo, monitores por *hardware* são completamente não intrusivos do ponto de vista do *software*, o que é muito bom para monitoração de restrições temporais em sistemas embarcados e capazes de observar eventos do mais baixo nível em grande detalhe, mas sua construção é altamente complexa, custosa e muito difícil para porte entre processadores com diferentes arquiteturas de barramento.

2.4.2 MONITORAÇÃO EXCLUSIVAMENTE POR *SOFTWARE*

A estratégia de monitoração unicamente por *software* consiste em estender o software existente com o objetivo de registro de eventos relevantes do programa. Há muitas vantagens no uso dos monitores por *software*: é desnecessário o acesso físico ao SUT, a implementação é simples quando comparada aos monitores por *hardware* e, em caso do monitor ser construído em linguagens e alto nível tem independência da arquitetura do processador usado no SUT. Há, por outro lado, várias desvantagens: a mais relevante é o alto nível de intrusão imposto pelo monitor baseado em *software*, que pode requerer várias considerações no projeto para acomodar as características do monitor. Além disso, o comportamento temporal da aplicação pode tornar-se não determinística devido ao comportamento também não determinístico dos canais de comunicação utilizados para troca de informações entre o SUT e o *host* (computador que controla o monitor).

A instrumentação do programa é o caminho usual para a implementação de um monitor baseado em *software*. Na seção 2.5 são mostrados os principais meios de instrumentação de programas.

2.4.3 MONITORAÇÃO HÍBRIDA

A monitoração híbrida é uma combinação entre as duas estratégias anteriormente expostas, trazendo a baixa intrusão da monitoração por *hardware* e a flexibilidade e portabilidade dos monitores por *software*.

Monitoração híbrida requer usualmente a instrumentação do programa do SUT, que é uma característica dos monitores por *software*. Apesar da necessidade de instrumentação, a quantidade de instruções usadas na monitoração híbrida é significantemente menor do que aquela empregada na monitoração exclusivamente por software para coleta das mesmas evidências de execução. Tais instruções têm a função de apenas sinalizar a ocorrência de eventos para o mundo externo.

Todas as tarefas de monitoração não executadas pelo SUT são delegadas a uma entidade externa, que corresponde à parte de *hardware* do sistema de monitoração. A responsabilidade deste sistema é de:

- Coletar os eventos gerados pela execução das instruções de instrumentação inseridas no programa do SUT;
- Associar tais eventos às necessárias marcações;
- Armazenamento das informações de *trace* e respectivas marcações;
- Comunicação com uma estação de trabalho para envio dos arquivos de trace.

A execução destas tarefas deve ser feita fora da plataforma do SUT, sendo apenas necessária a construção de um elemento de *hardware* no SUT para monitoração da interface usada para sinalização dos eventos para fora do SUT.

A base para construção de um monitor híbrido deve conter tanto elementos de *hardware* quanto de *software*. Em uma primeira análise seriam os seguintes elementos de *hardware* necessários:

- Interface de captura dos sinais provenientes do SUT
- Elementos de memória para armazenar sinais do SUT com velocidade superior à de geração pelo SUT
- Memória de massa com capacidade de armazenar sinais por longos períodos de tempo
- Interface de comunicação com o host, para recebimento de comandos e envio de dados de monitoração
- Elemento de controle das transferências entre a memória rápida e de massa e desta última para o canal de comunicação, usualmente um microprocessador. Este deve ter performance suficiente para que os *deadlines* do monitor não sejam perdidos.
- *Timers* para supervisão

Para que a flexibilidade dos monitores por *software* seja mantida, uma série de parâmetros precisa ser definida de maneira flexível. Alguns requisitos de *software* para manutenção desta flexibilidade:

- Existência ou criação de interface definida entre os elementos de hardware acima citados e o processador
- Característica multitarefa, com suporte de um RTOS.
- Compiladores e depuradores de código para a plataforma escolhida.

2.4.4 MONITORAÇÃO DE SISTEMAS EMBARCADOS DE TEMPO REAL

A monitoração de sistemas embarcados operando em tempo real possui algumas peculiaridades com relação à monitoração de outros tipos de sistema. Primeiramente, os recursos de um sistema embarcado são mais restritos que os demais. As restrições típicas que podem ser aplicadas em sistemas embarcados são: espaço em placa de circuito impresso, consumo de energia, custos de produção, tamanho e tecnologia da memória, velocidade do processador, número de interfaces externas, entre outras [Hennessy 96].

Na seção 2.2 foi conceituado um ERTS. Devido à restrição natural de recursos em tais sistemas, não é fácil para o projetista aumentar a quantidade de memória, espaço de placa de circuito impresso ou velocidade da CPU para propósitos de monitoração. Em vista disso, as atividades de monitoração deveriam impor ao sistema um baixo nível de exigência adicional de recursos.

Outro importante problema observado em sistemas de tempo real é o *probe-effect*. Para evitá-lo, uma instrumentação mínima pode ter caráter permanente. Este adicional de recursos será sempre exigido e deve ter, portanto, custo mínimo.

Pelo registro de todos os eventos de sincronização, escalonamento e comunicação, incluindo interações com processos externos, é possível verificar-se *off-line* o comportamento de tempo real do sistema sem usar observações intrusivas, que poderiam levar a *probe-effects*.

2.5 INSTRUMENTAÇÃO DE *SOFTWARE*

Há, pelo menos, quatro métodos usados para instrumentação de programas: modificação do código binário, modificação do código objeto, modificação do código *assembly* e por fim modificação do código-fonte. Estes métodos são aplicáveis tanto à monitoração exclusivamente por *software* quanto para monitoração híbrida.

2.5.1 MODIFICAÇÃO DO CÓDIGO BINÁRIO

Tal modificação envolve o *disassembly* do programa em formato executável, a localização dos blocos básicos do programa, inserção dos pontos de instrumentação, onde aplicáveis e o *reassembly* do programa em formato executável. Esta tarefa envolve conhecimento detalhado do formato binário bem como da linguagem *assembly* do processador do SUT. Com esta técnica é possível fazer o *trace* de programas cujos arquivos fonte não sejam conhecidos. Além disso, a complexidade desta tarefa para processadores CISC usualmente torna impossível a aplicação desta técnica.

2.5.2 MODIFICAÇÃO DO CÓDIGO OBJETO

Esta abordagem foi introduzida por Cargille [Cargille 92] para permitir a realização de *traces* de bibliotecas e chamadas de sistema sem a necessidade da alteração do *software* do usuário ou do sistema operacional. Esta técnica consiste em modificar-se a tabela de símbolos gerada pelo compilador, criando ligações entre as rotinas das bibliotecas ou chamadas de sistema e o código de usuário. Estas ligações são responsáveis pela execução das tarefas necessárias do *trace* e as funções originais das bibliotecas ou as chamadas de sistema. No retorno das funções, as ligações acima mencionadas podem também executar atividades finais de *trace*, tal como calcular o tempo gasto na execução da função e então retornar ao código de usuário. Esta técnica é limitada, dado que não se pode fazer o *trace* nem do código do usuário nem do sistema operacional.

2.5.3 MODIFICAÇÃO DO CÓDIGO *ASSEMBLY*

A modificação do código *assembly* é uma opção à complexidade da modificação dos arquivos binários e permitem que o monitor a obtenha informações mais detalhadas do que com as modificações em código objeto. A localização dos blocos básicos em linguagem *assembly* é uma tarefa factível. Com esta técnica é também possível localizar e instrumentar estruturas de baixo nível, não visíveis nos arquivos fonte, que foram incluídas no *assembly* pelo compilador quando da implementação de alguma estrutura de linguagem de alto nível ou mesmo de otimizações do código intermediário. Uma desvantagem desta técnica é a sua baixa portabilidade, dado que há ainda uma dependência da linguagem *assembly*.

2.5.4 MODIFICAÇÃO DO CÓDIGO FONTE

Por fim, a abordagem mais popular como forma de instrumentação. Pode ser aplicada manualmente pelo programador ou por alguma ferramenta de instrumentação de código fonte. A ferramenta de instrumentação identifica os blocos básicos nos arquivos fonte e insere as instruções de instrumentação adequadas em cada bloco, se requisitado. A saída da ferramenta de instrumentação é um código instrumentado, que será usado no processo de compilação e *link* para geração do executável final.

2.5.5 RESUMO

A **Tabela 2-1** mostra uma comparação entre os diversos métodos apresentados anteriormente. Dois itens são comparados: a complexidade da tarefa de instrumentar determinado código; a quantidade de informações que se pode obter quando as instruções advindas da instrumentação forem executadas.

Tipo de instrumentação	Modificação do código binário	Modificação do código objeto	Modificação do código assembly	Modificação no código-fonte
Complexidade	muito alta	média	alta	baixa
Quantidade de informação	média	média	alta	alta
Código fonte disponível	não	Sim	sim	sim
Instrumentação automatizável	não	Sim	sim	sim

Tabela 2-1 – Comparação entre métodos de instrumentação

Em resumo, a instrumentação do código fonte é fácil de aplicar e flexível, mas depende da disponibilidade do código fonte. Como forma de facilitar o uso da instrumentação, pode-se automatizar parte do processo, principalmente na parte do *software* da aplicação. A instrumentação automatizada pode gerar, porém, um grande volume de dados durante a execução do programa. Isto requer o uso de técnicas de filtragem e de extração apenas das informações relevantes para a resolução de determinado problema, apenas. Recentemente tem havido propostas, [Larus 94], de armazenar os arquivos de trace em forma comprimida, reduzindo a quantidade de memória necessária para os dados de *trace*.

2.6 FPGA – FIELD PROGRAMMABLE GATE ARRAY

2.6.1 FPGA – JUSTIFICATIVA DO EMPREGO

Apesar de um circuito digital poder ser, a rigor, obtido a partir da simples conexão de circuitos lógicos como portas, contadores, registradores, somadores, um monitor precisa empregar estruturas complexas e que dificultam tal abordagem. A dificuldade de alteração no circuito devido a alterações no algoritmo do monitor é outro fator que dificulta a implementação discreta de um monitor. Deverá haver o emprego de sinais com freqüências relativamente altas, da ordem de poucas centenas de MHz. Isto implica em uso de placa de circuito impresso de várias camadas e com projeto cuidadoso.

Outra opção é a do uso de um sistema de desenvolvimento que contivesse uma FPGA e outros elementos adicionais como memórias. Parte do *hardware* do monitor seria, então, constituída utilizando-se uma FPGA. A grande variedade de estruturas de controle necessárias, como máquinas de estado síncronas, memórias RAM com *dual port* e circuitos digitais de *phase-locked loop*, presentes em apenas um circuito integrado mostram grande vantagem sobre os circuitos discretos.

Um monitor demanda, porém, estruturas de colaboração com o usuário, gerenciamento e comunicação seriais, não realizáveis apenas em hardware. Tais demandas devem ser supridas por um processador. As opções para o processador são a de utilização de tal componente incorporado ao sistema de desenvolvimento, em conjunto com a FPGA. Deve ser ainda avaliada a existência ou não de um processador incorporado à FPGA.

2.6.2 FPGA - CONCEITOS BÁSICOS

Parte da sigla FPGA descreve a idéia básica por trás destes dispositivos: *Gate Array* ou arranjo de portas lógicas, em uma tradução livre, denota a disposição de várias portas lógicas em um único encapsulamento. Na metodologia tradicional de projeto um pequeno conjunto de portas lógicas era encapsulado em um circuito integrado de baixa escala de integração. A interconexão destes vários circuitos integrados perfaz a função desejada. Os GA podem, então, ser encarados como a evolução dos circuitos integrados tradicionais nos quesitos de quantidade de portas lógicas bem como em sua quantidade.

O restante da sigla denota a diferença destes dispositivos FPGA para os *Gate Arrays* – GA. Estes contêm várias portas lógicas conectadas de maneira definitiva e que portanto possuem função específica. Quando se adiciona a eles a possibilidade de modificação destas

conexões, passamos a chamá-los de arranjos de portas lógicas programáveis em campo – *Field Programmable Gate Arrays*. O esquema final de conexões tanto de um GA quanto de uma FPGA é chamado de *netlist*. O problema passa a ser a obtenção desta lista de conexões.

A interconexão entre os vários blocos é função direta do conteúdo de uma memória RAM, também localizada no interior da FPGA. A *netlist* é a base da definição do conteúdo desta memória. Decorrem deste fato duas importantes conseqüências: a) a função executada pela FPGA só é verdadeira após o correto preenchimento desta RAM. b) FPGAs são voláteis, ou seja, se o circuito onde a FPGA estiver inserida for desenergizado, a função lógica executada pela FPGA é suprimida até que o sistema seja novamente alimentado e a RAM de conexões novamente carregada.

Apesar da possibilidade de programação, algumas estruturas fixas, que se mostraram úteis para a maioria dos projetos, foram sendo incorporadas às FPGAs. Estas estruturas, por não possuírem a sobrecarga das linhas de interconexão programáveis são de densidade maior que a parcela programável. Comparando-se uma função que pode igualmente ser implementada em lógica programável bem como via estrutura fixa, a segunda ocupa menor área, consome menos energia e possui melhor performance de *timing*, mas perde para a primeira em flexibilidade.

2.6.3 ESTRUTURA INTERNA

A Figura 2-5 ilustra simplificadamente a estrutura da FPGA empregada na construção do monitor. A seguir uma descrição breve destes blocos:

- CLBs – Configurable Logic Blocks, elemento básico da FPGA que pode tanto conter lógica combinacional quanto Registradores
- IOBs – Input/Output Blocks, blocos especializados para conexão externa do dispositivo, com características como portas bidirecionais, portas com terceiro estado e ainda sinais diferenciais.
- BRAM - Block RAM, memória *dual-port* RAM.
- Multiplier Blocks, multiplicadores de duas palavras de 18 bits.
- DCM - Digital Clock Manager, possibilita multiplicação, divisão e deslocamento de fase para sinais de relógio.

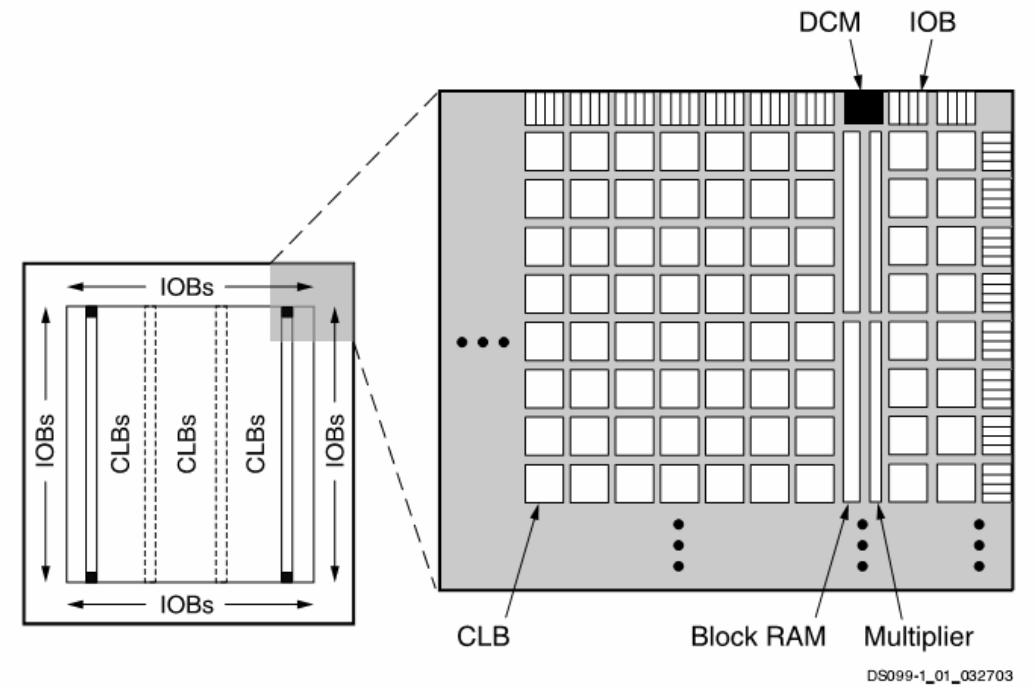

Figura 2-5 – Spartan-3 Family Architecture
Fonte: [Xilinx 05]

2.7 VHDL – VHSIC *HARDWARE DESCRIPTION LANGUAGE*

Uma das maneiras de gerar-se a *netlist* para circuitos programáveis é o uso de linguagens de descrição de hardware (HDL – *hardware description languages*). Tais linguagens permitem que a representação e então o projeto de circuitos digitais seqüenciais e combinacionais seja feito de maneira textual. Os comportamentos esperados do hardware são descritos nestas linguagens. Ao contrário de linguagens de programação de uso geral para computadores, tais linguagens permitem a descrição de processos verdadeiramente concorrentes. As linguagens de programação de computadores usualmente emulam o paralelismo de processos pela divisão do tempo de processamento de uma CPU entre os vários processos. Se a latência para o atendimento das demandas de cada processo for suficientemente pequena em relação à duração de cada processo, tem-se a percepção, ilusória, de paralelismo.

Dentre as mais comuns cita-se VHDL [Pedroni 04], Verilog e mais recentemente System C. Dada a experiência anterior e disponibilidade de ferramentas, optou-se por VHDL. Cohen [Cohen 03] cita algumas de suas características importantes: Linguagem não proprietária; largamente utilizada; similar a linguagens de programação como Ada e C;

suporte a simulação incluindo eventos, *timing* e concorrência; linguagem capaz de documentar estruturas de controle, máquinas de estado, hierarquias e finalmente capacidade de uso para síntese de circuitos em dispositivos programáveis como FPGAs.

2.7.1 NÍVEIS DE DESCRIÇÃO E INFERÊNCIA

Cohen descreve também níveis de descrição da linguagem. Neste trabalho foram majoritariamente utilizados os níveis estrutural e *Register Transfer Logic* – RTL. O primeiro serve à interconexão entre componentes dentro de um mesmo nível hierárquico ou ligando componentes de nível hierárquico diverso. A Figura 2-6 ilustra o modelo utilizado para representar comportamentos no nível RTL: *Registers* e *Combinatorial Logic*. *Datain*, *Dataout* representam sinais elétricos externos ao bloco, *Clock* sincroniza o armazenamento de dados pelos *Registers*. Sinais internos podem gerar outros sinais através de lógica combinacional – *Combinational Logic*.

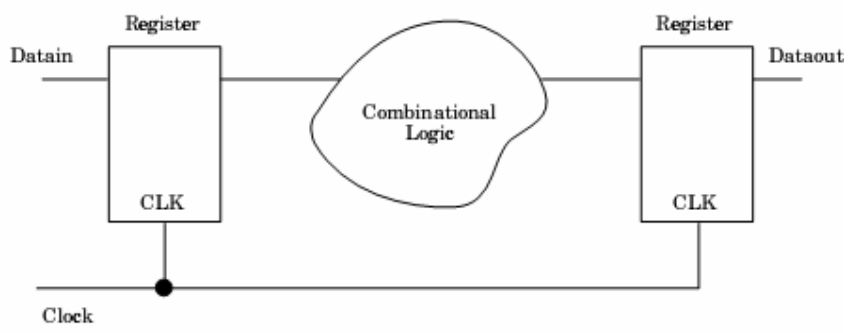

Figura 2-6 – *Register and cloud diagram*
Fonte: [Perry 02]

A utilização de VHDL para programação de FPGAs consiste em submeter o conjunto de arquivos texto constituintes de determinado projeto a uma ferramenta chamada compilador e depois a um sintetizador. A operação conjunta destas duas ferramentas produz a *netlist*.

Analisando-se o esquema simplificado apresentado na Figura 2-5, os elementos CLBs e Multiplicadores são, comumente, representados no nível RTL. IOBs, BRAMs, e DCMs têm suas conexões explicitamente definidas no nível estrutural. Uma característica da interação entre os compiladores VHDL e as ferramentas de síntese é a capacidade do conjunto de inferir o emprego de determinado bloco pelo uso, por parte do programador, de construções semânticas de padrão conhecido.

Algumas estruturas podem, no entanto, ser de difícil ou até mesmo impossível representação em termos de linguagem. Nestes casos a abordagem da representação estrutural deve ser usada e apenas as conexões externas dos blocos fixos representadas em termos textuais. Um exemplo desta situação é o uso de blocos fixos que possuem componentes analógicos, como PLLs. Uma série de parâmetros e as conexões externas são providas de maneira textual.

2.7.2 BLOCOS COM "PROPRIEDADE INTELECTUAL"

A evolução do uso tanto de FPGAs quanto de outros elementos programáveis levou ao surgimento de provedores de blocos funcionais chamados comumente de IPs . Tais blocos são comprados de companhias que garantem sua aderência a determinada especificação funcional. Podem ser adquiridos tanto na forma de código fonte VHDL quanto na de *netlist*, específico para determinada família de dispositivos ou mesmo para um dispositivo específico.

O esquema de licenciamento do uso dos blocos varia, podendo ser gratuito, por projeto específico ou até mesmo por unidade comercializada que contenha o *IP*. No caso específico da construção do MIMO, o processador *softcore* é um IP fornecido gratuitamente com o sistema de desenvolvimento, sendo, porém vedado o acesso ao código VHDL fonte do mesmo. O termo *softcore* refere-se ao fato deste IP ser constituído por estruturas programáveis da FPGA. Os processadores *hardcore*, por sua vez, são componentes fixos da FPGA. Novamente, um processador *hardcore* é mais poderoso e menos flexível. Por questões de otimização de performance, não raro é o fato de processadores *softcore* lançarem mão de estruturas fixas da FPGA. Exemplo freqüente é o uso de multiplicadores em tais processadores.

2.8 METASTABILITY

Quando se propõe sistemas de aquisição de dados, é de fundamental importância que se verifique em que domínio de *clock* determinado sinal é gerado e se este sinal deve passar para outro domínio de *clock*. Entende-se por domínio de *clock* o conjunto de sinais em um circuito digital síncrono comandado por determinado *clock*. Um sinal assíncrono não pertence a nenhum domínio de *clock*.

Algumas técnicas simples devem ser aplicadas a um sinal que precise cruzar as fronteiras de *clock*, ou seja passar a ser controlado por um *clock* diferente daquele que o

originou. Tais técnicas envolvem registros múltiplos do sinal que deve cruzar a fronteira de *clock* e a verificação da constância da mudança de valor do sinal de saída após múltiplos registros. A Figura 2-7 e a Figura 2-8 ilustram a questão. Em alguns casos é ainda necessária uma combinação lógica entre os sinais Q do bloco DEMET e Sync_signal (da Figura 2-8) para obtenção de sinais de sincronismo entre os dois domínios de *clock*.

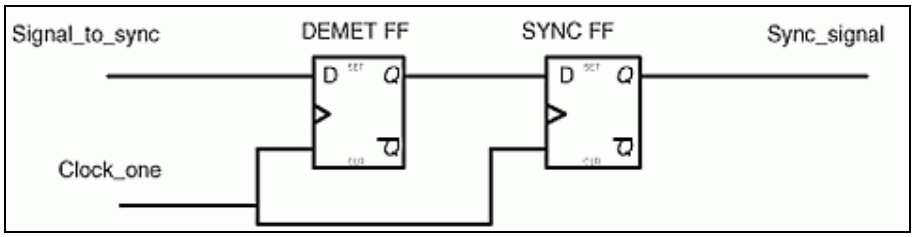

Figura 2-7 – Registros múltiplos para mudança de domínio de clock
Fonte: [Parker 04]

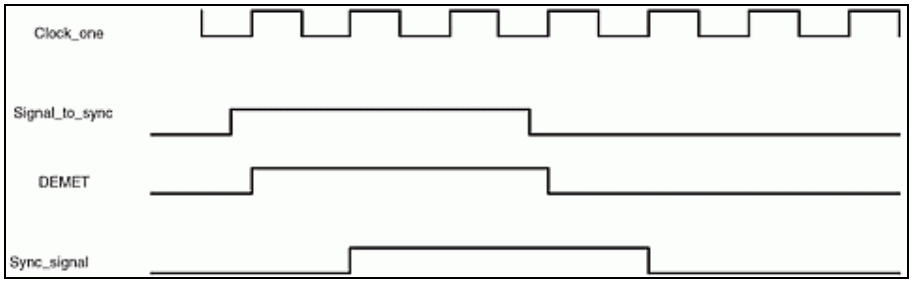

Figura 2-8 – Sinal resultante
Fonte: [Parker 04]

Uma condição básica para o uso das técnicas acima descritas é a de os sinais de entrada devem ter períodos de duração tanto em nível lógico alto quanto em nível lógico baixo maiores que o período do sinal de *clock* do domínio de *clock* de destino.

