# Reprodução de Faltas em Programas Concorrentes Através da Combinação de Múltiplos Históricos Parciais

Nuno Machado, Paolo Romano e Luís Rodrigues

Instituto Superior Técnico/INESC-ID
nuno.machado@ist.utl.pt, romano@inesc-id.pt, ler@ist.utl.pt

**Resumo** Este artigo aborda o problema de reproduzir uma execução de um programa concorrente em que tenha ocorrido uma falta, de modo a facilitar a depuração do mesmo. O artigo propõe uma nova técnica que permite realizar esta reprodução com base num número potencialmente muito elevado de históricos parciais. O uso de históricos parciais permite diminuir substancialmente o custo associado à execução de código instrumentado, mas levanta o problema de encontrar a combinação de históricos que permite reproduzir a falta. O artigo propõe uma heurística para fazer esta selecção. O sistema resultante é avaliado através de uma bancada de testes que permite ilustrar as vantagens e limitações da aproximação proposta.

**Abstract** This paper addresses the problem of reproducing an execution of a concurrent buggy program, in order to ease its debugging. This paper proposes a novel technique to provide execution replay based on a great number of partial logs. The use of partial logs allows to substantially reduce the overhead imposed by the instrumented code execution, but raises the problem of finding the combination of logs capable of replaying the fault. This paper also proposes an heuristic to perform this search. The system is evaluated through a benchmark that shows the advantages and limitations of the proposed approach.

## 1 Introdução

Estima-se que 40% das falhas que ocorrem na execução de aplicações sejam originadas por erros durante a fase de desenvolvimento [1]. Infelizmente, apesar do progresso que tem sido feito no desenvolvimento de técnicas que permitem prevenir e evitar erros durante a produção do software (e.g. métodos formais [2]), um número significativo destes erros continua a chegar aos utilizadores [3]. A generalização da utilização da programação concorrente, como forma de tirar partido dos sistemas multi-processador, veio amplificar este problema.

Deste modo, continua a ser importante desenvolver ferramentas que facilitem a depuração do código, por exemplo, através da reprodução da execução que gerou um erro previamente observado. Infelizmente, esta tarefa é não trivial [4], nomeadamente em programas concorrentes com múltiplos *fios de execução*, uma

vez que ocorrência do erro pode depender de um encadeamento particular dos fios de execução no acesso a dados partilhados.

Neste artigo abordamos o problema de assegurar a *reprodução determinista* de um erro de forma a facilitar o seu diagnóstico [5]. Tipicamente, para atingir este objectivo, é necessário instrumentar o código de forma a recolher, durante a sua execução, a informação necessária para permitir a sua posterior reprodução [4] (e.g. ordem de acessos a regiões de memória partilhada, sinais, etc). Infelizmente, a recolha desta informação, que pode recorrer a suporte de hardware [6,7,8] ou ser feita apenas por software [9,10,11,4,12,13], introduz uma degradação significativa (entre 10x até 100x) no desempenho das aplicações [9,10]. Para mitigar este problema, propomos uma solução baseada em *históricos parciais*, isto é, em cada execução regista-se apenas um pequeno subconjunto de toda a informação necessária para a reprodução. Posteriormente, diferentes históricos produzidos por execuções independentes são combinados, recorrendo a uma heurística desenvolvida para o efeito, de modo a obter um histórico completo que permite a reprodução do erro.

A nossa solução denomina-se LEAP Cooperativo, pois tem por base o sistema de reprodução determinista LEAP [11], e combina técnicas que minimizam a informação que é necessário registar durante a execução da aplicação com técnicas de depuração estatística [14,15], que recolhem e analisam informação relativa a execuções de diferentes utilizadores. A solução é avaliada através de uma bancada de testes que permite ilustrar as vantagens e limitações da aproximação proposta.

Em suma, este artigo faz as seguintes contribuições: *i)* a especificação de um algoritmo de registo e análise da informação de diferentes utilizadores, para reproduzir deterministicamente o erro; *ii)* a descrição de um protótipo do LEAP Cooperativo; e *iii)* uma avaliação experimental do sistema usando bancadas de testes.

O resto do artigo está organizado da seguinte forma: a Secção 2 refere os conceitos base da reprodução determinista e da depuração estatística, e apresenta uma visão geral do LEAP; a Secção 3 descreve a arquitectura do LEAP Cooperativo e a sua concretização, bem como a heurística de combinação de históricos parciais; a Secção 4 avalia o protótipo implementado e apresenta os resultados; a Secção 5 revê o trabalho relacionado e, finalmente, a Secção 6 conclui o artigo.