2.9 RESUMO

Descreveu-se a base teórica de monitoração de programas em sistemas embarcados, suas características principais, restrições a que cada método está submetido e sua terminologia básica. A instrumentação do programa a ser monitorado foi discutida em cada um dos tipos de monitoração apresentada.

Por fim, fizeram-se considerações sobre a técnica de realização possível para o monitor. Uma introdução aos elementos constituintes de FPGAs de forma genérica, linguagem de programação utilizada e mais especificamente detalhes de uma FPGA comercial. Considerações sobre *metastability* fecham o capítulo.

3 MONITORES EXISTENTES

3.1 INTRODUÇÃO

Neste capítulo serão apresentados os principais monitores encontrados na literatura. Será feita uma breve explanação a respeito de cada um deles e suas classificações quanto aos critérios já citados no capítulo de fundamentação teórica. São apresentados a seguir monitores por hardware como o VDS e TLM, um monitor híbrido CODETEST-HWIC e finalmente três monitores por software: CODETEST-SWIC, WINDVIEW e SHARK.

São analisadas características gerais de cada um deles como tipo de monitoração, plataforma alvo e tipos de interface externa para com os respectivos SUTs. Ao final do capítulo é feita uma comparação entre os monitores com os resultados apresentados conjuntamente em uma tabela. Conhecidas as vantagens e desvantagens de cada monitor, passa-se à discussão que tem por objetivo justificar a criação de um novo monitor que contemple as necessidades não cobertas pelos monitores descritos.

3.2 DESCRIÇÃO DOS MONITORES

3.2.1 VDS MONITOR

Um exemplo de equipamento construído especialmente para coletar informações por meio de amostragem de sinais de *hardware* é o módulo de interface proposto por Tsai e outros [Tsai 90]. Ele foi construído para monitorar processadores Motorola 68000 com sistema operacional UNIX. O módulo possui quatro blocos principais: Unidade de Controle da Interface, Unidade Duplamente Processada, Unidade de Qualificação e Memória de Alta Velocidade. Tal sistema é eminentemente um monitor baseado unicamente em *hardware*. Uma ilustração deste sistema de monitoração, incluindo o módulo de interface, é mostrada na Figura 3-1.

Neste modelo, o sistema de monitoração é ligado ao SUT através da Unidade de Controle da interface, que é capaz de acompanhar as atividades do processador principal sem interferir no sistema. A Unidade Duplamente Processada é equipada com dois processadores iguais ao processador do SUT. Eles são responsáveis por reproduzir as ações do processador principal a partir do ponto inicial de amostragem. As condições de início e de fim de amostragem são determinadas e processadas pela Unidade de Qualificação, enquanto a

Memória de Alta Velocidade registra os eventos ocorridos nos barramentos do processador principal para posterior tratamento pela Unidade Duplamente processada.

A monitoração é iniciada através de uma solicitação feita pela Unidade Duplamente Processada ao processador do SUT, utilizando uma interrupção de baixa prioridade. Na rotina de atendimento o SUT e o monitor sincronizam seus estados para que, a partir deste ponto, ambos possam executar em paralelo o mesmo código, com os mesmos valores de variáveis, obtendo os mesmos resultados. É fato que o processo de sincronização introduz uma determinada latência, mas a partir daí o SUT não sofrerá mais interferência do Módulo de Interface.

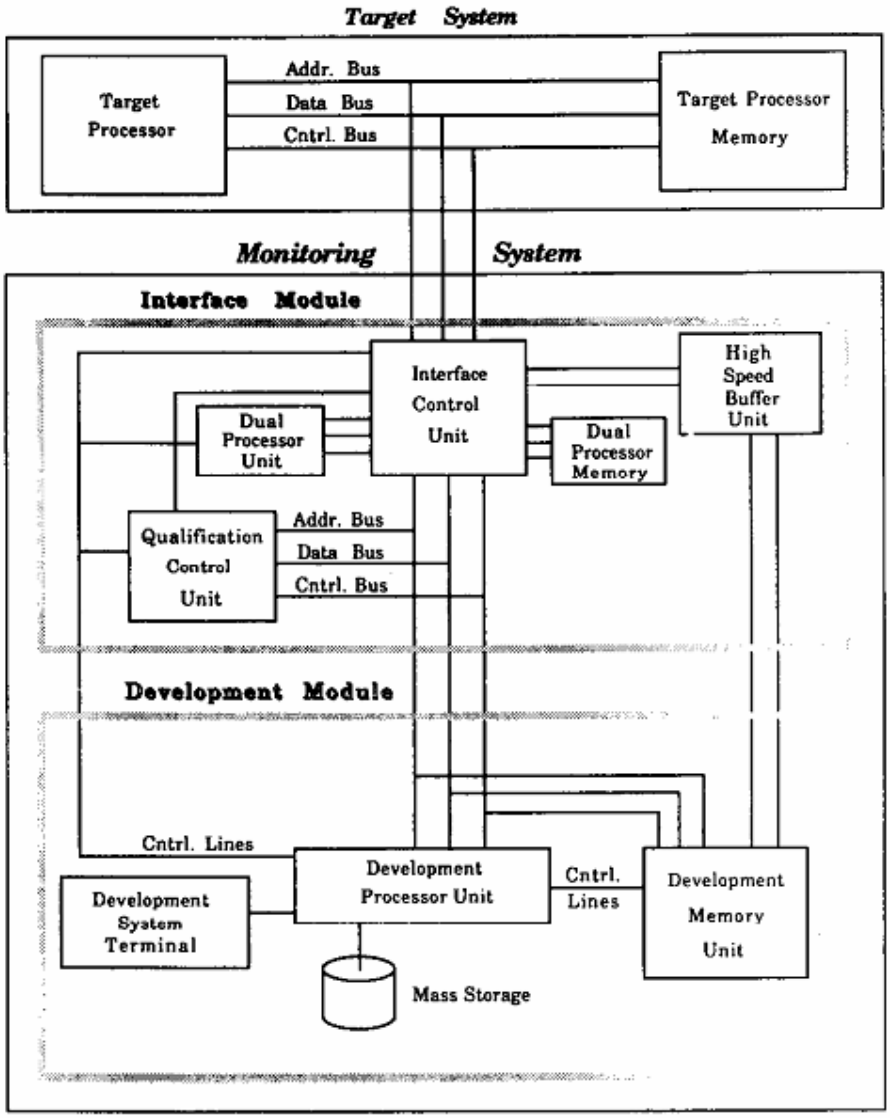

Figura 3-1 – Arquitetura de Monitor para interferência nula
Fonte: [Tsai 90]

Há porém um problema nesta abordagem: Na busca por faltas – ver definição na seção 2.2 -, a latência para começo da monitoração poderá representar até mesmo a não visualização das mesmas, caso a falta seja transiente. Pode-se argumentar que o processo de monitoração é iniciado por uma interrupção de baixa prioridade gerada para o SUT e que portanto, quando do atendimento da mesma, o SUT não sofreria os efeitos da intrusão. A questão que se coloca, então, é a de que tal monitor não poderia começar o armazenamento de eventos quando o SUT estivesse operando sob alta carga de processamento, ou seja quando os requisitos de tempo real do sistema estivessem mais à prova.

3.2.2 TLM

Braga [Braga 99] descreve a implementação de um sistema de monitoração chamado TLM – *Time Line Monitor*. Tal monitor pode ser usado para registro de eventos em um microcontrolador de 16 bits 80186. Entre outras funções, permite que as atividades do sistema operacional PET [Renaux 96] sejam registradas em um diagrama de linha de tempo, como o mostrado na Figura 3-2.

O TLM é composto por três módulos: Instrumentador, Monitor e Rastreador. O instrumentador modifica o código fonte original do programa a ser monitorado e permite variadas modalidades de monitoração, definidas como curta, normal e longa. A instrumentação curta é a menos intrusiva, disponibilizando menor quantidade de informação ao passo que a instrumentação longa permite a exteriorização de maior quantidade de informação com um conseqüente aumento na intrusão. A instrumentação normal é um meio-termo entre as duas citadas acima.

As instruções introduzidas fazem acesso a periféricos mapeados em endereços de memória. Desta forma pode-se exteriorizar um evento a partir da execução destas instruções de monitoração. O instrumentador pode ser usado tanto para códigos escritos em C quanto em C++, conforme mostra a Figura 3-3.

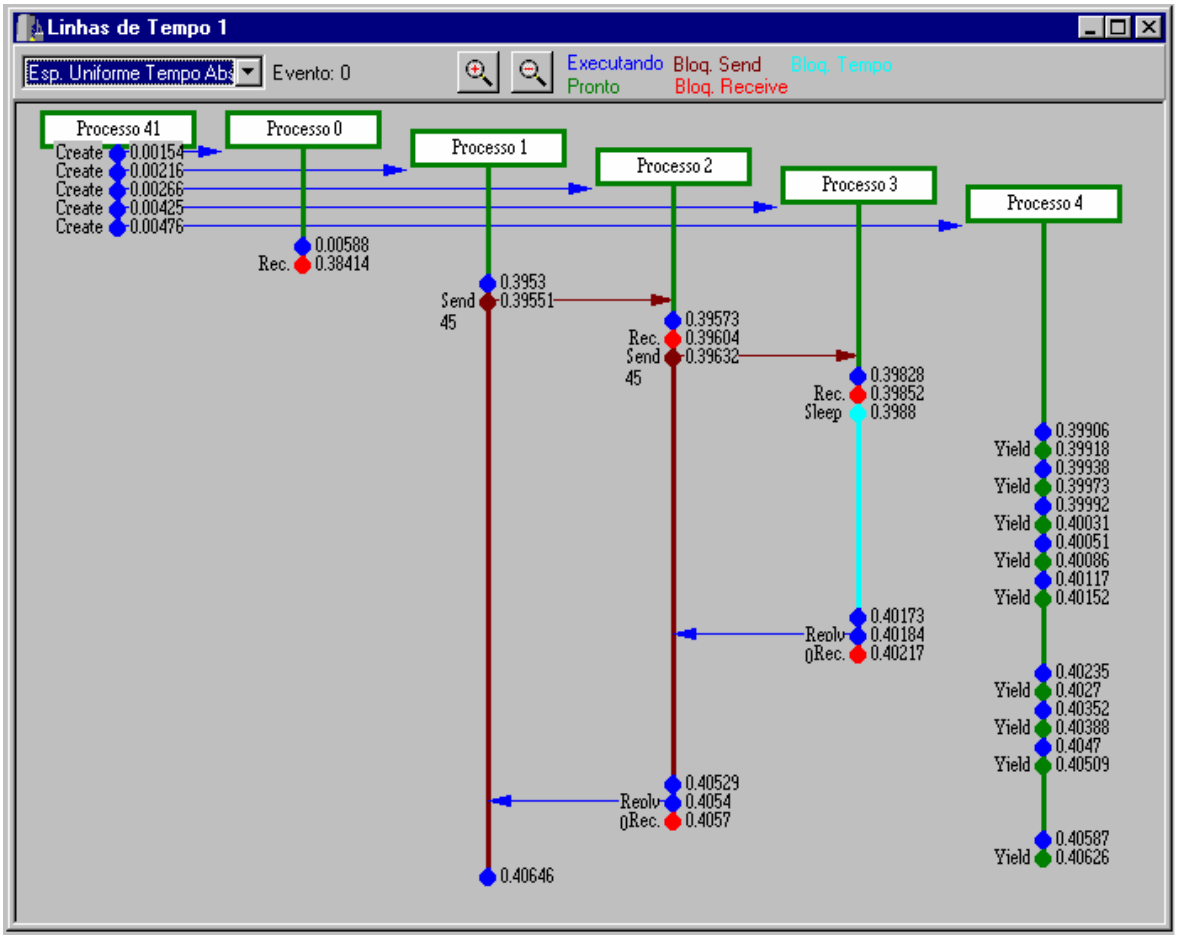

Figura 3-2 – Diagrama de linha de tempo
Fonte: [Braga 99]

```
  /* Definições em C */
if ( x > y ) {
     instr (grIf, 1);
     {
     int a = InitA();
     int b = InitB();
     char c;
     printf ("A=%d\n",a);
     printf ("B=%d\n",b);
     c = getch();
     ...     }
     }
```

```
     //Definições em C++
if ( x > y ) {
     instr (grIf, 1);
     {
     int a = InitA();
     printf ("A=%d\n",a);
     int b = InitB();
     printf ("B=%d\n",b);
     char c;
     c = getch();
          ... }
     }
```

Figura 3-3 – Instrumentação do código fonte
Fonte: [Braga 99]

O monitor externo pode registrar tanto eventos de *software* quanto de *hardware*. A Figura 3-4 ilustra a estrutura interna do monitor. O SST – Sistema Sob Teste é conectado ao monitor por um barramento paralelo que contém sinais de dados, endereços e outros

relevantes para o microprocessador do SST, no caso 80186EC. A sigla SST tem o mesmo significado que a sigla SUT. Foi utilizada em consonância com o autor [Braga 99].

Os eventos são detectados pelo bloco de Identificação de eventos: um evento de software é caracterizado pela identificação de endereços reservados às instruções de monitoração; eventos de hardware são resultado da comparação de valores dos sinais de interesse lidos em dois *clocks* subseqüentes. Havendo diferença entre os valores dos sinais um evento de *hardware* é armazenado. Para os dois tipos de evento, um temporizador integrado ao monitor fornece a informação de tempo relativo a ser registrada junto com as outras informações do monitor.

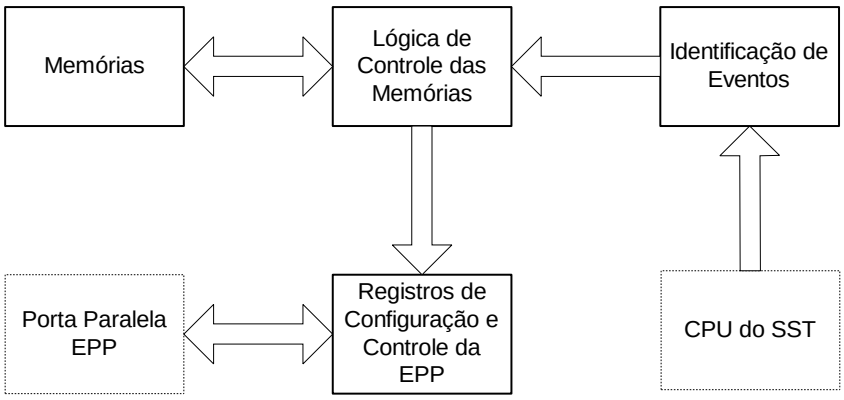

Figura 3-4 – Diagrama de blocos do TLM
Fonte: [Braga 99]

O rastreador controla o monitor e permite a visualização dos dados coletados pelo monitor. Baseado em arquivos de descrição gerados em tempo de instrumentação, permite a identificação de eventos de hardware e software. Controla, ainda, qual das vistas será usada para apresentação dos dados de monitoração. A vista apresentada no início desta seção é a linhas de tempo, mas também podem ser gerados grafos de segmentos, visualização do código fonte correspondente aos arquivos de descrição e histogramas apresentando a ocupação do processador.

Devido a restrições dinâmicas, a captura de eventos simultâneos de hardware e de software não é suportada. Outra limitação é a da aquisição de um evento de hardware consumir 3 ciclos de clock.

3.2.3 CODETEST

Codetest [Greenwalt 03] é uma ferramenta de análise de software da empresa Metrowerks. Além de interfaces de controle e de apresentação de resultados, possui uma estrutura de monitoração de execução de software. Para tanto, pode lançar mão de dois tipos de monitor: HWIC (*hardware in circuit*) e SWIC (*software in circuit*). O primeiro é um monitor híbrido e o segundo um monitor por software.

A Figura 3-5 mostra esquematicamente o monitor com o HWIC e a Figura 3-6 com o SWIC.

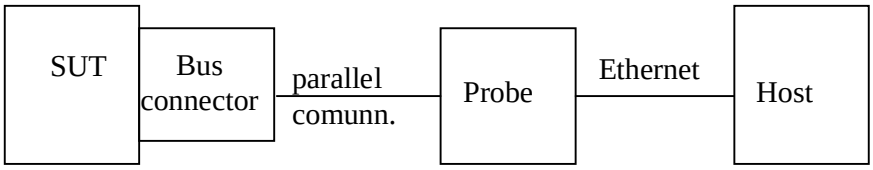

Figura 3-5 – CodeTest *hybrid monitor*
Baseado em [Greenwalt 03]

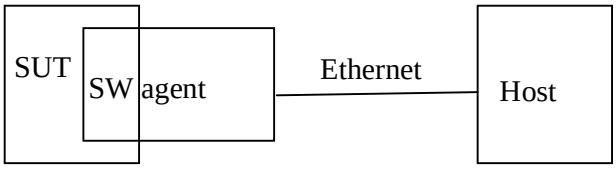

Figura 3-6 – CodeTest *software based monitor*
Baseado em [Braga 99]

O primeiro é mais caro e de operação mais difícil, devido à necessidade de acesso aos barramentos do processador via *Bus Connector*. A intrusão na primeira abordagem é menor que na segunda, dadas as diferenças entre monitores híbridos e por *software* já discutidas anteriormente. Outro ponto relevante é a menor precisão no registro do *timestamping* relativo a cada evento, dado que o agente de software compilado junto com o código do SUT impõe alguma sobrecarga ao SUT para cálculo desta marcação. O autor cita o fato de ser o segundo mais fácil de ser portado para um RTOS, não indicando, porém qual seria a razão precisa. Uma possível explicação para tal afirmação seria a da excessiva intrusão do segundo monitor, inviabilizando seu uso quando da existência de restrições temporais.

De maneira análoga ao segundo monitor apresentado (TLM), um instrumentador é necessário. Este é responsável pela inserção dos *tags*. Tais elementos são os sensores que gerarão os eventos a serem armazenados no *host*. Existem *tags* para indicação de início e fim de funções bem como para geração de informações a respeito do "caminho" de execução de determinada parte do software. Uma aplicação de *tags* pelo CodeTest é mostrada na Figura 3-7 onde aparecem tanto o código não instrumentado, à esquerda, quanto o mesmo após instrumentação, à direita.

Embora a quantidade de informação de cada um destes *tags* seja diferente, uma mesma quantidade de *bytes* é usada para sua exteriorização (4 *bytes*, 32 bits). Também não há possibilidade de geração de eventos com quantidade de informação superior a 32 bits.

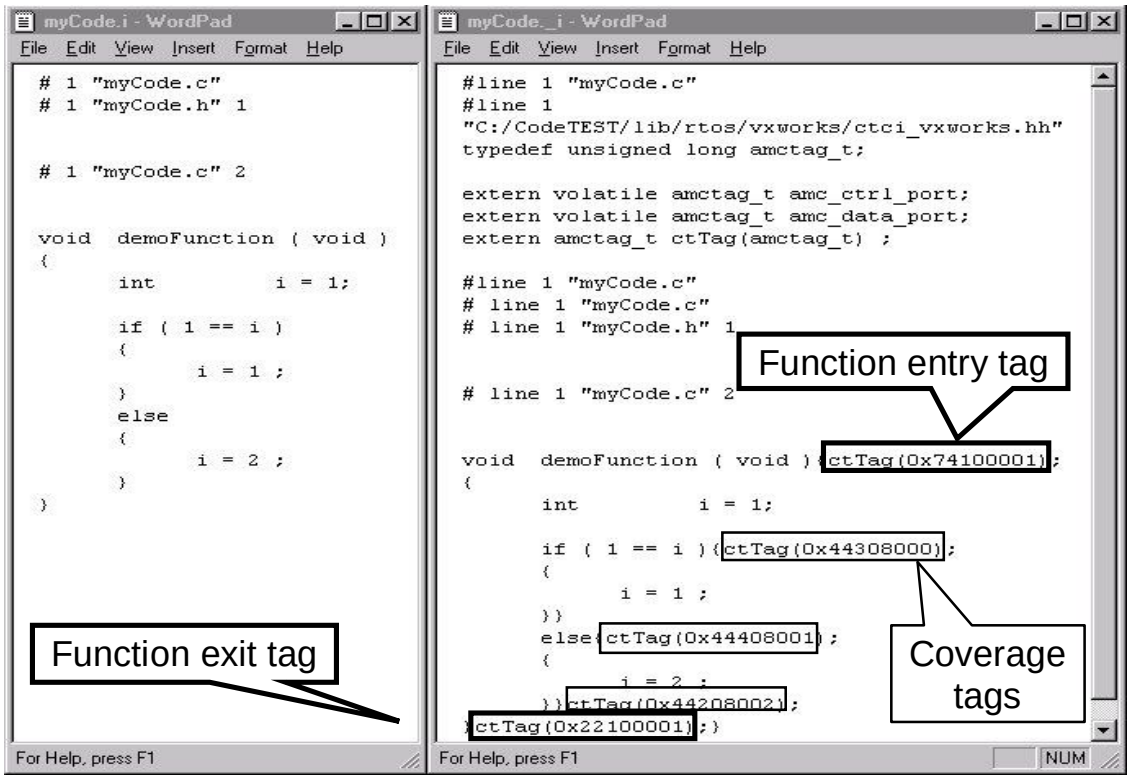

Figura 3-7 – Instrumentador da ferramenta CodeTest
Fonte: [Greenwalt 03]

3.2.4 WINDVIEW

Este é um monitor exclusivamente por software para o RTOS VxWorks [Wind 95], da empresa WindRiver. Foi apresentado inicialmente por Willner [Wilner 95]. Para que possa funcionar, o conjunto aplicação – sistema operacional deve ser recompilado em conjunto com uma biblioteca específica para monitoração. Segundo recomendações do fornecedor deste sistema [Wind 99], tal biblioteca deve ser removida antes da produção para entrega final. Na Figura 3-8, podemos ver um exemplo de saída das informações coletadas pelo monitor e já processadas.

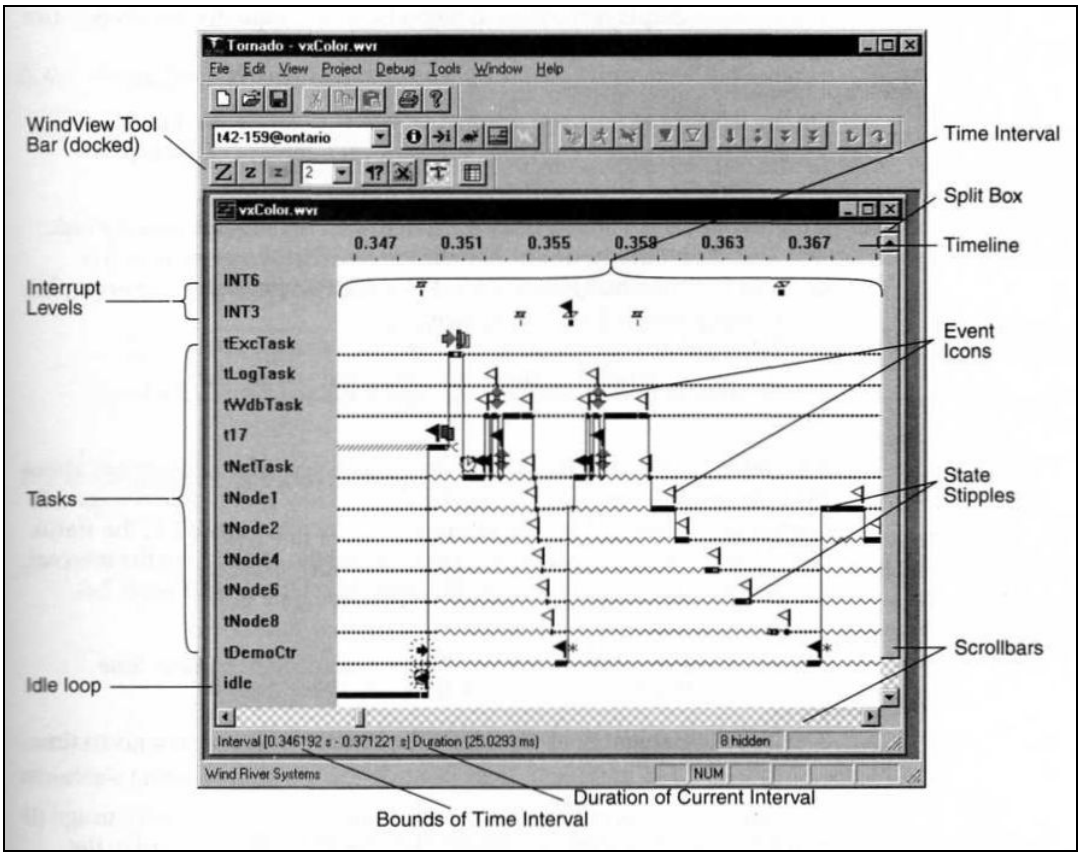

Figura 3-8 – Resultado da captura de dados da ferramenta CodeTest
Fonte: [**Wind 99**]

Permite três níveis diferentes de monitoração, com diferentes níveis de intrusão. O primeiro armazena apenas as informações de troca de contexto (*Context Switch Logging Level*). O contexto atual do sistema refere-se a qualquer seqüência de execução, incluindo-se aí as *tasks*, Interrupt Service Routines (ISRs) ou o *idle loop* do *kernel* do sistema operacional. Este é o menos intrusivo dos três modos de operação.

O segundo modo de operação (TST – *Task State Transition Level*) inclui o registro de todas as possíveis transições de estado, sendo que estas podem ou não causar trocas de contexto, a depender das condições do SUT.

Se o detalhamento necessário do funcionamento do SUT for ainda maior, o AIL level (*Additional Instrumentation Logging Level*) é requerido, tendo, porém, a maior intrusão entre os três modos. Todas as informações dos dois níveis anteriores são armazenadas, como também todas as operações relativas ao sistema operacional, tendo as mesmas sido ou não relevantes para uma transição de estado ou de contexto.

A funcionalidade de *trigger* está disponível e o monitor pode ser programado para que comece a captura de informações assim que o SUT receber o evento desejado.

Este monitor é dedicado a um sistema operacional específico, de código fechado e está sujeito ao *probe-effect*, já que a monitoração é retirada do *software* do SUT de produção.

3.2.5 S.HA.R.K. MONITOR

Valpereiro e Pinho apresentam uma abordagem para a monitoração do sistema oparacional S.HA.R.K. e suas aplicações [Valpereiro 06]. O foco dos autores está na separação das tarefas de monitoração daquelas do sistema operacional e das aplicações usuais. Na Figura 3-9 vê-se o diagrama geral do *framework* de monitoração, partindo-se do nível superior, com os requisitos de monitoração até o nível inferior, onde é mostrada a aplicação de monitoração como mais uma das aplicações do sistema.

Além da aplicação de monitoração, é necessária a adição dos *probes* nos pontos do sistema operacional ou da aplicação que se deseja monitorar. A Figura 3-10 mostra o exemplo da inserção de um *probe* em uma função do sistema operacional.

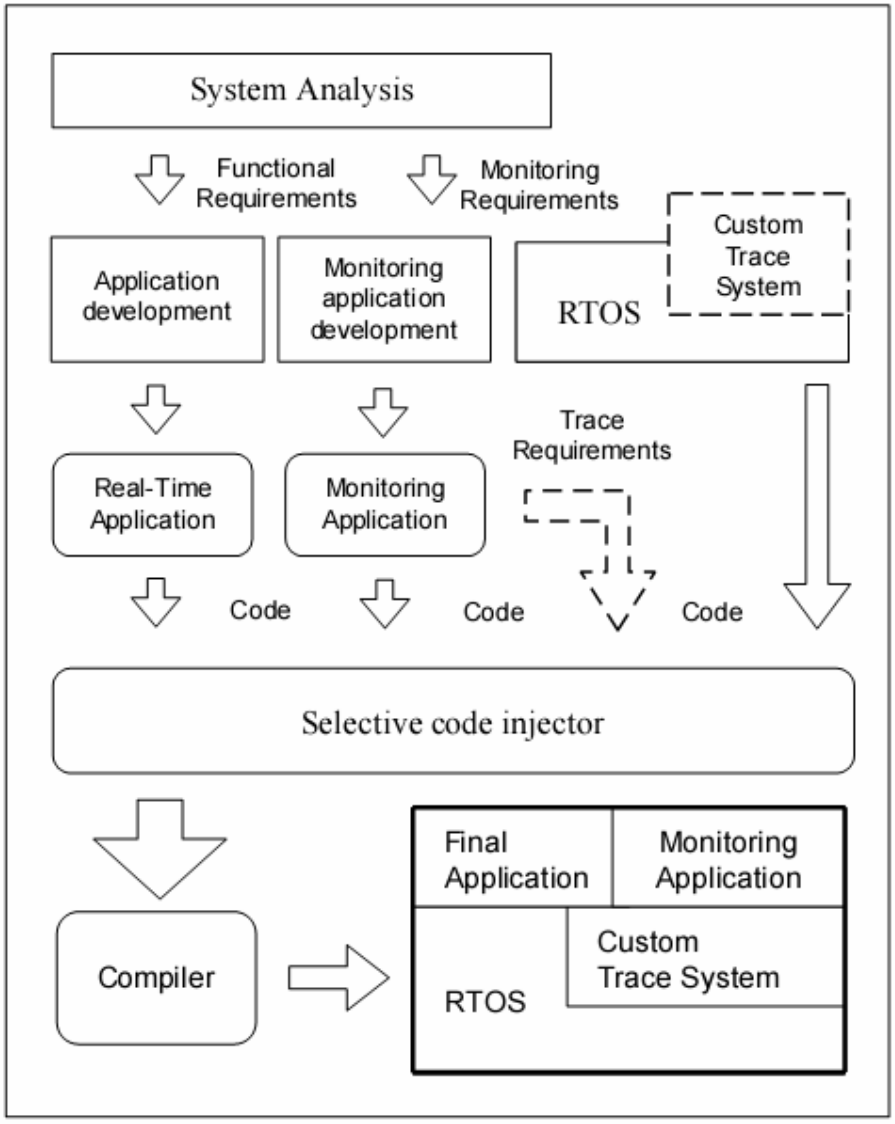

Figura 3-9 – *Run –Time monitoring framework*
Fonte: [Valpereiro 06]

```
int mutex_lock (mutex_t *mutex) {
   int val;
   mutex_resource_des *m;

   mutex_func_event e =
       MUTEX_EVENT_LOCK_START (mutex);

   // Check for init errors ...

   // Get the module for this mutex policy
   m = resource_table[mutex->mutexlevel];

   // Start lock
   TRACE_EVENT(e, sizeof(mutex_func_event));
   val = m->lock (mutex->mutexlevel, mutex);

   e = MUTEX_EVENT_LOCK_END (mutex);
   // End lock
   TRACE_EVENT(e, sizeof(mutex_func_event));

   return val;
}
```

Figura 3-10 – Colocação de *probe* em uma função do sistema operacional
Fonte: [Valpereiro 06]

Além da colocação dos *probes*, o framework gera um esqueleto da aplicação de monitoração que deve ser preenchido pelo programador. Esta aplicação é responsável pela coleta das informações de trace coletadas pelos probes. Os autores citam uma série de cuidados a serem tomados quando da inserção dos probes para que não haja inversão da ordem de geração de eventos.

3.3 RESUMO

Foram listadas as principais características de um conjunto de monitores. Estes foram escolhidos tanto entre monitores para uso acadêmico quanto para comercial, baseados em software, hardware ou híbridos e ainda com variados processadores e RTOS. A **Tabela 3-1** consolida as principais informações vistas neste capítulo.

Monitor	VDS 1	TLM 2	CodeTest		WindView 5	S.HA.R.K 6
			HWIC 3	SWIC 4		
Tipo de monitoração	*hardware*	*hardware*/ híbrida	híbrida	*software*	*Software*	*software*
SUT	Motorola 68000	80186EC	processadores de 16 ou 32 bits		processadores com suporte para o RTOS VxWorks	processadores com suporte para o RTOS S.Ha.R.K.
Observações	Latência no início da monitora-ção	Restrições dinâmicas de desempenho	*sem suporte à validação de restrições temporais em ERTS*			
Uso	acadêmico	acadêmico	Comercial		comercial	acadêmico

Tabela 3-1 – Resumo das características dos monitores existentes

As informações mostram a falta de um monitor que possua as seguintes características:

- Portabilidade entre diferentes processadores: parcialmente coberta pelos monitores de 3 a 6.
- Intrusão mínima e acurácia: ofertada por 1, 2 e 3
- Imunidade ao *probe-effect*: ausente de todos os monitores considerados
- Alta velocidade de aquisição de eventos: excluindo-se os monitores por software (4, 5 e 6), suportado por 1, com latência alta para início de monitoração, por 2 com restrições, conforme tabela e por 3, com o problema de falta de suporte para análise de dados de monitoração com vistas à validação de restrições temporais.
- Armazenamento de eventos com número variável de *words*: não suportado.

4 MIMO – MINIMAL INVASIVE MONITOR

Baseando-se na análise feita ao final do capítulo 3 e devido à necessidade de prova de conceito para o método Dyretiva [Cadamuro 07], decidiu-se pela busca de uma alternativa aos monitores existentes à época, ano de 2006. A especificação de requisitos do monitor foi definida:

- Intrusão mínima;
- Armazenamento de eventos com número variável de *words*.
- Resolução de 20 ns (tempo mínimo entre dois eventos consecutivos ou entre duas *words* pertencentes ao mesmo evento);
- Imunidade ao *probe-effect* ;
- Latência nula;
- Abordagem de monitoração híbrida;
- Características elétricas do sinal de entrada conforme Figura 4-5 e Figura 4-6.

Propôs-se, então, uma arquitetura para o monitor e uma metodologia de desenvolvimento. Em linhas gerais uma FPGA com processador *softcore* e memória DDR externa são os principais constituintes do monitor. As próximas seções detalham os aspectos relevantes do monitor proposto: A seção 7.1 fornece mais detalhes sobre a colocação do MIMO dentro do método Dyretiva.

4.1 ARQUITETURA DO MONITOR

Neste capítulo é detalhada a proposta de arquitetura do MIMO. Uma visão geral do monitor e a descrição suas interfaces é seguida pela decomposição do monitor em blocos e suas subseqüentes descrições. Nestas são realçados os principais componentes e as máquinas de estado empregadas no monitor.

As tarefas reservadas ao *software* de controle do monitor, protocolo de comunicação serial do MIMO e uma explanação a respeito dos comandos do monitor fecham esta seção.

4.1.1 VISÃO GERAL

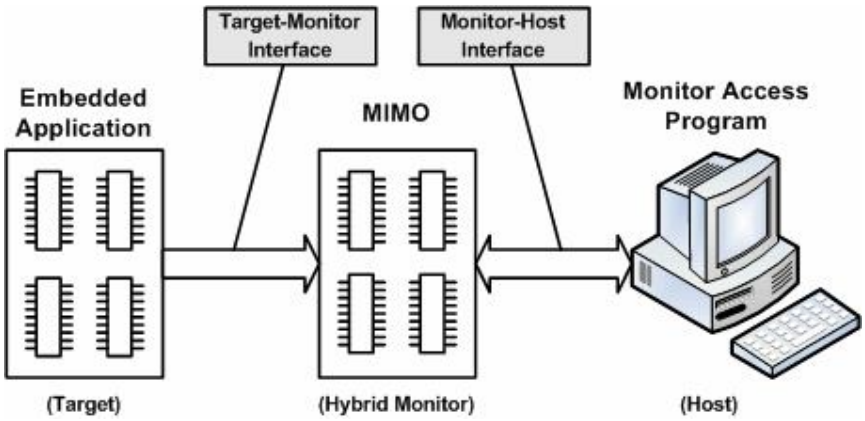

Figura 4-1 – Visão geral do MIMO
Fonte: [Copetti 07]

A Figura 4-1figura 1 mostra a aplicação do MIMO e as interfaces tanto com o sistema a ser monitorado (SUT), quanto com o computador que controla o monitor (Host).

4.1.2 INTERFACE COM O *HOST*

A Figura 4-2 mostra o funcionamento do monitor na forma de *Statechart*, método de representação formal criado por Harel [Harel 87]. Desta figura, pode-se notar que há duas atividades independentes no monitor. A parte superior representa a interface com o host e responde aos vários comandos enviados pelo usuário do monitor. Os comandos estão sumarizados na Tabela 4-1. É representada pelo nome de “Controle do Monitor”.

A segunda atividade, chamada de “Amostrador”, ocorre ininterruptamente, na metade inferior da Figura 4-2, desde o instante que o monitor é ligado. Ela coleta eventos e os armazena mesmo no caso em que não há uma monitoração em curso, que teria sido iniciada pela seção de Controle do Monitor.

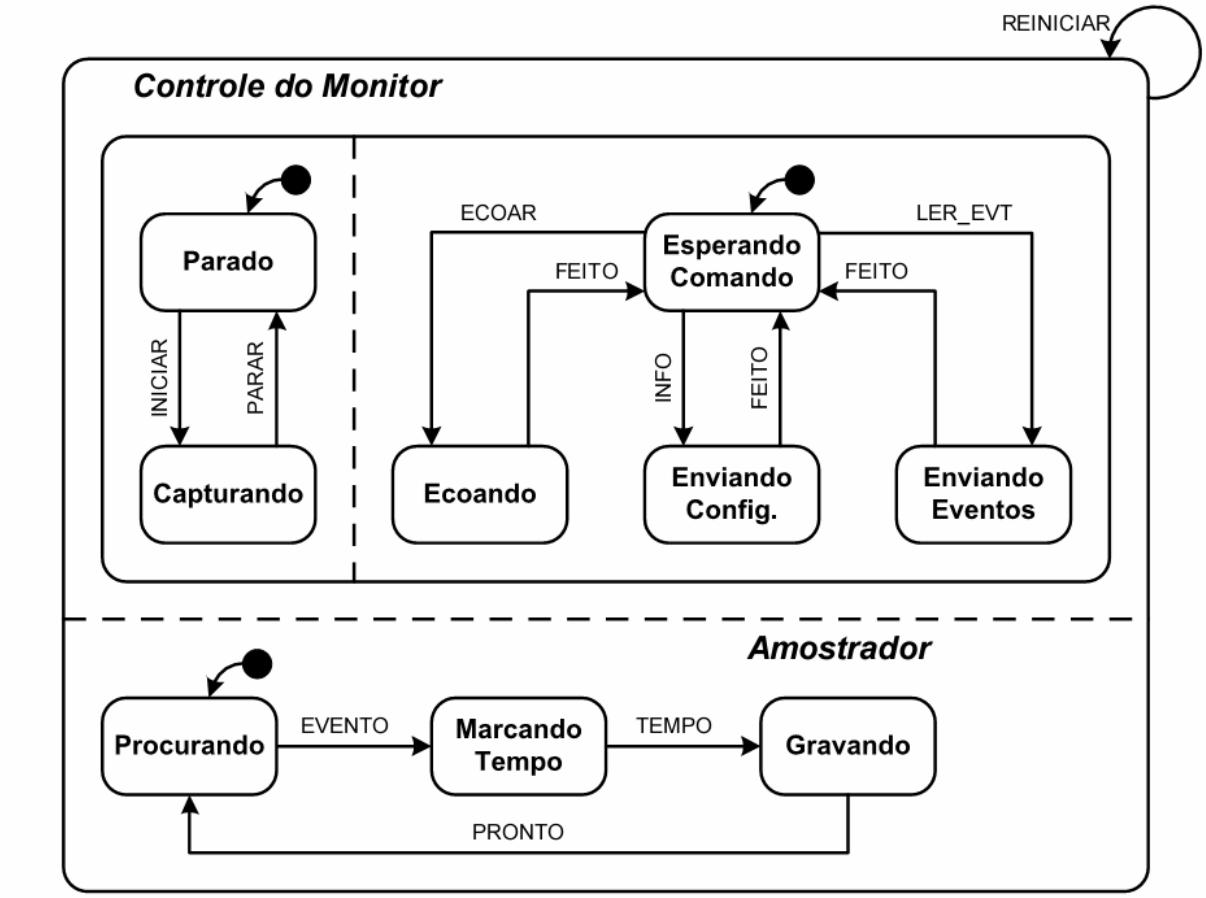

Figura 4-2 – *State Chart* do monitor MIMO

Evento	Descrição
REINICIAR	Comando para reiniciar o monitor.
EVENTO	Indicação de que um novo evento foi detectado na porta de medição.
TEMPO	Indicação de que o relógio de temporização foi lido e seu valor está disponível.
PRONTO	Indicação de que um evento e seu instante de tempo foram adequadamente tratados pela máquina de estados da amostragem, e que esta está livre para começar pelo próximo evento.
INICIAR	Comando para iniciar a amostragem.
PARAR	Comando de parar amostragem. Este evento pode ser gerado a partir de uma solicitação do usuário ou pelo próprio monitor quando a memória de eventos ficar lotada.
ECOAR	Comando para ecoar a mensagem recebida.
INFO	Comando de leitura das informações de configuração.
LER_EVT	Comando de leitura da memória de eventos.
FEITO	Indicação de que um comando de ECOAR, INFO ou LER_EVT foi respondido para o programa de controle do monitor.

Tabela 4-1 – Eventos de Controle do MIMO

4.1.3 INTERFACE COM O *SUT*

A interface de comunicação SUT-Monitor é composta de um barramento paralelo de 16 bits de dados e mais um sinal de *strobe*. Esta interface está em consonância com os requisitos do método Dyretiva. Cada ativação do sinal de *strobe* corresponde a uma *word* – palavra binária de 16 bits que deve ser armazenada. Um evento válido pode ser composto de uma única palavra de 16 bits ou várias, em seqüência. Não há um evento específico para sincronização do sistema, devido ao fato de todas as 65536 possibilidades de eventos serem válidas. Na seção 4.1.5 será discutida a questão da sincronização do início da captura de eventos.

O monitor coleta os eventos do SUT e os armazena, juntamente com sua correspondente marcação de tempo. O termo *timestamping* será usado para tais marcações de tempo. Estes dados são colocados em memórias rápidas, porém de baixa capacidade de armazenamento. Ciclicamente tais informações são transferidas a uma memória de massa, de grande capacidade. Tal transferência é efetuada por meio de DMA. Quando um ciclo de coleta de dados é completado, estes são transferidos ao host através de um canal de comunicação serial.

4.1.4 DIAGRAMA DE BLOCOS

Para um melhor entendimento do MIMO, a Figura 4-3 mostra um diagrama com as partes relevantes do monitor. Na parte superior esquerda, temos o elemento de *hardware* denominado *Sampler Dedicated Core*. Este elemento detecta os eventos gerados pelo SUT, gera a marcação de tempo correspondente à chegado do evento e os armazena (eventos e correspondentes marcações de tempo) numa memória do tipo *Dual Ported* RAM. Tal memória permite que sejam gravados ou lidos dados em quaisquer de suas duas portas. Doravante será denominada de *Input RAM*.

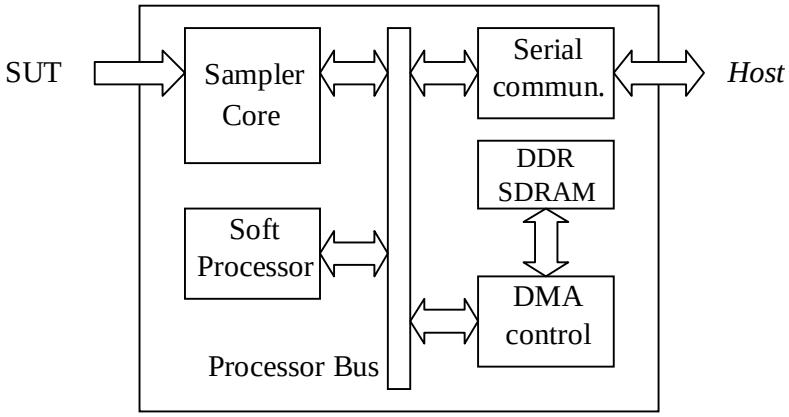

Figura 4-3 – Diagrama de blocos do MIMO.

O conteúdo da *Input* RAM é periodicamente transferido, por meio de DMA através do *Processor Bus*, à memória DDR SDRAM. Quando esta última estiver cheia, o *Sampler Core* pára a transferência de dados para a DDR e também a coleta de eventos do SUT. Os dados são, então, transferidos pelo canal serial, para o *host*. O controle destas transferências, bem como do canal serial é feito por meio de tarefas do *Soft Processor*, descritas no capítulo 4.1.4.3. A *input* RAM possui largura de 16 bits no lado de aquisição de dados e de 32 bits no lado conectado ao controlador de DMA.

O monitor foi construído usando-se o *Xilinx Embedded Development Kit* (EDK) , que provê bibliotecas para parte dos componentes do MIMO, como o *Soft Processor* Microblaze, controlador de comunicação serial, controlador de DMA e controlador de DDR SDRAM. A exceção notável é o *Sampler Core*, que foi criado especialmente para atender as necessidades específicas do MIMO. O *Sampler Core* foi desenvolvido em VHDL, uma linguagem de descrição de *hardware*, descrita na seção 2.7.

4.1.4.1 Sampler Core

Um diagrama mais detalhado do *Sampler Core* é visto na figura 3. O *Sampler Core* é elemento escravo na arquitetura do *Soft Processor*. Ele pode ser controlado por qualquer um dos outros Mestres na arquitetura utilizada, tais como o próprio *Soft Processor* ou o controlador de DMA. A input RAM é o local de armazenamento temporário dos eventos coletados do SUT e seus respectivos *timestamps*.

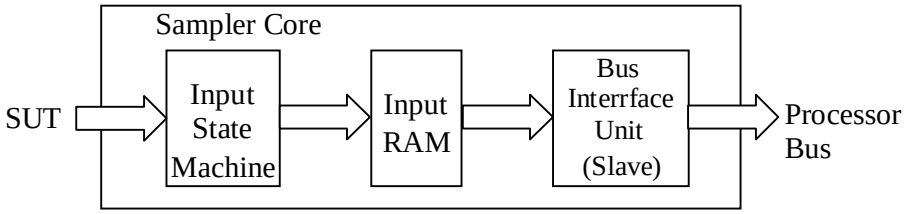

Figura 4-4 – Diagrama do *Sampler Core*

O SUT gera eventos de ordem completamente assíncrona com relação aos sinais de *clock* do sistema de aquisição – MIMO. O sinal de strobe é, porém, sincronizado com os dados que compõem o evento. Para que o evento seja corretamente armazenado, é necessária a transferência do *strobe* para o domínio de *clock* do MIMO. Uma vez que os sinais de dados com largura de 16 bits provêem de um *latch*, é necessária a garantia de que o sinal de *strobe* foi corretamente captado e sincronizado com o *clock* do sistema. A seção 2.8, que versa sobre *metastability*, forneceu informações para transferência do sinal *strobe* para o domínio de clock do monitor.

A Figura 4-5 e a Figura 4-6 mostram os sinais provenientes do SUT e sua relação temporal com o *strobe*. A partir do sinal de *strobe* é gerado um sinal correspondente, porém sincronizado com o *clock* do sistema, de 75 MHz. Deste sinal sincronizado, obtém-se um sinal que corresponde à borda de descida do sinal sincronizado. Este sinal pode estar desde 1 até 3 ciclos de *clock* atrasado em relação à borda de descida, ou seja 13,3 ns a 40 ns. Em ambos os casos, o dado a ser capturado está estável e pode ser registrado.