## 2 Contexto

Nesta secção fazemos um breve sumário destas técnicas de reprodução determinista e depuração estatística, e da ferramenta LEAP, nas quais baseamos o nosso trabalho.

### 2.1 Reprodução Determinista

A *reprodução determinista* pretende resolver os problemas associados à reprodução de erros em programas concorrentes. Neste tipo de programas, a reexecução não guiada da aplicação não garante a reprodução do erro [16], visto que

este pode só se manifestar num conjunto reduzido dos encadeamentos possíveis dos fios de execução. Assim, esta abordagem começa por capturar informação relativa aos eventos não-deterministas da execução, guardando-a num ficheiro de *histórico*. Depois, re-executa a aplicação, consultando o histórico para forçar a reprodução dos eventos não-deterministas de acordo com a ordem observada na execução original.

Os eventos não-deterministas podem ter origem em factores internos (e.g. escalonamento de fios de execução) ou externos (e.g. dados vindos do teclado ou da rede, interrupções). No caso de sistemas multi-processador, é necessário registar também a ordem de acesso a variáveis partilhadas, incluindo a ordem de entrada nos pontos de sincronização (e.g. monitores), uma vez que os fios de execução podem executar-se de forma independente em diferentes processadores. No limite, é necessário registar cada acesso a memória partilhada, o que acarreta uma penalização significativa na execução original.

Apesar do seu custo, a necessidade de considerar erros ligados à concorrência é incontornável, devido à tendência de crescimento do seu número e proporção ao longo dos anos [17]. No nosso trabalho, centramo-nos na reprodução de erros de concorrência relacionados com violações de atomicidade e *corridas* no acesso aos dados. Assim, não consideramos a reprodução de outras fontes de não-determinismo, tais como dados inseridos pelo utilizador, entradas da rede, sinais e valores de retorno de chamadas ao sistema (e.g. `gettimeofday`).

### 2.2 Depuração Estatística

A depuração estatística [14] também procura facilitar o processo de depuração de programas de software, mas, ao contrário da reprodução determinista, preocupa-se mais em isolar e diagnosticar as causas do erro do que em reproduzi-lo. Esta abordagem é motivada pela observação de que as aplicações de software são, geralmente, executadas por grandes comunidades de utilizadores. Por isso, a depuração estatística procura acelerar o processo de detecção do erro através da distribuição da monitorização por diferentes clientes, aplicando depois modelos estatísticos sobre os dados recolhidos. Para diminuir a penalização imposta pela gravação de informação, assenta em períodos aleatórios de amostragem esparsa [14] .

### 2.3 LEAP

O LEAP [11] é um sistema recente de suporte à reprodução determinista de programas concorrentes, escritos em Java, em multiprocessadores. Este sistema baseia-se na observação de que basta registar localmente a ordem de acessos dos vários fios de execução a cada variável partilhada, ao invés da ordem global. Recorrendo a modelos matemáticos, os autores demonstram que esta informação localizada é suficiente para assegurar a reprodução determinista.

Para registar os acessos dos fio de execução, o LEAP associa um *vector de acesso* a cada variável partilhada. Durante a execução, sempre que um fio de execução lê ou escreve numa variável partilhada, o seu identificador é guardado no vector de acesso. Por exemplo, assuma-se a existência de um programa $P$ que

corre com dois fios de execução ($f_1$ e $f_2$) e contém uma variável partilhada $x$. Se, durante a execução de $P$, $x$ for acedida uma vez por $f_1$ e, posteriormente, duas vezes por $f_2$, o vector de acesso de $x$ será `<f1,f2,f2>`.

Esta abordagem permite tornar o processo de gravação do LEAP mais leve, sendo reportados valores para a degradação do desempenho na ordem dos 10% [11]. Contudo, o LEAP tende a produzir ficheiros de registo com tamanhos significativos (51 até 37760 KB/segundo) e não é eficiente para programas que acedem a múltiplas variáveis partilhadas em ciclos com muitas iterações. Para mitigar o problema do tamanho do histórico e reduzir ainda mais os custos de gravação, a nossa solução expande o LEAP de forma a aproveitar a cooperação de diferentes utilizadores.

## 3 LEAP Cooperativo

Nesta secção, descreve-se a arquitectura da nossa solução, a heurística de combinação de históricos parciais e a sua implementação.