De acordo com as premissas do método Dyretiva, o contador de *timestamp* deve ser incrementado a cada 20 ns e o seu valor deve ser sempre relacionado ao último evento recebido. Como a referência para captura é de 75 MHz e não de 50 MHz, como requerido, o valor do contador de *timestamp* é multiplicado por dois e dividido por *3*. A primeira operação é trivial, mas a segunda envolveu um divisor por 3, construído em lógica combinacional. O valor de freqüência de 75 MHz é a maior disponível no monitor e foi escolhida para obtenção da maior performance possível na aquisição dos eventos.

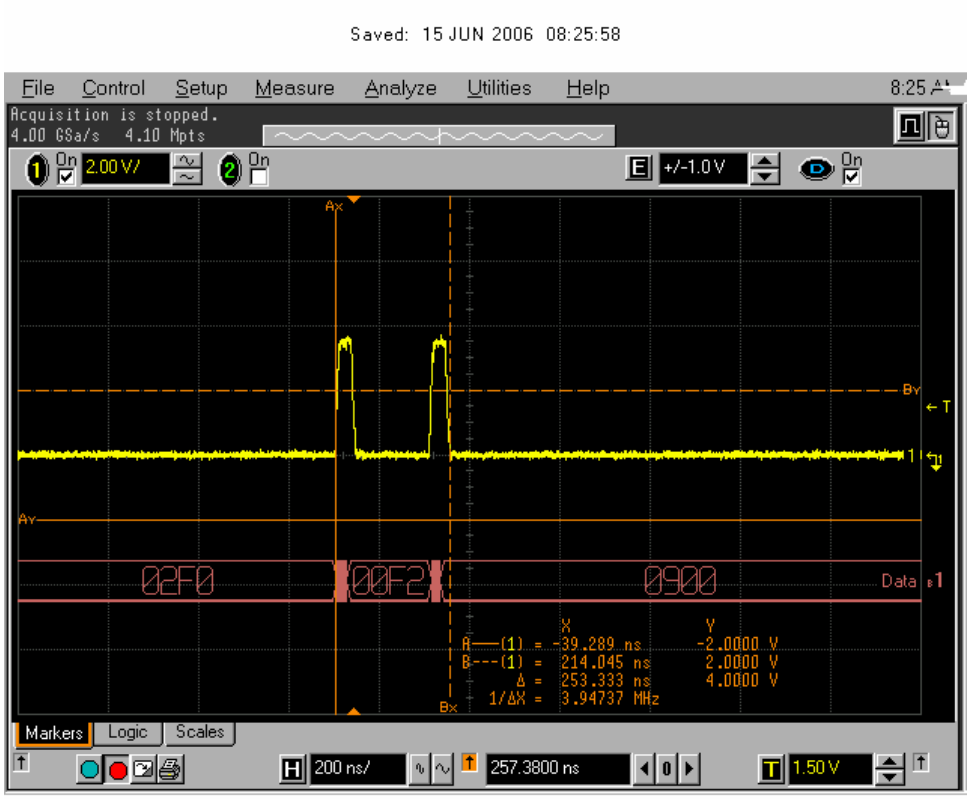

Figura 4-5 – Sinais provenientes do SUT – 1

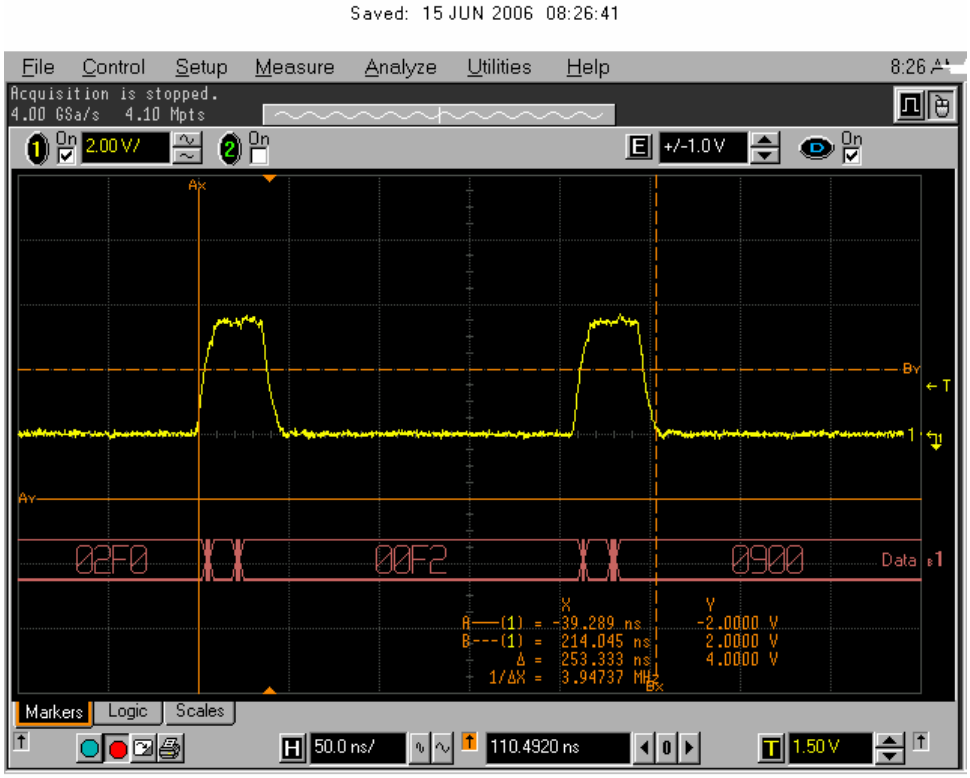

Figura 4-6 – Sinais provenientes do SUT – 2

A máquina de estados de entrada é a parte do hardware responsável pela detecção, decodificação, marcação de tempo (timestamping) e armazenamento dos eventos. Uma versão simplificada desta máquina de estados é apresentada na figura 4. Logo após sua ativação, a máquina vai do estado Idle para Run, onde passa a esperar por eventos provenientes do SUT. Quando detecta um evento, A transição para o próximo estado depende da decodificação feita na primeira palavra de 16 bits lida em conjunto com o sinal de strobe. Neste estado também é armazenado o valor do timestamp.

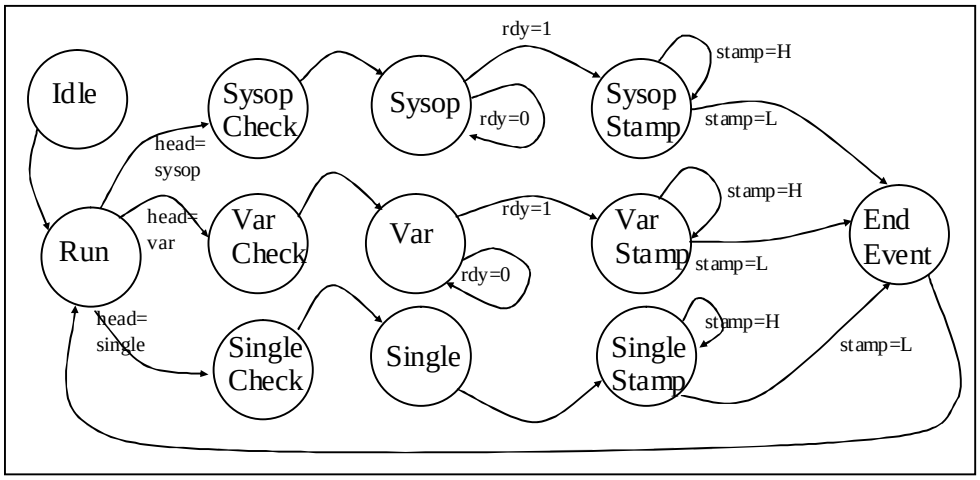

Figura 4-7 – Máquina de estados de entrada – simplificada

Se for obtido como resultado da decodificação um evento “single”, passa-se ao estado “Single Check”, onde se verifica a existência ou não de espaço na *Input* RAM. No caso normal, onde o espaço existente é suficiente, passa-se ao estado “Single” e por fim “Single Stamp”. No estado “Single” é feito o armazenamento do valor característico do evento e no estado “Single stamp”, do *timestamp* referente a este evento. Uma vez que se passa pelo estado “End Event”, a máquina de estados retorna ao estado “Run”, onde passa a aguardar novos eventos.

No caso do evento decodificado ser um “data event”, o *timestamp* é lido e a máquina move-se para o estado “Var Check”, onde o espaço necessário para o armazenamento de eventos de dados com 16 ou 32 bits e seu *timestamp* é avaliado. Na fase de decodificação anteriormente citada, também se tem a definição do tamanho do evento – 16 ou 32 bits. Caso o espaço disponível seja adequado, a máquina de estados move-se para o estado “Var”, onde se armazena o valor do evento (palavra de 16 bits recebida com o *strobe*). No estado subseqüente, “var stamp”, são armazenadas duas palavras de 16 bits para o caso de *data events* de 32 bits e uma palavra no caso de *data events* de 16 bits. Estas duas operações de

armazenamento só ocorrem quando da chegada das palavras adicionais, sincronizadas pelo sinal de strobe. Depois deste estado, a máquina transita para o estado “var stamp”, onde o *timestamp* é armazenado. Por fim, o estado *End Event* é atingido e daí volta-se ao estado “Run”

A última possibilidade para a máquina de estado de entrada é a chegada de um evento de sistema operacional. De maneira análoga aos dois tipos de eventos anteriores, quando da chegada de um evento deste tipo, o *timestamp* é preservado para ser armazenado em estado subseqüente e o número de palavras de 16 bits é obtido. Passa-se ao estado “Sysop Check” para verificação de disponibilidade de memória. No caso de ter-se espaço adequado, passa-se aos estados “Sysop”, onde tanto o evento propriamente dito como também as palavras adicionais são armazenadas. Ressalva deve ser feita no sentido de que o armazenamento de tais palavras adicionais é sincronizado por ativações do sinal *strobe* para cada uma delas. Segue-se para o estado “Sysop Stamp”, onde é armazenado o timestamp e finalmente atinge-se o estado End Event para posterior transição ao estado Run.

Por construção, não deve ocorrer, em funcionamento normal, a condição de não disponibilidade de memória na *input* RAM. Tal fato poderia vir a acontecer em caso da taxa de chegada de eventos exceder o máximo admissível, ver 4.2 sobre cálculo de desempenho;

Novamente, de acordo com o método Dyretiva, o contador do *timestamp* deve ser incrementado em passos de 20 ns e o seu valor deve .relacionado ao último evento recebido. Em virtude desta diretriz, sempre que se atinge o estado “Run”, tal contador é zerado, após ter sido lido.

4.1.4.2 Transferências via DMA

Cada vez que a input RAM é preenchida com um número de *bytes* correspondente à sua metade, este conteúdo é transferido, por meio de DMA para a SDRAM DDR. Tal processo é comandado pelo controlador de DMA, e em última instância pelo *softprocessor* – ver tarefas do Software. A cada nova transferência, o ponteiro de destino do DMA, para a DDR é incrementado em um valor equivalente à metade da *input* RAM. O ponteiro de origem do DMA, ou seja do lado da *input* RAM, tem um valor fixo. A cada leitura, o endereço interno da *input* RAM é incrementado até que faça o *wrap around* para zero.

4.1.4.3 Tarefas do *Software*

O software desempenha funções não críticas do ponto de vista temporal, de controle e coordenação do hardware do MIMO. Este software foi desenvolvido seguindo-se o padrão de linguagem de programação ANSI C e um microkernel provido pela empresa Xilinx para uso com o Soft Processor Microblaze. Este kernel disponibiliza o suporte básico para programação concorrente, comunicação entre threads e mecanismos de sincronização. São três estas tarefas, começando pela de prioridade mais elevada: Controle do HW, protocolo do MIMO, e por fim, comunicação serial. Além disso, tem-se uma rotina de tratamento de interrupção, ou ISR, no equivalente em inglês. A seguir uma descrição de cada uma delas.

Controle do HW. Esta tarefa do sistema operacional opera baseada em semáforos, que controlam se determinadas funções do HW devem ou não ser ativadas. Sua principal função é de desencadear o processo de DMA da *input* RAM para a DDR, quando o monitor estiver capturando dados. Neste caso, a tarefa permanece bloqueada até o momento em que o semáforo responsável por este bloqueio tenha seu valor alterado pela rotina de interrupção, discutida posteriormente neste texto.

MIMO *protocol*, interage com a tarefa de comunicação serial para receber os dados do *host*, em forma de *string* de caracteres. Valida ou não tais cadeias de caracteres, traduzindo-as em comandos válidos e conseqüentes ações para a tarefa de controle do HW. Em outro sentido faz também o papel de agrupar o resultado dos *traces*, formatá-los de acordo com o protocolo e enviá-los para a tarefa de comunicação serial.

A tarefa de comunicação serial recebe *bytes* do host e os disponibiliza em uma área onde a tarefa do MIMO *protocol* possa lê-los. Perfaz também a comunicação em sentido contrário, repassando respostas de comandos ao *host*, sejam elas simples reconhecimentos de comandos, mensagens de erro ou dados de monitoração. Interage diretamente com o componente UARTLITE, provido pela plataforma. O fluxo de dados, na porta serial, é transferido em uma taxa de 115 kbps, sem paridade, 1 bit de parada e sem controle de fluxo.

A rotina de tratamento de interrupção é responsável pelo tratamento da única interrupção no sistema relacionada ao funcionamento do monitor propriamente dito. A outra interrupção é responsável pelo escalonamento das tarefas e foge ao escopo desta dissertação. A interrupção do monitor pode ocorrer somente se o monitor estiver capturando dados e, conseqüentemente, a input RAM estiver sendo preenchida. Quando houver o preenchimento de metade da input RAM, é gerado o sinal de interrupção.

4.1.5 COMANDOS DO MIMO

Para que o usuário interaja com o monitor são necessários alguns comandos. Estes são enviados ao monitor através de um terminal serial RS-232 com 115 kbps, sem paridade, um bit de parada e sem controle de fluxo. Ao ser ligado, o monitor deve apresentar uma mensagem em seu LCD indicando o seu correto funcionamento.

Uma observação importante: o monitor deve ser ligado antes do SUT. Como não há eventos de sincronização a ser enviados do SUT para o monitor, este deve estar apto a receber os primeiros eventos do SUT. Desta forma é garantida a perfeita sincronização entre SUT e monitor.

As funções de controle e supervisão do monitor são: *Alive*, *Start*, *Stop*, *Info* e *Upload*. A função *Alive* tem como único objetivo verificar se o monitor está recebendo e transmitindo corretamente os *bytes* no formato definido pelo protocolo. Podem ocorrer uma resposta afirmativa de funcionamento, uma negativa e ainda, se nada retornar do MIMO, uma mensagem de transcurso de tempo (timeout). Apenas no primeiro caso, pode-se proceder ao uso do MIMO.

As funções start e stop controlam o início e o fim da monitoração respectivamente. Provêem mecanismos de alerta em caso do sistema já se encontrar no estado que se almeja. O comando Info permite ver as características da última monitoração efetuada, como número de *bytes* recebidos e número de eventos associados. O comando Upload marca o início da transferência de dados resultantes de capturas para o host. Caso o sistema esteja no estado de aquisição de dados, emite uma mensagem de que somente fora do estado de aquisição podem ser transferidos tais dados. A Tabela 4-2 sumariza os comandos do MIMO.

Comando	Significado
? ou help	Apresentam informações sobre os comandos disponíveis.
alive	Envia ao monitor um comando de ecoar, informando ao usuário se o monitor está pronto para operar ou não.
exit ou quit	Encerra o programa.
info	Envia ao monitor um comando de ler configuração, apresentando ao usuário os dados lidos ou o motivo de o comando não haver sido completado.
reset	Envia ao monitor um comando de reiniciar.
start	Envia ao monitor o comando para iniciar a amostragem, informando ao usuário se o comando foi confirmado pelo monitor ou não.
stop	Envia ao monitor o comando para finalizar a amostragem, informando ao usuário se o comando foi bem sucedido ou não.
upload	Lê a configuração do monitor e, caso ele esteja parado e com eventos armazenados na memória, realiza a leitura dos eventos e monta um arquivo com as informações de rastreamento coletadas.
version	Informa a versão do programa de controle do monitor.

Tabela 4-2 – Sumário das operações do MIMO

4.2 CÁLCULO DE DESEMPENHO

Embora a input RAM tenha apenas 2048 *bytes* de capacidade, deve continuar a receber dados do SUT mesmo durante as transferências via DMA, conforme descrito no capítulo 4.1.4.2

Para solução deste problema, a input RAM foi dividida em duas partes, cada uma com 1024 *bytes*. Enquanto os eventos provenientes do SUT e seus respectivos *timestamps* são armazenados em uma metade da *input* RAM, a outra metade é objeto de transferência para a DDR SDRAM. Para o *Sampler Core*, o pior caso em termos de uso de lagura de banda de memória é a chegada contínua de eventos simples. Para recebê-los, e armazená-los junto com seu *timestamp*, são necessários seis ciclos de *clock* e consumidos 6 *bytes* da *input RAM*, dois para o evento e quatro para os 32 bits do *timestamp*. O *clock* correntemente utilizado é de 75 MHz, levando a termos 80 ns para deteção e armazenamento completo de um evento simples. Se a chegada de eventos for contínua, metade da input RAM será preenchida em 13.65µs, e portanto poderia-se suportar um máximo de 12500000 eventos por segundo.

Para efetuarem-se as transferências entre a *input* RAM e a DDR SDRAM, é utilizada a característica de *burst mode* do controlador de DMA. Um analisador lógico, incorporado dentro da Fpga para inspeção de seus sinais, mostrou que para cada 64 *bytes* são necessários 986,7 ns, incluídos aí a transferência propriamente dita e os intervalos entre dois *bursts*

consecutivos. Com estes dados, chega-se ao tempo de transferência de metade da *input* RAM para a DDR SDRAM: 15,79 μs.

Como este valor é cerca de 2 μs maior do que o necessário para preencher a input RAM no pior caso (13,65 us), passa a ser ele a limitação para as transferências de dados. Com 30 MBytes disponíveis para armazenamento de dados na DDR SDRAM, e a geração de eventos simples sendo a única atividade do SUT, a DDR inteira seria preenchida em aproximadamente meio segundo. Este é o extremo máximo absoluto em termos de performance e não acontece em situações reais.

4.3 RESUMO

Neste capítulo foi apresentado o monitor para intrusão mínima – MIMO. O primeiro tópico discutido foi com respeito aos requisitos funcionais impostos pelo método Dyretiva. Suguem-se as descrições das interfaces do monitor tanto com o host quanto com o SUT. Uma visão da arquitetura do monitor é então apresentada com o intuito de preparar o leitor para uma apresentação mais detalhada das partes relevantes do MIMO.

A primeira, o *Sampler Core*, é um IP desenhado especificamente para o monitor. Possui especificidades que não permitiram o uso de componente de hardware pré-existente. A seção 2.7.2 trata do tema IP. A transferência dos dados gravados pelo *Sampler Core* na memória rápida do monitor é apresentada na seqüência.

Concluída a explanação sobre o hardware do monitor, passou-se às tarefas de controle do mesmo, executadas pelo software executado em um microprocessador *softcore* de 32 bits, Microblaze [Xilinx 05].

5 VALIDAÇÃO DO MONITOR E ESTUDO DE CASO

A validação do monitor se dá através da utilização do mesmo no registro de eventos de um ERTS comercial. As 4 fases que constituíram a validação do MIMO foram:

1) Validação do hardware do monitor: o SUT envia continuamente padrões repetitivos para que se verifique a captura adequada dos eventos pelo monitor. Testes do monitor foram feitos com uma aplicação especial e o monitor foi corrigido até que reportasse os eventos correspondentes à Figura 4-6. Nela pode-se ver o sinal de *strobe* e o conjunto de 16 bits que é transmitido do monitor para o SUT.

2) Validação do monitor em períodos longos de tempo quanto à concisão das informações registradas. Uma das condições para a validação do monitor foi a de ter toda a memória preenchida em uma taxa ligeiramente menor que a máxima permitida. Esta taxa foi discutida na seção 4.2, sobre performance do monitor. Novamente uma aplicação especialmente concebida gerou eventos em uma taxa muito maior do que a usual em aplicações normais. Não se conseguiu ultrapassar, com o SUT visto na seção 5.1 o limite de tempo entre duas aquisições consecutivas. Em outras palavras, o SUT não conseguiu gerar eventos em uma taxa superior àquela que o monitor conseguisse absorver.

3) O SoftScope, ferramenta de suporte ao método Dyretiva, possui um monitor baseado em software que foi usado para produzir *traces* antes do monitor MIMO estar operacional. Estes *traces* serviram de comparação para a verificação da correção dos dados associados a cada evento. Como as informações de *trace*, tanto do monitor por software do SoftScope quanto as geradas pelo MIMO, nas mesmas condições de execução foram equivalentes, foi atingida a condição descrita no item 2) para validação do monitor. É importante ressaltar que estas informações podem ser coerentes do ponto de vista de seu conteúdo, mas não da marcação de tempo ou *timestamping*. Além dos tempos relativos entre eventos serem maiores, pode haver inversão na ordem dos eventos.

4) Casos se teste reais foram construídos para exploração das diversas características do monitor. As próximas seções detalham as condições em que foram efetuados os testes funcionais do monitor MIMO, começando pelo ambiente de testes, seguindo pela caracterização da aplicação de teste e terminando com os casos de teste utilizados.

5.1 AMBIENTE DE TESTE

Conforme ilustra a Figura 5-1 o ambiente de testes é composto pelo SUT, o monitor MIMO, o *host* para o monitor executando o SoftScope e um terminal VT-100 para troca de mensagens com o sistema operacional PET# que esta sendo executado no SUT. A Figura 5-2 mostra os principais elementos constituintes do SUT. Este é conectado ao monitor por um *flat-cable* de 20 vias.

O SUT é composto por um processador de 32 bits Freescale MPC850 Power PC operando a um *clock* de 48 MHz. O SUT é proveniente da empresa eSysTech [eSysTech 03]. O processador possui 2 *KBytes* de memória *cache* de instruções, e 1 KBytes de memória *cache* de dados, controlador para diversos bancos de memória e controladores de comunicação. O módulo em questão, e850Lite, tem 32 MBytes de memória SDR-SDRAM, 8 MBytes de memória flash, duas portas seriais RS-232, uma porta USB, uma porta Ethernet e uma saída paralela de 16-bits.

O sistema operacional de tempo real utilizado no SUT é o PET#. Ele é uma variante do PET, desenvolvido por Douglas Renaux [Renaux 96]. O PET não possuía a característica de preempção. Daí a razão da necessidade da derivação do PET# a partir do original PET. A preempção é requisito para a análise de escalonamento por taxa monotônica [Briand 99]. A preempção é a característica de interrupção e posterior bloqueio de uma tarefa de menor prioridade para que outra tarefa, de maior prioridade, possa passar ao estado de execução.

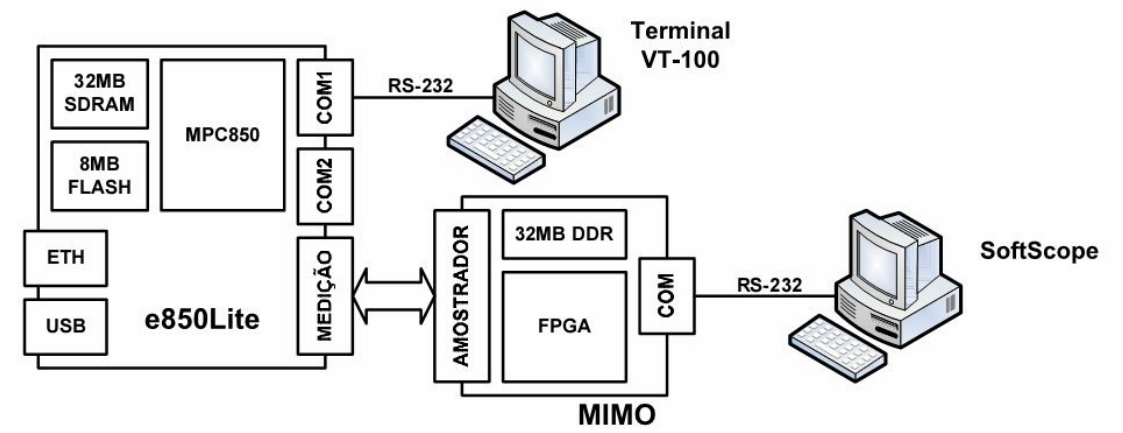

Figura 5-1 – Ambiente de teste com monitor MIMO

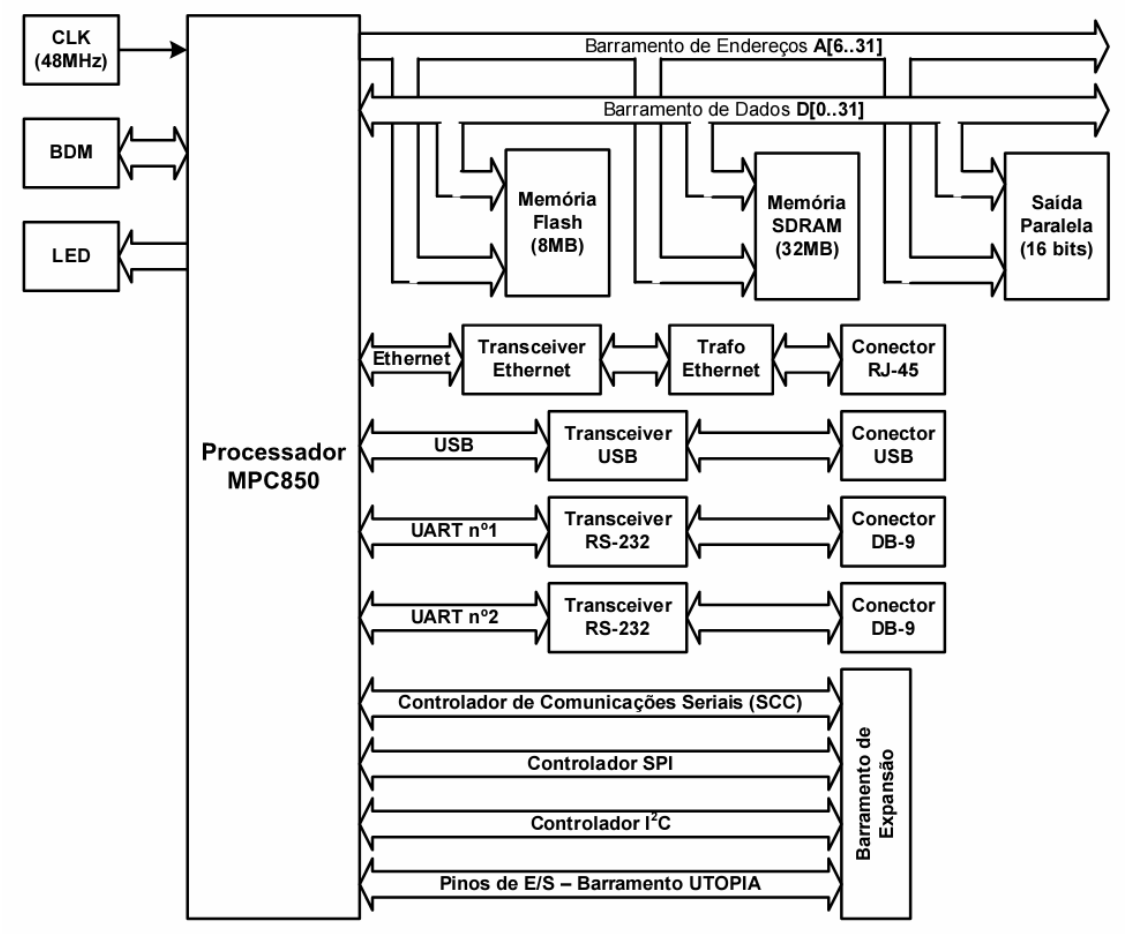

Figura 5-2 – Diagrama em blocos do SUT e850Lite
Fonte: [eSysTech 03]

5.2 APLICAÇÃO DE TESTE

Para uso como teste do monitor buscou-se uma aplicação ou conjunto de aplicações que pudesse impor carga variável ao processador do SUT. A aplicação usada para teste foi uma implementação do algoritmo de criptografia Blowfish [Schneier 1994]. Prara simplificação a aplicação de teste será chamada de TA (*Test Application*).

Através da interface de usuário com o SUT, obtêm–se a informação da composição da TA. É composta por 12 tarefas, sendo 6 relativas ao sistema operacional, 3 de suporte ao *hardware* e 3 efetivamente associadas à TA. À esquerda da Figura 5-3 aparecem os identificadores das tarefas, seguidos pelos seus respectivos nomes, estados em que se encontram, prioridades, mensagens assíncrona esperando tratamento, tamanho total da pilha (*stack*) de cada tarefa e por fim parcela livre de cada uma das pilhas. Prioridades numericamente maiores indicam tarefas com menor precedência. Sendo assim a tarefa

IdleTask é a de menor prioridade e é colocada para execução sempre que não houver nenhuma outra tarefa pronta para execução.

```
PET# for MPC8xx, version 1.0.0 (Jul  8 2007 - 20:19:12)


*******************************************************************************
*                   PET# - A Preemptive Real-Time Microkernel                 *
*                    By Joao Cadamuro Junior (CPGEI / UTFPR)                  *
*******************************************************************************
*                         Welcome to PET# Embedded Shell                      *
*               Type "help" to view the list of available commands            *
*******************************************************************************
[PET#]>
[PET#]>
[PET#]> thread

-------------------------------------------------------------------------
| TID | NAME                |   STATE    | PRIO | MSGs |  STACK | %FREE |
-------------------------------------------------------------------------
|   0 | IdleTask            | READY      |  31  |   0  |    512 |   64% |
|   1 | TimeManager         | RX_BLOCKED |   0  |   0  |   2048 |   86% |
|   2 | DeviceManager       | RX_BLOCKED |   1  |   0  |   2048 |   89% |
|   3 | ttyManager          | RX_BLOCKED |   7  |   0  |   2048 |   82% |
|   4 | EmbeddedShell#0     | READY      |  20  |   0  |   4096 |   73% |
|   5 | EmbeddedShell#1     | TX_BLOCKED |  20  |   0  |   4096 |   80% |
|   6 | Heartbeat(GREEN)    | RX_BLOCKED |  16  |   0  |   2048 |   86% |
|   7 | Heartbeat(RED)      | RX_BLOCKED |  25  |   0  |   2048 |   86% |
|   8 | FlashDriver#0       | RX_BLOCKED |   5  |   0  |   2048 |   88% |
|   9 | CryptoManager       | RX_BLOCKED |  22  |   0  | 133120 |    1% |
|  10 | BlowfishEncrypter   | RX_BLOCKED |  21  |   0  |   2048 |   88% |
|  11 | BlowfishDecrypter   | RX_BLOCKED |  21  |   0  |   2048 |   88% |
-------------------------------------------------------------------------

[PET#]>
```

Figura 5-3 – Componentes da aplicação de teste

A escolha de prioridades permite que a visualização dos LEDs (*Light-Emitting Diode*) ligados aos processos 6 e 7, Heartbeat (GREEN) e Heartbeat (RED) indique em termos grosseiros a carga do ERTS. Como o processo 6 possui prioridade maior que as tarefas da aplicação (9, 10 e 11), esta não é fortemente afetada pela maior cargas das mesmas, que possuem prioridades 22,21 e 21 respectivamente. O LED verde serve então como indicação de funcionamento normal do ERTS. Já o processo que aciona o LED vermelho tem prioridade mais baixa que os processos da aplicação e sendo, então, pouco acionada. A freqüência de acionamento do LED vermelho indica então o estado de carga das tarefas da aplicação. Quanto mais lento o acionamento, maior é a sobrecarga associada às tarefas 9, 10 e 11.

Para análise da monitoração no SUT em questão, as tarefas relevantes são: Cryptomanager, BlowFishEncrypter e BlowFishDecrypter. A tarefa Cryptomanager cria conjuntos de dados aleatórios a cada meio segundo. Tais dados são criptografados por

BlowFishEncrypter e decriptografados por BlowFishDecrypter. Comparando-se os dados originais que foram submetidos à encriptação / decriptação com os dados resultantes deste processo, verifica-se o funcionamento ou não do ERTS.

5.3 CASOS DE TESTE

Os seguintes casos de teste foram escolhidos para verificação do funcionamento do monitor MIMO:

- Desempenho do sistema operacional
- Taxa de utilização do processador
- Tempo de execução de funções
- Verificação de restrições temporais
- Intrusão da monitoração híbrida

5.3.1 DESEMPENHO DO SISTEMA OPERACIONAL

Em um sistema multitarefa o número de operações típicas de sistema operacional é significativa para que o sistema computacional forneça a percepção de paralelismo aos solicitantes externos e internos de seus serviços. Tais operações envolvem, por exemplo, trocas de contexto e intercâmbio de mensagens entre as tarefas. Dada a sua freqüência, é um bom exercício para verificação da capacidade do MIMO capturar os eventos gerados quando da execução das tarefas de sistema operacional. Por um período de aproximadamente 2 minutos, foram gerados e coletados dados em uma taxa variando de 0 a 512 kbps. A Tabela 5-1 mostra a compilação de resultados obtidos a partir de vários arquivos de monitoração.

Serviço	Execuções	OBCET	OTCET	OWCET
PutMessage	6.680	13,4µs	17,8µs	19,7µs
SendMessage	6.834	17,4µs	22,6µs	25,6µs
ReceiveMessage (vazia)	13.878	9,9µs	12,8µs	21,0µs
ReceiveMessage (Put)	13.077	11,8µs	13,3µs	15,6µs
ReceiveMessage (sem nova)	5.998	16,7µs	17,0µs	17,7µs
ReceiveMessage (SendMessage)	6.834	13,3µs	15,2µs	22,3µs
ReplyMessage	6.835	14,2µs	17,0µs	19,2µs

Tabela 5-1 – Eventos de Controle do MIMO

Para cada tipo de serviço analisado tem-se o número de execuções de cada um dos serviços do sistema operacional, o melhor caso de tempo de execução OBCET, tempo típico OTCET e pior tempo OWCET. Em ReceiveMessage(SendMessage) são observadas algumas

discrepâncias que podem ser explicadas por possíveis preempções na referida tarefa. Com o auxílio do SoftScope pôde-se determinar a origem de tal preempção: um interrupção de relógio do sistema. No PET# permite-se a preempção das funções do núcleo do sistema operacional para diminuir a latência no atendimento de interrupções.

5.3.2 TAXA DE UTILIZAÇÃO DO PROCESSADOR

Outra importante medida em ERTS é a da divisão do tempo entre as aplicações que realizam trabalho útil com relação à finalidade do sistema analisado e as tarefas do sistema operacional, que poderiam ser consideradas como sobrecarga das tarefas úteis. Complementando, a quantificação do tempo ocioso dá, ao projetista ou usuário do ERTS, uma estimativa da carga total do processador e de quanta capacidade de processamento ainda resta.

Para este experimento a taxa de geração de dados foi de 256 kbps durante um período de aproximadamente 20 segundos. Uma informação importante obtida da Tabela 5-2 é a participação percentual das atividades do sistema operacional em relação ao tempo total, de cerca de 1%. Detalhes da divisão de tempo entre as aplicações podem ser vistos na Tabela 5-3.

Descrição	Tempo	Tempo (%)
Tempo ocioso	14,4s	71,76%
Tempo de sistema (execução dos serviços de troca de mensagem no núcleo)	238,3ms	1,18%
Tempo utilizado pelas tarefas	5,4s	27,06%
Tempo total	20,1s	100%

Tabela 5-2 – Divisão do tempo entre sistema operacional e aplicações

Tarefa	Prioridade	Preempções	Tempo de CPU	Tempo de CPU (%)
TimeManager	0	0	71,2ms	0,35%
ttyManager	7	0	64,9ms	0,32%
Hearbeat(GREEN)	16	0	1,3ms	0,01%
Heartbeat(RED)	25	0	1,2ms	0,01%
CryptoManager	22	40	235ms	1,17%
BlowfishEncrypter	21	120	2,5s	12,58%
BlowfishDecrypter	21	120	2,5s	12,62%

Tabela 5-3 – Detalhamento do tempo de cada tarefa da aplicação

Da análise da Tabela 5-3 pode-se também verificar o alto consumo de recursos pelas tarefas de criptografia e decriptação, seguidas pela tarefa CryptoManager. As duas tarefas de Heartbeat perfazem acionamento simples de dispositivos de hardware e não são significativas do ponto de vista temporal. As duas tarefas restantes, TimeManager e ttyManager perfazem, em nível de aplicação, tarefas normalmente deixadas a encargo dos núcleos de sistema operacional. Possuem baixa participação no tempo total.

5.3.3 TEMPO DE EXECUÇÃO DE FUNÇÕES

Como continuação natural das medidas de divisão de tempo entre sistema operacional, aplicações e tempo ocioso, analisou-se o arquivo de rastros até o nível das funções. Este detalhamento teve como objetivo a verificação da correção dos dados capturados pelo monitor MIMO. Como exemplo de teste de consistência, o número de execuções das funções BlowfishEncrypt e BlowfishDecrypt foi exatamente o mesmo, 81920, conforme mostra a terceira coluna da Figura 5-4. É um resultado esperado pois as duas operações são de natureza simétrica.

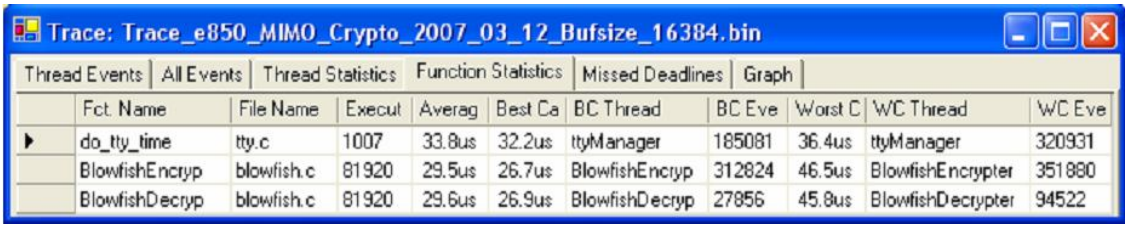

Figura 5-4 – Dados da execução de funções.

Diversas análises de coerência foram feitas para que assegurar-se que o monitor estava coletando evidências de execução corretamente. Como exemplo podemos citar a primeira e última linhas da Figura 5-5 onde são apresentados eventos de usuário, USER 1 1 e USER 1 2, responsáveis pela identificação do início e fim da função BlowfishEncrypt. Este tipo de eventos é comum devido à instrumentação do início e fim das funções. Esta é uma outra simetria que permite validar dados capturados pelo MIMO. Na mesma figura aparecem, como nas linhas 351881 e 351884 eventos relacionados ao tratamento de exceções. São eles PET_EXCEPTION_ENT e PET_EXCEPTION_EXIT. A ordem em que aparecem permite confirmar a correção da captura destes eventos.

Trace: Trace_e850_MIMO_Crypto_2007_03_12_Bufsize_16384.bin

Thread Events | All Events | Thread Statistics | Function Statistics | Missed Deadlines | Graph

89%	Event Nr	Time Sta	Event	Event Name	Nr	Param 1	Param 2	Par	Para
	351880	1280	USER	USER	1	1	0	0	0
	351881	13480	PET#	PET_EXCEPTION_ENT	1	Decrementer	0	0	0
	351882	9280	PET#	PET_PUT_MSG_ISR	1	TimeManager	0	0	0
	351883	10800	PET#	PET_SCHEDULE	1	TimeManager	0	0	0
	351884	1280	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351885	8120	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351886	6020	PET#	PET_RECEIVE_PUT_M	1	TimeManager	0	0	0
	351887	7260	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351888	6180	PET#	PET_TIMER_EXPIRED	1	ttyManager	0	0	0
	351889	5120	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351890	7500	PET#	PET_PUT_MSG	2	TimeManager	ttyManager	0	0
	351891	10360	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351892	11480	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351893	5300	PET#	PET_RECEIVE_MSG_E	1	TimeManager	0	0	0
	351894	5720	PET#	PET_SCHEDULE	1	ttyManager	0	0	0
	351895	2020	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351896	7820	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351897	7260	PET#	PET_RECEIVE_PUT_M	1	ttyManager	0	0	0
	351898	6160	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351899	4200	USER	USER	1	5	0	0	0
	351900	34580	USER	USER	1	6	0	0	0
	351901	9520	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351902	7900	PET#	PET_SEND_MSG	2	ttyManager	TimeManager	0	0
	351903	14200	PET#	PET_SCHEDULE	1	TimeManager	0	0	0
	351904	1060	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351905	9200	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351906	9600	PET#	PET_RECEIVE_SEND_	2	TimeManager	ttyManager	0	0
	351907	5620	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351908	6560	PET#	PET_SET_TIMER	2	ttyManager	20	0	0
	351909	10600	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351910	10240	PET#	PET_REPLY_MSG	2	TimeManager	ttyManager	0	0
	351911	7220	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351912	6600	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351913	5480	PET#	PET_RECEIVE_MSG_E	1	TimeManager	0	0	0
	351914	5220	PET#	PET_SCHEDULE	1	ttyManager	0	0	0
	351915	2220	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351916	8800	PET#	PET_EXCEPTION_ENT	1	SystemCall	0	0	0
	351917	8800	PET#	PET_RECEIVE_MSG_N	1	ttyManager	0	0	0
	351918	6520	PET#	PET_SCHEDULE	1	BlowfishEncr	0	0	0
	351919	1440	PET#	PET_EXCEPTION_EXIT	0	0	0	0	0
	351920	33060	USER	USER	1	2	0	0	0

Figura 5-5 – Eventos decodificados para teste de coerência

5.3.4 VERIFICAÇÃO DE RESTRIÇÕES TEMPORAIS

O último caso de teste para validação da integridade dos dados coletados pelo MIMO é baseado na configuração do arquivo de restrições temporais e da aplicação de tal forma que haja necessariamente violações e portanto que o SoftScope as detecte corretamente. Para que a detecção da violação seja feita, os eventos que a desencadeiam precisam ser corretamente armazenados. A Figura 5-6 ilustra o estabelecimento de um deadline de 259 ms, das tarefas envolvidas e de sua quantidade entre outros fatores. Já a Figura 5-7 ilustra exatamente a ocorrência da perda de *deadlines*.

```
[NUMBER_OF_TIME_REQUIREMENTS = 1]
[TIME_REQUIREMENT_NAME = Blowfish]
[START_EVENT = OS, 2, 82, 9, 10]
[END_EVENT = OS, 2, 87, 11, 9]
[RELATED_THREADS = 3]
[THREAD1 = CryptoManager]
[THREAD2 = BlowfishEncrypter]
[THREAD3 = BlowfishDecrypter]
[DEADLINE = 259]
[INTERVAL = 500]
```

Figura 5-6 – Especificação de *deadline*

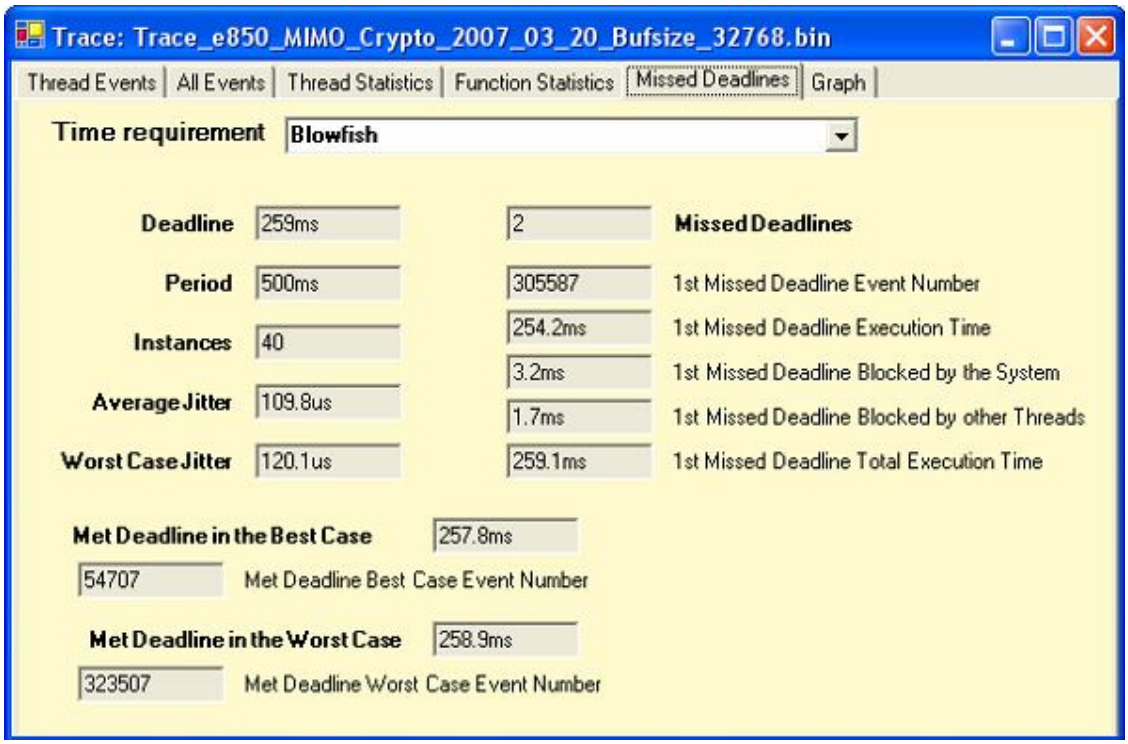

Figura 5-7 – Relatório de *deadlines* perdidos

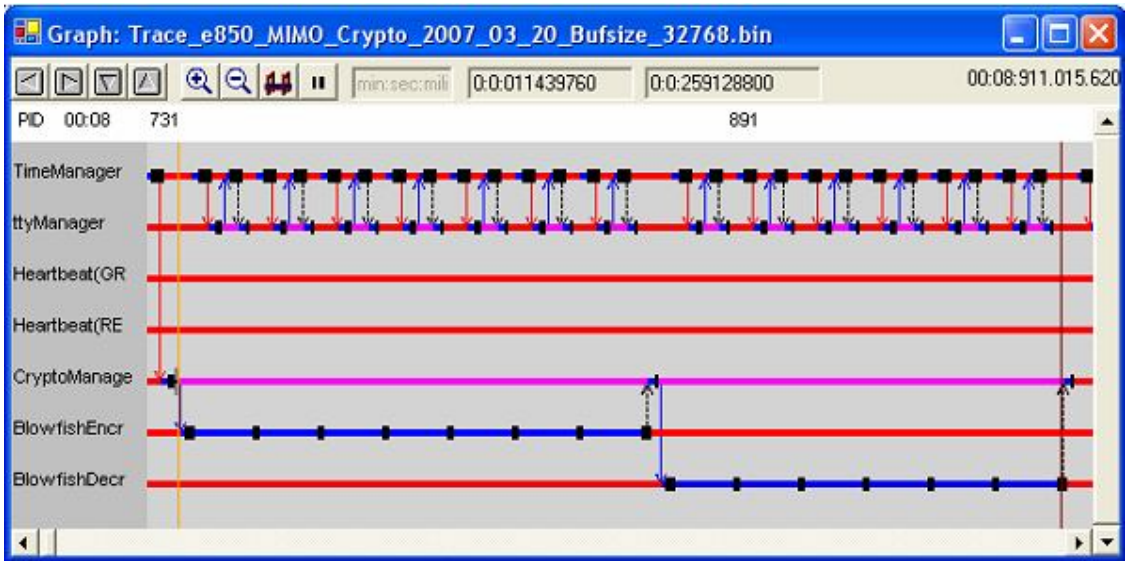

Figura 5-8 – Diagrama em blocos do SUT e850Lite

Para fechamento do teste de correta captura dos eventos necessários à detecção de uma violação temporal, a Figura 5-8 mostra a distribuição de tarefas ao longo do tempo e apresenta uma aparente normalidade. Pode-se ver as tarefas de menor prioridade nas linhas inferiores sendo preemptadas pelas de prioridade mais elevada, nas linhas superiores. A violação temporal só foi gerada e detectada pelo estabelecimento de um *deadline* irreal apenas para efeitos de teste.