### 3.1 Arquitectura

A Figura 1 ilustra a arquitectura do nosso sistema, indicando os módulos que a constituem. Durante a execução, o módulo *gravador* grava os vectores de acesso para um subconjunto de variáveis partilhadas aleatoriamente escolhidas aquando da instrumentação do programa (pelo *transformador* do LEAP [11]). Assumindo que o programa irá ser executado por um grande número de utilizadores, este mecanismo irá permitir recolher, com elevada probabilidade, vectores de acesso que cubram todo o universo de variáveis partilhadas existentes. Fazendo isto, evita-se a penalização no desempenho imposta pela necessidade de gravar todos os vectores de acesso em cada cliente.

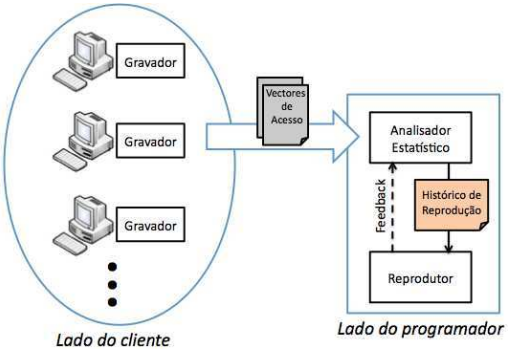


**Figura 1.** Visão geral da arquitectura proposta.

Posteriormente, cada cliente enviará os seus históricos parciais com os vectores de acesso gravados, juntamente com a informação adicional acerca do su-

cesso ou falha da execução, para ser analisado no lado do programador. Aqui, o *analisador estatístico* começa por associar um identificador único a cada vector de acesso, através da aplicação de uma função de dispersão (*hash*) sobre a informação gravada. Depois, executa a heurística (descrita na Secção 3.2) sobre os históricos parciais recebidos para gerar um *histórico de reprodução* que permita a reprodução do erro.

Por fim, o módulo *reprodutor* é responsável por controlar a re-execução do programa. Para isso, consulta o histórico de reprodução para forçar que a ordem de acessos às variáveis partilhadas seja a estabelecida no ficheiro. Infelizmente, nem todas as combinações de históricos parciais criam um histórico completo capaz de despoletar o erro. Como consequência, a tentativa de reprodução da falha pode não ter sucesso. Neste caso, o analisador estatístico é notificado e um novo histórico de reprodução é gerado. Este processo termina quando o erro se manifestar ou o limite máximo de tentativas tiver sido atingido.

### 3.2 Heurística de Combinação de Históricos Parciais

Devido aos eventos não-deterministas referidos anteriormente, raramente se obtêm duas execuções totalmente idênticas do programa, pelo que é necessário procurar históricos parciais cuja intersecção contenha informação o mais análoga possível (i.e. com vectores de acesso iguais). Para tal, a nossa heurística de análise estatística para combinação de históricos parciais utiliza quatro passos:

*1. Calcular o grau de semelhança entre os históricos parciais* - o primeiro passo consiste em calcular a *semelhança* de cada histórico parcial com os restantes históricos do universo de amostras. Sejam $h_1$ e $h_2$ dois históricos parciais, a sua semelhança é dada pela seguinte equação:

$$\text{Semelhança}(h_1, h_2) = \frac{VA_{iguais}(h_1, h_2)}{VA_{iguais}(h_1, h_2) + VA_{diferentes}(h_1, h_2)} \tag{1}$$

em que, para a intersecção dos conjuntos de variáveis gravadas por $h_1$ e por $h_2$, $VA_{iguais}(h_1, h_2)$ é o número de variáveis partilhadas que apresentam os mesmos vectores de acesso e $VA_{diferentes}(h_1, h_2)$ o número de variáveis partilhadas que apresentam vectores de acesso diferentes.

Com estes valores consegue-se identificar, para cada histórico parcial, os $k$ históricos com maior semelhança, criando-se assim vários subconjuntos de históricos complementares. De referir que no nosso protótipo considerámos $k \leq 5$, sendo que dois históricos serão assumidos como complementares se possuírem um índice de semelhança maior ou igual a 30%. Isto permite evitar que históricos pouco semelhantes façam parte do conjunto de complementares.