5.3.5 INTRUSÃO DA MONITORAÇÃO HÍBRIDA

Este último caso de teste tem uma função adicional em relação aos outros quatro casos de teste. Além de verificar o funcionamento do monitor em uma determinada condição de operação, visa determinar quanto de intrusão a monitoração acarreta ao ERTS. A quantidade de intrusão é crucial devido ao fato de o método Dyretiva preconizar o uso da instrumentação permanente com o objetivo de evitar o *probe-effect*.

A primeira abordagem para a medição de intrusão era baseada no fato de que o monitor tem total conhecimento da quantidade de eventos e dos instantes onde cada evento acontece. Como a duração de cada instrução de instrumentação é conhecida, vislumbrou-se a possibilidade de contabilizar-se este tempo e dividi-lo pelo tempo total de monitoração, obtendo-se a parcela de intrusão.

Há porém um fator complicador: o tempo decorrido desde a busca de uma determinada instrução de instrumentação na memória, subseqüente decodificação até sua efetiva execução não é mensurável devido às características de *pipeline*, memórias *cache*

entre outras. Este tempo estaria, então, deixando de ser considerado. Passou-se então para a abordagem de executar-se uma quantidade finita de trabalho, com e sem instrumentação dos códigos de aplicação e do sistema operacional. A diferença de tempo de execução do trabalho total nos dois casos representa a intrusão.

A aplicação de teste anteriormente utilizada perfazia as seguintes atividades: geração de um bloco aleatório de dados; criptografia e decriptação dos mesmos; comparação dos dados originais com os resultantes da criptografia / decriptação. Este processo era repetido durante o tempo necessário para que a monitoração adquirisse as evidências de execução para cada caso. A modificação consiste na retirada das tarefas CryptoManager, BlowfishEncrypt e BlowfishDecrypt e inclusão de uma única aplicação que criptografe, decripte e faça as comparações necessárias um bloco de 2 MBytes de dados gerados aleatoriamente. Tal aplicação pode ser chamada por comando direto do usuário para o SUT.

Tal aplicação de teste é executada em baixa prioridade de modo a sofrer preempção por parte de outras tarefas de maior prioridade e pelo sistema operacional. A instrumentação é colocada no início e no final das funções que perfazem a criptografia e a decriptação. Estas funções operam sobre blocos de dados de 64 bits. São, então, necessárias 262144 chamadas para a função de criptografia e mais 262144 chamadas para a decriptação. Como o início e fim das funções são instrumentados, tem-se um total de 1048576 chamadas a funções de instrumentação.

Situação	Média do Tempo de Execução
Sem Instrumentação	52,504038867 segundos
Com Instrumentação	53,564690734 segundos
Diferença :	1,98%

Tabela 5-4 – Resultados da medição indireta de intrusão

5.4 RESUMO

O monitor foi extensivamente testado para validação das interfaces com o SUT e com o *host.* Casos de teste que refletissem aplicações reais foram usados para verificação do funcionamento do monitor em condições diversas e com carga de geração de eventos variável.

Tanto a aplicação de teste quanto o ambiente e o SUT foram descritos de forma a permitir o entendimento do conjunto SUT – monitor – *host*. Em cada um dos casos de teste procurou-se ressaltar quais características do monitor estavam a prova.

O último caso de teste, em especial, tem importância pois permitiu a medição da intrusão adicionada ao SUT pela monitoração. Uma baixa intrusão é uma das características básicas de motivação para a construção deste monitor.

Adicionalmente, todo o ambiente proposto pelo método Dyretiva foi utilizado, servindo de auxílio à validação deste método.

6 CONCLUSÃO

O monitor MIMO foi construído com o objetivo de servir de prova do conceito de monitoração híbrida do método Dyretiva. Foi testado em condições reais de uso tanto em situações de baixa carga do processador quanto em situação perto do limite de aquisição do monitor.

As características de alta performance de aquisição do monitor foram verificadas. Durante os testes, em nenhum momento a capacidade máxima calculada de 12,5 milhões de eventos foi atingida.

Fator determinante na monitoração permanente de sistemas embarcados de tempo real, a intrusão foi medida com resultados excelentes: O custo da monitoração foi de cerca cerca de 2%. O monitor mostrou-se útil na medição de tempos de execução de funções, tarefas quanto na de verificação de restrições impostas aos ERTS.

6.1 PUBLICAÇÕES

Os resultados experimentais obtidos com o MIMO para monitoração de um sistema embarcado de tempo-real foram apresentados em um artigo no WTR2007 – Workshop de Tempo Real [Copetti 07]. Também a estrutura do monitor e considerações sobre performance foram consideradas.

6.2 SUGESTÕES PARA TRABALHOS FUTUROS

- A abordagem de monitoração com a coleta dos dados de monitoração separada do *upload* dos mesmos para o *host* é em grande parte função do meio de comunicação utilizado. Propõe-se, com trabalho futuro, a substituição do canal serial por uma interface mais rápida, tal como Ethernet, SPI ou USB.
- Implementação de *triggers* para que o armazenamento dos dados em função da ocorrência de determinado evento. A arquitetura atual do MIMO coleta as informações geradas pelo *target* de maneira ininterrupta a partir do momento em que são ligados (*target* / monitor). Caso a gravação dos dados não tenha ainda sido ordenada, o conteúdo da *Dual-Port* RAM é constantemente sobrescrito.

7 DYRETIVA, SOFTSCOPE E PERF

São mostrados aqui alguns tópicos de especial interesse para a perfeita compreensão do ambiente em que o monitor MIMO está inserido. São eles, o método Dyretiva, o conjunto de ferramentas SoftScope [Cadamuro 06] e o projeto PERF. Por uma questão de organização o método de instrumentação usado em conjunto com o monitor MIMO é descrito em conjunto com o método Dyretiva.

7.1 O MÉTODO DIRETYVA

7.1.1 INTRODUÇÃO

Dyretiva (*Dynamic Real-Time Violation Analyzer*) é o objeto da tese de Doutorado de João Cadamuro Junior [Cadamuro 07]. É proposto com duas finalidades: monitoração para teste e depuração e monitoração para correção e avaliação de desempenho. A Figura 7-1 ilustra estes dois usos possíveis. O código de aplicação e/ou Sistema operacional, objetos das ações acima, precisam ser instrumentados para que possam gerar eventos para a monitoração. O método permite instrumentação automática da aplicação, com possibilidade de seleção mais ampla ou mais restrita dos pontos de instrumentação, permitindo assim manter a intrusão no nível mais baixo possível. Para o sistema operacional, no entanto, o método Dyretiva suporta a instrumentação manual do código, onde não apenas os eventos do sistema operacional devem ser sinalizados mas também detalhes a eles relacionados devem ser passados ao monitor.

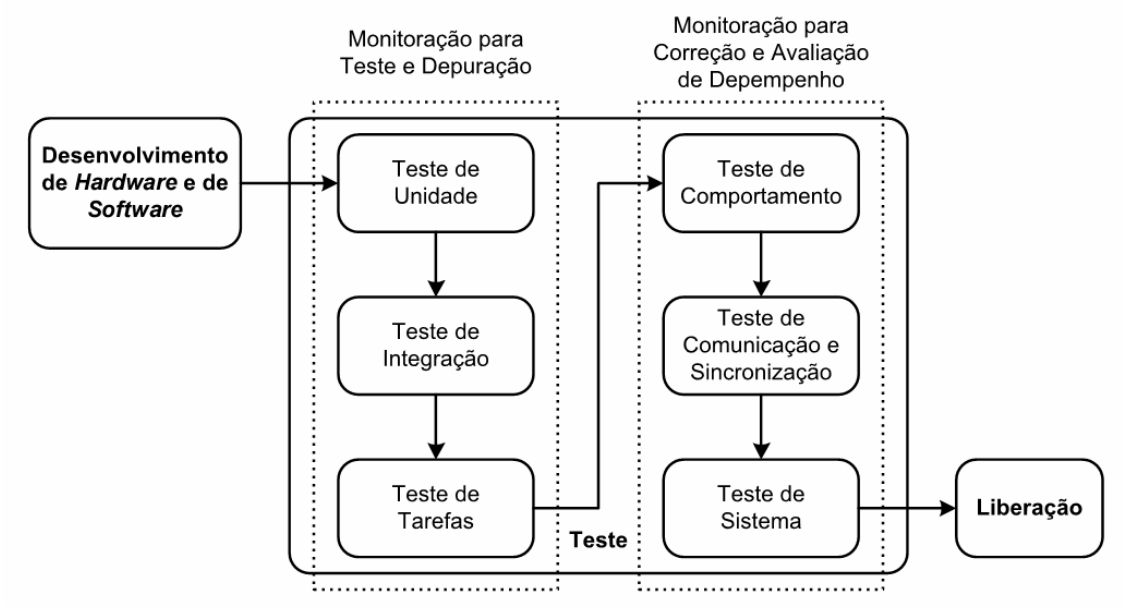

Figura 7-1 – Aspectos de uso do Diretyva

Fonte: [Cadamuro 07]

7.1.2 UTILIZAÇÃO DO MÉTODO

A Figura 7-2 mostra como seria um típico fluxo de desenvolvimento de um SETR que tivesse suporte do método. Em paralelo ao desenvolvimento do *software* e do *hardware*, há a especificação dos requisitos temporais do sistema. O código é instrumentado e após o processo de compilação e ligação é carregado no SUT. A execução normal das tarefas no SUT ocorre concomitantemente com a das instruções decorrentes da instrumentação. Esta última deixa assim um “rastro” ou “traço” da execução normal das aplicações e do sistema operacional. O monitor registra os eventos decorrentes daí decorrentes gerando um conjunto de informações sobre o histórico de execução do SUT.

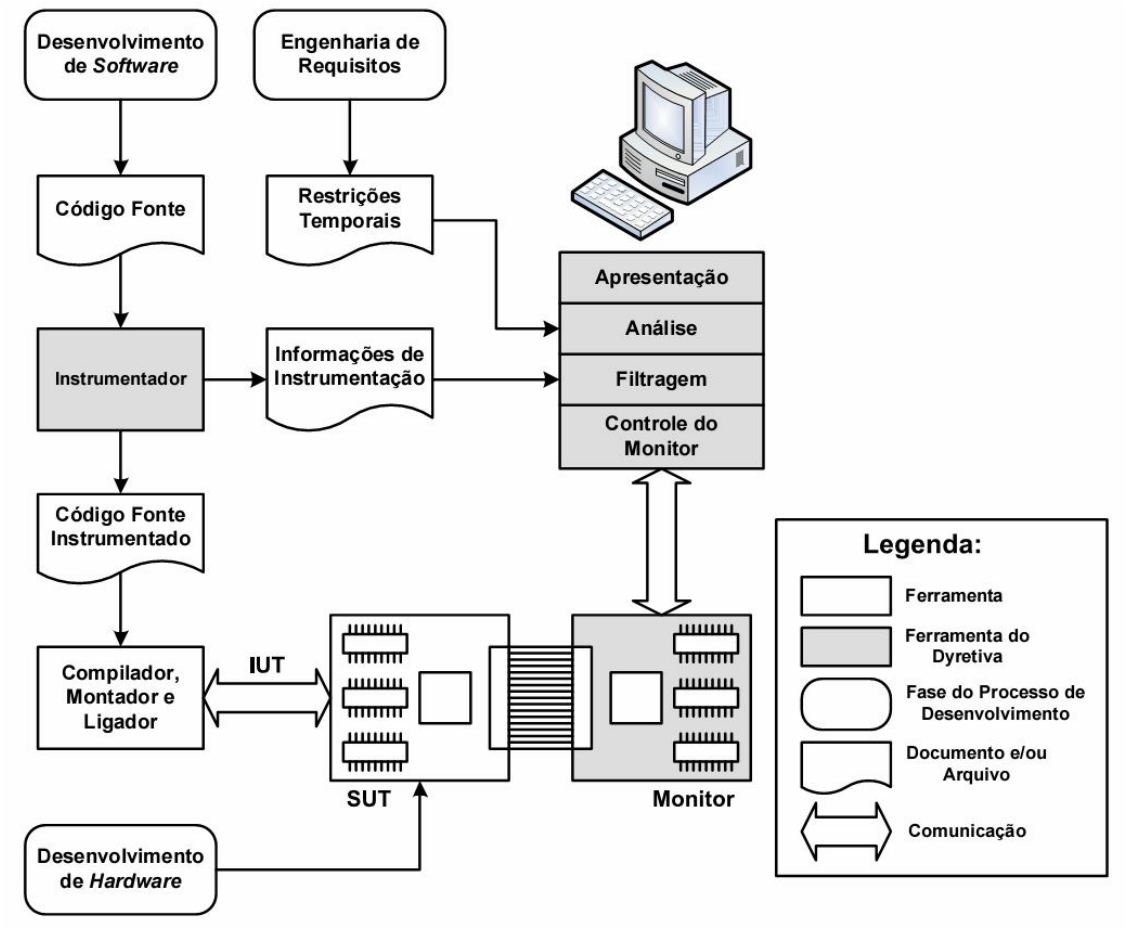

Figura 7-2 – Modelo de uso do Dyretiva

Fonte: [Cadamuro 07 b]

Ainda com relação à Figura acima, o registro dos eventos gerados pelo SUT é comandado pelo bloco “Controle do Monitor”. Este define o início e fim da captura dos eventos de monitoração. Os dados coletados são então filtrados para se conformarem ao padrão de entrada das ferramentas de análise e apresentação. Se o objetivo é de avaliação de desempenho, o sistema pode ser configurado para que as informações relativas ao tempo de uso da CPU pelo Sistema operacional bem como pelas tarefas sejam coletadas. Os dados de saída, após filtragem e análise podem gerar gráficos de setores para a ocupação percentual do processador pelas diversas tarefas. Gráficos de Gannt permitem representar o comportamento dinâmico das tarefas, como tempos de execução e seqüenciamento das mesmas.

7.1.3 MODELO DE FALTA

Três são as faltas que o método Dyretiva permite identificar: concorrência, processamento e perda de prazo. Uma premissa para a utilização do método Dyretiva é a de que seja possível a reconstrução das linhas de execução das tarefas e seus respectivos tempos. Esta premissa restringe o uso do Dyretiva aos sistemas operacionais como o PET# que permitem a determinação, em um instante qualquer de tempo, qual é o estado do sistema operacional. Para a recuperação das linhas de execução, um algoritmo é aplicado sobre o arquivo resultante da monitoração. Através da simulação da lógica do escalonador, este algoritmo reproduz as transições de estado do sistema operacional na situação monitorada.

7.1.3.1 Faltas de Concorrência

Acontecem quando uma determinada tarefa não cumpriu seu prazo pela concomitância (concorrência) de outras atividades. Esta interferência de outras tarefas pode ser devida à execução de tarefas de maior prioridade – preempção; constantes tratamentos de exceção ou interrupções de hardware – interrupção e finalmente devida à sincronização. Esta falta ocorre se a tarefa fica bloqueada por elementos de sincronização do sistema operacional tais como semáforos e filas.

7.1.3.2 Faltas de Processamento

Tais faltas ocorrem se uma tarefa utiliza o processador por mais tempo que o esperado. Pode ser resultado de uma tarefa excessivamente longa devido a mau planejamento e a laços de *busy wait*, onde o processador executa o *poll* de determinado recurso indisponível de maneira contínua. *Poll* ó o termo comumente usado para a leitura de determinada informação.

7.1.3.3 Faltas de Prazo

Tais faltas ocorrem se uma task utiliza o processador por mais tempo que o esperado. Pode ser resultado de uma task excessivamente longa devido a mau planejamento e a laços de *busy wait*, onde o processador executa o *poll* de determinado recurso temporariamente indisponível de maneira contínua.

7.2 SOFTSCOPE

7.2.1 INTRODUÇÃO

O SoftScope é um conjunto de ferramentas concebido para dar suporte ao método Dyretiva. A concepção de tais ferramentas é apresentada por Cadamuro [Cadamuro 06]. Uma explanação dos diversos componentes é mostrada a seguir devido ao fato de o monitor, objeto deste trabalho, ser parte constituinte do SoftScope. A Figura 7-3 apresenta o diagrama de fluxo que indica como suas diversas ferramentas são utilizadas.

7.2.2 DESCRIÇÃO DAS ATIVIDADES

Tomando-se como base a Figura 7-3, o pimeiro passo a ser seguido pelo usuário das ferramentas é a instrumentação do código. Esta instrumentação é detalhada na próxima seção. O código das aplicações do ERTS deve estar, neste momento, pronto para ser carregado no *target*. Para que o usuário não precise se ater às particularidades do código de instrumentação, uma ferramenta de instrumentação foi desenvolvida com esta finalidade. De maneira similar às aplicações, o sistema operacional também deve ser instrumentado para que suas diferentes atividades produzam evidências de execução. Tais atividades englobam atendimento de interrupções, escalonamento e preempção de tasks e troca de mensagens, dentre outras. Além da adição das instruções de instrumentação ao código, há o correspondente armazenamento de informações correlatas a elas em um *database*, conforme ilustração.

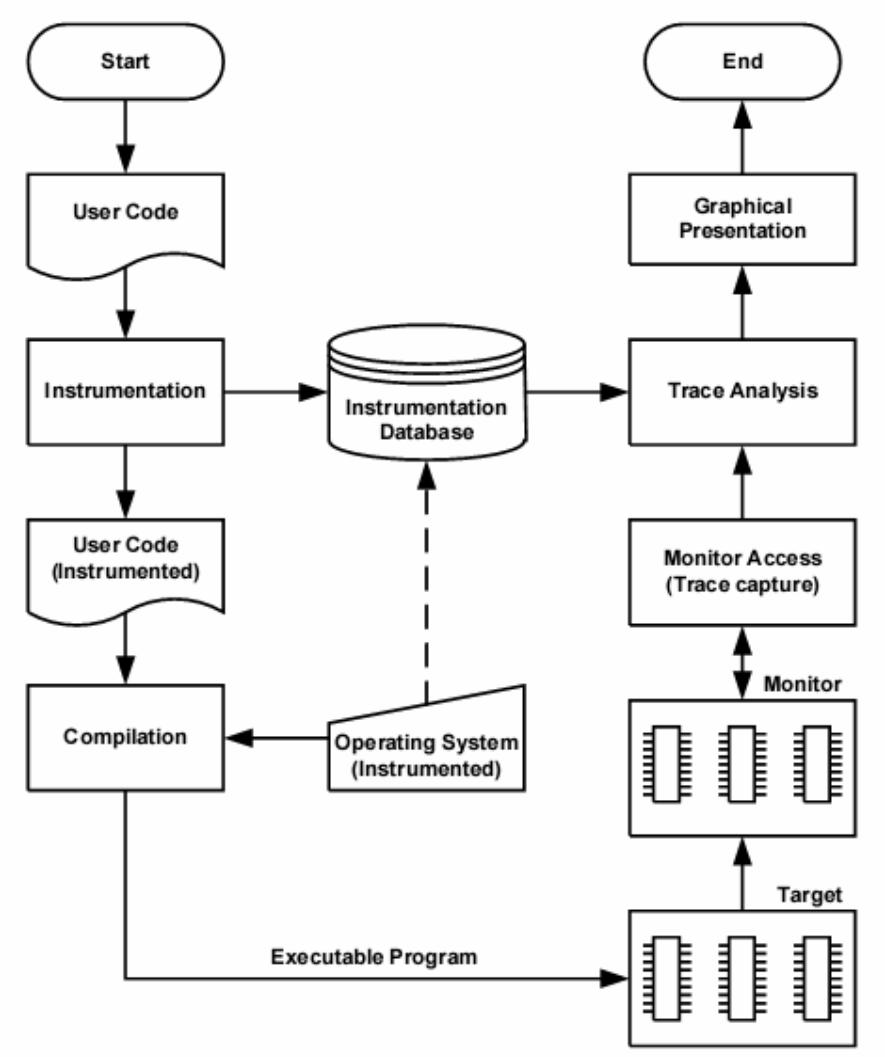

Figura 7-3 – Diagrama de fluxo do SoftScope
Fonte: [Cadamuro 06]

Para a obtenção de um arquivo executável, tanto o código de aplicação quanto o sistema operacional instrumentado são compilados. Este arquivo é então carregado no *target* para que possa ser executado de maneira idêntica àquela que teria lugar no caso da ausência completa de instrumentação. Durante a execução do código instrumentado, as instruções adicionadas provocam a geração de sinais que são enviados a uma porta de saída do *target*. O monitor captura estas informações e as armazena juntamente com a informação de tempo relativo em que ocorreram. O início e final do armazenamento das informações capturadas pelo monitor são controlados por uma entidade de controle do monitor, baseada em um *host* externo ao mesmo.

Quando uma captura é finalizada, o processo de análise começa a decodificação dos dados capturados com o auxílio das informações armazenadas no database durante a fase de instrumentação. O resultante do processo de decodificação é um arquivo ainda de difícil compreensão. Tais dados são melhor visualizados através de ferramentas de apresentação gráfica, parte integrante do SoftScope.

Quando da publicação do artigo a que se referem os parágrafos acima, o monitor MIMO não havia sido finalizado, razão pela qual se optou pela inclusão, junto à aplicação e ao sistema operacional, de um simulador de monitor. Tal simulador fora executado como um agente no *target*, escondendo das ferramentas do *host* a inexistência de um monitor de fato. Esta abordagem permitiu a geração de uma série de arquivos de trace que puderam ser, em uma fase posterior à publicação do artigo, úteis na depuração do MIMO.

7.2.3 MÉTODO DE INSTRUMENTAÇÃO

O método de instrumentação utilizado pelo SoftScope segue os princípios estabelecidos pelo Dyretiva [Cadamuro 07], utilizando instrumentação por alteração do código fonte (Seção 2.5.4). Para auxiliar o usuário na instrumentação de grandes programas, foi desenvolvido um instrumentador automático de código fonte.

O instrumentador do SoftScope obedece aos seguintes requisitos:

- Permitir escolher o tipo de instrumentação a inserir.
- Permitir escolher em que funções a instrumentação deve ser inserida.
- Instrumentar chamadas de função.
- Permitir instrumentação manual, instrumentação automática e instrumentação semi-automática.
- Permitir acompanhar valores de variáveis de estado do sistema.
- Formato das instruções de instrumentação

7.2.3.1 Tipo de instrumentação

A instrumentação será constituída de chamadas a funções (ou substituição de macro) da forma { instr(event_id); }. O que faz a função/macro instr ficará a cargo do arquivo de cabeçalho “instr.h”, que deverá ser fornecido pelo usuário de acordo com os detalhes específicos do seu *hardware*. event_id deverá identificar o evento relacionado à

instrumentação no sistema de forma única e inequívoca, caso contrário não será possível saber qual ação do sistema gerou o evento.

Apesar de o arquivo “instr.h” poder ser incluído de forma automática pelo instrumentador, esta tarefa será intencionalmente deixada para o usuário, permitindo que ele escolha a localização do mais adequada para este arquivo na sua árvore de diretórios.

Se o monitor híbrido (MIMO) estiver sendo utilizado, *instr* será uma macro expandida para uma escrita de um dado de 16 bits em um endereço fixo de 32 bits. O identificador do evento será o valor do dado. Caso a monitoração estiver sendo feita apenas por *software*, é necessário que *instr* armazene no local apropriado além do identificador do evento, também o tempo em que ele ocorreu.