*2. Identificar um histórico parcial base* - o passo seguinte consiste em escolher o histórico parcial que servirá de base para se começar a construir o histórico completo. O histórico parcial base será aquele que apresentar maior *relevância*. Seja $h_1$ um histórico parcial e $Sem(h_1)$ o conjunto dos seus $k$ históricos mais semelhantes. A relevância de $h_1$ é dada por:

$$\begin{aligned} \text{Relevância}(h_1) = {} & 0.7 * \frac{Vars_{preenchidas}(h_1, Sem(h_1))}{Vars_{total}} \\ & + 0.3 * \frac{\sum_{n=1}^{k} \text{Semelhança}(h_1, hs_n)}{k}, hs_n \in Sem(h_1) \end{aligned} \tag{2}$$

sendo $Vars_{preenchidas}(h_1, Sem(h_1))$ o número de variáveis para o histórico completo que se conseguem preencher a partir da combinação do histórico parcial $h_1$ com os seus históricos complementares, e $Vars_{total}$ o número total de variáveis partilhadas do programa.

Como se pode verificar, a primeira parcela da equação tem um peso superior à segunda. Isto permite dar prioridade aos históricos parciais que, combinados com os seus complementares, consigam cobrir um maior número de variáveis partilhadas do programa, devendo por isso ser testados primeiro.

*3. Completar informação com históricos parciais "complementares por transitividade"* - mesmo juntando informação de históricos semelhantes, é possível que não se consiga gerar um histórico de reprodução completo. Para isso, tenta-se obter os vectores de acesso em falta a partir de históricos que, não pertencendo ao conjunto de semelhantes do histórico base, pertencem ao conjunto de semelhantes dos históricos que o são em relação ao histórico base. Isto é, se $h_2 \in Sem(h_1) \wedge h_3 \in Sem(h_2) \Rightarrow h_3 \in Sem_2(h_1)$, em que $Sem_n(h_1)$ contém os históricos complementares de $h_1$ de grau $n$ (neste exemplo, $h_3$ seria complementar de $h_1$ em 2º grau).

*4. Completar o histórico de reprodução* - se ainda assim não se conseguir completar um histórico de reprodução (os conjuntos de históricos semelhantes podem não ter elementos disjuntos entre si), complementa-se a informação através da aplicação de modelos estatísticos. Os vectores de acesso escolhidos neste passo devem ser aqueles que, não estando incluídos no conjunto de históricos parciais escolhidos nos passos anteriores, apresentam maior correlação com o erro. Segundo Liblit et al [14], isto significa que são igualmente *sensíveis* e *específicos*, ou seja, aparecem em muitas execuções falhadas e em poucas execuções com sucesso.

Desta forma, considere-se $F_{total}$ como sendo o número total de históricos parciais resultantes de execuções com erro. Para cada vector de acesso $v$ considere-se também os seguintes valores: $F(v)$ - número de execuções com erro em que $v$ foi observado, e $S(v)$ - número de execuções com sucesso em que $v$ foi observado. Usando estes valores, são calculadas as seguintes três métricas de classificação de um vector de acesso quanto à sua correlação com o erro:

$$\text{Sensibilidade}(v) = \frac{F(v)}{F_{total}} \tag{3}$$

$$\text{Especificidade}(v) = \frac{F(v)}{S(v) + F(v)} \tag{4}$$

$$\text{Importância}(v) = \frac{2}{\frac{1}{\text{Sensibilidade}(v)} + \frac{1}{\text{Especificidade}(v)}} \tag{5}$$

Assim, serão escolhidos os vectores de acesso que apresentarem maior valor de *importância* para preencher a informação em falta.

Como conclusão, pode afirmar-se que será tanto mais fácil gerar um histórico de reprodução capaz de despoletar o erro, quanto maior for o número de históricos parciais com uma relevância próxima de 1. Isto significa que existem históricos que combinados possuem vectores de acesso para praticamente todas as variáveis partilhadas do programa e apresentam grande grau de semelhança entre si.

### 3.3 Concretização

A nossa implementação do LEAP Cooperativo foi feita sobre o LEAP original[1], acrescentando-lhe as classes Java para gravação parcial e para a análise estatística antes da fase de reprodução. O LEAP usa a *framework Soot*[2] para localizar as variáveis partilhadas de um programa em formato *bytecode* de Java. Porém, como a localização destas variáveis é feita a partir de uma análise estática do programa, possui algumas limitações [11], tais como a incapacidade de distinguir posições independentes de um *array* partilhado.

## 4 Avaliação

Para avaliar a correcção da reprodução determinista da nossa solução, começámos por desenvolver uma *micro-bancada* que nos permitisse facilmente controlar o número de fios de execução e de variáveis partilhadas. Posteriormente, utilizaram-se alguns erros da bancada *ConTest* da IBM [18], que cobrem a maior parte dos tipos de erros de concorrência caracterizados no estudo feito por Lu et al. [5]. Finalmente, para fazer uma análise comparativa com o LEAP em termos de custos adicionais impostos à execução original, usámos a bancada *Java Grande Forum Benchmark Suite*[3]. Para os testes experimentais foi utilizada uma máquina de processador Intel Core 2 Duo de 2.26 Ghz, com 4GB de memória e sistema operativo Mac OS X.

### 4.1 Micro-bancada

A micro-bancada de teste consiste numa aplicação que simula transferências bancárias, cujo balanço final pode não ser correcto, devido à não existência de sincronização na execução concorrente dos fios de execução que actualizam as contas.

Configurou-se a micro-bancada de forma a ter 100 variáveis partilhadas, variando o número de fios de execução no intervalo [4,100] e a percentagem de variáveis gravadas nos seguintes patamares: 25%, 50% e 75%. Para cada caso, gravaram-se 300 históricos parciais de execuções onde o erro se manifestou. Posteriormente, adicionaram-se aos testes mais alguns históricos de execuções que tiveram sucesso, para aferir a influência desta informação na qualidade da heurística.

A Figura 2 d) apresenta os resultados para um máximo de 200 tentativas da heurística em reproduzir o erro (um número de tentativas igual a 200 significa que não se conseguiu reproduzir o erro). Por sua vez, na Figura 2 a) ilustra-se a percentagem de históricos parciais, do universo de amostras com 25% das variáveis gravadas, cujo valor de relevância (ver Equação 2): é inferior a 0.4, está entre 0.4 e 0.8, ou é maior do que 0.8. As Figuras 2 b) e 2 c) apresentam a mesma situação, mas para os modelos de gravação de 50% e 75%, respectivamente.

[1] Código disponível publicamente em http://sites.google.com/site/leaphkust
[2] http://www.sable.mcgill.ca/soot/
[3] http://www.epcc.ed.ac.uk/research/java-grande

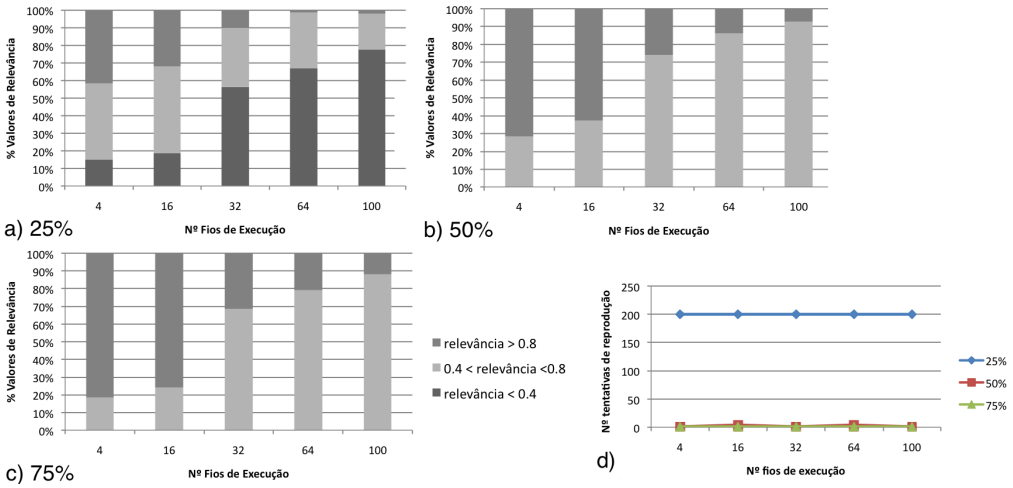


**Figura 2. a), b), c)** Percentagem de frequência dos valores de relevância para os três modelos de gravação. **d)** Número de tentativas necessárias para reproduzir o erro.

Como se pode ver na Figura 2 d), gravando apenas 25% das variáveis nunca se conseguiu reproduzir o erro, enquanto que com 50% e 75% ele foi sempre despoletado nas primeiras 5 tentativas. Isto significa que, para o primeiro caso, não havia nenhum subjconjunto de históricos complementares cuja combinação conseguisse cobrir o número total de variáveis partilhadas, sendo preciso completar o histórico de reprodução com informação potencialmente incompatível. De facto, observando os gráficos das percentagens, verifica-se que há uma diferença clara na proporção das gamas de relevância dos históricos. Por exemplo, para o modelo de gravação de 25% com 4 fios de execução, apenas 40% dos históricos possuíam relevância superior a 0.8 (sendo este valor superior 70% para os outros dois modelos de gravação).

Os gráficos mostram também que, nos três modelos de gravação, o aumento do número de fios de execução do programa traz uma diminuição da quantidade de históricos com relevância superior a 0.8. Isto deve-se ao facto de haver mais encadeamentos possíveis entre os fios de execução e, consequentemente, ser mais difícil obter duas execuções idênticas. Apesar disso, gravando 50% e 75% das variáveis partilhadas, a heurística conseguiu construir, quase sempre à primeira tentativa, um histórico completo que reproduzisse o erro. Pode-se concluir então que nestes casos se obteve sempre um subconjunto com históricos parciais semelhantes em número suficiente para gerar um histórico de reprodução completo.

Finalmente, convém referir que a adição de históricos de execuções sucedidas acabou por se revelar irrelevante. Como o erro se manifestava com bastantes execuções diferentes, havia sempre vários vectores de acesso com a mesma importância estatística para preencher a informação em falta.

### 4.2 Capacidade de Reprodução do Erro

Além dos testes na micro-bancada, aplicou-se a heurística a alguns programas da bancada *ConTest*, tal como se listam na Tabela 1. O resultados, apresentados na mesma tabela, incluem o número de iterações da heurística (considerando um máximo de 100 tentativas) que foram necessárias até se conseguir reproduzir o erro, para uma amostra de 100 clientes. De um modo geral, verifica-se que a heurística consegue reproduzir o erro quase sempre à primeira tentativa, com excepção dos casos em que se gravaram 25% e 50% das variáveis nos programas `Account` e `Manager` e 25% no programa `RandomNumbers`. Todos estes casos têm em comum o facto de não existir nenhum vector de acesso partilhado por históricos parciais diferentes, o que fez com que a heurística se resumisse à criação de históricos completos através de combinações de força bruta com os vectores de acesso que apresentavam maior índice de correlação com o erro. Esse cenário também aconteceu no programa `PingPong`, embora neste caso se tenha conseguido activar a falha à 18ª tentativa. Duas possíveis soluções para esta limitação seriam aumentar o tamanho da amostra de históricos parciais e/ou definir um subconjunto fixo de variáveis para serem sempre gravadas, o que aumentaria a probabilidade de existirem vectores de acesso em comum entre os históricos.

| **Nome do Erro** | **25%** | **50%** | **75%** | **Acessos Partilhados** | **Descrição do Erro** |
|---|---|---|---|---|---|
| PingPong | 18 | 18 | 1 | 502 | Não-atómico |
| RandomNumbers | X | 1 | 1 | 2417 | Bloqueio na secção crítica |
| Account | X | X | 1 | 12650 | *Lock* errado ou sem *lock* |
| LinkedList | 1 | 1 | 1 | 152 | Não-atómico |
| Critical | 1 | 1 | 1 | 219 | Não-atómico |
| Manager | X | X | 1 | 990 | Não-atómico |
| ProducerConsumer | 1 | 1 | 1 | 112 | Fio de execução orfão |

**Tabela 1.** Número de iterações da heurística para reproduzir o erro em programas da bancada IBM ConTest.

### 4.3 Penalização na Execução

A comparação entre a nossa solução e o LEAP baseou-se, essencialmente, em dois factores: tamanho dos históricos gravados e penalização de desempenho em tempo de execução. A Tabela 2 mostra esta análise comparativa, onde se nota que há benefícios claros em gravar históricos parciais.

De forma geral, verifica-se sempre uma diminuição dos custos adicionais de desempenho impostos comparando com o LEAP original, embora nos tamanhos dos históricos essa diminuição não seja relevante em alguns casos (*Series* e *SOR*, para 50% e 75%). O caso mais preponderante é o da aplicação *SparseMatmult*, onde se obteve uma uma redução da penalização de desempenho em relação ao

| Aplicação | Penalização no Desempenho | | | | Tamanho dos Históricos (KB) | | | |
|---|---|---|---|---|---|---|---|---|
| | **25%** | **50%** | **75%** | **LEAP** | **25%** | **50%** | **75%** | **LEAP** |
| Series | 0.1% | 0.2% | 2.3% | 6.5% | 4 | 4 | 4 | 4 |
| SOR | 1.1% | 2.0% | 2.4% | 2.7% | 4 | 16 | 16 | 16 |
| SparseMatmult | 5.6% | 25.6% | 2505.1% | 2606.7% | 4 | 4 | 401 | 406 |
| Raytracer | 9566.7% | 17452.1% | 44610.6% | 92908.4% | 7920 | 15800 | 33400 | 73700 |
| Montecarlo | 1.5% | 2.3% | 3.7% | 7.3% | 8 | 12 | 16 | 20 |

**Tabela 2.** Os custos adicionais impostos à execução original.

LEAP original de 465x e 102x para os casos de 25% e 50% de variáveis partilhadas gravadas, respectivamente. Para o mesmo programa, também foi possível gerar históricos 101x mais pequenos. Estas diminuições são explicadas pelo facto da maior parte dos acessos estarem confinados a um subgrupo pequeno de variáveis. Pelo que, se as variáveis que são acedidas mais vezes ficarem fora do subconjunto gravado, os custos adicionais serão automaticamente bastante inferiores. Uma alternativa possível para balancear a carga seria correr o programa instrumentado uma vez para perceber o número de vezes que cada variável é acedida e depois fazer nova instrumentação, escolhendo as variáveis para gravar tendo em conta a distribuição de carga.

No caso específico do *Raytracer*, verifica-se que ainda se incorre em penalizações significativas, devido ao elevado número de acessos às variáveis partilhadas. Porém, assiste-se igualmente a uma redução da penalização de desempenho em relação ao LEAP original de 9.7x, 5.3x e 2.1x para os casos de 25%, 50% e 75%, respectivamente. Esta tendência também é bastante visível no tamanho dos históricos gerados, onde se consegue reduzir em mais de metade a informação registada até mesmo gravando 75% das variáveis.

## 5 Trabalho Relacionado

A correcção de erros de software e a optimização do processo de depuração trazem inúmeros desafios, pelo que várias soluções têm sido apresentadas ao longo dos últimos anos. Nesta secção fazemos uma revisão geral de algumas delas, focando-nos nas de reprodução determinista e nas de depuração estatística.

Dentro das técnicas de reprodução determinista, o trabalho relacionado pode ser dividido em em duas categorias: as baseadas em hardware e as baseadas em software. As soluções baseadas em hardware pretendem mitigar as penalizações impostas pela gravação de informação através de extensões ao hardware. Como exemplos relevantes, tem-se os casos do FDR [6] e do BugNet [7] e, mais recentemente, do DeLorean [8]. Porém, estas abordagens requerem modificações significativas ao hardware que, além de serem caras, são complexas e não se encontram disponíveis actualmente, excepto em simulações.

O InstantReplay [9] foi o primeiro sistema de reprodução determinista baseado em software para multiprocessadores. Faz uso do protocolo *CREW* instrumentado para controlar e gravar os acessos a memória partilhada. O JaRec [10]

diminui as penalizações impostas pelo InstantReplay na medida em que assenta na ordem parcial dos fios de execução (em vez de ordem total), usando para isso *relógios de Lamport*. Porém, requer que o programa não tenha corridas de acesso a dados para garantir uma reprodução correcta, o que torna a solução pouco atractiva para a maioria das aplicações multiprocessador reais. Estas abordagens pretendem reproduzir o erro à primeira tentativa e, como tal, incorrem em elevados custos adicionais.

Algumas soluções recentes, como o PRES [4], o ODR [12] e o ESD [13], relaxam esta ideia, descurando uma reprodução determinista idêntica à execução original, a troco de menores custos de gravação. Para tal, aplicam heurísticas de inferência para completar a informação em falta.

A nossa solução também parte da observação de que não é fulcral conseguir reproduzir o erro à primeira tentativa, mas melhora estes sistemas na medida em que aproveita informação de múltiplos clientes para facilitar o processo de inferência, inspirando-se em técnicas de depuração estatística, como o CBI [14]. Neste último, são recolhidos relatórios de diferentes clientes com os valores gravados de certos predicados do programa (e.g. saltos condicionais, valores de retorno de funções, atribuições). Depois aplicam-se iterativamente modelos estatísticos que classificam os predicados de acordo com o seu grau correlação com o erro. Contudo, o CBI não consegue lidar com erros de concorrência. O CCI [15] ultrapassa esta dificuldade, ajustando os princípios do CBI para lidar com eventos não-deterministas, usando períodos de gravação mais longos, para ser possível gravar os múltiplos acessos a memória e os encadeamentos dos fios de execução.

## 6 Conclusão

Neste artigo apresentámos o LEAP Cooperativo - uma extensão ao LEAP [11] que tem como objectivo a reprodução determinista de erros de concorrência em software, através da combinação de múltiplos históricos parciais. Como cada utilizador grava apenas uma fracção das variáveis partilhadas do programa, consegue-se reduzir as penalizações impostas pela necessidade de registar todos os acessos a estas variáveis durante a execução original. Os resultados dos testes de desempenho que efectuamos com auxílio de uma bancada, mostraram que é possível reduzir os custos adicionais até 465x.

Apresentámos também uma heurística para combinação de históricos parciais, que demonstrou ser capaz de rapidamente gerar um histórico da execução completo capaz de reproduzir o erro, para os casos em que se gravavam 50% ou 75% das variáveis partilhadas do programa.

Como trabalho futuro, aponta-se a necessidade de estudar da viabilidade da heurística em aplicações reais de maior dimensão (e.g. *Tomcat*) e com períodos de execução mais longos. Também se pretende desenvolver novos métodos de definição da informação parcial a gravar, de modo a permitir a reprodução do erro mesmo com baixas percentagens de variáveis gravadas.

**Agradecimentos** Agradecemos a Jeff Huang pela imensa disponibilidade em esclarecer questões relativas ao LEAP. Este trabalho foi parcialmente suportado

pela FCT através do financiamento multianual do INESC-ID com fundos do Programa PIDDAC e através do projecto Europeu "FastFix"(FP7-ICT-2009-5).

## Referências

1. Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., Zhai, C.: Have things changed now?: an empirical study of bug characteristics in modern open source software. In: ASID '06. ASID, ACM (2006) 25–33
2. Hall, A.: Realising the benefits of formal methods. Journal of Universal Computer Science **13**(5) (2007) 669–678
3. Parnas, D.L.: Really rethinking 'formal methods'. Computer **43 (1)** (2010) 28–34
4. Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K.H., Lu, S.: Pres: probabilistic replay with execution sketching on multiprocessors. In: SOSP, ACM (2009) 177–192
5. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ASPLOS, ACM (2008) 329–339
6. Xu, M., Bodik, R., Hill, M.D.: A "flight data recorder"for enabling full-system multiprocessor deterministic replay. In: ISCA, ACM (2003) 122–135
7. Narayanasamy, S., Pokam, G., Calder, B.: Bugnet: Continuously recording program execution for deterministic replay debugging. In: ISCA, IEEE Computer Society (2005) 284–295
8. Pablo Montesinos, L.C., Torrellas, J.: Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In: ISCA, IEEE Computer Society (2008) 123–134
9. LeBlanc, T.J., Mellor-Crummey, J.M.: Debugging parallel programs with instant replay. IEEE Trans. Comput. **36** (April 1987) 471–482
10. Georges, A., Christiaens, M., Ronsse, M., De Bosschere, K.: Jarec: a portable record/replay environment for multi-threaded java applications. Softw. Pract. Exper. (May 2004) 523–547
11. Huang, J., Liu, P., Zhang, C.: Leap: lightweight deterministic multi-processor replay of concurrent java programs. In: FSE, ACM (2010) 385–386
12. Altekar, G., Stoica, I.: Odr: output-deterministic replay for multicore debugging. In: SOSP, ACM (2009) 193–206
13. Zamfir, C., Candea, G.: Execution synthesis: a technique for automated software debugging. In: EuroSys, ACM (2010) 321–334
14. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. SIGPLAN Not. **40** (June 2005) 15–26
15. Jin, G., Thakur, A., Liblit, B., Lu, S.: Instrumentation and sampling strategies for cooperative concurrency bug isolation. In: OOPSLA, ACM (2010) 241–255
16. Cornelis, F., Georges, A., Christiaens, M., Ronsse, M., Ghesquiere, T., Bosschere, K.D.: A taxonomy of execution replay systems. In: Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet. (2003)
17. Fonseca, P., Li, C., Singhal, V., Rodrigues, R.: A study of the internal and external effects of concurrency bugs. In: DSN, IEEE Computer Society (2010) 221–230
18. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: IPDPS, IEEE Computer Society (2003) 286–293