7.2.3.2 Escolha das funções a instrumentar

A escolha das funções a instrumentar deverá ser feita pelo programador diretamente no código fonte, utilizando para isso os formatos especiais de comentários que já são suportados:

```
/* instr_total */
/* instr_func */
/* instr_start */
/* instr_null */
```

Instrumentação de chamadas de função de biblioteca:

Todas as chamadas de biblioteca que se desejar monitorar terão as chamadas desviadas para uma função que: marcará a hora de entrada, chamará a função original e, quando esta retornar, marcará a hora de saída.

O padrão de mudança de nome das funções monitoradas será feito acrescentando-se à função de empacotamento a cadeia de caracteres “_wrapper”. Por exemplo, se desejarmos monitorar a função de biblioteca write cujo código fonte não esteja disponível, as chamadas a write no código fonte serão substituídas por chamadas a write_wrapper. Protótipos para todas as funções de empacotamento deverão existir em um arquivo de cabeçalho chamado wrapper.h, que deverá estar visível no caminho de inclusão de arquivos de cabeçalho do compilador (opção –I do compilador GNU gcc, por exemplo).

Uma forma possível de implementação deste tipo de instrumentação poderia ser através do fornecimento, pelo usuário, de um arquivo texto com os protótipos das funções de biblioteca que ele deseja monitorar. Com isto, o instrumentador poderia ler este arquivo texto com os protótipos, realizar as substituições necessárias nos arquivos fonte do projeto, e ainda gerar de forma automática os arquivos wrappers.c e wrappers.h em um diretório específico indicado pelo usuário.

7.2.3.3 Formas de instrumentação

Existirão três formas distintas de inserir instruções de instrumentação no(s) arquivo(s) a ser analisados: automática, semi-automática e manual.

A instrumentação automática é a inclusão de chamadas à função de instrumentação intr(event_id); inseridas de forma autônoma pelo instrumentador de acordo com as instruções do usuário (/* instr_total */, /* instr_func */, /* instr_start */, /* instr_null */).

A instrumentação semi-automática é aquela em que o usuário deseja incluir uma instrução de instrumentação isoladamente em um determinado ponto de código, mas como não pode prever qual identificador único será atribuído pelo instrumentador ao evento, precisa apenas indicar o lugar da inserção. Para suportar este tipo de instrumentação, será criado um comando adicional para o instrumentador na forma de comentário, que será o /* instr_user */.

A instrumentação manual é a forma prevista de instrumentação para o sistema operacional.

7.2.3.4 Monitoração de variáveis

O monitoramento de variáveis foi introduzido no Dyretiva para permitir que o usuário possa acompanhar a mudança de valor em variáveis chave do sistema, como por exemplo nas variáveis que representam estados no sistema.

Para que todas as atribuições a uma variável possam ser monitoradas no Dyretiva, é necessário que o usuário dê a esta variável um nome único no sistema, e que este nome único termine com a cadeia de caracteres “_TRACED”. Assim, se uma variável têm nome device1_state e o usuário deseja que esta variável seja monitorada, seu nome deve ser alterado para device1_state_TRACED. Neste caso, quando o instrumentado encontrar a declaração desta variável, ele reconhecerá nela uma variável que precisa ser monitorada, associará a ela um identificador único, localizará todas as atribuições que são feitas a ela no

código fonte, e acrescentará após cada atribuição a chamada a uma das funções de monitoração de valores de variáveis, que pode ser instr_data8(data_id,val8), instr_data16(data_id,val16) ou instr_data32(data_id,val32), dependendo se a variável for de 8, 16 ou de 32 bits.

7.2.3.5 Formato das instruções de instrumentação

Para cumprir os objetivos do Dyretiva, é necessário que a intrusão causada pelas instruções de instrumentação seja mínima. Em virtude disto, e para minimizar as necessidades de suporte de hardware no sistema sob teste, foi decidido que as instruções de instrumentação seriam escritas de 16 bits no barramento de dados, capturadas e externadas através de um latch.

Com 16 bits de dados é possível sinalizar 65.536 eventos simples, que podem ser início/final de função, estrutura de decisão (*if/then/else*), estrutura de repetição (*do/while*), ou ponto de verificação do usuário. No entanto, nem todos os eventos que se deseja monitorar são simples. Em especial a monitoração do sistema operacional e das variáveis de um programa não são contempladas apenas com uma escrita de 16 bits de largura. Por isso as instruções de instrumentação foram divididas em três formatos distintos: instruções de instrumentação simples, instruções de instrumentação de variáveis e instruções de instrumentação do sistema operacional, conforme a tabela abaixo:

Formato da instrução de instrumentação	Faixa de valores
Simples	0x0000 até 0xDFFF
Variáveis	0xE000 até 0xEFFF
Sistema operacional	0xF000 até 0xFFFF

Tabela 7-1 – Valores dos códigos de instrumentação

Com instruções simples é possível monitorar até 57344 eventos.

Já com as instruções de instrumentação de variáveis pode-se acompanhar o valor de até 2048 variáveis do sistema, conforme o seguinte modelo:

Número dos bits															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	sz	veid										

Tabela 7-2 – Instrumentação de variáveis

No *nibble* superior da palavra encontra-se o valor hexadecimal ‘E’ (binário “1110”), indicando que esta é uma instrução de instrumentação de uma variável. O bit número 11 indica se a variável monitorada é de 16 bits (bit 11 = 0) ou de 32 bits (bit 11 = 1). Se a variável monitorada for de 16 bits (bit 11 = 0), a próxima escrita na porta será o valor da variável monitorada. Por outro lado, se a variável monitorada for de 32 bits (bit 11 = 1), então as duas próximas escritas serão o valor da variável monitorada, sendo que a primeira escrita conterá o valor da metade superior da palavra, e a segunda o valor da metade inferior da palavra. Por fim, os bits de 10 até 0 são o identificador único da variável monitorada, permitindo o acompanhamento de até 2048 variáveis em um sistema.

A instrumentação do sistema operacional é a última categoria. O formato de uma instrução de instrumentação de sistema operacional é mostrada na tabela a seguir.

Número dos bits															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Size			oseid								

Tabela 7-3 – Instrumentação do sistema operacional

No *nibble* superior da palavra encontra-se o valor hexadecimal ‘F’ (binário “1111”), indicando que esta é uma instrução de instrumentação do sistema operacional. Os bits de 11 até 9 indicam quantas escritas subseqüentes (quantidades de 16 bits) serão necessárias para descrever completamente o evento do sistema operacional. Por fim, os bits de 8 até 0 são responsáveis por conter o identificador evento do sistema operacional, num total de até 512 eventos diferentes deste tipo. Este tipo de instrução de instrumentação não é inserida no código fonte de forma automática ou semi-automática, mas sim de forma manual pelos projetistas do sistema operacional.

7.2.4 ANÁLISE, FILTRAGEM E APRESENTAÇÃO DOS RESULTADOS

Na seção anterior foi mostrado o processo de instrumentação do utilizado no SoftScope. Quando o SUT executa o código instrumentado e o monitor coleta os estímulos correspondentes, é gerado um arquivo binário que contém dados brutos correspondentes à escrita na porta de monitoração do SUT (código do evento e informações específicas deste) e o valor de *timestamp* gerado pelo monitor.

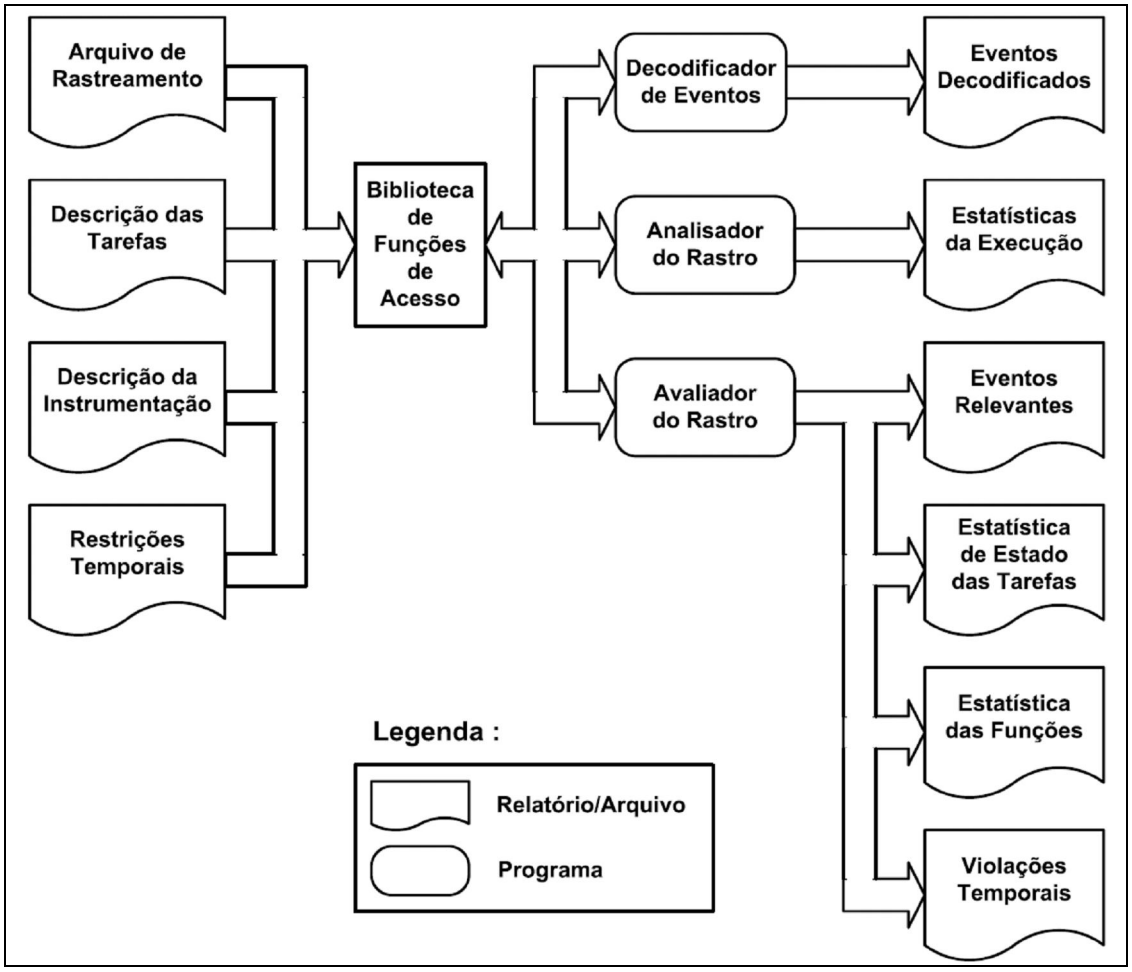

Figura 7-4 – Análise e filtragem no SoftScope

Fonte: [Cadamuro 07 b]

A Figura 7-4 mostra, à esquerda: o arquivo de rastreamento, mencionado no parágrafo anterior; arquivos descrição de tarefas e de descrição de instrumentação, obtidos no processo de instrumentação do código e o arquivo de restrições temporais. Este último arquivo é gerado pelo usuário do ERTS e que contém as restrições, de acordo com o modelo de falta do Dyretiva - seção 7.1.3.

Ainda na Figura 7-4 mostra-se a biblioteca de funções de acesso, necessária para:

- Decodificação de eventos, seguindo a tabela Tabela 7-1.
- Geração de estatísticas de execução, atrvés da análise dos *traces.*
- Separação dos eventos relevantes; geração da estatística da permanência em cada estado; geração da estatística de execução das funções e geração do relatório de violações temporais. Todos estes dados são gerados pelo avaliador de *traces.*

Tipo do Evento	Significado
1	Início da execução de uma função.
2	Final da execução de uma função.
3	Cláusula `case` executada.
4	Cláusula `default` executada.
5	Início da execução de um `if`.
6	Fim da execução de um `if`.
7	Início da execução de um `else`.
8	Início da execução de um `switch`.
9	Final da execução de um `switch`.
10	Início da execução de um laço `while`.
11	Final da execução de um laço `while`.
12	Início da execução de um laço `do`.
13	Final da execução de um laço `do`.
14	Início da execução de um laço `for`.
15	Final da execução de um laço `for`.
16	Cláusula `goto` executada.
17	Cláusula `continue` executada.
18	Cláusula `break` executada.
19	Cláusula `return` executada.
20	Instrumentação casual executada.
21	Valor reservado.
22	Início de chamada de função de biblioteca.
23	Retorno de chamada de função de biblioteca.

Tabela 7-4 – Tipos de eventos de monitoração

Fonte: [Cadamuro 07]

As diversas modalidades de saída de dados são exemplificadas a seguir. São mostrados os formatos de saída gerados no âmbito do SoftScope e também o resultado do processamento destes dados com vistas a torná-los não só de mais fácil compreensão como também permitir que análises rápidas sejam feitas de maneira mais segura, evitando erros de interpretação de dados numéricos. Este processamento adicional é fruto de um trabalho desenvolvido a partir do trabalho de Navarro [Navarro 04]. No trabalho original era proposto um ambiente para melhorar o entendimento das características dinâmicas de ERTS. Apenas as saídas relevantes do sistema de pós-processamento são mostradas. Referências à configuração deste sistema podem ser encontradas em [Cadamuro 06].

7.2.4.1 Eventos decodificados

Com base na decodificação do arquivo de saída do monitor, obtém-se um outro arquivo como o da Figura 7-5. A coluna da esquerda mostra o tempo em nanosegundos entre dois eventos consecutivos. A segunda coluna mostra duas informações separadas pelo sinal

de dois pontos: posição em bytes desde o início do arquivo e número de seqüência também contado a partir do início do arquivo. O resto de cada linha refere-se à descrição do arquivo propriamente dita.

```
0001995200 [000000098:00000015]: Decrementer exception
0000005360 [000000106:00000016]: End of exception
0001995200 [000000112:00000017]: Decrementer exception
0000006340 [000000120:00000018]: Put MSG from ISR to "TimeManager"
0000010000 [000000128:00000019]: Schedule - running now is thread "TimeManager"
0000001280 [000000136:00000020]: End of exception
0000009340 [000000142:00000021]: System Call exception
0000006040 [000000150:00000022]: Receive Put MSG in thread "TimeManager"
0000007260 [000000158:00000023]: End of exception
0000006500 [000000164:00000024]: Timer expired for thread "ttyManager"
0000004840 [000000172:00000025]: System Call exception
0000007680 [000000180:00000026]: Put MSG from "TimeManager" to "ttyManager"
0000010280 [000000190:00000027]: End of exception
0000011040 [000000196:00000028]: System Call exception
0000005000 [000000204:00000029]: Receive MSG without any MSG - thread "TimeManager" blocked
0000005640 [000000212:00000030]: Schedule - running now is thread "ttyManager"
0000002020 [000000220:00000031]: End of exception
0000007800 [000000226:00000032]: System Call exception
0000006820 [000000234:00000033]: Receive Put MSG in thread "ttyManager"
0000006200 [000000242:00000034]: End of exception
0000004200 [000000248:00000035]: User event: (5)
0000032800 [000000254:00000036]: User event: (6)
```

Figura 7-5 – Exemplo de saída do decodificador de eventos

Na Figura 7-6 é mostrada a lista de eventos de forma mais amigável

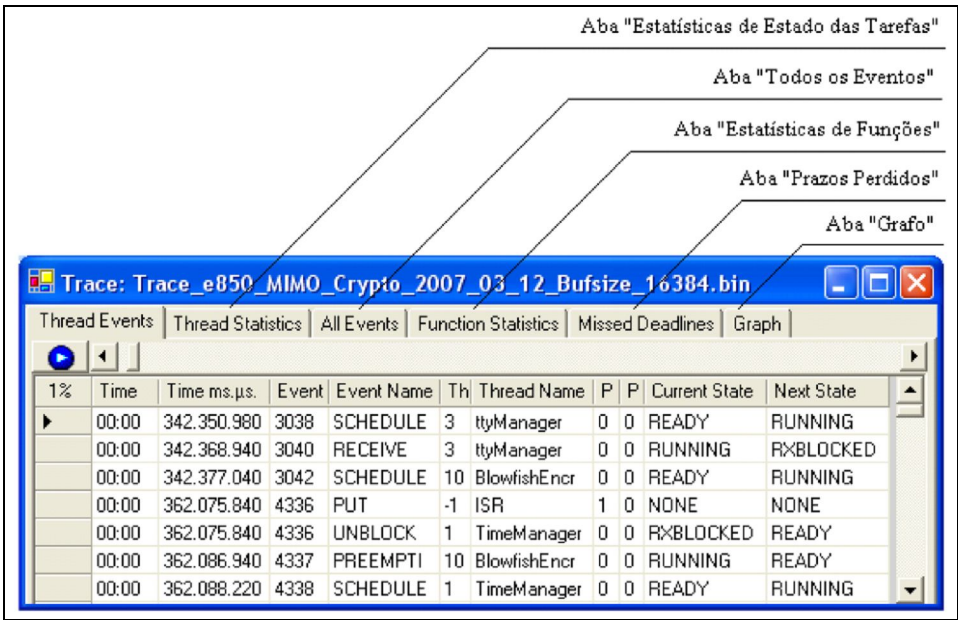

Figura 7-6 – Visualização de eventos

7.2.4.2 Estatísticas de execução

O analisador de rastro gera um arquivo como o visto na Figura 7-7. É composto por informações como: divisão do tempo entre tarefas (*threads*), tempo ocioso (*Idle*) e sistema operacional (SYS). As outras informações dizem respeito ao tempo gasto com exceções (*Exceptions*), chamadas do sistema operacional (*system services*), serviço de interrupções (*interrupt handling*) e as tarefas propriamente ditas (threads). A Figura 7-8 e a Figura 7-9 mostram informações da mesma natureza.

Uma observação importante a respeito da Figura 7-7 é a referência às siglas OBCET (*Observed Best Case Execution Time*), OTCET (*Observed Typical Execution Time*) e OWCET (*Observed Worst Case Execution Time*). A diferença entre estes valores e aquele mencionados na seção sobre ERTS é a de que trata-se dos tempos observados e não estimados ou mesmo os tempos reais, daí a adição da letra O indicando observado ou *observed*.

```
Trace total time: 19.9s
CPU Idle time: 16.9s (84.66%)
CPU SYS time: 232.1ms (1.16%)
Threads time: 2.8s (14.17%)

Exceptions: "- Exception Name: [invocations], [OBCET], [OTCET], [OWCET];"
- Decrementer: <9965>, 5.3us, 7.3us, 23.2us;
- System Call: <10124>, 9.9us, 15.7us, 23.8us;

System services: "- System Service: [number_of_calls], [OBCET], [OTCET], [OWCET];"
- PutMessage   : <1116>, 14.5us, 17.8us, 19.4us;
- SendMessage  : <1156>, 17.4us, 22.5us, 23.8us;
- ReceiveMessage (empty)   : <2352>, 9.9us, 12.7us, 14.3us;
- ReceiveMessage (Put)     : <2192>, 11.9us, 13.1us, 14.9us;
- ReceiveMessage (no match): <996>, 16.7us, 17.0us, 17.6us;
- ReceiveMessage (Send)    : <1156>, 13.6us, 15.2us, 15.7us;
- ReplyMessage : <1156>, 14.2us, 17.0us, 19.1us;

Interrupt handling "- IRQ Name: [invocations], [OBCET], [OTCET], [OWCET];"

Threads: "- Thread Name: [number_of_schedules], [CPU_Time] (CPU_Time%);"
- TimeManager: <8728>, 70.5ms (0.35%);
- ttyManager: <4990>, 83.6ms (0.42%);
- Heartbeat(GREEN): <120>, 1.3ms (0.01%);
- Heartbeat(RED): <120>, 1.2ms (0.01%);
- CryptoManager: <202>, 118.1ms (0.59%);
- BlowfishEncrypter: <82680>, 1.3s (6.34%);
- BlowfishDecrypter: <82720>, 1.3s (6.37%);
```

Figura 7-7 – Exemplo de saída do analisador de rastro

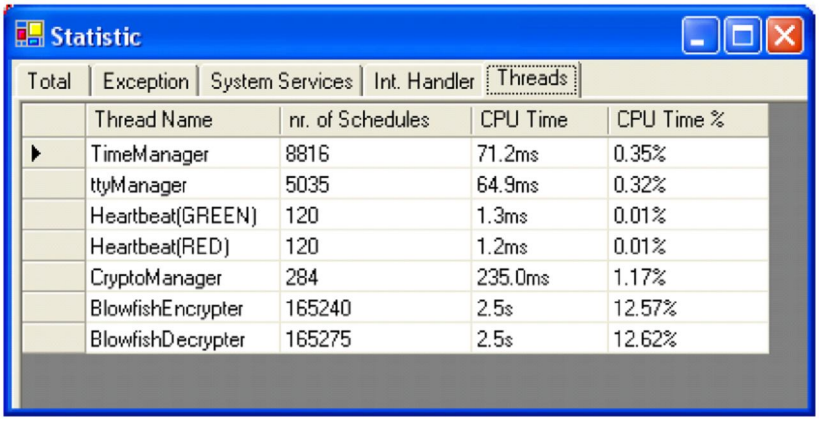

Figura 7-8 – Estatísticas geradas pelo analisador de rastro

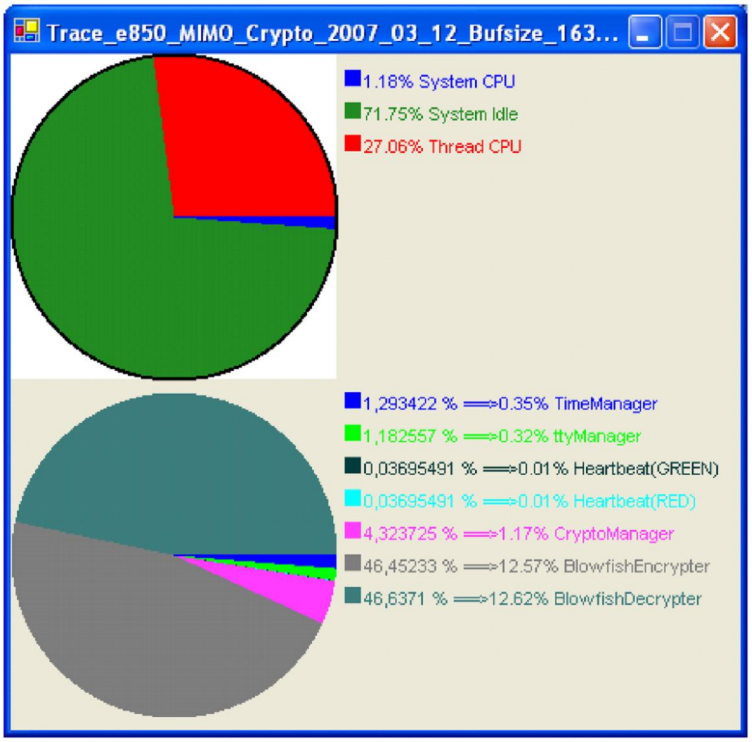

Figura 7-9 – Estatísticas de utilização do processador

7.2.4.3 Eventos relevantes

O analisador de rastro pode também selecionar uma lista de evento mais significativos, que permitem reconstruir a seqüência de execução das tarefas conforme exemplo mostrado na Figura 7-10. O gráfico de Gannt mostrado na Figura 7-11 mostra um a acupação do processador por várias tarefas. A ferramenta permite que a visualização das

tarefas seja feita proporcionalmente ao tempo de cada tarefa ou de maneira que as tarefas ocupem espaços iguais. Isto permite a visualização de tarefas com duração muito reduzida.

```
0000102319740, [000002426:00000326], SCHEDULE, 0, IdleTask, 0, 0, READY, RUNNING;
0000122042580, [000002566:00000346], PUT, -1, ISR, 1, 0, NONE, NONE;
0000122042580, [000002566:00000346], UNBLOCK, 1, TimeManager, 0, 0, RX BLOCKED, READY;
0000122053260, [000002574:00000347], PREEMPTION, 0, IdleTask, 0, 0, RUNNING, READY;
0000122054540, [000002582:00000348], SCHEDULE, 1, TimeManager, 0, 0, READY, RUNNING;
0000122069860, [000002596:00000350], RECEIVE, 1, TimeManager, 0, 0, RUNNING, RUNNING;
0000122097680, [000002626:00000354], PUT, 1, TimeManager, 3, 0, RUNNING, RUNNING;
0000122097680, [000002626:00000354], UNBLOCK, 3, ttyManager, 0, 0, RX BLOCKED, READY;
0000122124700, [000002650:00000357], RECEIVE, 1, TimeManager, 0, 0, RUNNING, RX BLOCKED;
0000122132020, [000002666:00000359], SCHEDULE, 3, ttyManager, 0, 0, READY, RUNNING;
0000122146620, [000002680:00000361], RECEIVE, 3, ttyManager, 0, 0, RUNNING, RUNNING;
```

Figura 7-10 – Lista de eventos relevantes

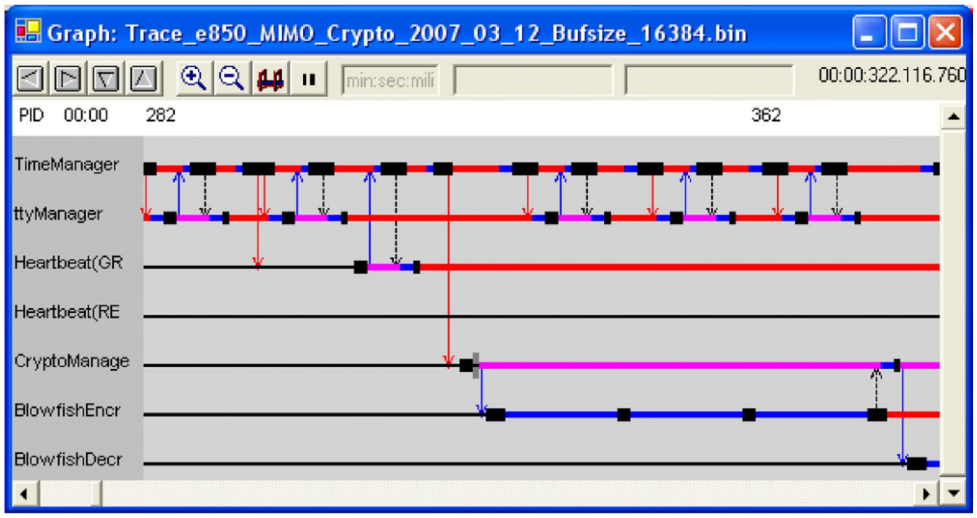

Figura 7-11 – Gráfico de Gannt com a seqüência de tarefas

7.2.4.4 Estatística de estados de uma tarefa

Uma tarefa pode ser analisada separadamente conforme visto na Figura 7-12. Cada tarefa pode estar em um dos estados do sistema opearcional PET#. Uma tarefa fica no estado UNKNOW a partir da início da monitoração do SUT até que sejam coletadas informações suficiente para que passe a um estado conhecido.

```
Thread name: CryptoManager (preempted 40 times)
--> Time spent in the UNKNOWN state: 308.2ms (1.53%)
--> Time spent in the READY state: 14.7ms (0.07%)
--> Time spent in the RUNNING state: 237.7ms (1.18%)
--> Time spent in the RX BLOCKED state: 14.4s (71.56%)
--> Time spent in the TX BLOCKED state: 5.2s (25.66%)
```

Figura 7-12 – Exeemplo de estatística dos estados de uma tarefa

7.2.4.5 Estatística de execução de funções

De maneira similar à mencionada na seção anterior com respeito a determinada tarefa, uma análise mais acurada do comportamento de uma função pode ser obtido conforme mostra a Figura 7-13.

```
Function [BlowfishEncrypt] in file [blowfish.c] has been executed [81920] times
--> Average execution time   : 29.5us
--> Best  case execution time: 26.7us ([BlowfishEncrypter], trace offset [1950284], event# [312824])
--> Worst case execution time: 46.5us ([BlowfishEncrypter], trace offset [2193614], event# [351880])
Function [BlowfishDecrypt] in file [blowfish.c] has been executed [81920] times
--> Average execution time   : 29.6us
--> Best  case execution time: 26.9us ([BlowfishDecrypter], trace offset [173538], event# [27856])
--> Worst case execution time: 45.8us ([BlowfishDecrypter], trace offset [589124], event# [94522])
```

Figura 7-13 – Exemplo de estatística de execução de funções

7.2.4.6 Violações temporais

O último, porém não menos importante, conjunto de informações diz respeito às violações que podem ocorrer nas especificações temporais. Aproximadamente no meio do arquivo de especificação de restrições temporais (Figura 7-14) pode ser vista a especificação de um *deadline* para a ocorrência de determinado evento como sendo de 300 ms. A Figura 7-15 mostra a saída de um arquivo com violações temporais onde pode-se notar uma restrição temporal violda com 301,2 ms, na décima linha.

```
[NUMBER_OF_TIME_REQUIREMENTS = 2]
[TIME_REQUIREMENT_NAME = UmaRestricaoTemporal]
[START_EVENT = OS, 2, 82, 9, 10]
[END_EVENT = OS, 2, 87, 11, 9]
[RELATED_THREADS = 2]
[THREAD1 = NomeDaPrimeiraTarefaEnvolvida]
[THREAD2 = NomeDaSegundaTarefaEnvolvida]
[DEADLINE = 300]
[INTERVAL = 500]
[TIME_REQUIREMENT_NAME = OutraRestricaoTemporal]
[START_EVENT = OS, 2, 82, 9, 12]
[END_EVENT = OS, 2, 87, 13, 9]
[RELATED_THREADS = 1]
[THREAD1 = NomeDaTarefaEnvolvida]
[DEADLINE = 200]
[INTERVAL = 1000]
```

Figura 7-14 – Arquivo de especificação de restrições temporais

```
Time requirement [UmaRestricaoTemporal]
Deadline  [300ms]
Period    [500ms]
Instances [40]
Average jitter    [150.2us]
Worst case jitter [210.1us]
Missed deadlines [2]
--> 1st missed deadline offset         [1872890]
--> 1st missed deadline event number   [305587]
--> 1st missed deadline total execution time [301.2ms]
--> 1st missed deadline feature execution time [296.1ms]
--> 1st missed deadline feature blocked by the system [3.2ms]
--> 1st missed deadline feature blocked by other threads [1.9ms]
Met deadline in the best case [275.8ms]
--> Met deadline best case offset [336138]
--> Met deadline best case event number [54707]
Met deadline in the worst case [298.9ms]
--> Met deadline worst case offset [1982658]
--> Met deadline worst case event number [323507]
```

Figura 7-15 – Resultado da análise de violações temporais

7.3 PERF

O PERF é um projeto de longa duração no CPGEI [Kawamura 99][Renaux 99][Góes 01a][Renaux 02] , . Consiste de ferramentas integradas para a estimação de tempos de execução de programas executados em ERTSs. A Figura 7-16 mostra as diversas janelas do aplicativo usado pelo PERF para interface com o usuário. A idéia básica do método de análise de tempos é a montagem de grafos de segmentos, visto à esquerda na figura Figura 7-16. Cada segmento é composto por instruções que são executadas obrigatoriamente em seqüência. Caso haja desvios tais como saltos condicionais ou incondicionais, chamadas de funções entre outras, são necessariamente criados novos nós e segmentos para contemplar-se a execução seqüencial e ininterrupta das instruções formadoras de cada segmento.

Como exemplo, as áreas em cinza na Figura 7-16 evidenciam um laço. Correspondente a este laço pode-se ver a listagem do arquivo em *assembly* e em código fonte C. A janela menor, no canto superior direito, pede intervenção do usuário para fornecimento de valores mínimos, típicos e máximos (*worst-case*) para o número de iterações que o laço irá, hipoteticamente, perfazer. Isto é feito devido ao fato da ferramenta não possuir subsídios para inferência de tal valor, que deverá provir da experiência do usuário ou de projetos semelhantes.

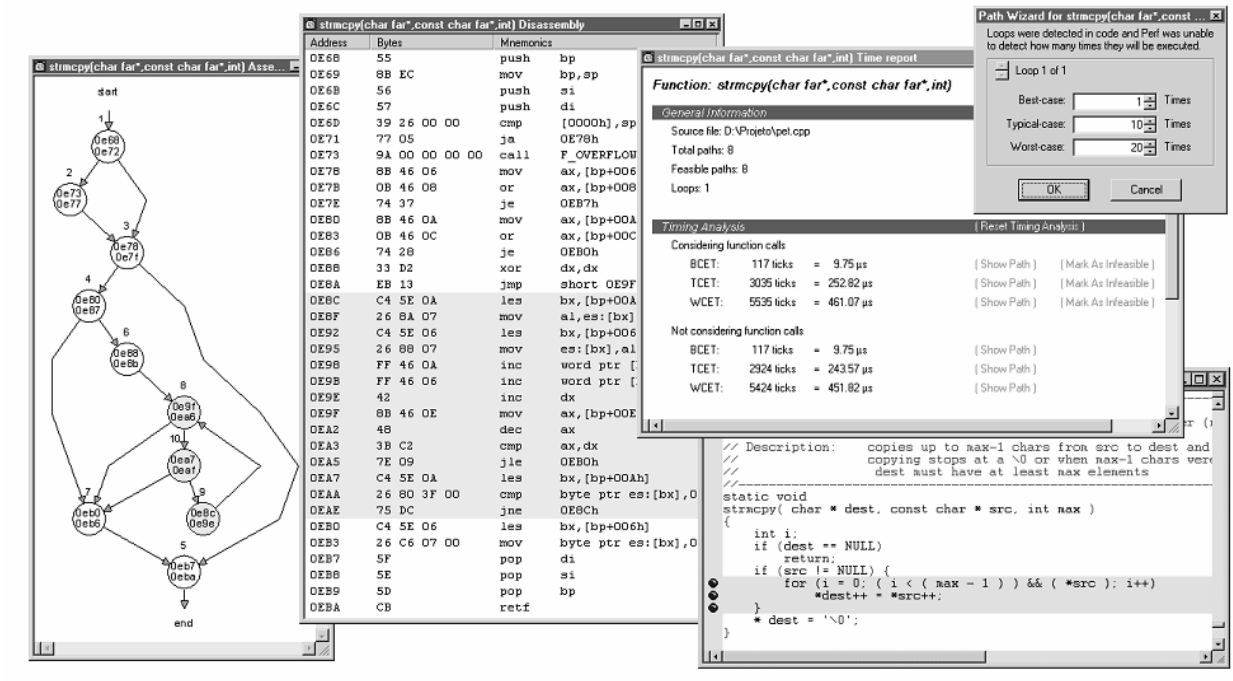

Figura 7-16 – Exemplo de análise da função *strmcpy*
Fonte: [Góes 01]

As últimas informações referem-se aos valores obtidos para BCET – Best-Case Execution Time e para TCET – Typical Case Execution Time e WCET - Worst Case Execution Time. Tais valores representam o melhor caso de execução ou seja menor tempo, o tempo típico e a pior hipótese que é a de maior tempo, respectivamente.

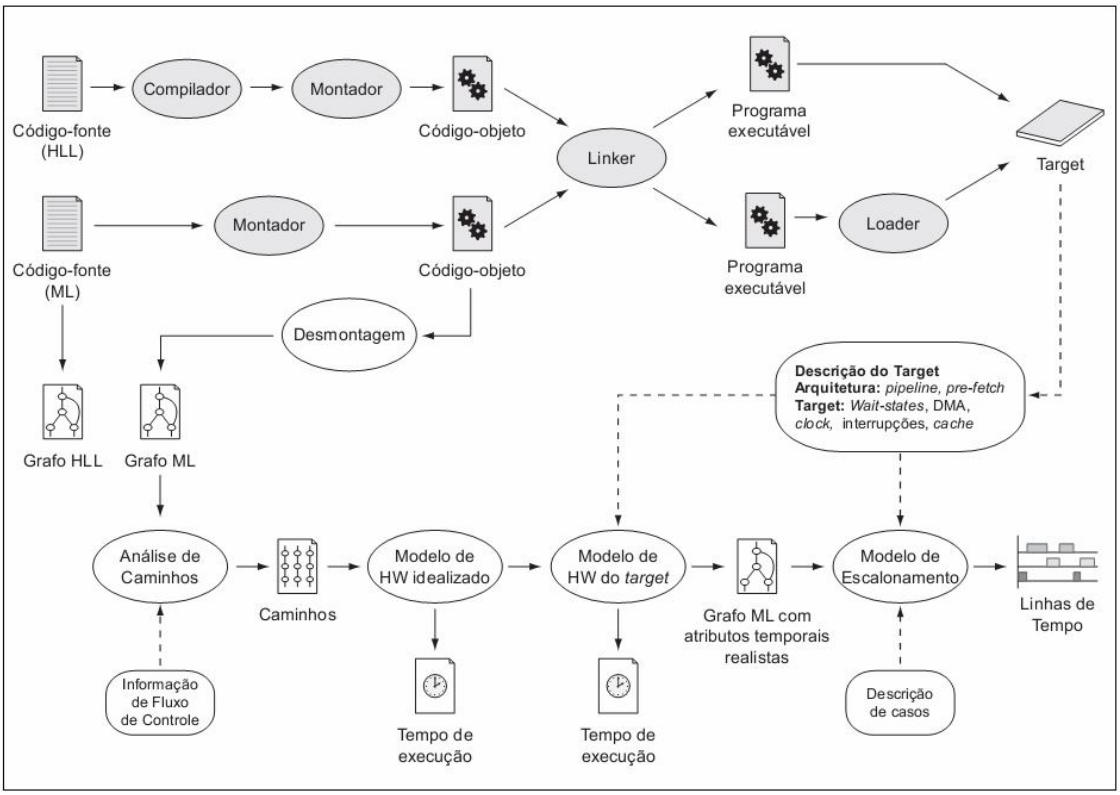

Figura 7-17 – Estimação de tempo no ambiente PERF
Fonte: [Góes 01]

A Figura 7-17 ilustra o processo completo de uso do PERF. Na parte superior vê-se os itens relativos ao processamento tanto de linguagens de alto nível HLL – *High Level Language*, quanto baixo nível, ML – *Machine Language*. O PERF permite a construção dos grafos de segmento para análise dos caminhos de execução a partir dos códigos fonte e também de engenharia recersa dos arquivos objeto. Estes grafos não levam em consideração as interferências a que um ERTS está sujeito em uso. Tais interferências são modeladas como se vê na parte inferior da Figura 7-17. Por fim é aplicado o modelo de escalonamento para a obtenção do diagrama de linhas de tempo.

7.4 RESUMO

Foram apresentados, de forma sucinta o método Dyretiva, suas ferramentas de suporte que integram o SoftScope e por fim uma ilustração de como um método estático de estimação temporal opera. Os formatos de instrumentação, relevantes para o entendimento do monitor MIMO foram descritos nesta seção, devido à sua foret relação com o método Dyretiva.

8 REFERÊNCIAS

[Braga 99] Braga, A. S. – *TLM – Uma Ferramenta de Apoio ao Teste de Restrições Temporais em Sistemas Dedicados Operando emTempo Real*. Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. CEFET-PR, Curitiba, 1999.

[Briand 99] Briand, L. P.; Roy, D. M. – Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach. Editora IEEE Computer Society Press. ISBN 0-8186-7406-7, E.U.A., 1999.

[Burns 97] Burns, A.; Wellings, A. – *Real-Time Systems and Programming Languages*. Editora Addison-Wesley – 2ª Edição, Inglaterra, 1997.

[Cadamuro 06] Cadamuro, João; Renaux, Douglas P. B. – *SoftScope: Embedded Real-Time Systems Verification Tool Set* RTSS 2006 - 27th IEEE International Real-Time Systems Symposium

[Cadamuro 07] Cadamuro, João; – *Dyretiva : um método para a verificação das restrições temporais em sistemas embarcados.* Tese de Doutorado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. UTFPR-PR, Curitiba, 2007

[Cargille 92] Cargille, J.; Miller, B. P. – *Binary Wrapping: A Technique for Instrumenting Object Code*. ACM SIGPLAN Notices. Pág. 17-18, v. 27, n. 6, Junho/1992. Association for Computing Machinery.

[Cohen 03] Cohen, Ben – *Vhdl Coding Styles Methodologies.* 2ª edição. Kluver Academic Publishers, 2003.

[Copetti 07] Copetti, L. F.; Cadamuro Jr., João; Renaux, D. P. B.; Pedroni, V. A. – *MIMO: A Hybrid Monitor for Embedded Real-Time Systems*. IX Workshop on Real-Time Systems (WTR 2007), Pág. 39-46, Belém, Maio/2007.

[eSysTech 03] *Manual do Usuário da Placa de Avaliação e850Lite*. Documento PR-ESYS-E850L-MAN-0100b, versão 1.00b. eSysTech Ltda., Brasil, Julho/2003.

[Góes 01] Góes, J. – *PERF: Ambiente de Desenvolvimento e Estimação Temporal de Sistemas em Tempo Real*. Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. CEFET-PR, Curitiba, 2001.

[Góes 01a] Góes, J.; Linhares, R.; Renaux, D. – *Estimação de Tempo de Execução de Programas no Ambiente PERF*. III Workshop em Sistemas em Tempo Real. Florianópolis, Maio/2001.

[Greenwalt 03] Greenwalt, T. – *Code Test: Logic Analyzer for Software Engineers*. Smart Networks Developer Forum 2003. Relatório da Seção Técnica K220, 35 páginas, Dallas, Texas, E.U.A., Março/2003. Motorola, Inc.

[Harel 87] Harel, D. – *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming 8, pág. 231-274, 1987. Elsevier Science Publishers B.V.,North-Holland.

[Hennessy 96] Hennesy, J. L.; Patterson, D. A. – *Computer Architecture: A Quantitative Approach*. Editora Morgan Kaufmann – 2ª Edição, E.U.A., 1996.

[Kawamura 99] Kawamura, A. – *Análise Estática de Programas para Predição de Tempos de Execução*. Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. CEFET-PR, Curitiba, 1999.

[Larus 94] Larus, J. R.; Ball, T. – *Optimally Profiling and Tracing Programs*. Em ACM Transactions on Programming Languages and Systems. Pág. 1319-1360, v. 16, n. 4, Julho/1994. Association for Computing Machinery.

[Linhares 01] Linhares, R. R. – *Modelamento de Hardware Visando À Estimação do Tempo de Execução de Programas*. Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial. CEFET-PR, Curitiba, 2001.

[Mahrenholz 01] Mahrenholz, D. – *Minimal Invasive Monitoring*. Em Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'01). Pág. 0251, Maio/2001. IEEE Computer Society.

[Navarro 04] Navarro, C.; Renaux, D. P. B. – *Ambiente Didático para Monitoração de Sistema Operacional de Tempo Real*. Em WTR2004 -VI Workshop de Tempo Real. Pág. 29-36, Gramado, Brasil, Maio/2004.

[Parker 04] Parker, Roy H. - *Caution: Clock Crossing*. Chip Design Magazine http://www.chipdesignmag.com/display.php?articleId=32&issueId=5, Junho/2004.

[Pedroni 04] Pedroni, Volnei A. – *Circuit design with VHDL:* MIT Press, 2004.

[Perry 02] Perry, Douglas L. – *VHDL: Programming by Example 4th ed.* Editora McGraw-Hill, 2002.

[Puschner 89] Puschner, P.; Koza, C. – *Calculating the Maximum Execution Time of Real Time Programs*. Real-Time Systems, v. 1, n. 2, pág. 159-176.

[Renaux 96] Renaux, D. P. B. – *PET – A Small Real-Time Support System for Microcontroller without Virtual Memory*. Relatório Técnico, CPGEI. CEFET-PR, Curitiba, Maio/1996.

[Renaux 99] Renaux, D. P. B.; Braga, A.; Kawamura, A. – *PERF: Um Ambiente para Avaliação Temporal de Sistemas em Tempo Real*. II Workshop de Sistemas em Tempo Real, Salvador. Pág. 76-87, 1999.

[Renaux 02] Renaux, D. P. B.; Góes, J.; Linhares, R. – *WCET Estimation From Object Code Implemented in the PERF*. Em Proceeding of the Second International Workshop on Worst Case Execution Time Analysis, evento satélite ao 14th Euromicro Conference on Real-Time Systms. Viena, Áustria. Pág. 28-35, 2002.

[Schneier 94] Schneier, B.; – *Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)*. Cambridge Security Workshop Proceeding. Pág. 191-204, 1994. Springer-Verlag.

[Schroeder 95] Schroeder, B. A.; – *On-Line Monitoring: A Tutorial*. Em IEEE Computer. Pág. 72-78, v. 28, n. 6, Junho/95. IEEE Computer Society.

[Stunkel 91] Stunkel, C. B.; Janssens, B.; Fuchs, W. K. – *Address Tracing for Parallel Machines*. Em IEEE Computer. Pág. 31-38, v. 24, n. 1, Janeiro/91. IEEE Computer Society.

[Thane 00] Thane, H.; Hansson, H. – *Using Deterministic Replay for Debugging Distributed Real-Time Systems*. Nos anais de 12th Euromicro Conference on Real-Time Systems (Euromicro-RTS 2000). Pág. 265, Junho/2000.

[Tsai 90] Tsai, J. J. P.; Fang, K.; Chen, H; Bi, Y. – *A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging*. Em IEEE Transactions on Software Engineering. Vol. 16, nº 8, Págs. 897-916, Agosto/1990.

[Turley 03] Turley, J. – *The Two Percent Solution*. Embedded Systems Programming – v. 16, n. 1 – Janeiro/2003.

[Valpereiro 06] Valpereiro, Filipe; Pinho Miguel – Run-time Monitoring Approach for the Shark Kernel. Polytechnic Institute of Porto (ISEP-IPP), Janeiro de 2006.

[Wilner 95] Wilner, D. – *WindView: A Tool for Understanding Real-Time Embedded Software Through System Visualization*. Em II ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems. Orlando, Flórida, E.U.A., Junho/1995. Association for Computing Machinery.

[Wind 95] *VxWorks 5.3 Programmer's Guide*. Manual do Tornado. Documento DOC-11045-ZD-01, 1ª Edição. Wind River Systems, Inc., E.U.A., Dezembro/1995.

[Wind 99] *WindView 2.0.1 User's Guide.* Manual do Tornado II. Documento DOC-12393-ZD-02, 1ª Edição. Wind River Systems, Inc., E.U.A., Março/1999.

[Xilinx 05] *Spartan-3 Preliminary Product Specification DS0999-1* v1.4 Xilinx Corporation, E.U.A., Janeiro/2005.

RESUMO:

Este trabalho apresenta o MIMO (Minimal Invasive MOnitor), um monitor híbrido usado em pesquisa acadêmica, mas com possibilidades de uso prático, para o teste de sistemas embarcados de tempo real. Monitores híbridos oferecem a acurácia e baixa intrusão típica de monitores baseados exclusivamente em hardware aliados à flexibilidade dos monitores construídos com base apenas em software. O monitor, em si, é um sistema embarcado de aquisição de dados, operando em tempo real, com seus próprios requisitos como eficiência e flexibilidade. Os princípios de concepção do MIMO, a estratégia para sua validação e o caso de teste são também apresentados. Um protótipo do MIMO foi construído e assim sua performance pode ser avaliada, com base na monitoração das atividades de um sistema embarcado de tempo real, multi-tarefa e preemptivo.

PALAVRAS-CHAVE

Monitoração híbrida, RTOS Debug, FPGA, Software embarcado

ÁREA/SUB-ÁREA DE CONHECIMENTO

1.03.00.00 – 7 Ciência da Computação

1.03.04.00-2 Sistemas de Computação

ano
2008

N°:477

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo