Técnicas e Arquitetura para Captura de Traços e Execução Especulativa

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por João Paulo Porto e aprovada pela Banca Examinadora.

Campinas, 15 Fevereiro de 2011.

Guido Araujo

Prof. Dr. Guido Costa Souza de Araujo
(Orientador)

Tese apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

Porto, João Paulo

P838t Técnicas e arquitetura para captura de traços e execução especulativa/João Paulo Porto-- Campinas, [S.P. : s.n.], 2011.

Orientador : Guido Costa Souza de Araujo.

Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1.Otimização. 2.Processamento paralelo (Computadores). 3.Compiladores (Programas de computador). I. Araujo, Guido Costa Souza de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Techniques and architecture for trace detection and speculative execution

Palavras-chave em inglês (Keywords): 1. Optimization. 2.Parallel processing (Electronic computers). 3.Compilers (Computer programs).

Área de concentração: Arquitetura de Computadores

Titulação: Doutor em Ciência da Computação

Banca examinadora: Prof. Dr. Guido Costa Souza de Araujo (IC – UNICAMP)
Prof. Dr. Philippe Olivier Alexandre Navaux (UFRGS)
Dr. Maurício Breternitz Jr (AMD)
Prof. Dr. Luis Eduardo Buzato (IC – UNICAMP)
Prof. Dr. Edson Borin (IC - UNICAMP)

Data da defesa: 15/02/2011

Programa de Pós-Graduação: Doutorado em Ciência da Computação

TERMO DE APROVAÇÃO

Tese Defendida e Aprovada em 15 de fevereiro de 2011, pela Banca examinadora composta pelos Professores Doutores:

[illegible]

Prof. Dr. Guido Costa Souza de Araújo
IC / UNICAMP

[illegible]

Prof. Dr. Philippe Olivier Alexandre Navaux
INF / UFRGS

[illegible]

Dr. Maurício Breternitz Junior
Advanced Micro Devices

[illegible]

Prof. Dr. Luiz Eduardo Buzato
IC / UNICAMP

Edson Borin

Prof. Dr. Edson Borin
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Técnicas e Arquitetura para Captura de Traços e Execução Especulativa

João Paulo Porto[1]

Fevereiro de 2011

[1]**Suporte financeiro da CAPES, Intel Corporation e Microsoft Research**

Resumo

É sabido que o modelo de desenvolvimento de micro-processadores baseado na extração de *Instruction-Level Parallelism* (ILP) de código sequencial atingiu seu limite. Encontrar soluções escaláveis e eficientes que permitam a manutenção de inúmeras instruções em execução simultaneamente tem se mostrado um desafio maior que o imaginado.

Neste sentido, arquitetos e micro-arquitetos de computadores vêm buscando soluções alternativas para o desenvolvimento de novas arquiteturas. Dentre as soluções existentes, vêm ganhando força as baseadas na extração de *Thread-Level Parallelism* (TLP). Resumidamente, TLP é um tipo de paralelismo que tenta quebrar um programa sequencial em *tarefas* relativamente independentes entre si para executá-las em paralelo.

TLP pode ser extraído por *hardware* ou *software*. Idealmente, uma solução híbrida deve ser utilizada, com o *software* realizando a identificação das oportunidades de extração de TLP, e com o *hardware* provendo suporte para execução do código gerado. Com tal solução de compromisso, o *hardware* fica livre da necessidade de especular, e o *software* pode trabalhar com maiores garantias.

Nesta Tese, estudaram-se formas automáticas de paralelização e extração de TLP. Inicialmente, focou-se em *traces* dinâmicos de execução de programas sequenciais. Técnicas existentes (tais como *MRET* e Trace Trees) mostraram-se inapropriadas, de modo que desenvolveu-se uma nova técnica chamada *Compact Trace Tree* (CTT), que mostrou-se mais rápida que *Trace Trees*. *Trace Tree* (TT) também apresentam grande nível de especialização de código (*tail duplication*), característica ausente em *MRET*.

Além de CTT, esta Tese apresenta *Trace Execution Automata* (TEA), um autômato que representa *traces* de execução. Esta representação revelou, em nossos experimentos, quase 80% de economia de espaço quando comparada com a representação usual.

A seguir, o foco da Tese foi voltado para laços de execução e para paralelização estática de código sequencial através de *Decoupled Software Pipeline* (DSWP). Nosso primeiro resultado nesta direção, usando Java, mostrou claramente que sem nenhum suporte em *hardware*, a paralelização estática de programas poderia atingir um ganho de desempenho médio de 48% nas aplicações paralelizadas.

Finalmente, a Tese propõe um modelo de execução paralelo baseado em DSWP que permite a consistência de dados entre as diversas *threads* de programas paralelizados. Apesar de não avaliar esta arquitetura completamente, os resultados iniciais são promissores. Além disso, o suporte necessário em *hardware* é simples e acomoda-se sobre o protocolo de coerência de *cache* existente, sem alterações sensíveis no processador.

Abstract

The usual, *Insturction-Level Parallelism* (ILP)-oriented, microprocessor development model is known to have reached a hard-to-break limit. Finding scalable and efficient solutions that keep several instructions on-the-fly simultaneously has proven to be more difficult than imagined.

In this sense, computer architects and micro-architects have been seeking alternatives to develop new architectures. Among all, the TLP-based solutions are gaining strength. In short, TLP strives to break a sequential program into quasi-independent tasks in order to execute them in parallel.

TLP can be extracted either by hardware or software. Ideally, a hybrid solution would be employed, with the software being responsible to identifying TLP opportunities, and the hardware offering support for the parallel code execution. With such solution, the hardware is free from the heavy speculation burden, whilst the software can be parallelized with more warranties.

In this Thesis, automatic parallelization and TLP strategies were studied. The research first focused on dynamic execution traces. Existing techniques, such as *MRET* and *Trace Trees* proved unsuitable for our goals, which led us to develop a new trace identification technique called *Compact Trace Trees*, which showed to be faster than *Trace Trees*. *Compact Trace Trees* also present trace specialization, which *MRET* lacks.

Besides *Compact Trace Trees*, this Thesis presents a new trace *representation* called *Trace Execution Automata*, an automaton representing the execution traces. This technique revealed nearly 80% memory size savings when compared to the usual, code duplication representation.

Next, the Thesis' focus shifted to parallelizing loops statically. Our initial result in this direction, using Java and *without* any hardware support, clearly revealed that static parallelization of sequential programs could reach a 48% average speedup when compared to their sequential execution.

Finally, a new, *Decoupled Software Pipelining*-based execution model with automatic data coherence amongst parallelized programś threads is proposed by the Thesis. Despite the lack of a full model evaluation, the initial results are promising. Differently from other

proposals, the hardware support necessary for this architecture is simple and builds upon the existing cache coherence protocol, without any modifications to this sensitive system component.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Algoritmos

Lista de Acrônimos

AFD	Autômatos Finitos Determinísticos	51
AMAS-BT	*Architectural and Micro-Architectural Support for Binary Translation*	29
CFC	Componente Fortemente Conexa	11
CFP	*Continual Flow Pipeline*	21
CMS	*Code Morphing Software*	20
CPR	*Checkpoint Processing and Recovery*	21
CTT	*Compact Trace Tree*	vii
DSMTx	*Distributed SMTx*	14
DSWP	*Decoupled Software Pipeline*	vii
E/S	Entrada e Saída	23
ECM	Eliminação de Código Morto	20
ESC	Eliminação de Subexpressões Comuns	20
GDP	Grafo de Dependências do Programa	11
GFC	Grafo de Fluxo de Controle	2
HTM	*Hardware Transactional Memory*	27
ICMC	*Inter-Core Memory Coherence*	25
ILP	*Instruction-Level Parallelism*	vii
ISCA	*International Symposium on Computer Architecture*	1
JVM	*Java Virtual Machine*	16
LEI	*Last Executed Iteration*	19
LLP	*Loop-Level Parallelism*	27
LNCS	*Lecture Notes in Computer Science*	51
MDT	*Memory Disambiguation Table*	24
MFET	*Most Frequently Executed Tail*	19
MRET	*Most Recently Executed Tail*	2
MRET^2	*Two-pass MRET*	20
MSI	*Modified – Shared – Invalid*	18
MSSP	*Master/Slave Speculative Parallelization*	28
MSI	*Modified – Shared – Invalid*	18
MESI	*Modified – Exclusive – Shared – Invalid*	27

MOSI	*Modified – Owner – Shared – Invalid*	137
NET	*Next Executed Tail*	2
PS-DSWP	*Parallel Stage DSWP*	13
RAW	*Read After Write*	23
SPAA	*Symposium on Parallelism in Algorithms and Architectures*	97
SMP	*Symmetric MultiProcessors*	23
Spec-DSWP	*Speculative DSWP*	11
SVC	*Speculative Versioning Cache*	23
SMT	*Symmetric Multi-Threading*	28
SMTx	*Software Multi-threaded Transactions*	14
TEA	*Trace Execution Automata*	vii
TLP	*Thread-Level Parallelism*	vii
TLS	*Thread-Level Speculation*	5
TLDS	*Thread-Level Data Speculation*	23
TT	*Trace Tree*	vii
SMT	*Symmetric Multi-Thread*	28
UCP	Unidade Central de Processamento	1
VLIW	*Very-Long Instruction Word*	27
WAR	*Write After Read*	23
WAW	*Write After Write*	23
WiDGET	*Wisconsin Decoupled Grid Execution Tiles*	28

Capítulo 1

Introdução

La Microarchitecure est Morte. Longue Vie à la Microarchitecture!.

A frase, título de um painel do *International Symposium on Computer Architecture* (ISCA) de 2010, resume, de maneira magistral, o presente momento para arquitetura e micro-arquitetura de computadores. A versão original (em francês *Le Roi est mort. Vive le Roi!* – O Rei morreu. Vivas ao Rei!) é utilizada quando da ascensão de um monarca, após o falecimento do monarca anterior, e denota a continuidade do governo.

É sabido que o desenvolvimento de processadores nos moldes empregados nas últimas décadas chegou ao ápice, e que novas técnicas serão necessárias para que novos *chips* sejam mais eficientes que seus predecessores, de modo que a *velha* micro-arquitetura morreu, e *novas* abordagens fazem-se necessárias.

Existem muitas proposta de novos modelos de execução (veja a Seção 2.2), bem como antigas idéias recicladas [19] para os novos tempos da micro-arquitetura. A existência de tal variedade de abordagens aponta claramente para um ponto de inflexão no desenvolvimento de processadores e, de certa forma, na maneira de se programar os novos *chips*.

Por um longo tempo, o foco no aumento de desempenho dos computadores foi a Unidade Central de Processamento (UCP). Manter mais de uma instrução em execução foi o subterfúgio mais utilizado pelos arquitetos de computadores, no início em processadores com *pipeline* em ordem [43] até os mais recentes processadores capazes de executar instruções fora de ordem [61]. Este modelo foi possível graças à evoluções micro-arquiteturais e avanços nos processos de fabricação que permitiram a manutenção da tendência histórica de dobrar a disponibilidade de transistores sem aumentar a área do *chip*, aumentando a frequência de operação destes transistores.

Entretanto, operar em altas frequências aumenta a dissipação de potência de um *chip*. Por exemplo, o processador Pentium ® IV (mono-núcleo) chegou a dissipar até 115W [29] (*prescott* @ 3.8 GHz). Assim, a Indústria adotou os processadores com múltiplos núcleos como solução para os problemas enfrentados pelo modelo anterior.

Neste novo modelo, para aproveitar todo o potencial de processamento disponibilizado pelas UCP, é necessária a programação explícita em múltiplas linhas de execução (*multi-thread programming*). Apesar dos esforços para popularizar e simplificar a programação *multi-threaded* [14, 53], a adoção deste novo modelo de programação tem sido lenta e mais difícil do que o originalmente imaginado.

Nas Seções seguintes resume-se os problemas de pesquisa encontrados e as contribuições feitas por esta Tese.

1.1 Paralelismo em *Traces*

Traces são utilizados como forma de detectar regiões quentes em programas. Usualmente, estas regiões consistem de algumas poucas instruções que são responsáveis por grande parte do tempo de execução de programas. Além de pequenos, *traces* usualmente contém apenas os caminhos efetivamente exercitados durante a execução.

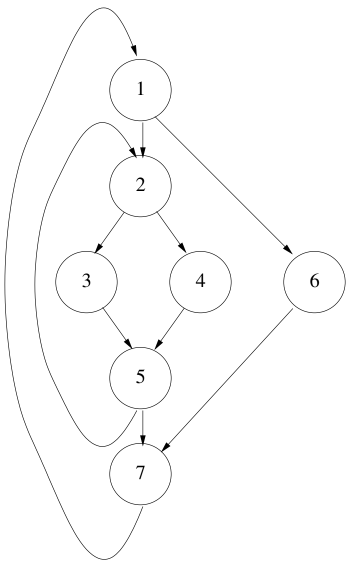
Por estas duas razões, *traces* são ótimos candidatos à otimização. Por serem usualmente pequenos, os algoritmos de fluxo de dados empregados por otimizadores executam rapidamente; por serem responsáveis por grande parte do tempo de execução, qualquer ganho oriundo da otimização tende a ser potencializado.

Antes, porém, de iniciar o estudo em oportunidades de paralelização em *traces*, estudaram-se técnicas de identificação dos mesmos. As técnicas *Most Recently Executed Tail* (MRET) [16] (também conhecido por *Next Executed Tail* (NET)) e TT [18], utilizadas durante a pesquisa, estão brevemente explicadas abaixo. Descrita abaixo também está a CTT [46], uma contribuição desta Tese. Veja a Seção 2.1 para outras técnicas.

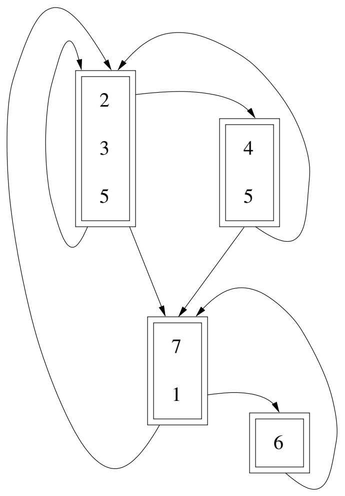
Possivelmente a técnica mais conhecida para identificação de *traces*, MRET é também a mais simples e a que apresenta o menor custo em tempo de execução. Para identificar *traces*, MRET instrumenta as instruções (ou blocos básicos) do programa que são alvos de saltos para trás (*i.e.*, saltos cujo alvo estão em instruções prévias no código do programa). MRET tende, desta forma, a instrumentar *loop headers*.

Assim que uma instrução é identificada como "quente" (*i.e.*, executa além de um limite determinado), todas as instruções executadas pelo programa devem ser marcadas até que (1) o programa execute uma instrução previamente marcada; ou (2) o número de instruções marcadas atinja um determinado limite. As instruções marcadas são consideradas um *trace* de execução.

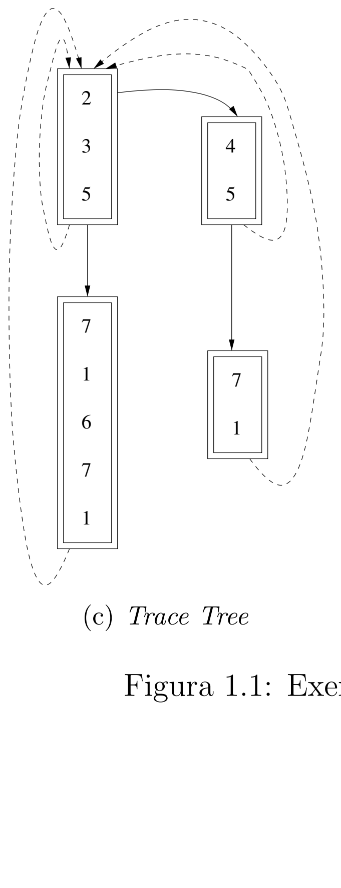
Por exemplo, o *trace* da Figura 1.1(b) representa os *traces* MRET decorrentes da execução do programa (com os retângulos representando sequências de instruções com uma única entrada e múltiplas saídas, *i.e.*, super blocos), cujo Grafo de Fluxo de Controle (GFC) está representado na Figura 1.1(a). Repare que os *traces* representados são capazes



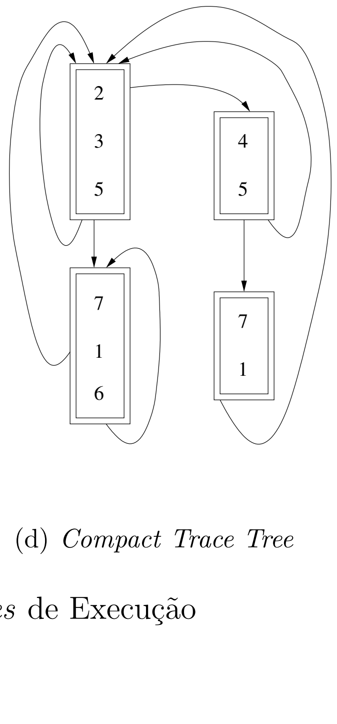
(a) CFG



(b) *Traces* MRET



(c) *Trace Tree*



(d) *Compact Trace Tree*

Figura 1.1: Exemplos de *Traces* de Execução

Tabela 1.1: Resumo das características de MRET, TT e CTT

Técnica	Especialização	Duplicação
MRET	Baixa	Mínima
TT	Altíssima	Altíssima
CTT	Alta	Média

de capturar qualquer execução do programa.

Entretanto, MRET falha ao não ser capaz de identificar *traces* especializados da execução do programa. Tal especialização é desejável para potencializar as possibilidades de otimização. Por exemplo, note que não é possível identificar com clareza se todas as saídas do laço interno (aresta $5 \rightarrow 7$ no GFC original) passam pela aresta $1 \rightarrow 6$. Os *traces* apenas *indicam* que tal aresta foi utilizada, não fornecendo informações adicionais (*e.g.*, se a última iteração do laço interno a executar for $2 \rightarrow 4 \rightarrow 5$, então a aresta $1 \rightarrow 6$ nunca será exercitada).

Gal *et al.* [18] propõem *Trace Tree* (TT). Diferentemente de MRET, TT identifica caminhos especializados da execução do programa. A forma de identificar instruções quentes em TT é idêntica a de MRET; a condição de *parada* de gravação de TT, contudo, é bem diferente.

TT marca as instruções executadas após a identificação de uma instrução quente até que o fluxo de execução do programa retorne à primeira instrução marcada. Assim, em TT as arestas de retorno (*back edges*) são implícitas, sendo por isso representada com arestas tracejadas nos *traces* da Figura 1.1(c), que ilustra uma possível TT para a execução do código na Figura 1.1(a).

Ao contrário de MRET, a TT representada não consegue capturar todos os fluxos de execução possíveis para o código original. Repare, em particular, que os ramos que representam o laço externo (super blocos $\boxed{7,1,6,7,1}$ e $\boxed{7,1}$) podem crescer ilimitadamente. Cientes destes problemas, os autores propõem heurísticas para limitar o crescimento da árvore. Entretanto, em nossos experimentos, as heurísticas propostas foram ineficientes.

1.1.1 Contribuições da Tese

CTT (veja o Capítulo 3 ou [46]) surgiu como uma técnica intermediária entre MRET e TT. Nosso objetivo inicial era obter *traces* mais especializados que MRET e mais compactos que TT. Para identificar o término de um *trace*, CTT identifica saltos para instruções que iniciem um *trace* no caminho que vai da raiz do *trace* (*i.e.*, o primeiro ramo identificado), até o *trace* sendo gravado. Isso permite que CTT identifique outros laços além do laço interno, permitindo uma representação mais compacta. A Tabela 1.1 resume as principais características de cada técnica.

A Figura 1.1(d) ilustra os *traces* que CTT gera. Note que a especialização de caminhos permite identificar instruções que nunca serão executadas dependendo do fluxo de execução do programa.

Outro resultado obtido por esta Tese foi o TEA (Capítulo 4 ou [45]), uma nova forma de representar *traces* que é independente da estratégia utilizada para identificá-los. TEA representa um *trace* t como um autômato. Para criar um TEA T a partir de um *trace* t existente, basta criar um estado E_i em T para cada instrução i de t, adicionando em T as transições $E_i \xrightarrow{j} E_j$ para todos os pares de instruções (i, j) de t tal que exista uma aresta de fluxo de controle $i \rightarrow j$.

TEA não duplica código, executando as instruções originais do programa. As Figuras 1.2 e 1.3 ilustram a TT da Figura 1.1(c) representada por um TEA. Na Figura 1.2, o programa está prestes a fazer a transição $5 \rightarrow 7$, com o estado atual do autômato (e a instrução correspondente à 5) em destaque; a Figura 1.3 mostra o programa após a transição.

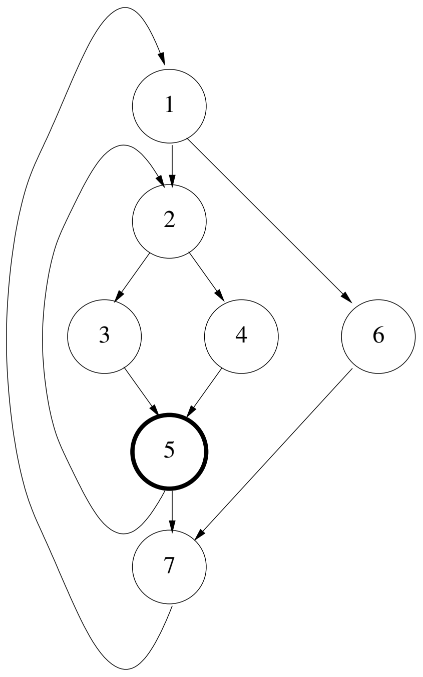
TEA pode ser atualizado dinamicamente para, por exemplo, indicar a detecção de um novo *trace*. Partindo de um autômato vazio (*i.e.*, um TEA somente com o estado "Código Frio") é possível adicionar os *traces* detectados durante a execução de um programa. Com isso, TEA pode ser utilizada como representação única em um ambiente de otimização dinâmico.

O *overhead* em gerenciar a detecção, criação e otimização de *traces* exclusivamente em *software* em tempo de execução mostrou-se elevado demais (veja os Capítulos 4 e 3 para maiores detalhes) para que ganhos oriundos de otimização viessem a sobrepujá-lo. Assim, o foco do estudo mudou para otimização estática de laços.

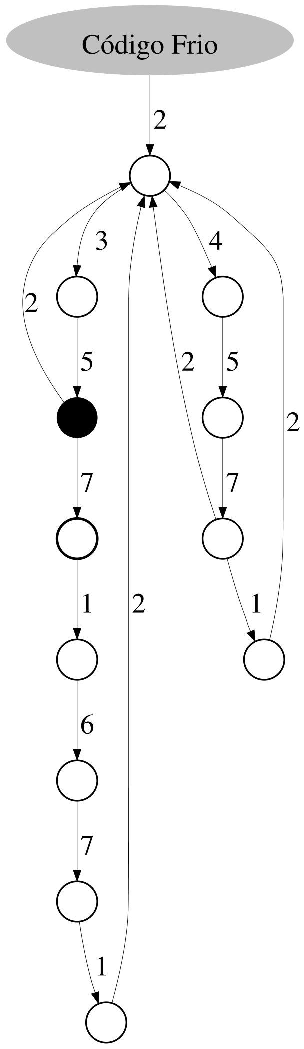
1.2 Paralelismo em Laços Usando *Decoupled Software Pipeline*

A otimização estática de laços regulares é um problema resolvido (veja a Seção 5.2). Entretanto, a paralelização de laços não regulares continua sendo um problema em aberto. Existe ampla disponibilidade bibliográfica sobre *hardwares* especializados com suporte a paralelização oferecendo, entre outros, mecanismos que garantem a coerência entre as diversas linhas de execução do programa paralelo. Estas soluções são usualmente chamadas de *Thread-Level Speculation* (TLS).

TLS é uma área da arquitetura de computadores que utiliza algum tipo de especulação (de dados, de controle ou ambos) para aumentar o desempenho de aplicações. Como a execução ocorre de forma *especulativa*, mecanismos para recuperação de falhas são necessários para garantir a corretude da execução paralela.

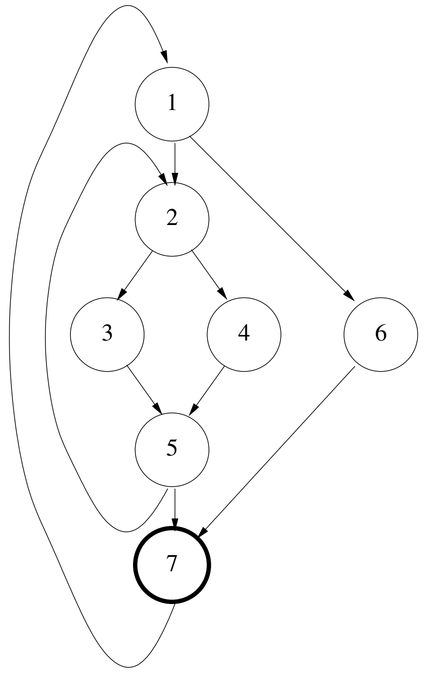


(a) GFC

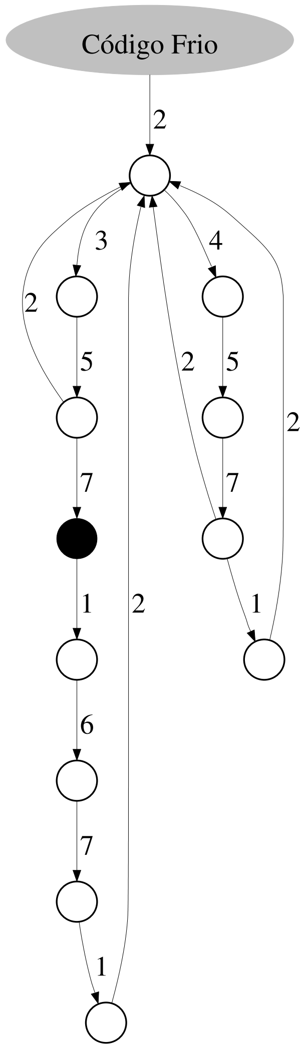


(b) TEA

Figura 1.2: Antes da Transição 5 → 7



(a) GFC



(b) TEA

Figura 1.3: Após a Transição 5 → 7

Em TLS esta Tese propõe um modelo de execução paralelo que, por sua regularidade, simplifica o suporte necessário para execução. Uma prova de corretude para tal modelo é apresentada no Apêndice A. Como o modelo de execução é baseado em DSWP, a Seção 1.2.2 descreve esta técnica. Por serem utilizadas para a explicação de DSWP, as técnicas tradicionais de paralelização DOALL e DOACROSS são ilustradas na próxima Seção.

1.2.1 Técnicas Tradicionais de Paralelização de Laços

Tradicionalmente, as técnicas DOALL e DOACROSS são utilizadas para paralelizar laços estaticamente. Das duas, DOALL é a mais simples: laços paralelizados com esta técnica precisam que suas iterações sejam independentes entre si (*i.e.*, não pode haver dependências *loop carried*). A Figura 1.4(a) ilustra um laço que pode ser paralelizado empregando-se DOALL. As arestas sólidas são arestas de fluxo de controle, e as arestas pontilhadas representam dependências de dados.

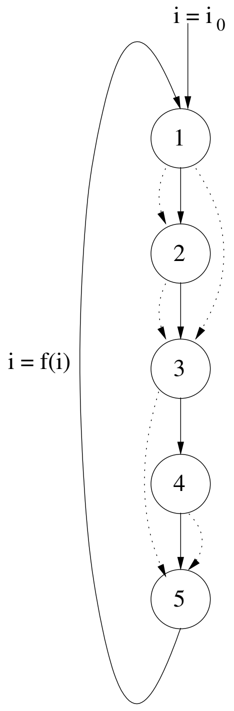
DOALL é uma técnica escalável com o número de núcleos: o programa paralelo pode identificar, em tempo de execução, quantos núcleos estão disponíveis para execução do laço, criando um número ótimo de *threads*.

DOACROSS é ligeiramente mais geral que DOALL: dependências entre iterações são permitidas, sendo comunicadas em tempo de execução. A idéia básica por trás de DOACROSS é gerar todas as dependências de iterações futuras o mais cedo possível, iniciando as novas iterações conforme suas dependências são satisfeitas. A Figura 1.5 ilustra um laço que não pode ser paralelizado com DOALL: repare a dependência de dados $3 \rightarrow 2$. Esta única aresta faz com que o laço da Figura 1.5(a) não possa ser tratado por DOALL, enquanto a Figura 1.4(a) possa.

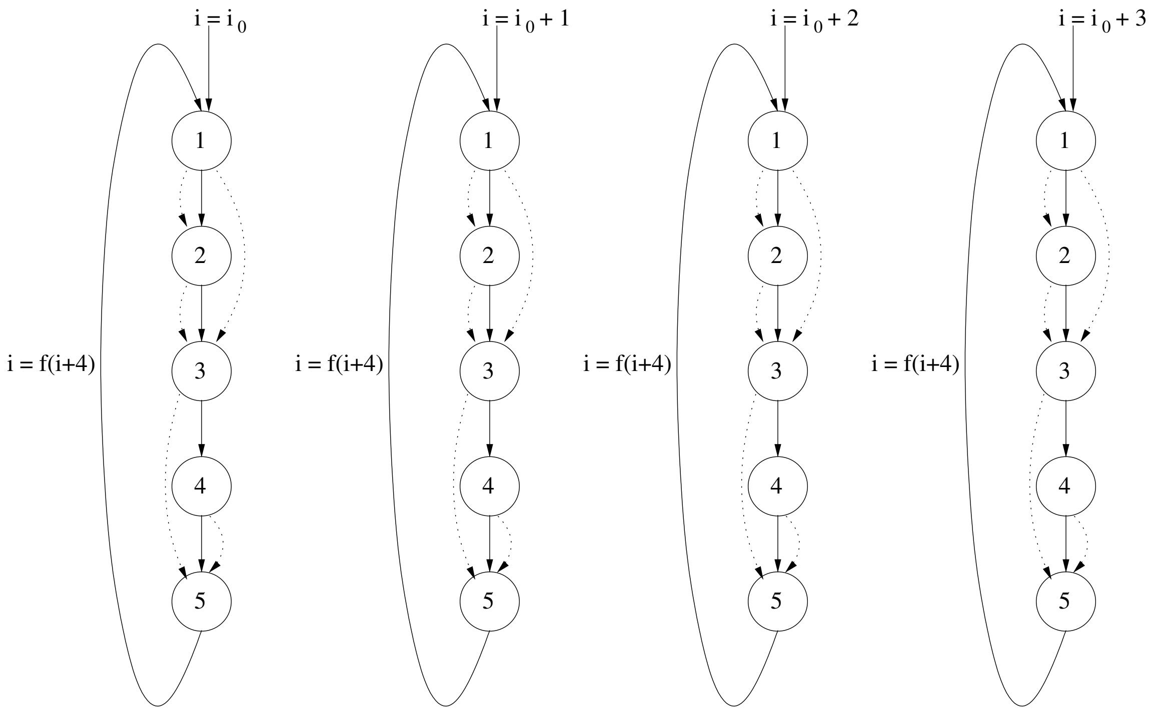
A Figura 1.5(b) ilustra a paralelização do laço apresentado na Figura 1.5(a). Diferentemente de DOALL, DOACROSS não é escalável com o número de núcleos: o código é gerado com um número pré-determinado de *threads* necessárias para sua execução. No exemplo da Figura, o número de *threads* assumido para paralelização é dois.

1.2.2 DSWP

DSWP foi inicialmente proposto por Ottoni *et al.* [41]. Neste trabalho, os autores introduzem a nova técnica de TLS, descrevendo os algoritmos utilizados para geração de código paralelo e o mecanismo de comunicação inter-*thread*. Além disso, os autores apresentam uma extensa avaliação experimental da técnica, incluindo uma avaliação de tamanho e latência de várias filas de comunicação. A avaliação foi feita utilizando-se um simulador do processador Itanium® 2 [44] e o compilador IMPACT [4].

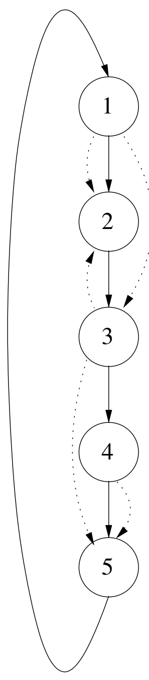


(a) Sequencial

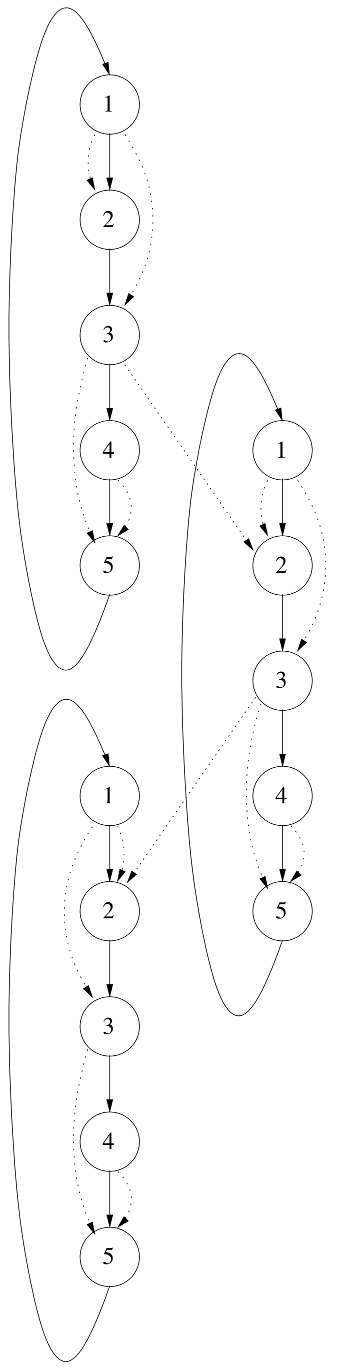


(b) Paralelo

Figura 1.4: Laço com Iterações Independentes e sua Paralelização com DOALL.



(a) Sequencial



(b) Paralelo

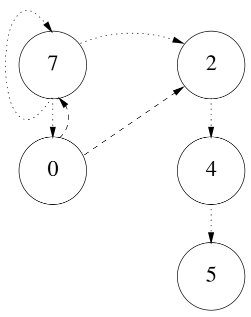
Figura 1.5: Laço Paralelizado com DOACROSS.

```
...
for ( node* n = theList; n != 0; n = n->next )
    n->sum += 1;
...
```

(a) Sequencial – MIPS

```
       ...
start: beq   $zero , $t1   , end (0)
       nop                       (1)
       lw    $t2   , ($t1)4      (2)
       addi  $t2   , $t2   , 1   (4)
       sw    ($t1)4, $t2         (5)
       j     start               (6)
       lw    $t1   , ($t1)8      (7)
end:   ...
```

(b) Sequencial – MIPS



(c) Grafo de Dependências do Programa

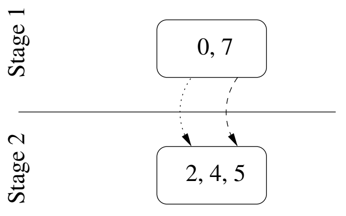
Figura 1.6: Atualização de Elemento de Lista Ligada

Como ilustração para DSWP, considere o laço apresentado na Figure 1.6(a), que percorre uma lista ligada somando 1 ao valor do campo *sum*. A Figura 1.6(b) mostra a codificação do laço em linguagem de montagem do processador MIPS já com os *delay slots* preenchidos. Os números entre parênteses indicam o ciclo em que as instruções entrariam no *pipeline*, e são usados para identificá-las nos exemplos.

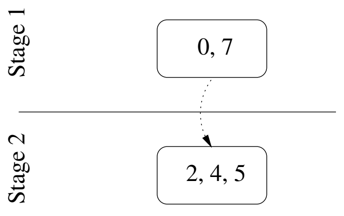
O Grafo de Dependências do Programa (GDP), apresentado na Figura 1.6(c), representa dependências de dados com arestas pontilhadas e dependências de controle com arestas tracejadas. No GDP, vemos a existência de uma Componente Fortemente Conexa (CFC) contendo a instrução 7. Ele mostra também a existência de duas cadeias de dependências de dados: uma entre as instruções 2, 4 e 5, e outra entre as instruções 7 e 0. Há, ainda, dependências de controle entre as instruções 0 e 2 e 0 e 7.

O laço paralelizado utilizando-se DSWP é apresentado na Figura 1.7(a). Como é possível ver na Figura, DSWP gera dois estágios para o laço do exemplo, sendo o primeiro responsável por caminhar na lista ligada e, o segundo, por atualizar os elementos da lista. Note a existência de duas arestas entre o primeiro e o segundo estágio. Estas dependências serão honradas em tempo de execução com as filas de comunicação entre os núcleos. Vale ressaltar que as filas devem ser suficientemente rápidas para que a comunicação de dados não inviabilize a técnica.

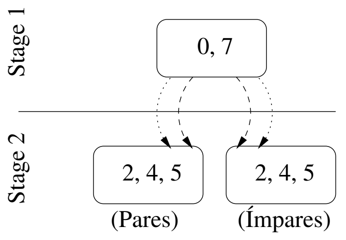
O próximo trabalho baseado em DSWP foi publicado por Vachharajani *et al.* [64] e é chamado *Speculative DSWP* (Spec-DSWP). Os autores observaram que, apesar do bom desempenho de DSWP, havia ainda desempenho que poderia ser extraído utilizando-se especulação em dependências *infrequentes* que são removidas antes do compilador otimizar



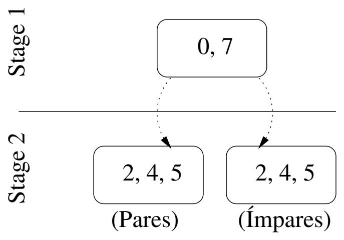
(a) DSWP



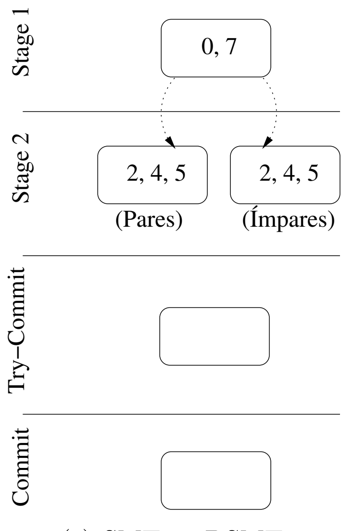
(b) specDSWP



(c) PS-DSWP



(d) PS-DSWP & specDSWP



(e) SMTx e DSMTx

Figura 1.7: Paralelização do Código da Figura 1.6

o código com DSWP. Nenhum suporte extra em *hardware* além da fila de comunicação é necessário para Spec-DSWP, de modo que o código gerado deve ser capaz de detectar e recuperar-se de falhas de especulação.

Continuando com o exemplo da lista ligada (Figura 1.6(b)), se a lista for suficientemente longa, a dependência de controle para a saída do laço ocorre infrequentemente. Desta forma, o compilador pode especulativamente remover a dependência, $0 \rightarrow 2$, conforme mostrado na Figura 1.7(b), de modo que a comunicação entre os estágios será, neste exemplo, diminuída. Obviamente, o compilador deverá inserir código de compensação devido à remoção da aresta de dependência.

O trabalho de Raman *et al.* [50] abandonou o suporte à especulação adicionado por [64] e utilizou outro subterfúgio para aumentar o desempenho de DSWP. *Após* o código paralelo ser gerado pelo compilador, a nova técnica, chamada *Parallel Stage DSWP* (PS-DSWP), aplica otimizações clássicas como DOALL [31] e DOACROSS [13] aos *estágios* gerados.

A Figura 1.7(c) mostra o resultado da aplicação da técnica de paralelização DOALL ao segundo estágio do programa paralelizado com DSWP. No exemplo, fazer esta otimização permite ao programa tolerar maiores latências no acesso à memória com o uso de uma *thread* extra. Note que a aresta de dependência de controle está presente.

Uma extensiva análise do desempenho de DSWP pode ser encontrada em [51]. Neste texto, os autores avaliam quais os fatores são possíveis gargalos para que DSWP gere código com o maior aumento de desempenho possível. Os autores avaliam diversas configurações de processadores (de dois até oito núcleos) e várias configurações distintas para a fila de comunicação. Eles introduzem uma nova variável nas análises: a banda disponível para o acesso à fila de comunicação. Os autores derivam expressões analíticas que podem ser utilizadas no momento da partição do programa em *threads* para determinar se o custo de comunicação é tolerável.

O trabalho de Huang *et al.* [28] é semelhante à união de [50] e [64], com a diferença que, neste novo trabalho, os autores aplicam outras técnicas de otimização aos *estágios* do *pipeline* de *threads* além de DOALL, a saber, LOCALWRITE [23] e SpecDOALL [52]. A avaliação experimental deste artigo foi feita, diferentemente dos anteriores, em um processador com oito núcleos comercial, logo, todo o suporte para as aplicações paralelizadas foi provido em *software*.

A união de Spec-DSWP e PS-DSWP é ilustrada na Figura 1.7(d). Note que o segundo estágio foi otimizado com DOALL e que a aresta de dependência de controle foi removida.

Todos os trabalhos envolvendo DSWP relacionados acima têm uma característica em comum: eles assumem a disponibilidade de informações completas sobre os acessos à memória realizados pelo código que será alvo da paralelização. Sem tal informação, o compilador não é capaz de particionar o código em estágios. Ainda pior, não há nenhum suporte, tanto em *hardware* quando em *software*, para detecção de falhas na atualização da memória do programa.

Software Multi-threaded Transactions (SMTx), proposto por Raman *et al.* [49], executa cada estágio do laço paralelizado em um processo distinto, diferentemente das propostas anteriores, que utilizavam *threads* para executar os estágios. Em SMTx, cada iteração do laço original é uma transação (uma *Tx*), sendo cada estágio gerado na paralelização chamado de *subTx*. Há, ainda, dois outros estágios especiais cujas funções serão explicadas nos parágrafos seguintes. Para um exemplo da arquitetura proposta por SMTx, veja a Figura 1.7(e).

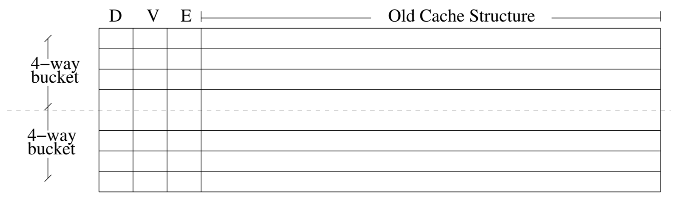
Inicialmente, todas as *subTx* possuem o mesmo mapeamento da memória virtual, sendo que as páginas estão protegidas contra escrita. No momento em que uma *subTx* s solicita uma escrita na página p, o sistema operacional cria uma nova página p' (cópia de p) para s. Assim, cada *subTx* executa isoladamente das outras. Caso o novo valor escrito seja necessário em uma *subTx* t diferente de s, SMTx propõe um sistema de comunicação de dependências entre *subTx*.

O código paralelo é também instrumentado de modo que todos os acessos à memória em todas as *subTx* sejam armazenados em um *log*, cujas entradas consistem de três campos: *Operação* (*load* ou *store*), *Endereço* e *Valor*.

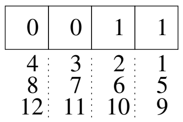
Quando todas as *subTxs* de uma *Tx* terminam de executar, seus *logs* são combinados para formar um único *log*. Uma *subTx* especial chamada *Try-Commit* recebe este *log* e o processa. Tal processamento consiste em *gravar* o conteúdo do campo *Valor* em *Endereço* (no caso de uma entrada do tipo *store* no *log*) ou verificar se o valor presente na memória em *Endereço* é idêntico à *Valor* (*loads*). A lógica por trás deste processamento é que o *log* contém os acessos à memória da mesma forma que eles seriam executados pelo programa sequencial original. O resultado do processamento é informado a última *subTx*, chamada *Commit*.

Commit utiliza o resultado do processamento do *log* de uma *Tx* para identificar se execução paralela da iteração que gerou o *log* foi bem-sucedida ou não. O *log* de uma *Tx* bem-sucedida é executado por *Commit* (*i.e.*, as entradas do tipo *store* no *log* são utilizadas para atualizar o conteúdo da memória) de modo a consolidar o estado do programa sequencial. Uma *Tx* mal-sucedida dispara a recuperação de falhas.

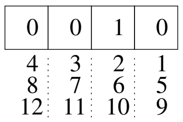
SMTx foi proposto e avaliado para ser utilizado em computadores com múltiplos processadores. Kim *et al.* [33] propõem *Distributed SMTx* (DSMTx), que é a generalização de SMTx para execução em *clusters* de computadores onde o acoplamento não seja tão forte. O custo de comunicação em tais ambientes é consideravelmente mais alto que em computadores multiprocessados, de modo que os autores avaliam formas de *buferização* de dados.



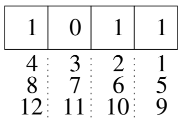
(a) Cache com os Campos Adicionados



(b) D – Dependências



(c) V– Iteração



(d) E – Dados Presentes

Figura 1.8: *Bits* Extra Utilizados para Consistência

1.2.3 Contribuições da Tese

As contribuições desta Tese em DSWP (Capítulo 5) foram feitas usando Java em processadores de prateleira. A fila de comunicação, indisponível nos ambientes de teste, foi simulada em *software* usando diversas estratégias de sincronização. Contudo, os resultados mostraram que as diferentes estratégias não influenciam diretamente o desempenho da aplicação. Em média, o aumento no desempenho das aplicações chegou à 48%. Os problemas encontrados durante estes experimentos foram abordados no próximo trabalho.

Grande parte dos problemas oriundos da paralelização de código sequencial emergem de acessos à memória: a dificuldade de determinar, incontestavelmente, se duas instruções que acessam a memória endereçam posições distintas limita as possibilidades de otimização. Além disso, a dificuldade em determinar se um conflito ocorreu (bem como recuperar o estado do programa na eventualidade de um conflito) orientaram o estudo do Capítulo 6.

Como resultado, propõe-se ali um esquema de consistência de memória, baseado em versionamento automático de dados nas *caches* dos processadores, e construído sobre o protocolo de coerência de *cache* existente, sem alterações no mesmo. Alguns *bits* extras são adicionados à *cache*, conforme ilustrado na Figura 1.8(a).

Os *bits* extra compõem três *tags*, cada uma com uma função distinta, cooperando para manter a consistência em programas paralelizados (ou, ao menos, detectando violações da semântica do programa sequencial). A lógica para gerenciar estas *tags* foi cuidadosamente elaborada para ser simples e eficiente.

Na arquitetura proposta, cada iteração de laços paralelizados é representada univocamente por um único *bit*. Para evitar a necessidade de grandes campos na *cache* (e nas mensagens de coerência), a arquitetura proposta limita a quantidade máxima de iterações ativas ao mesmo tempo. Além disso, utilizar um único *bit* simplifica a lógica necessária para o versionamento.

D é utilizado para manter as dependências de uma dada linha da *cache*. Na Figura 1.8(b) a existência de dois *bits* 1 indica que a linha l atrelada àquele campo é especulativa nas iterações correspondentes aos *bits* ligados. Quaisquer escritas a l nas iterações indicadas em D indica uma violação da semântica do programa sequencial.

V é utilizado para indicar a qual iteração a linha pertence. Se todos os *bits* deste campo forem zero, a linha atrelada a *tag* não é especulativa.

E é utilizado durante as operações do *commit* para evitar que a *cache* seja colocada em um estado inválido com múltiplas entradas para o mesmo endereço da memória. Colateralmente, E evita que a arquitetura proposta necessite de tráfego de rajada durante *commits*.

1.3 Principais Contribuições

De forma sucinta, as contribuições desta Tese são:

- Uma nova técnica de gravação de *traces* de execução de programas;
- Uma representação mais compacta para *traces*;
- Um algoritmo simples para paralelização de programas Java que, mesmo sem nenhum tipo de suporte específico da *Java Virtual Machine* (JVM), provê bons resultados;
- Uma arquitetura paralela com suporte à paralelização automática de laços sem a necessidade de *profiling*; e
- Uma prova de corretude para a arquitetura proposta sem a necessidade da criação de um protocolo de coerência específico.

1.4 Publicações

Durante a realização dos estudos desta Tese, foram publicados ou submetidos, em ordem cronológica, os seguintes trabalhos:

- J. P. Porto, Y. Wu, E. Borin e C. Wang. Compact Trace Trees in Dynamic Binary Optimization. Aplicação de patente no USPTO sob o número # 20100083236.
- J. P. Porto, G. Araujo, Y. Wu, E. Borin e C. Wang. Compact Trace Trees in Dynamic Binary Translators. Em *AMAS-BT 2009: 2nd Workshop on Architectural and MicroArchitectural Support for Binary Translation*. Junho de 2009. Austin, Texas, EUA.

- J. P. Porto, G. Araujo, E. Borin e Y. Wu. Trace Execution Automata in Dynamic Binary Translation. Em *AMAS-BT 2010: 3rd Workshop on Architectural and MicroArchitectural Support for Binary Translation.* Junho de 2010. St.-Malo, França.
- J. P. Porto, A. Oliveira, G. Araujo e M. Breternitz. Cache-Based Cross-Iteration Coherence for Loop Parallelization. Em *The 23rd ACM Symposium on Parallelism in Algorithms and Architectures.* Junho de 2011. San Jose, California, EUA. (*Submetido*)
- J. P. Porto e G. Araujo. Extending Decoupled Software Pipeline to Parallelize Java Programs. Em *Software Practice and Experience.* (*Submetido*)

1.5 Organização desta Tese

Antes de descrever a organização do texto, uma pequena nota ao leitor. Esta Tese, em concordância com as normas atuais que regem a pós-graduação da Universidade Estadual de Campinas, possui partes do texto escritas em língua estrangeira, correspondentes a artigos publicados ou submetidos a jornais científicos e conferências internacionais. Ainda de acordo com esta norma, o conteúdo desses textos reproduz fielmente aqueles dos artigos originais. Contudo, a formatação dos mesmos foi adequada para ficar compatível com o estilo do restante desse documento.

O Capítulo 2 lista trabalhos anteriores relacionados à esta Tese. Nele são detalhadas técnicas anteriores de TLS.

O Capítulo 3 inicia a coletânea com o primeiro artigo publicado durante a pesquisa deste trabalho. Nele, descreve-se uma nova técnica ara identificação de *traces*.

Durante os estudos sobre identificação de *traces*, desenvolvemos uma nova forma para representação dos mesmos, que é descrita no Capítulo 4.

O Capítulo 5 apresenta uma extensão de DSWP para utilização em nível de código-fonte em Java. Nos experimentos realizados, nenhum suporte extra foi adicionado à JVM. O trabalho de paralelização, embora tenha sido feito manualmente, foi de extrema importância para o resultado final desta Tese por ter exposto os problemas intrínsecos à DSWP, em especial os problemas de *alias* de memória. As soluções para os problemas encontrados neste estudo foram úteis para a arquitetura descrita no Capítulo 6.

O Capítulo 6 apresenta a principal contribuição desta tese: uma arquitetura paralela com suporte à execução de código paralelizado utilizando estratégias baseadas no modelo *DOPIPE* [42] de paralelização como DSWP.

As conclusões da Tese são apresentadas no Capítulo 7, que também lista trabalhos futuros possíveis.

O Apêndice A apresenta uma prova de corretude para o mecanismo de coerência utilizado pela arquitetura proposta pelo Capítulo 6. Esta prova utiliza protocolo *Modified – Shared – Invalid* (MSI) de coerência de *cache* sem alterações.

Capítulo 2

Trabalhos Relacionados

Neste Capítulo, é feita uma revisão dos trabalhos anteriores em *Traces* (Seção 2.1) e em paralelização não-tradicional de laços usando TLS (Seção 2.2). As técnicas de paralelização tradicionais, utilizadas para paralelização de laços regulares, são descritas na Seção 5.2.

2.1 *Traces*

Ao contrário de TLS, a literatura a respeito de *traces* não é tão extensa e, os trabalhos, não tão diversificados. Ainda assim, como uma importante parte deste trabalho, esta Seção dedica-se a listar alguns trabalhos relacionados detecção e usos de *traces*.

Cifuentes *et al.* [10] propõem *Most Frequently Executed Tail* (MFET). Nesta técnica, as *arestas* do GFC do programa são instrumentadas. MFET também instrumenta as instruções alvos de salto para trás. Quando uma instrução é identificada como "quente", MFET utiliza a informação sobre a frequência de execução das arestas para determinar o *trace* que deve ser formado.

Duesterwald *et al.* [16] propõem MRET. Assim como MFET, os inícios de *traces* são identificados em instruções que são alvo de saltos para trás. Entretanto, MRET segue o fluxo da execução do programa para determinar o *trace* a ser gravado. A idéia é evitar a custosa instrumentação em todas as arestas do GFC do programa original.

Last Executed Iteration (LEI) [26], proposto por Hiniker *et al.*, procura identificar código quente mais significativo que MRET. Basicamente, a técnica utiliza um *buffer* com os últimos saltos executados pelo programa. Uma entrada repetida neste *buffer* indica a detecção de um *loop header*. Quando este *loop header* executar mais que um número determinado de vezes, LEI grava a execução do programa até que um ciclo seja formado, ou até que a execução do programa atinja um *trace* pré-existente.

Como MRET não é seletivo em relação ao caminho escolhido para gerar um *trace*, *Two-pass MRET* (MRET^2)[2] foi desenvolvida. Como o nome indica, ela é derivada de MRET. Seu funcionamento é bem simples: ao invés de gravar um *trace* assim que seu código se torna quente, MRET^2 grava um *trace potencial*, e programa continua a executar código normal. Quando um segundo *trace* for identificado, MRET^2 faz a intersecção de ambos os *traces potenciais*. Esta heurística tende a identificar caminhos mais significativos para execução do programa.

Há uma ampla literatura de ambientes de otimização de código baseados em *traces*. Por exemplo, Baraz *et al.* [6] introduz IA32-EL, um tradutor binário dinâmico que traduz código *x86* para execução em sistemas Itanium ®. Este tradutor otimizante mostrou um desempenho superior à solução inicialmente empregada (*i.e.*, uso de um núcleo de um processador 386 para execução nativa de código x86). Este ganho de desempenho somente foi possível pois IA32-EL identificava *traces* de execução, armazenando-os em uma *trace cache* para posterior otimização.

Dehnert *et al.* [15] descreve *Code Morphing Software* (CMS), a grande novidade da Transmeta em seus processadores: ao invés de manter a compatibilidade com *x86* em *hardware*, uma camada de *software* (a CMS) foi utilizada. Desta forma, os engenheiros do processador se viram livres das idiossincrasias de *x86*. CMS também mantém os *traces* descobertos em uma *trace cache*, focando maior esforço de otimização nos *traces* mais executados.

Digital FX!32, proposto por Hookway [27], é um subsistema presente nos sistemas operacionais Windows NT 4.0 em sistemas Alpha para execução de aplicações *x86* sem a necessidade de recompilação para a nova arquitetura. Diferentemente dos trabalhos anteriores, FX!32 inicialmente *interpretava* as aplicações, instrumentando o código executado. A otimização dos códigos mais executados era feita *offline*. As unidades de otimização eram *traces* de execução.

Gal *et al.* [17] propõem TraceMonkey, um compilador dinâmico para JavaScript que emprega TT para identificar código quente em aplicações *Web*. Compilar JavaScript para execução nativa é extremamente interessante, haja vista a quantidade de código JavaScript disponível que pode beneficiar-se direta e automaticamente da compilação.

Em comum, todos estes trabalhos não identificam oportunidades de paralelização de código em nível de *threads*, optimizando, contudo, o código com otimizações tradicionais, tais quais Eliminação de Subexpressões Comuns (ESC) e Eliminação de Código Morto (ECM) [1, 39].

[2]USPTO #20070079293

2.2 *Thread-Level Speculation*

Laços irregulares requerem técnicas mais poderosas de paralelização. Além da informação estática sobre o código, estas técnicas requerem informações sobre a *execução* dos programas. Usualmente, tais informações são obtidas com o emprego de instrumentação. Entretanto, outros meios podem ser utilizados, tal como anotação do código-fonte. A seguir, uma seleção de trabalhos anteriores em TLS.

- Akkary *et al.* [2]: Uma interessante abordagem em *hardware* para TLS, que quebra um programa sequencial em *threads* automaticamente. Para isso, o *hardware* procura agressivamente por laços e chamadas de função que são usados para delimitar o início de uma nova *thread*. Como estas são criadas antes dos valores de entrada (*live-in*) estarem prontos, os autores propõe um esquema de predição de valores. Eles também provêm e explicam o suporte para falhas de especulação.

- *Checkpoint Processing and Recovery* (CPR) [3] e, posteriormente, *Continual Flow Pipeline* (CFP) [57], introduzem micro-arquiteturas que são capazes de terminar (em inglês, *retire*) instruções fora da ordem original do programa. Para tanto, os autores introduzem novas estruturas nos processadores. Ambos os modelos baseiam a execução no processamento de *checkpoints*. Estas micro-arquiteturas, no entanto, apresentam elevado consumo de energia, inviabilizando-as. Hilton *et al.* [25], propõe o uso do modelo de execução de processamento de *checkpoints* em processadores *in-order* como forma de amenizar o custo de instruções de acesso à memória.

- Balakrishnam *et al.* [5]: os autores propõem a multiplexação de um programa pelo compilador e a demultiplexação pelo processador. Neste esquema, o compilador determina, estaticamente, a relação de dependência entre métodos e funções de um programa e expõe esta informação para o processador. Para executar o programa, o *hardware* executa as funções que tem todas as suas dependências satisfeitas. A medida que as funções terminam, novas dependências são satisfeitas e, assim, novas funções estão aptas para serem executadas.

- Bhowmik *et al.* [7]: Este trabalho descreve um compilador (baseado no compilador SUIF [66]) que faz a decomposição de um programa sequencial em *threads*. Para tanto, o compilador utiliza heurísticas e informações obtidas com instrumentação. A avaliação experimental do compilador foi feita com um simulador que executa *traces* de execução e, aparentemente, nenhum suporte especial em *hardware* foi necessário para execução do código paralelo.

- Bridges *et al.* [8] discorre sobre a necessidade de especulação para que técnicas automáticas de paralelização consigam efetivamente extrair paralelismo de aplicações irregulares. Após uma pequena revisão de DSWP os autores apresentam duas anotações de código que estendem a semântica de estruturas sequenciais de linguagens de programação tradicionais. A primeira anotação (@YBRANCH) indica saltos condicionais (*i.e.*, *if*s em linguagens de programação) não-determinísticos. Esta anotação indica ao compilador que o código protegido pela condição (*i.e.*, o corpo do *if*) pode ser executado independente do resultado da condição. Esta anotação inclui também a probabilidade com que o corpo do *if* deve ser executado. A segunda anotação (@Commutative) é utilizada para anotar funções cuja ordem de invocação no programa paralelo não precise ser idêntica à ordem do programa sequencial (funções de alocação de memória são, em teoria, comutativas).

- Bulk [9, 62]: estes artigos descrevem uma maneira simples de verificar se um programa paralelizado viola a consistência sequencial durante a sua execução. Para detectar tais violações, os autores propõem um esquema baseado em assinaturas dos endereços de memória atualizados por cada uma das *threads*. Além das assinaturas, um esquema de *checkpoint* dos registradores é utilizado de modo a ser possível a restauração do estado correto da aplicação em resposta a detecção de conflitos. Nenhum suporte de *software* é proposto.

- Cintra *et al.* [11]: este trabalho propõe uma arquitetura escalável para *TLS*. Os autores criam seu próprio protocolo de coerência de cache para esta arquitetura. Além disso, as operações de *commit* e *squash* são potencialmente lentas: a primeira faz invalidação em rajada (*burst-invalidate*) de todos os dados quando do final de uma *thread*, enquanto a segunda força a sincronização dos processadores que executam a aplicação. Além disso, como os dados são invalidados ao término de uma *thread*, a *cache* do sistema ficará potencialmente fria.

- Collins *et al.* [12] propõe o uso de TLS para realização de pré-busca (*pre-fetching*) de *loads* "delinquentes". Um *load* é delinquente se é responsável por grande parte das faltas no acesso à *cache* (em inglês, *cache miss*). Em outras palavras, este *load* não é facilmente predito pelos mecanismos usuais de pré-busca. Para a avaliação, os autores implementaram um compilador que, identificando um *load* delinquente, emite código para a criação da *thread* responsável pela pré-busca dos dados. Algum suporte de *hardware* também é necessário. Assim, os autores utilizaram um simulador baseado em SMTSIM [63]. Os resultados mostram que eliminar os *loads* delinquentes é equivalente ao aumento de desempenho obtido pela utilização de memória penalidade de falha igual a zero (no original, *zero-miss-penalty memories*).

- Sohi *et al.* propõe um modelo de execução hierárquica, o Wisconsin Multiscalar [56]. Neste modelo, o fluxo dinâmico das instruções é partido em tarefas (*tasks*). Para isso, um *hardware* especializado prediz qual a próxima tarefa a ser executada e inicia sua execução em algum processador livre. Esta predição pode falhar, levando ao cancelamento de tarefas (*task squashing*) disparadas especulativamente. Neste modelo, a tarefa mais antiga (*i.e.*, criada antes de todas as outras tarefas executando no sistema) é dita não especulativa e sempre termina. Neste contexto, Gopal *et al.* propõe *Speculative Versioning Cache* (SVC) [20], que é um tipo de *cache* versionada coerente. Organizado de maneira similar às *caches* em processadores *Symmetric MultiProcessors* (SMP), SVC detecta acessos à memória que violem a semântica do programa sequencial. SVC depende de um mecanismo centralizado que faz a busca em uma lista de dados sempre que há uma falha de leitura à *cache* ou um acerto de leitura (em inglês, *cache hit*) a um dado não especulativo.

- Hydra CMP: Hammond *et al.* [21] descreve o processador SMP Hydra, que é composto por quatro processadores MIPS. Além da *cache* privativa, Hydra provê também uma grande *cache* compartilhada por todos os processadores. Fazendo a interligação dos processadores com a *cache* compartilhada e com o restante do sistema (*e.g.*, controlador de memória, controladores de Entrada e Saída (E/S) etc) existe um barramento largo o suficiente para acomodar um linha inteira de *cache*. Existe, ainda, um barramento mais estreito utilizado tanto para fazer atualizações no nível compartilhado da *cache* como para invalidar as cópias da linha sendo escrita nos outros processadores. A arquitetura Hydra também oferece suporte à TLS adicionando *hardware* especializados para tal, tanto nos núcleos como fora deles.

- *Thread-Level Data Speculation* (TLDS) é proposto por Steffan *et al.* [58] como um subterfúgio para aumentar a eficiência de TLS. Neste trabalho, os autores evitam anti-dependências (*Write After Read* (WAR) e *Write After Write* (WAW)) forçando os dados escritos especulativamente a residirem no primeiro nível da cache, o que limita a quantidade de dados escritos em modo especulativo. A detecção de conflitos *Read After Write* (RAW) é feita quando da escrita de um dado em modo especulativo simplesmente anexando (em inglês *piggybacking*) informações suficientes nas mensagens do protocolo de coerência. Um suporte para Hydra similar foi proposto por Hammond *et al.* [22] que adicionou a passagem (*forwarding*) automática de dados entre *threads* especulativas, além de algumas melhorias como a ausência de tráfego de rajada (*burst traffic*) ao término de *threads* especulativas. Apesar disso, o suporte descrito para Hydra requer um sofisticado suporte em *hardware*.

- Johnson *et al.* [30] propõe um algoritmo para o particionamento de um programa sequencial em *threads*. O algoritmo proposto opera em nível de blocos básicos, utilizando informações estáticas e dinâmicas, estas obtidas com instrumentação, tentando maximizar a quantidade de paralelismo que é exposta ao *hardware*, que deve obrigatoriamente oferecer suporte à especulação, utilizando uma série de parâmetros tais quais o tamanho das *threads* geradas, o desbalanceamento de carga (no original, *load imbalance*) e a possibilidade de gerar cancelamentos por causa do escalonamento.

- *Alchemist* [67] é um ambiente de instrumentação que identifica dinamicamente candidatos a pré-cômputo. Por exemplo, um comando *if* que execute após um comando *while* e não dependa deste pode ser executado em paralelo com o mesmo. Apesar de identificar os candidatos ao pré-cômputo, *Alchemist* não faz a paralelização automaticamente, ficando a cargo do programador acatar ou não a sugestão do ambiente. Um estudo mais detalhado de pré-cômputo é feito em [32] por Kim *et al.*. Neste trabalho, os autores analisam diversos algoritmos em diversos compiladores. Este trabalho requer um suporte minimo do *hardware* para a criação das *threads*, e não requer nenhum suporte para consistência.

- Krishnan *et al.* [34, 35] propõe neste artigo um sistema no qual o compilador é responsável por anotar o código binário com informações sobre o início e o fim de regiões as quais devem ser paralelizadas pelo *hardware* em tempo de execução. Diferentemente de grande parte dos trabalho que dependem de um compilador, neste os autores analisam o código binário, e não o código-fonte da aplicação, o que torna esta proposta uma alternativa para paralelização de código legado. O trabalho foca na paralelização de laços internos e o *hardware* é responsável por criar as *threads* para executar especulativamente iterações sucessivas de tais laços. As dependências de registradores são honradas pelo processador com o uso de um *hardware* específico chamado *synchronizing scoreboard*. As dependências dinâmicas são tratadas por uma tabela chamada *Memory Disambiguation Table* (MDT) que está localizada junto ao último nível da hierarquia de *cache*.

- Madrilles *et al.* propõe *Anaphase* [36] como outra solução para suporte à execução especulativa de *threads*. Este estudo focou no agrupamento de dois núcleos que podem operar de maneira independente ou cooperativa. Para o modo cooperativo, alguns *bits* são adicionados ao primeiro e ao segundo nível (que é compartilhado) da *cache*. Além deste suporte, o artigo introduz o módulo de coerência de memória entre núcleos (*Inter-Core Memory Coherence* (ICMC)). Este módulo é responsável por garantir que as atualizações à memória ocorram da mesma forma como ocorreriam se o programa fosse executado sequencialmente. O conjunto de dois núcleos, uma memória *cache* compartilhada e um ICMC formam um *tile*. Neste artigo os autores apresentam uma estimativa da área necessária para implementar o sistema proposto.

- Marcuello *et al.* [37]: outra alternativa que depende unicamente do *hardware* para paralelizar programas sequenciais. Como a maioria dos estudos de TLS, este foca em laços, que não necessariamente são os laços internos de um programa. Como outras abordagens exclusivamente em *hardware*, este trabalho faz especulação do fluxo de controle para geração das *threads*, bem como para as dependências de dados. Aquelas são feitas baseadas na observação de que a maior parte das iterações tende a seguir o mesmo caminho dentro de um laço, enquanto estas são tratadas com o uso de uma tabela para predição de valores.

- Marcuello *et al.* [38]: Neste artigo, os autores apresentam uma alternativa para a criação de *threads* para especulação. Os resultados experimentais mostram que o sistema proposto (baseado em instrumentação) é mais eficiente que as heurísticas tradicionais, que usualmente criam *threads* para executar iterações, continuação de laços ou de subrotinas. Uma outra avaliação feita mostra que os preditores de valor não precisam de esquemas elaborados caso o número de *threads* a serem criadas não seja grande.

- *Pinot* [40]: os autores propõem uma arquitetura para execução especulativa de *threads* que conecta os núcleos do processador utilizando uma conexão em anel. Este *design* foi escolhido pela simplicidade de implementação da rede em anel, bem como pela alta velocidade que ela proporciona para a transferência de dados entre núcleos. Esta eficiência na transferência de dados entre núcleos adjacentes é útil neste modelo de execução, no qual cada *thread* cria um única outra *thread* para adiantar a computação que seria executada na sua continuação. Também é descrito um suporte em *hardware* para versionamento de memória como forma de aumentar o suporte a várias granularidades de *TLP*. Para criar as *threads* antes de serem necessárias o *hardware* faz predição de fluxo de controle. O *hardware* não é responsável por extrair as *threads*: um otimizador *offline*, com base em informações de instrumentação, delimita as *threads* no código do programa.

- Prvulovic *et al.* [47] analisa as ineficiências de arquiteturas com suporte a execução especulativa de *threads* e determinam que o custo de fazer *commit*, o esgotamento de espaço para armazenar dados especulativos e o tráfego de coerência gerado pela especulação juntos dominam o custo de executar código especulativo. Para o primeiro gargalo é proposta uma solução que permite ao *commit* não escoar os dados especulativos para memória. Para o segundo gargalo (esgotamento de espaço para dados especulativos) é proposta uma solução que permite que uma região da memória principal seja utilizada como área de escape que pode ser redimensionada caso haja necessidade. Por fim, para contornar o excesso de mensagens de coerência devido à especulação, é proposta uma alteração no protocolo de coerência da *cache* que permita ao processador identificar linhas de cache que não precisam de coerência fina, diminuindo assim a quantidade de mensagens geradas pelo acesso a tais linhas.

- Compilador Mitosis (Quiñones *et al.*) [48]: quando da criação de uma *thread*, a maior parte das arquiteturas com suporte à TLS provê algum tipo de suporte em *hardware* para materialização dos valores iniciais dos registradores do processador que executará a nova *thread*. Neste artigo, os autores propõem que esta geração seja feita por *p-slices* – uma ou mais instruções que efetivamente calculam o valor dos registradores vivos. Este código de prólogo pode facilmente negar os benefícios oriundos da paralelização. Para evitar que isso ocorra, os autores propõem heurísticas para otimizar as *p-slices*. O *commit* ocorre quando a *thread* não-especulativa atinge o início de uma *thread* especulativa. Neste ponto, o estado não especulativo é comparado com o estado predito e, caso sejam idênticos, a *thread* não-especulativa termina e a *thread* especulativa torna-se não-especulativa. Caso os estados sejam distintos, a *thread* especulativa e todas as *threads* por ela geradas são canceladas e a *thread* não especulativa continua a execução do programa, criando novas *threads* quando necessário.

- Raman *et al.* [49] propõe SMTx como forma viabilizar DSWP em processadores atuais. Os autores propõem o uso do sistema de memória virtual dos processadores atuais como forma de garantir que, no evento de uma falha de especulação, seja possível restaurar o estado correto do programa sequencial. Entretanto, esta estratégia ainda depende de informações de instrumentação para funcionar corretamente: cabe ao compilador identificar as dependências entre os estágios do *pipeline*.

- Renau *et al.* [54]: TLS surgiu como uma alternativa aos processadores superescalares. A existência de múltiplos núcleos em um mesmo processador possibilita que técnicas agressivas de otimização sejam empregadas para aumentar o desempenho de aplicações. Entretanto, todo o suporte necessário para execução correta de TLS era visto como algo que poderia consumir uma grande quantidade de energia se implementado. Contrariando o esperado, os autores mostram que, com algumas otimizações em *software* e *hardware*, TLS pode ser eficiente em termos de consumo de energia. O estudo também aponta algumas ineficiências inerentes à estilos de projeto de sistemas com suporte à TLS para os arquitetos (e micro-arquitetos) de *hardware* evitem-los.

- Steffan *et al.* [59] faz uma proposta de *hardware* para TLS que diz ser escalável. O esquema proposto é um composto de processadores que estão interligados através de barramentos e de uma *cache* compartilhada. Apenas as iterações de laços são paralelizadas, de modo que a execução do código paralelo ocorre de forma semelhante à execução paralela de um laço DOALL, exceto que, neste caso, podem haver dependências entre as iterações, dependências estas que são honradas pelo *hardware* em tempo de execução. A coerência entre as iterações é mantida por um protocolo baseado em *Modified – Exclusive – Shared – Invalid* (MESI), mas com extensões para o suporte à especulação. Contrariando o dito em [47], esta proposta necessita do tráfego de rajada para fazer *commits* e *squashs*, de modo que estas operações tendem a ser um fator limitante muito severo no desempenho geral do sistema.

- O modelo *Copy or Discard* [60], proposto por Tian *et al.*, é um exemplo de técnica de TLS extremamente simples que não depende de nenhum suporte específico em *hardware* para funcionar. Ela trabalha paralelizando iterações de laços. A novidade neste esquema é o modo como a especulação termina. Como o nome da técnica sugere, terminar uma iteração é uma questão de copiar o estado especulativo de uma área privativa alocada para cada *thread* ou simplesmente descartar o estado especulativo de uma execução que falhe. Os autores apresentam técnicas para otimizar a cópia dos valores da área privativa para a área compartilhada.

- Voltron (Zhong *et al.* [68]): a proposta deste artigo não se limita à TLS. Os autores propõe uma arquitetura que seja capaz de capturar três formas distintas de paralelismo, a saber: paralelismo em nível de instrução (em inglês, ILP), paralelismo em nível de *thread* (TLP) e paralelismo em nível de laço (*Loop-Level Parallelism* (LLP)). Para o primeiro tipo de paralelismo (ILP), o sistema proposto faz a fusão de núcleos de modo a criar um *cluster* para execução semelhante ao encontrado em um processador *Very-Long Instruction Word* (VLIW). Para TLP e LLP o sistema oferece um modo de execução desacoplado e com suporte a *Hardware Transactional Memory* (HTM). Entretanto, esta abordagem apresenta dois problemas, sendo o primeiro deles a falta de suporte automático à coerência de dados (dependendo, assim, de um compilador mais elaborado) e o segundo oriundo do fato de o modelo de fusão de núcleos (também aplicado em *Wisconsin Decoupled Grid Execution Tiles* (WiDGET) [65]) já ter sofrido críticas quanto à sua eficiência [55]. Apesar destas observações, este é um dos poucos trabalhos a abordar de maneira tão abrangente os diversos tipos de paralelismo existente em aplicações sequenciais.

- Zilles *et al.* [69] observam que uma pequena parte das instruções estáticas de um programa tem um efeito negativo significativo no desempenho em tempo de execução das aplicações dada a dificuldade em antecipar, com as técnicas atuais de pré-busca ou predição de salto, seu comportamento dinâmico. Para contornar o problema, os autores propõe um esquema de pré-aquecer o preditor de saltos ou a *cache* para estas instruções problemáticas. Usando contextos ociosos de *threads* em processadores *Symmetric Multi-Thread* (SMT) com algumas alterações, os autores propõe a pré-execução de instruções problemáticas em paralelo com o programa principal. Esta *thread* auxiliar é responsável por tolerar a grande latência destas instruções.

- *Master/Slave Speculative Parallelization* (MSSP) é proposta por Zilles *et al.* [70]. Nesta abordagem, existe um processador mestre (*Master*) e vários processadores escravos (*Slaves*). O programa principal, executado pelo processador mestre, é, na verdade, uma aproximação do programa original que não necessariamente tem o mesmo resultado do programa original. Executar o programa original é responsabilidade dos processadores escravos. Assim, o programa aproximado cria *threads* verificadoras em pontos determinados para que estas confirmem o resultado da especulação. Esta abordagem possibilita a otimização agressiva do programa original, fazendo o caso comum extremamente eficiente. Como forma de garantir o progresso da execução do programa, o *hardware* pode, temporariamente, abandonar a especulação e executar a versão exata (e sequencial) do código.

Capítulo 3

Compact Trace Trees in Dynamic Binary Translators

Prólogo

Neste trabalho inicial, estudaram-se duas técnicas conhecidas utilizadas para detecção e gravação de *traces*. A primeira, conhecida como MRET ou NET [16], é uma técnica que faz *profiling* apenas de blocos básicos que são alvos de *back edges*. Neste esquema, quando um bloco torna-se quente (*i.e.*, executa mais que um determinado número de vezes) o *trace* é gravado acompanhado-se o fluxo natural do programa até que alguma condição de término seja detectada. A segunda técnica é TT [18]. Uma grande diferença entre TT e MRET é a restrição quanto ao término da gravação do *trace*: TT requer que um *trace* termine com um desvio para o início do *trace*. Além disso, a tomada de saídas laterais (*side exits*) em TT forçam a expansão do *trace*.

Por causa da expansão dos *traces*, TT apresenta potencial para um alto nível de duplicação de código (e, de fato, a duplicação observada foi alta) mas expõe mais oportunidades de otimização. MRET é bem comportada no quesito duplicação, mas os *traces* gerados são menos propensos à otimizações. Por causa disso, desenvolvemos uma técnica híbrida chamada CTT que apresenta um bom compromisso entre a alta especialização de TT e a baixa duplicação de MRET.

O seguinte artigo foi apresentado na segunda edição da *Architectural and Micro-Architectural Support for Binary Translation* (AMAS-BT) em junho de 2009 em Austin, Texas, Estados Unidos. Derivado deste trabalho, há um pedido de patente (US PTO# 20100083236, Compact Trace Trees in Dynamic Binary Optimization) pendente.

Abstract

Trace Tree (TT) is a technique to collect program execution traces, which is commonly used in JIT environments. Its main features are the ability to perform loop unrolling and function inlining at no cost, while detecting application loop kernels. In this paper we evaluate a TT implementation in a DBT environment. We show that, under DBT, trace trees suffer from severe code duplication, considerably degrading its performance. In order to take advantage of the TTs interesting features in DBTs, we propose a variation called *Compact Trace Trees* (CTTs), which we show to be faster and to reduce code duplication.

3.1 Introduction

Binary compatibility has been the standard for the microprocessor industry for several good reasons. First of all, maintaining binary compatibility implies the ability to run the existing (*legacy*) code. System programmers also benefit from backward compatibility, as previously acquired tool knowledge is still valid. For microprocessor architects, however, legacy compatibility also means that a lot of effort has to be put into newer chip versions, given that new products must run legacy code correctly, as if they were running in a previous chip set generation.

On the other hand, binary compatibility can also be implemented by using a special software layer. For example, in the Transmeta Crusoe [16] this is achieved by means of the *code morphing* layer. Intel has also explored software binary compatibility by using a tool called IA-32 EL [2], which allows x86 code to run on top of the Itanium processor.

As far as industry is concerned, dynamic binary translation, or simply DBT, is a viable way to keep existing code running on newer systems until new software comes up for the new processors. As a drawback, DBT adds a considerable amount of overhead to legacy code execution. In order to amortize runtime overhead and to enable legacy code to run faster on newer machines, the runtime system must be able to perform code optimizations. Several techniques can be used, but knowing what to optimize is a key feature in any DBT. Hot trace detection is a set of algorithms that try to find these so called *hot regions*.

In order to achieve a low optimization overhead, while producing a decent runtime performance, hot traces must capture the kernel of the running applications. Tying more execution time to less code often means that translation and optimization will be much faster, thus allowing for even higher speedups.

Several techniques can be used to detect hot regions, varying according to the overhead they add to the running time. One well-known technique is MRET (*Most Recently Executed Tail* [1, 8]), which combines backward edges (and super blocks' side exits) to target profiling. MRET adds little overhead, when compared to the basic block profiling needed, for example, by the *Most Frequently Executed Tail* [7]). MRET fails, however, at detecting application's kernel, as the generated traces look like a dynamic CFG for the running application.

Franz et al [12] described a technique called *trace tree* (TT). Their experimental results show that TTs are capable of finding kernel-like traces. Also, the way trace trees are created, functions tend to be inlined, loops unrolled, or even inverted. The inner loop, where the program might spend most of its execution time, is likely to be discovered sooner by the runtime environment. Optimizing this early-discovered traces is an excellent way to hide optimizations' overhead.

Unfortunately, trace trees can become potentially large due to its most notable characteristic: every path in a TT must end with a branch instruction to the entry point of the tree. To avoid this code explosion, long paths (i.e., paths longer than a maximum allowed threshold L) are discarded. Although it is possible to define L for a specific application, defining a general L requires a lot of effort. Also, defining L to be too small limits the effectiveness of TTs. In order to address these issues we propose at this paper the idea of *Compact Trace Trees* (CTTs), which is an extension of TTs which we show is less sensitive to L than TTs.

This article is organized as follow: Sections 3.2 details the basic concepts behind TTs. Section 3.3 describes our implementation of the TT technique in a DBT runtime environment. Section 3.4 presents *Compact Trace Tree*, our *relaxed* approach to the TT technique. Section 3.5 presents the experimental results and a comparative analysis of TTs and CTTs. Previous and related work is discussed in section 3.6. Finally, section 3.7 presents the conclusions.

3.2 Background on Trace Trees

Trace trees were first implemented on a Java Virtual Machine(JVM) for PowerPC [17], to support JIT compilation of an application most executed methods and loops. Algorithm 3.2.1 describes the rules for detecting trace trees. Before moving into the algorithm, some definitions are needed.

DEFINITION 1. *A trace tree T consists of several super blocks called tree-branches. The tree-branches are created as a set of the original program's basic block. The set of basic blocks in a Trace Tree T is denoted by $bbs(T)$.*

DEFINITION 2. *The first created tree-branch is called* ***root*** *of the TT, denoted by $root(T)$.*

DEFINITION 3. *The first basic block of the* ***root*** *of the TT is the* ***anchor****, denoted $anchor(T)$*

Algoritmo 3.2.1: Trace Tree Creation rules

```
    Input: BB: the next basic block to be executed by the program
    Input: T: the TT being created or expanded
 1  if $BB = anchor(T)$ then
 2  |  return Success
 3  else if current edge is a back edge then
 4  |  if there are too many back edges in T then
 5  |  |  return Abort
 6  |  else
 7  |  |  increment back edge count in T
 8  else if BB is root for any TT in the program then
 9  |  return Abort
10  else if adding BB to T makes it too big then
11  |  return Abort
12
13  add BB to T
14  return Continue
```

Algorithm 3.2.1, which is responsible for building the TT, is a very small automata, and is invoked whenever a branch instructions transfer the control flow from one basic block to another. It examines the current state (the parameter T) and decides whether it (1) continues the TT creation / expansion; (2) successfully terminates the process; or (3) aborts the process.

The first case happens when the algorithm reaches line 13. In that step, BB is added to T, and becomes part of the *tree-branch* being built. Line 14 indicates to the runtime environment that T is not yet completed, so the running program should continue to be executed in single-step (one basic block at a time) mode.

The second online case happens at line 2. If the process reaches that state, then the algorithm returns *Success* to the runtime environment to notify the successful creation or expansion of T, and that T is ready to be *linked*.

The last case (failure) happens when the algorithm reaches a state that makes it prohibitive to further record the trace. The algorithm then returns *Abort* to notify the runtime that T should be discarded. The issues mentioned in section 3.1 happens when the test on line 10 evaluates to true.

After a *Success*, T needs to be linked before it is ready to be used by the runtime system. During link-time, side exit stubs are created and unnecessary jumps are removed as shown in algorithm 3.2.2. The complete Trace Tree formation algorithm can be found in [12], and the interested reader is advised to read it for all the details.

Algoritmo 3.2.2: Trace Tree link rules

```
    Input: T: the TT being linked
1   foreach $BB$ in $bbs(T)$ do
2       if $BB$ branches to $anchor(T)$ then
3           link $BB$ in $T$ with the anchor
4       else if $BB$ branches to tree-branch $B$ of $T$ then
            // $B$ was created to expand $T$
5
6           link $BB$ with $B$
7       else
8           create side exit stub $S$
9           link $B$ with $S$
```

3.2.1 Sample Trace Tree Creation

Consider the sample code in Figure 3.1(a). It scans a linked list in a loop, counting the number of zero elements, and invoking functions *func1* and *func2* (not shown in the example) for non-zero elements.

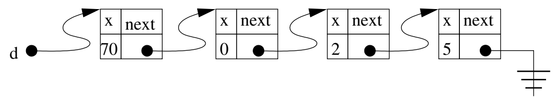
Considering the sample input in Figure 3.1(b), and assuming that the first element on the list is large enough to trigger the TT creation, a *tree-branch* is created, like the one shown in Figure 3.2(a). Notice that there is no back edge from block 15 to block 10, as, in TTs, the last instruction in a tree-branch jumps back to the first instruction in the tree, thus making back edges implicit.

After the creation of this first tree-branch, program execution continues from the first instruction on it. However, this time, the side exit at node 10 is taken. What happens now is somewhat particular to TTs: the structure is expanded. If the new tree-branch conforms with the TT formation rules, then the newly created super block will be created as a *private* tree-branch for the tree, and will end with a branch to the first instruction in the TT. By doing so, the technique is capable of providing more detailed runtime information about the paths taken by the application. In the example, after the first expansion, the resulting tree can be seen in Figure 3.2(b).

After the inner loop finishes the execution, the side exit at node 15 is taken. When this happens, the trace expansion module starts to record the new tree-branch. After this expansion, the resulting trace can be found in Figure 3.2(c). Notice that nodes 17, 4 and 5 appear twice in this new tree-branch. This happens because of the "jump to the header" condition: every TT tree-branch **must** end with a jump back to the trace header. As a result, TTs might grow indefinitely, until a control transfer instruction jumps back to the header. Thus, some trimming rules are required to limit the TT growth, such as

```
int func( data* d, int* sum ) {
   unsigned int i;
   int count = 0;
   while ( d ) {
     if ( (*d).x == 0 )
       ++count;
     else {
       i = (*d).x;
       do {
         if ( i % 2 )
           *sum = func1();
         else
           *sum = func2();
         --i;
       } while ( i ¿ 0 );
     }
     d = (*d).next;
   }
   return count;
}
```

(a) Sample C code



(b) The input data

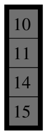
Figure 3.1: Sample function

the maximum trace height, size or number of taken back edges.

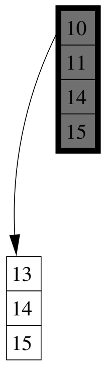
Finally, the function will process the last node in the linked list, and the resulting tree in figure 3.2(d) results.

3.3 Trace trees on DBT

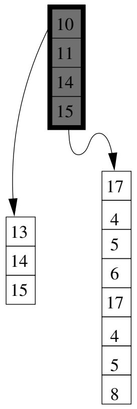
One of the goals of this paper is to evaluate TTs using the SPEC2000 suite [21] running in a native execution environment. To do that, we implemented TTs in a DBT research framework, called StarDBT [22] developed by Intel. This framework is capable of running the whole SPEC2000 suite, as well as *regular* applications like Mozilla Firefox. The following sections gives an overview of our implementation, as well as describes some of our design decisions.



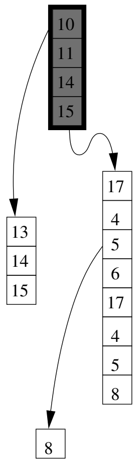
(a) The *root* branch



(b) After first expansion



(c) After exiting the inner loop



(d) A possible trace tree

Figure 3.2: Sample Trace Tree creation

3.3.1 Detecting Anchors

One interesting problem our implementation rose is how to detect *source file* anchors on an optimized binary. Since the compiler has the freedom to reorder basic blocks during code generation, we can not rely on backward branches to detect *source file* dominators. An obvious solution for this is to modify the compiler so basic blocks would not be reordered. Although this is a feasible, it is not practical, and could mask the results, as we intend to understand the technique behavior on native, fully optimized binaries.

Yet another solution would be to keep an array with the recently executed basic blocks. This solution would work fine if it did not raise two issues. First, the array needs to be traversed every time a control transfer instruction is executed. As the average basic block is small, the array would be scanned every five or so instructions, considerably degrading performance. The second issue is the array length. We have no means to predict how many basic blocks we need to keep in order to detect the loops. If the array is not big enough, and is used in a circular fashion, no anchor will eventually be found.

After evaluating these possibilities, we decided to use the backward branch heuristic. If a backward branch occurs in the compiled program, then it is likely to be part of a loop in the original source. Either that, or the profiled edge is not part of a loop, in which case it is unlikely that it would trigger a trace creation. Our experimental observations show that this approach works pretty well in practice.

3.3.2 Indirect Branches

On IA-32 architecture, indirect branches are hard to handle due to the processor's byte addressable memory, variable instruction size and no alignment requirement for the instructions. In other words, indirect jumps can jump anywhere into the processor's address space, so care must be taken when handling indirect branches on DBT systems.

Had the IA-32 ISA alignment restrictions or only fixed-size instructions, a table-based address translation could be used (as in [23]) to perform instruction address lookup on traces' indirect branches. However, this table would be considerably large, possibly requiring all (or even more) memory the computer can address.

To overcome this problem, our implementation keeps a list of possible targets for each indirect branch in the trace which was executed at least once. Indirect branches are generally not very common in the SPEC2000, with the exception of GCC and C++ programs, as indirect branches are used to implement virtual function calls. Thus our experiments did not show a severe impact on the translated binary performance. Needless to say, memory requirements were much lower and, most importantly, could be met.

Several techniques for handling indirect branches are described by Hisser et al. [15]. They perform several experiments regarding indirect branches and provide experimental evidence that there is no best technique. In our system, we use a local version of the *Indirect Branch Translation Cache* technique.

3.3.3 Trace Expansion

When dealing with the expansion of the trace, the variable size instruction on the IA-32 posed another obstacle to our implementation.

Every time a trace tree is linked, its side exits are compiled to branch to expansion stubs. These stubs set up data needed for the trace expansion before switching to the runtime environment that will actually start the expansion. Example of such data are pointers to the current executing trace, that are used to patch after expansion.

To overcome this, we forced branches on side exit to use the 32-bit branch instruction format. These instructions are generally 6 bytes long, as opposed to the 2-byte short format. By using the long format, we added some overhead to the processors' front-end (*decoding unit*), as well as increased L1 cache usage. We could not identify other simpler way to implement this feature and, since trace expansion is key to the TT technique, long branches were used.

3.3.4 Code Duplication Due to Path Specialization

The way TTs were designed, every time a side-exit is expanded, a new tree-branch is generated. This new tree-branch can be thought as an specialization since there will be no join points (as shown in figure 3.2(d)). In other words, tail duplication might lead to severe code duplication.

Unfortunately, as discussed in our experimental results (section 3.5.1), duplication was somewhat high. In order to address this issue, we could either choose to have a smaller dynamic coverage to keep the duplication (memory usage) low, or have a high duplication level, thus increasing the coverage.

Tail duplication is an important transformation that enables some optimizations and may expose parallelism [5, 19]. In order to still maintain tail duplication, and to avoid some of the duplication, we propose a modified, less strict set of trace formation rules for TTs. We named this relaxed form *Compact Trace Trees* (CTTs).

3.4 Compact Trace Trees on DBT

Compact trace trees were named *compact* as it is an attempt to have some interesting features of TTs, while keeping duplication low. Before explaining our TT modification, some definitions from TTs need to be relaxed.

Algoritmo 3.4.1: Compact Trace Tree Creation rules

```
Input: BB: the next basic block to be executed by the program
Input: B: the new tree-branch being created
Input: T: the CTT being created or expanded
if $BB \in bbs(B)$ then
    add $B$ to $T$
    return Success
else if $BB \in anchor(path(B))$ then
    add $B$ to $T$
    return Success
else if BB is root for any TT in the program then
    add $B$ to $T$
    return Success
else if adding BB to T makes it too big then
    discard $B$
    return Abort
else
    add $BB$ to $B$
    return Continue
```

DEFINITION 1. *A CTT consists of several super blocks called tree-branches. The tree-branches are created as a set of the original program's basic blocks. The set of basic blocks in a tree-branch B is denoted by bbs(B).*
DEFINITION 2. *The first created tree-branch is called **root** of the CTT.*
DEFINITION 3. *Every tree-branch's first basic block is an **anchor**. For a tree-branch B, anchor(B) denotes its anchor.*
DEFINITION 4. *Path is the tree-branch sequence from the current tree-branch B to the root, denoted by path(B).*

The rules to create the CTTs based on the relaxed definition of TTs are shown in Algorithm 3.4.1. When comparing it to the Algorithm 3.2.1, one can notice that there is now only one condition that might make the trace creation/expansion to abort. The relaxed *conditions* enable the implementation of a relaxed *automata*.

The *link-time* part of the implementation (algorithm 3.4.2) needed some small changes to work. Line 3 needs to scan through a list of tree-branches to search for the successor of all the branches it links.

Another interesting feature is found on line 6, where one CTT is linked with another CTT. Notice that this link only allows one CTT to jump to the *entry* of another CTT.

The modifications we implemented make impossible to infer back edges for the trees. In fact, a tree-branch might even end with a branch to a *cold-block*. This might happen

Algoritmo 3.4.2: Compact Trace Tree link rules

```
Input: T: the CTT being linked
1 foreach $B \in T$ do
2     foreach $BB \in B$ do
3         if $BB$ branches to an $A$ anchor(path(B)) then
4             link $BB$ with $A$
5         else if $BB$ branches to any CTT $C$ then
6             link $BB$ with $C$
7         else
8             create side exit stub $S$
9             link $BB$ with $S$
```

when the algorithm returns *Success* on line 3 of algorithm 3.4.1. Because of this feature, all control flow edges must be explicitly represented in CTTs, as seen in Figure 3.3.

3.4.1 Compact Trace Tree Creation Example

The first two CTT tree-branches are created the very same way as previously seen on Figures 3.2(a) and 3.2(b), so this step will not be shown this time. Figure 3.3(a) shows the initial CTT right before the side exit at node 15 is taken. The Figure is presented here to show the (now) non-implicit back edges.

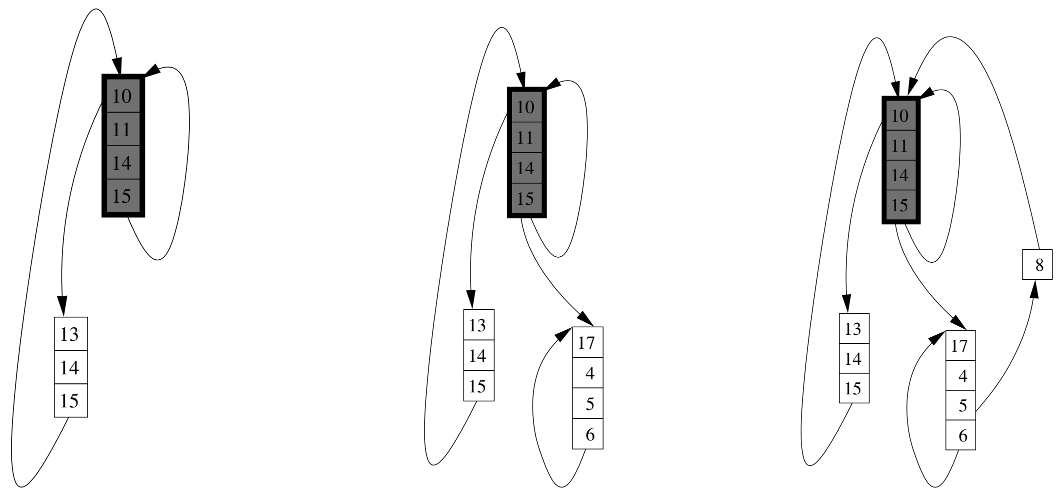
When node 15's side exit is taken, the trace expansion module starts recording a new branch for the CTT.This time, however, when the application jumps back to instruction 17, the trace expansion module will detect this as a trace recording stop condition, and will stop recording the branch. During link time, the jump to node 17 will be treated as a jump to an anchor, and the resulting CTT is shown in Figure 3.3(b).

Now, when instruction 5 takes the side exit to instruction 8, trace expansion is triggered again. The instruction that follows 8 is 10, which is the anchor of the root branch. This way, expansion will be completed and the resulting CTT is shown in Figure 3.3(c).

3.5 Experimental Results

In order to compare TTs and CTTs, we ran experiments using both techniques and different maximum tree heights values, where *tree height* is defined as the basic block count from the *root*'s *anchor* of TT or CTT up to the farthest block from it.

Also, we ran experiments with the MRET trace formation rules. For MRET traces, the maximum tree height served as a superblock size limit. Since MRET traces are not



(a) The *root* branch and the first expansion

(b) The second expansion

(c) The final CTT

Figure 3.3: Sample Compact Trace Tree creation

expanded, this parameter would have no other logical meaning.

The experiments were ran under Ubuntu 8.04.1 running on an Intel Q6600 2.4 GHz, with 4GB of RAM 1066MHz and two 500GB SATA disks. We used the 32 bit Linux distribution since our DBT is targeted at IA32. The whole SPEC2000 suite was used, under the reference input, and all the binaries were statically compiled, with optimization level 2.

The parameter (tree height) ranged from twenty blocks to a hundred and twenty, in twenty blocks steps. We also ran a special experiment with maximum tree height of five blocks. Table 3.1 shows how dynamic coverage changes with this parameter. It also shows that TTs and CTTs are sensitive to it. As expected, the greater the trace height, the better the coverage.

Interesting enough, notice that MRET traces are not sensitive to the height. MRET gets the higher coverage numbers of all techniques no matter the tree height is used (or block count for MRET super blocks). As anticipated, TT shows the expected behavior of increasing coverage as tree height becomes higher. It also exhibits an asymptotic behavior around 80%. For CTT, coverage seems slightly better, as the asymptotic behavior shows up around 86%.

Time spent within the runtime is shown in table 3.2. Again, MRET shows no sensitivity with respect to the trace height. CTT seems to be stabilizing as the maximum trace height is increased, which is good, since a large height would not affect the technique much. TT however, seems to be suffering with the higher trees. A quick notice, regarding the last two values for CTT on this table: they are smaller than the others since the last

Block count	CTT(%)	MRET(%)	TT(%)
5	50.53	97.70	48.79
20	74.72	97.59	67.42
40	81.97	97.62	74.32
60	82.78	97.63	76.13
80	85.05	97.60	76.37
100	85.56	97.60	76.94
120	85.64	97.60	77.88

Table 3.1: Average coverage for all experiments

Block count	CTT	MRET	TT
5	105	68	97
20	96	68	96
40	117	68	115
60	137	68	162
80	137	68	264
100	141	68	225
120	145	68	242

Table 3.2: Total time on the framework (minutes)

instance of bzip2 could not finish. Had it finished, its runtime would be higher.

Table 3.3 shows the duplication numbers for our instances. As shown, our technique fulfilled our primary goal to decrease duplication. Again, the last two entries for TT lack the last bzip2 instances, being slightly different than what was expected.

Finally, table 3.4 indicates the memory usage for a complete run of the SPEC2000 suite. It is noticeable how memory usage increases for CTT and TT, as a higher trace height was used. Again, the MRET implementation is independent of the parameter.

3.5.1 TT on DBT

Trace trees did not perform as well in DBT as it did on the Java Virtual Machine. The technique seems to be really dependent upon a good estimate of trace height to achieve good results. Selecting a small trace height value is bad, since it might lead the technique to under-perform. For example, in Figure 3.4(a) the technique performed well until the last three experiments. The only parameter that changed was the maximum trace height.

In the same benchmark, as shown Figure 3.4(b), duplication became an issue from the fourth experiment on, and, if duplication rises, so does memory usage. Notice the poor performance on Figure 3.4(a). Since duplication increases exponentially, the runtime

Block count	CTT	MRET	TT
5	0.08	0.42	0.06
20	0.62	0.52	0.33
40	2.34	0.54	5.83
60	4.43	0.54	28.36
80	5.37	0.54	73.75
100	6.75	0.54	69.21
120	6.85	0.54	76.19

Table 3.3: Average duplication

Block count	CTT	MRET	TT
5	939	980	939
20	989	989	963
40	1130	990	1205
60	1311	990	2105
80	1402	990	3498
100	1535	990	3407
120	1549	990	3700

Table 3.4: Total memory usage (in MB)

system takes much longer to create the TT.

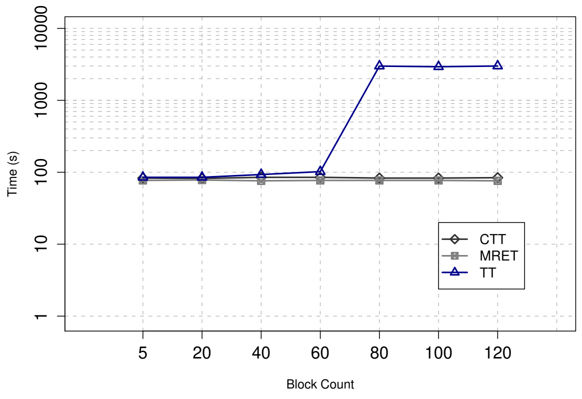
The bzip2 benchmark also suffered with bad performance. As shown in Figure 3.5, TT was not capable of finding loop kernels, and this led to the code explosion seen in Figure 3.5(b).

Also, the technique suffered from the highest duplication (see table 3.3). Specifically bzip2 had problems with the last two instances of the last two experiments: system memory was exhausted. Since each experiment is composed of 3 instances, we only show the data for the instance that finished the experiments.

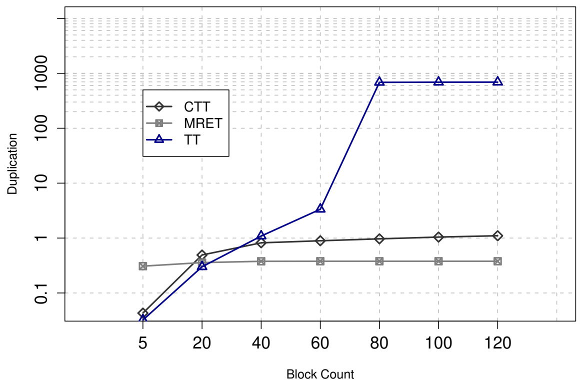
The problems faced here by TT do not invalidate it as a trace technique. They just show some aspects that need to be addressed by system writers if they want to use TT.

3.5.2 CTT on DBT

CTT successfully addresses the issues found with TT in our system. As tables 3.1, 3.2 and 3.3 show, CTT's coverage was higher than TT's, duplication was smaller, as was execution time. It is interesting to notice that CTTs are less sensitive to a poor trace height parameter, at least with respect to the execution time. Table 3.4 shows that CTT consumed less memory than TT.



(a) Time



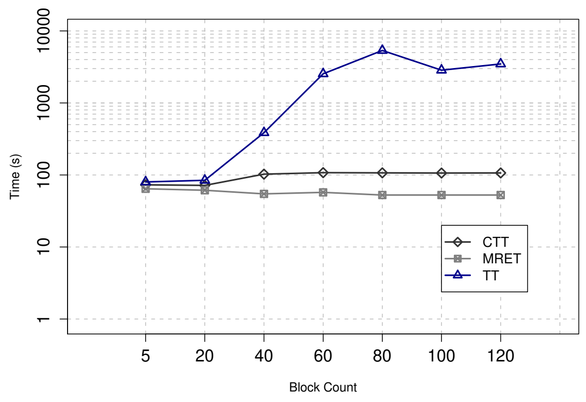
(b) Duplication

Figure 3.4: 181.mcf – SPEC2000 int

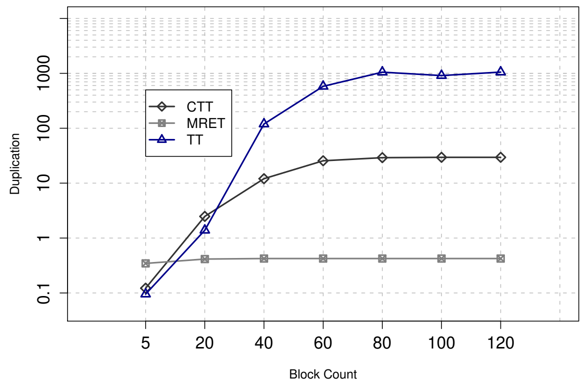
Figure 3.4 shows that, with a high enough trace height parameter, CTT can behave much like MRET. Particularly, Figure 3.4(a) shows that CTT was as fast as MRET. However, Figure 3.5(b) shows that CTT can be very different from MRET, regarding duplication. Overall, the technique is competitive (figure 3.5(a)) with MRET.

3.5.3 MRET, TT or CTT?

The three trace techniques are valid trace techniques that have their own advantages as well as disadvantages. MRET is, by far, the fastest algorithm. Also, as expected, it is not sensitive to trace size parameter, making it the ideal choice when there is no available time for calibration. Unfortunately, it lacks tail duplication, and runtime analysis of the



(a) Time



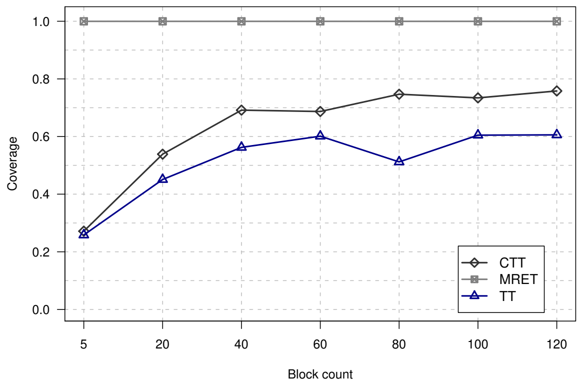
(b) Duplication

Figure 3.5: 256.bzip2 – SPEC2000 int

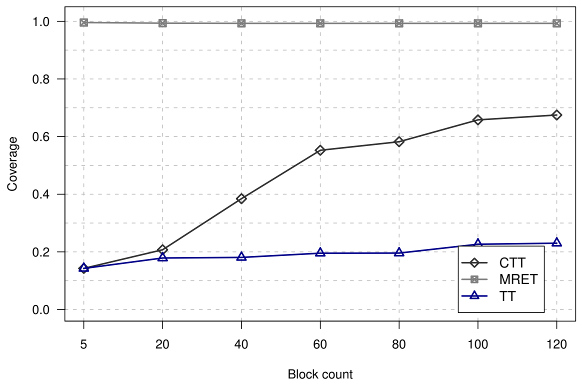
generated dynamic CFG might be slow.

TTs tend to perform poorly on DBT systems. We observed that the technique is slow and is not capable of detecting loop kernels (or even loops) in some applications. This is unexpected as the previous results [12] were good, and might be explained by the fact that an IA-32 DBT presents a much more challenging environment than a Java Virtual Machine. TTs also suffers a performance hit when the tree height is too big.

CTTs is a good choice for DBTs. It provides an efficient algorithm runtime with good coverage results, while keeping duplication at reasonable levels. Our experiments show that CTTs will still perform well even if the selected tree height is larger than necessary, thus eliminating the need to find its best value.



(a) 197.parser – SPEC2000 int



(b) 255.vortex – SPEC2000 int

Figure 3.6: Interesting results

3.5.4 Interesting Results

Although all the techniques behaved consistently, some interesting results were found. Two interesting results were selected and are show in Figure 3.6. They are briefly discussed bellow.

Figure 3.6(a) shows that coverage may drop even if we increase the maximum trace height. This result is counter-intuitive, since it is expected that, as the maximum allowed trace height is increased, coverage increases as well. This may be happening because, with a higher tree, the runtime environment keeps trying to create a long tree branch (which ends up being aborted), while the runtime with the more strict rules gives up on the long trace and end up detecting a smaller, more important trace. It is curious to notice that both CTT and TT suffered the same problem, but at different tree height values.

Figure 3.6(b) shows two other interesting aspects. First, it looks like CTT coverage would still increase as trace height increases. In other words, it seems that the loops in vortex are big. In other words, CTT needs a big tree height to be able to capture the whole loops.

Regarding the same benchmark, it is interesting to notice that, albeit being very similar when compared to CTT, TT coverage did not increase much with the parameter change. Here, the very restrictive *branch to the anchor rule* (line 7 of algorithm 3.2.1) prevent the algorithm to find the loops.

Also, for both benchmarks shown in Figure 3.6, MRET performs perfectly, as it does on most of benchmarks from SPEC200 suite (both floating point and integer benchmarks). As a result MRET can be used as a upper bound on the maximum coverage that can be obtained by CTT or TT.

3.6 Related and Previous Work

Dynamic binary translation is used in DAISY [9–11, 13] to achieve compatibility between two different architectures. The work is not tied to any particular architecture, thus making the techniques suitable for almost any ISA available. Unlike DAISY, our system translates IA-32 to IA-32. Also, the authors describe a page-based translation scheme, whereas our system operates on classical basic blocks that have no page alignment requirements.

The effects of tail duplication on hyperblock formation can be found on [18, 19]. In addition to tail duplication, [18] performs *head* duplication to perform convergent hyperblock formation.

Traces can be used to speed up system simulators (as described in [20]). Traces are also good units for parallelization [3, 4].

Prior to Trace Trees, Havanki et al [14] used tree regions code generation units. However, the *treegion scheduling* was statically performed to generate code for wide issue processors, whereas Trace Trees (and Compact Trace Trees) use a dynamic approach to generate the trees.

Our framework use software profiling without hardware support to collect hot regions. Chen et al. [6] describe hardware sampling to collect traces with a lower overhead.

3.7 Conclusions

This paper presents an evaluation of the Trace Tree technique running in a IA32 DBT. It presents experimental evidences that the technique, as described previously, is not suited for the challenges presented by both DBT and IA32.

This paper also presents a novel technique for trace construction that seems to perform well under a DBT, being a middle ground between traditional MRET traces and Trace Trees.

We also present experimental evidences that MRET is a good choice when no tail duplication is needed, and when runtime performance must be high and memory usage should be kept low. Also, MRET seems to be an upper bound estimate on trace techniques behavior.

3.8 Acknowledgments

This work was partially sponsored by CNPq, CAPES and Intel Corporation. We would like to thank Dr. Mauricio Breternitz Jr. for bringing TTs to our attention. We also would like to thank the reviewers who helped improving this paper with invaluable suggestions.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, 2000.

[2] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skalesky, Y. Wang, and Y. Zemach. IA-32 execution layer: A two phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. In *36th International Symp. on Microarchitecture*, pages 191–202, 2003.

[3] B. J. Bradel and T. S. Abdelrahman. The potential of trace-level parallelism in java programs. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 167–174, New York, NY, USA, 2007. ACM.

[4] B. J. Bradel and T. S. Abdelrahman. A study of potential parallelism among traces in java programs. *Sci. Comput. Program.*, 74(5-6):296–313, 2009.

[5] P. P. Chang, S. A. Mahlke, and W.-m. W. Hwu. Using profile information to assist classic code optimizations. *Softw. Pract. Exper.*, 21(12):1301–1321, 1991.

[6] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *CGO '03: Proceedings of the*

international symposium on Code generation and optimization, pages 79–90, Washington, DC, USA, 2003. IEEE Computer Society.

[7] C. Cifuentes and M. V. Emmerik. Uqbt: Adaptable binary translation at low cost. *IEEE Computer*, pages 60 – 66, March 2000.

[8] E. Duesterwald and V. Bala. Software profiling for hot path prediction: less is more. In *Proceedings of the 9th International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 202 – 211, November 2000.

[9] K. Ebcioğlu and E. R. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. Technical Report RC-20538, T. J. Watson Research Center, May 1996.

[10] K. Ebcioğlu and E. R. Altman. Daisy: dynamic compilation for 100% architectural compatibility. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 26–37, New York, NY, USA, 1997. ACM.

[11] K. Ebcioğlu, E. R. Altman, M. Gschwind, and S. Sathaye. Optimizations and oracle parallelism with dynamic translation. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 284–295, Washington, DC, USA, 1999. IEEE Computer Society.

[12] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical Report 06-16, Donald Bren School of Information and Computer Science, University of California, Irvine, November 2006.

[13] M. Gschwind, K. Ebcioğlu, E. Altman, and S. Sathaye. Binary translation and architecture convergence issues for ibm system/390. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 336–347, New York, NY, USA, 2000. ACM.

[14] W. Havanki, S. Banerjia, and T. Conte. Treegion scheduling for wide issue processors. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 266–276, New York, NY, USA, 1998. ACM.

[15] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 61–73, Washington, DC, USA, 2007. IEEE Computer Society.

[16] A. Klaiber. *The technology behind Crusoe^{TM} processors.* Tansmeta Corporation, January 2000.

[17] R. Lougher. JamVM virtual machine, June 2009. jamvm.sourceforge.net.

[18] B. A. Maher, A. Smith, D. Burger, and K. S. McKinley. Merging head and tail duplication for convergent hyperblock formation. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 65–76, Washington, DC, USA, 2006. IEEE Computer Society.

[19] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[20] W. S. Mong and J. Zhu. Dynamosim: a trace-based dynamically compiled instruction set simulator. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 131–136, Washington, DC, USA, 2004. IEEE Computer Society.

[21] The SPEC Corporation. www.spec.org.

[22] C. Wang, S. Hu, H. Kim, S. R. Nair, M. Breternitz, Z. Ying, and Y. Wu. Stardbt: An efficient multi-platform dynamic binary translation system. In *Asia-Pacific Computer Systems Architecture Conference*, 2007.

[23] E. Yardimci. *Exploiting parallelism to improve the performance of sequential binary executables.* PhD thesis, University of Irvine, Carlifornia, 2006.

Capítulo 4

Trace Execution Automata in Dynamic Binary Translation

Prólogo

Após a detecção, *Traces* são usualmente criados duplicando-se o código que os definem. Esta representação implícita foi a responsável pelos números obtidos para TT no Capítulo 3 e que levaram à necessidade do desenvolvimento da CTT. Entretanto, é possível utilizar uma representação implícita para *traces*.

Neste trabalho, publicado na terceira edição da AMAS-BT em junho de 2010 em Saint-Malo, França, propôs-se uma representação para *traces* baseada em Autômatos Finitos Determinísticos (AFD) chamada TEA. Além de mais compacta que a representação usual, TEA pode ser utilizada como forma de serialização de *traces* de modo que estes sejam reutilizados.

Além da apresentação acima mencionada, este trabalho aguarda publicação em volume da série *Lecture Notes in Computer Science* (LNCS) publicado pela editora Springer.

Abstract

Program performance can be dynamically improved by optimizing its frequent execution traces. Once traces are collected, they can be analyzed and optimized based on the dynamic information derived from the program's previous runs. The ability to record traces is thus central to any dynamic binary translation system. Recording traces, as well as loading them for use in different runs, requires code replication to represent the trace. This paper presents a novel technique which records execution traces by using an automaton called TEA (Trace Execution Automata). Contrary to other approaches, TEA stores traces implicitly, without the need to replicate execution code. TEA can also be used to *simulate* the trace execution in a separate environment, to store profile information about the generated traces, as well to instrument optimized versions of the traces. In our experiments, we showed that TEA decreases memory needs to represent the traces (nearly 80% savings).

4.1 Introduction

Dynamic Binary Translators (DBTs) rely on information about the dynamic behavior of a program to improve its performance. This is done by detecting and optimizing code fragments, known as *hot code*, which accounts for the largest share of the program execution time.

To optimize hot code, a DBT might use a trace selection technique. Several techniques have been proposed in the literature [1, 3, 5, 9, 15] which address the same issue: how can hot code be easily selected (i.e. with the least possible overhead)? The description of such techniques, as well as their advantages or disadvantages are beyond the scope of this paper, which describes a technique to *record* and *replay* traces.

The *Trace Execution Automata* technique uses a Deterministic Finite Automaton (DFA) to map executing instructions to instructions or basic blocks in previously recorded traces. When operating in *recording* mode, our technique builds a DFA that represents basic blocks (or instructions) from traces. During the *replay* mode, the transition between instructions in the executing program are mapped to transitions in the DFA, which turns into a precise map from the currently executing instructions to the represented basic blocks (or instructions) in the DFA. We found this technique useful in multiple contexts, among them:

- Building traces in one system, e.g. by using a DBT, and collecting statistics and profiling information for them on a second system, e.g. by replaying the traces on a cycle accurate simulator.

- Building and profiling traces without the need for actual trace construction (*e.g* without the need for code replication, code linking and original code patching). This is useful when collecting accurate profiling information before the actual traces code is generated. It is also useful when investigating trace formation techniques because it enables us to focus on the trace formation techniques without concerning about the trace code compilation correctness.
- Storing trace shape and profiling information for reuse in future executions.

This paper is organized as follows. Section 4.2 presents a motivation for TEA. Section 4.3 discusses how traces and DFAs are related to each other and shows how to build DFAs out of traces. Section 4.4 describes our experimental evaluation of TEA. Section 4.5 lists the previous work on trace recording techniques. Finally, Sect. 4.6 concludes the paper and presents the future works.

4.2 Motivation

Dynamic binary translation usually relies on dynamic profiling information to record and aggressively optimize traces. In this section we show why collecting accurate profiling information before building the actual traces may be challenging.

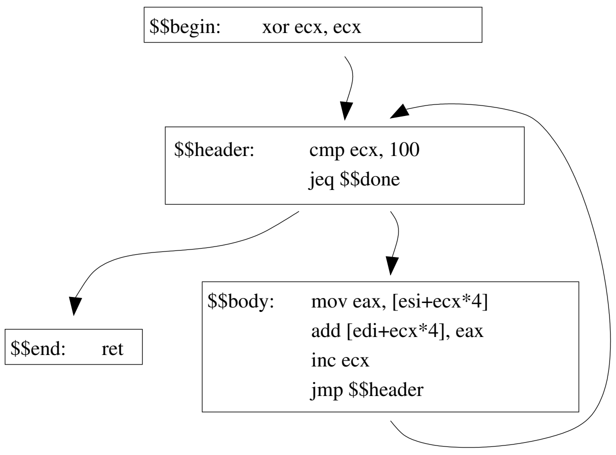
The code on Fig. 4.1(a) adds one hundred words from the array pointed to by *esi* to the array pointed to by *edi*. Although simple, this optimized code presents a challenge to runtime environments which could eventually optimize it: the values in the registers are not known until the application is executed, and might even change across different executions.

Assuming that the code was executed, and the loop it contains was detected as hot code, the trace of Fig. 4.1(b) comes up.

With Algorithm 4.3.1 it is possible to create a DFA to simulate that trace's execution. That DFA can now be loaded into a profiling tool (such as our profiling tool described in Sect. 4.4) and the profile information for the traces can be gathered.

An obvious question we are yet to answer is why not collecting the profile information as the traces are recorded. The simple, straightforward answer is that it might be easier to implement the trace recording algorithm in an environment where gathering profile data is substantially harder than in another environment. In our experiment, recording the traces was easily done in our DBT environment [19], whereas gathering profile information was easier under Pin [14] as the profile code was ordinary C functions instead of assembly language stubs.

Now, assume that traces are optimized using the profile information collected by replaying the DFA. For example, let's suppose the optimizer unrolled the trace by a factor



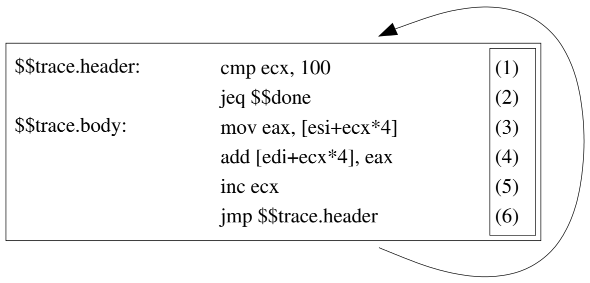
```
$$begin:    xor ecx, ecx

$$header:   cmp ecx, 100
            jeq $$done

$$end:   ret

$$body:     mov eax, [esi+ecx*4]
            add [edi+ecx*4], eax
            inc ecx
            jmp $$header
```

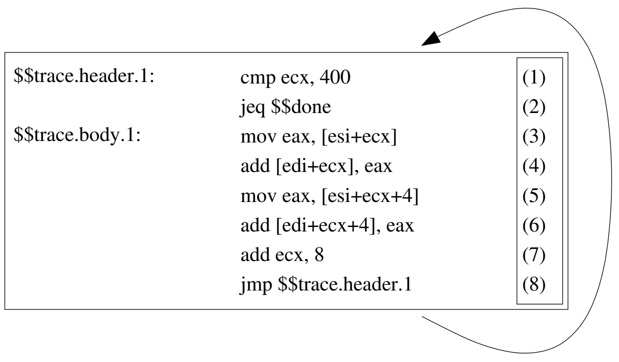
(a) CFG



```
$$trace.header:   cmp ecx, 100              (1)
                  jeq $$done                (2)
$$trace.body:     mov eax, [esi+ecx*4]      (3)
                  add [edi+ecx*4], eax      (4)
                  inc ecx                   (5)
                  jmp $$trace.header        (6)
```

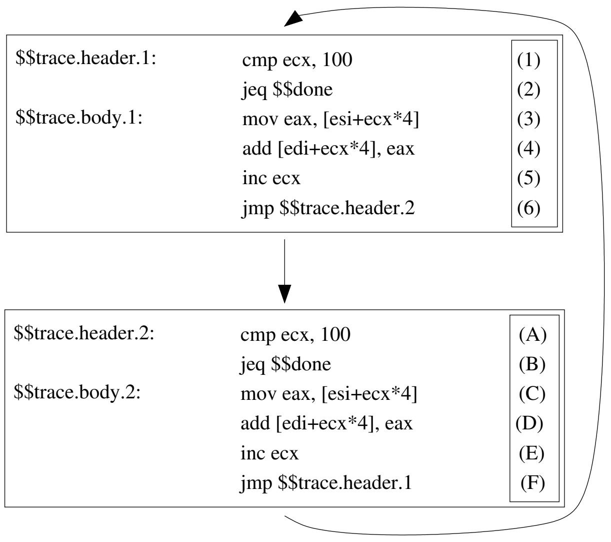
(b) Trace

Figure 4.1: Code Snippet and Resulting Trace



```
$$trace.header.1:      cmp ecx, 400             (1)
                       jeq $$done               (2)
$$trace.body.1:        mov eax, [esi+ecx]       (3)
                       add [edi+ecx], eax       (4)
                       mov eax, [esi+ecx+4]     (5)
                       add [edi+ecx+4], eax     (6)
                       add ecx, 8               (7)
                       jmp $$trace.header.1     (8)
```

(a) Trace After Unrolling



```
$$trace.header.1:      cmp ecx, 100             (1)
                       jeq $$done               (2)
$$trace.body.1:        mov eax, [esi+ecx*4]     (3)
                       add [edi+ecx*4], eax     (4)
                       inc ecx                  (5)
                       jmp $$trace.header.2     (6)

$$trace.header.2:      cmp ecx, 100             (A)
                       jeq $$done               (B)
$$trace.body.2:        mov eax, [esi+ecx*4]     (C)
                       add [edi+ecx*4], eax     (D)
                       inc ecx                  (E)
                       jmp $$trace.header.1     (F)
```

(b) Duplicated Trace

Figure 4.2: Optimization for Fig. 4.1

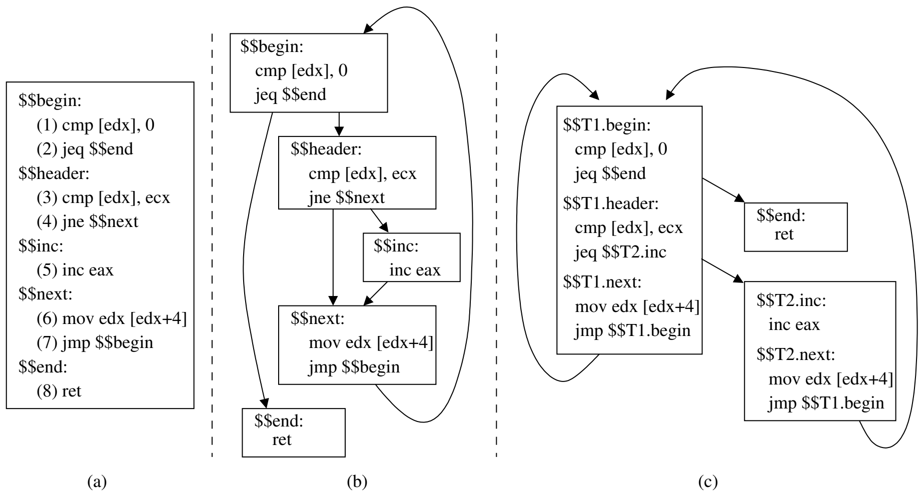

$$begin:
(1) cmp [edx], 0
(2) jeq $$end
$$header:
(3) cmp [edx], ecx
(4) jne $$next
$$inc:
(5) inc eax
$$next:
(6) mov edx [edx+4]
(7) jmp $$begin
$$end:
(8) ret
(a)
$$begin: cmp [edx], 0 jeq $$end
$$header: cmp [edx], ecx jne $$next
$$inc: inc eax
$$next: mov edx [edx+4] jmp $$begin
$$end: ret
(b)
$$T1.begin: cmp [edx], 0 jeq $$end
$$T1.header: cmp [edx], ecx jeq $$T2.inc
$$T1.next: mov edx [edx+4] jmp $$T1.begin
$$end: ret
$$T2.inc: inc eax
$$T2.next: mov edx [edx+4] jmp $$T1.begin
(c)

Figure 4.3: (a) Sample code. (b) CFG for the sample code. (c) MRET traces.

of two as seen on Fig. 4.2(a). There are now two options to determine the new profiling option.

The first option (the easy one) is to *conservatively* propagate the profiling information for the new instructions. For example, assume that instructions (3) and (4) in Fig. 4.1(b) alias. If this information is conservatively propagated to the unrolled trace, this information is likely to constrain any further optimizations.

The second, hard option, is to recollect the profiling information. The DFA can not be used to simulate the unrolled loop. Since it does not generate specialized code, the state for the trace would find no corresponding executable code in the executable. However, it is possible to easily work around this: the trace can be **duplicated** instead of **unrolled**. The duplicated trace is shown in Fig. 4.2(b).

The resulting DFA after the trace has been duplicated can be safely loaded alongside the original program for profiling. This new profile data can then be used after unrolling: instructions (C) and (D) in Fig. 4.2(b) are the same as instructions (3) and (4) in Fig. 4.2(a), thus the collected profile information can be used to optimize the unrolled loop. With the new, specialized information the runtime can accurately optimize the code.

The use for the DFA in profiling can be thought as the ability to label duplicate instructions differently for every copy of it in the running program.

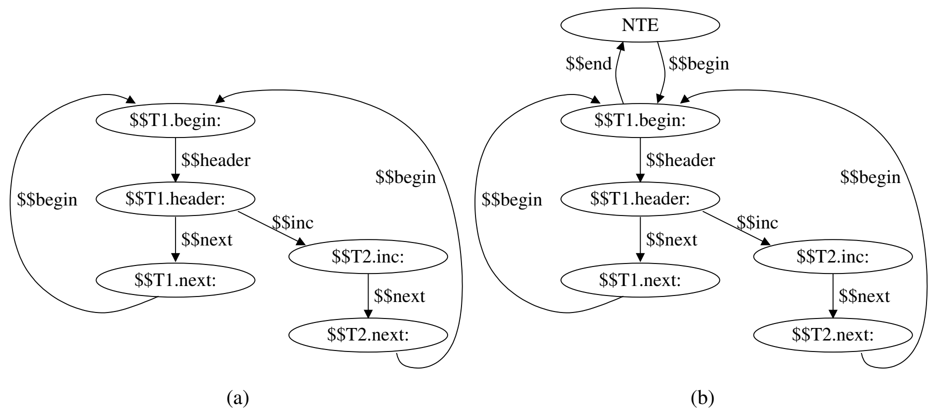


Figure 4.4: (a) DFA for MRET traces. (b) TEA for whole program.

4.3 From Traces to TEA

In this section we illustrate the relationship between Traces and DFAs and provide an algorithm to build TEA out of traces.

Suppose that a runtime system that builds traces[1] using the MRET (Most Recently Executed Tail) strategy [1, 5] is running the compiled code shown, as x86 assembly language, in Fig. 4.3 (a). That piece of code scans a linked list structure pointed to by register *edx* and updates *eax* with the number of times that the value in *ecx* appears on the list. It might take a few iterations for the runtime system to identify the hot code and invoke the trace recording subsystem. The generated traces heavily depend on the trace selection strategy used as well as on the program's input data. Figure 4.3 (c) shows two traces (T1 and T2) that could eventually be recorded by using the MRET trace selection strategy. Trace T1 is formed by the basic blocks $$begin, $$header and $$next, while trace T2 is formed by the basic blocks $$inc and $$next.

In our examples, we use the format $$*Ti.block* when referring to a block that belongs to a trace. This format allow us to distinguish blocks that are duplicated (e.g. $$T1.next and $$T2.next) and avoid confusion with the original block name (e.g. $$next). A trace, or a collection of traces, implicitly defines a DFA. As an example, the DFA for traces on Fig. 4.3 (c) can be seen on Fig. 4.4 (a). Each node in the DFA represents a basic block that is part of a trace. The transitions between nodes represent the control flow in the traces. The label in a transition indicates the address, or the *Program Counter*, that triggers such transition. Notice that the automaton for the trace does not contain the transition from $$T1.begin to $$end. This happens because basic block $$end does not belong to any trace, therefore this transition does not represent control flow inside or between traces.

[1]the word trace will be used from now on as a synonym for hot traces

Suppose that the traces at Fig. 4.3 (c) represent all the hot code for the sample program. To generate a DFA for the whole program all transitions must be accounted by the automaton, including transitions to and from hot code. To model this *whole program* DFA, a special state labeled NTE, which stands for *No Trace being Executed*, is generated. The program is on the NTE state whenever it is not executing any trace. Transitions from NTE to traces are labeled with the traces' start addresses. Transitions from traces to NTE represent control flow between traces and cold code. We call the *whole program* DFA generated from the execution traces and the NTE state **TEA**. For this sample program, the TEA is represented on Fig. 4.4 (b). The TEA is logically similar to the dynamic control flow graph (DCFG) for the traces seen on Fig. 4.3 (c). TEA, however, contains just the *state* information, whereas the DCFG contains code replication. TEA also models the whole program execution with the aid of the NTE state, while the DCFG only represents the hot code.

The generated TEA can be used to replay trace executions without running actual trace code. As an example, we could re-execute the program at Fig. 4.3 (a) on a different system and replay the MRET traces execution by feeding the program counter into the generated TEA. The TEA states provides an accurate mapping from the current program counter to the previously recorded traces. For instance, during the re-execution of the sample program, if the current program counter points to $$next we can precisely tell whether it corresponds to the execution of the original $$next, $$T1.next or $$T2.next by looking at the TEA current state.

The following two sections presents an algorithm that, given a set of traces, builds the corresponding TEA (Sect. 4.3.1) and provides some insights on how TEA can be used online to record traces (Sect. 4.3.2).

4.3.1 Building TEA out of Traces

The initial motivation for TEA was the ability to generate traces in one environment and to load and execute them in another. This simple problem becomes hard when the two different environments are extremely different. TEA enabled us to generate the traces in one system, and execute them on another system. Algorithm 4.3.1 shows how we converted the generated traces to TEA, but first, some definitions are needed.

Definition 4.3.1. *A Basic Block (BB) is a sequence of instructions with a single entry point and a single exit point.*

Usually a BB is terminated by a branch instruction. However, different runtime systems detect basic blocks differently. For instance, on Fig. 4.3 (a) some runtime system might be able to identify block *$$inc* as a basic block, even though it does not end in a branch instruction. Usually, however, DBTs merge blocks *$$inc* and *$$next*. Notice that either way, Definition 4.3.1 correctly identifies BBs. Since each BB might be on several different traces and, depending on the recording algorithm, might appear several times on the same trace, there should be a way to uniquely identify each BB on the traces.

Definition 4.3.2. *A Trace Basic Block (TBB) is an instance of a BB in a trace. Each occurrence of a BB will generate a unique TBB.*

Given Definition 4.3.2, even if BB b occurs several times in the set of traces of a program, it is possible to distinguish between the different *instances* of b. As an example, Fig. 4.3 (c) shows two MRET traces, T1 and T2, with two different instances of BB $$next: $$T1.next and $$T2.next.

Definition 4.3.3. *A Trace is a collection of TBBs and the control flow edges between them.*

Definition 4.3.3 encompasses many different kinds of traces, from traces made of superblocks[2] to Trace Trees [9]. With the previous definitions, it is possible to explain Algorithm 4.3.1 and to prove its correctness.

The first step in Algorithm 4.3.1, lines 1 to 2, initializes the TEA with a single state (NTE) and an empty set of transitions. As discussed, the NTE state represents the execution of basic blocks (or instructions) that does not belong to traces. The next step, lines 3 to 5, adds the states to represent the TBBs. Since each TBB is unique, and exactly one state is created for each TBB, the resulting TEA has the following property:

Property 4.3.1. *The resulting TEA is capable of representing the execution of every TBB.*

The third step, lines 6 to 17, adds the transitions between the states together with the labels that trigger the transitions. First, it adds the transitions that originates at TBBs (lines 7 to 14) by processing the TBBs successor basic blocks. If the successor of the TBB is not a block in a trace, a transition from the TBB to NTE is added, representing a transition from the trace to cold code. Finally, it adds the edges between the state NTE and the TBBs, representing the start of traces execution (lines 15 to 17). Thus the following holds:

Property 4.3.2. *The resulting TEA contains all transitions for every TBB represented.*

Properties 4.3.1 and 4.3.2 ensure the resulting TEA models the exact behavior of the program's traces, thus proving the algorithm correctness.

4.3.2 Recording TEA instead of Traces

As previously mentioned, TEA can be used as a online technique for trace recording. It is built by Algorithm 4.3.2, which is invoked every time the running program finishes a BB

[2]a superblock is a single-entry, multiple-exit sequence of instructions

Algoritmo 4.3.1: Converting Traces to TEA

```
    Input: Ts: The set of Traces in a Program
    Output: TEA: the TEA
 1  TEA.States ← {NTE}
 2  TEA.Transitions ← $\emptyset$
 3  foreach $T \in Ts$ do
 4      foreach $TBB \in T$ do
 5          TEA.States ← TEA.States $\bigcup$ {$TBB$}
 6  foreach $T \in Ts$ do
 7      foreach $TBB \in T$ do
 8          foreach Successor $S$ of $TBB$ do
 9              if $S$ is a trace block then
10                  TEA.Transitions ← TEA.Transitions $\bigcup$
11                          {$TBB \rightarrow S$, Label($S$)}
12              else
13                  TEA.Transitions ← TEA.Transitions $\bigcup$
14                          {$TBB \rightarrow$ NTE, Label($S$)}
15      foreach $TBB \in EntryBlocks(T)$ do
16          TEA.Transitions ← TEA.Transitions $\bigcup$
17                  {NTE $\rightarrow TBB$, Label($TBB$)}
18  return TEA
```

execution but before the next BB is executed. Trace recording is expressed as a three-state State Machine. The possible states are "*Initial*", "*Executing*" and "*Creating*", each of which with its own well-defined rule.

State "*Initial*" is executed before the program starts its real execution. It simply sets up an empty TEA (*i.e.* a TEA with only the NTE state) and indicates that the program is in the "*Executing*" state.

In the "*Executing*" state the application is either running cold code or executing a previously created trace. Depending on the trace recording rules (line 7) the state machine switches to the "*Creating*" state.

Trace recording takes place in the "*Creating*" state. Again, depending on the algorithm being used for trace selection, the state machine decides whether or not to end trace recording (line 12).

Algoritmo 4.3.2: Using TEA to Record Traces

```
Input: Current: The BB Previously Executed
Input: Next: The Next BB to be Executed
Input: State: The Recording Algorithm's Current state
switch State do
    case Initial
        InitializeTEA(TEA)
        State ← Executing
    case Executing
        ChangeState(TEA, Current, Next)
        if TriggerTraceRecording(Current, Next) then
            StartCreatingTrace(Current, Next)
            State ← Creating
    case Creating
        AddTBBToTrace(Current, Next)
        if DoneTraceRecording(Current, Next) then
            FinishTrace(Current, Next)
            State ← Executing
```

4.4 Experimental Results

For this paper, our goals were (1) to evaluate how TEA would decrease memory required to represent traces; (2) to evaluate how effective TEA is for replaying previously recorded traces on unmodified program executables; and (3) to evaluate TEA's effectiveness as a trace recording tool itself. All the experiments were executed in Ubuntu 9.10 in a virtual machine running under Windows 7 in a Core i7 EE 975 with 12 GB of DDR3 1333 MHz DRAM. Our experimental setup included two different DBT frameworks, pin [14] and StarDBT [19].

Pin is a well-known runtime environment which allows programmers to develop their own profiling tool (called "pintools") composed of instrumentation and analysis routines. Pin offers a rich set of APIs that offers great flexibility. It is indeed a very important tool for binary translation experiments, among other uses. For this paper, we implemented a pintool that loads traces from a input file and uses the traces for program execution. Our tool is also capable of recording traces if they are not available prior to program execution.

StarDBT is a DBT runtime environment which translates IA-32 to IA-32. It is less flexible than Pin, but it offers a greater control over how instrumentation and analysis are done. It was used as a baseline for memory requirements to represent traces. The generated traces were also used by our pintool during the "trace replaying" experiment.

Table 4.1: Size Savings with TEA

benchmark	MRET			CTT			TT		
	DBT	TEA	Savings	DBT	TEA	Savings	DBT	TEA	Savings
168.wupwise	329	81	75%	64	14	78%	63	14	78%
171.swim	538	110	79%	998	205	79%	193	38	80%
172.mgrid	671	138	79%	940	198	79%	278	61	78%
173.applu	648	124	81%	1005	187	81%	437	76	82%
177.mesa	583	127	78%	605	126	79%	238	56	76%
178.galgel	1011	238	76%	2083	463	78%	1766	388	78%
179.art	354	90	75%	441	110	75%	322	82	75%
183.equake	442	108	74%	683	157	77%	529	130	75%
187.facerec	674	152	73%	989	211	79%	535	114	79%
188.ammp	551	130	76%	903	197	78%	341	73	78%
189.lucas	113	19	83%	542	103	81%	673	124	81%
191.fma3d	1446	336	77%	1445	294	80%	419	91	78%
200.sixtrack	2162	500	77%	3055	613	80%	1148	225	80%
301.apsi	1346	304	77%	2119	423	80%	695	135	81%
164.gzip	2110	533	75%	51601	11318	78%	598533	143665	76%
175.vpr	1918	457	76%	13893	3093	78%	30687	7298	76%
176.gcc	53203	13147	75%	204203	44728	78%	89358	18917	79%
181.mcf	360	86	76%	855	224	74%	3430	908	74%
186.crafty	1980	493	75%	105018	22224	79%	14829	2998	80%
197.parser	3352	867	74%	25231	5534	78%	17202	3489	80%
252.eon	6217	1007	84%	10218	1677	84%	3732	554	85%
253.perlbmk	17333	4031	86%	78361	16819	79%	48287	9774	80%
254.gap	3183	684	79%	9869	1969	80%	6836	1358	80%
255.vortex	14854	3426	77%	17478	3497	80%	2188	488	78%
256.bzip2	1031	257	75%	59053	13177	78%	1801870	351738	80%
300.twolf	1632	408	75%	9848	2297	77%	7008	1518	78%
GeoMean			77%			79%			79%

Table 4.1 shows the data regarding the size needed to represent the traces. We recorded traces using three different techniques: MRET, CTT (Compact Trace Trees) [15] and TT (Trace Trees) [9]. Previous work by Porto *et al.* [15] showed that memory requirements for the three techniques were different from one another. We wanted to evaluate if TEA was sensitive to the technique. The columns labeled "DBT" indicate the memory requirements (in KB) to represent the recorded traces, whereas the columns "TEA" indicate the memory requirements (also in KB) to represent traces using TEA. The "Savings" column indicates the memory usage savings achieved by representing traces with TEA instead of the usual strategy (*i.e.* replicating the code) to be around 80%. TEA achieves this space savings by avoiding code specialization for trace representation.

Table 4.2 shows the runtime aspects of trace replaying. We again compare our TEA implementation in the Pintool against our "baseline", which are the StarDBT collected traces. The "coverage" columns show how much runtime instructions were executed inside the traces. The "time" column under TEA shows the amount of time needed to replay the traces in our pintool, and under DBT shows the amount of time needed to record the traces in DBT. Since the table displays information about trace *replaying*, it is expected that the coverage for TEA is slightly higher than DBT's coverage since our tool will execute less cold code. This is true for all but one benchmark: 177.mesa. The 0.2% difference in coverage on this particular benchmark occur since Pin and StarDBT use slightly different algorithm to detect individual instructions. Nevertheless, the results are close enough to be considered valid.

Regarding "Time", it is noticeable that TEA presents a somewhat high overhead when compared to DBT's execution. There are at least two reasons for this difference. The first reason is the way Pin inserts the instrumentation code to manipulate the TEA. Usually, pin will insert function calls to the pintool's instrumentation routines, which adds considerable overhead to the program's execution. The other reason is related to TEA's transition function. Every branch instruction is proceeded by a call to a function that eventually searches for the target trace in some sort of data structure. By replicating code to represent the traces, DBT does not need a transition function. The results on this Table (as well as the ones on Table 4.3) were collected with an optimized transition function. The optimizations are described in Sect. 4.4.2.

Table 4.3 shows the data regarding to our experiment on TEA's ability to record traces. For this experiment, we implemented the MRET [1, 5] trace strategy in our pintool. The columns in the table have the same meaning as they have on Table 4.2, except "Time" which means "recording time" for both Pin and DBT. Again, the recorded traces present a slightly different coverage and take more time to record. The reasons for the later are the same as the ones for the replaying experiment. The reasons for the former are the difference in how StarDBT and Pin count runtime instructions as well as subtle algorithm implementation differences.

4.4.1 Implementation Challenges

The most challenging issue faced during the experiments was related to how dynamic basic blocks are identified. StarDBT identifies a TBB as starting at an address which is target of a branching instruction and ending in a branch instruction. Besides this heuristic, Pin also create dynamic basic blocks on some unexpected instructions (*e.g.* x86's cpuid) and instructions with REP prefixes. To address this issues, our pintool inserts the instrumentation code on the taken and fall through edges instead of at the beginning of the TBBs. This guarantees that our pintool will see the same transitions StarDBT saw during trace recording.

Another small issue is related to instruction count. StarDBT counts every instruction to be one instructions, even if it is an instruction with a REP prefix that will iterate

Table 4.2: TEA Runtime Aspects – Replaying

Benchmark	TEA		DBT	
	Coverage	Time	Coverage	Time
168.wupwise	100%	2209	100%	151
171.swim	100%	614	100%	100
172.mgrid	100%	802	100%	144
173.applu	100%	725	100%	79
177.mesa	99.8%	1105	100%	87
178.galgel	100%	1412	100%	175
179.art	99.8%	1881	99.5%	110
183.equake	100%	324	100%	38
187.facerec	100%	1189	100%	95
188.ammp	100%	1558	100%	125
189.lucas	90.4%	670	89.3%	86
191.fma3d	94.2%	636	94.1%	98
200.sixtrack	99.1%	1358	99.1%	129
301.apsi	100%	1560	100%	134
164.gzip	99.8%	2913	99.6%	157
175.vpr	100%	1441	99.9%	97
176.gcc	98.1%	2160	97.6%	203
181.mcf	99.9%	635	99.9%	48
186.crafty	95.6%	2058	95.5%	146
197.parser	100%	3482	100%	163
252.eon	91.0%	9417	90.9%	814
253.perlbmk	83.3%	4890	82.9%	253
254.gap	88.3%	2186	87.9%	111
255.vortex	99.4%	3188	99.3%	242
256.bzip2	99.9%	2077	99.9%	117
300.twolf	100%	2977	100%	181
GeoMean	97.5%	1559	97.4%	129

some times. Pin, on the other hand, creates a loop for these instructions, and counts each instruction of each iteration as one instruction. For this reason, the number of dynamic instructions seen by StarDBT and Pin are slightly different. This is why Tables 4.2 and 4.3 do not show instruction count, but coverage instead.

4.4.2 Analyzing TEA's Performance

The numbers presented in this paper show that our implementation of TEA poses a heavy overhead for programs. Before collecting these results, we experimented with several different implementations for the transition function. This Sect. describes the changes our pintool underwent to improve its performance.

Table 4.4 contains six different entries for each benchmark. The first column ("Native") indicates the native performance numbers for the benchmarks. For each benchmark, every entry is normalized with respect to this value, thus all entries in this column being

Table 4.3: TEA Runtime Aspects – Recording

Benchmark	TEA		DBT	
	Coverage	Time	Coverage	Time
168.wupwise	99.7%	2697	100%	151
171.swim	100%	617	100%	100
172.mgrid	100%	867	100%	144
173.applu	100%	767	100%	79
177.mesa	96.9%	1332	100%	87
178.galgel	100%	1513	100%	175
179.art	100%	1827	99.5%	110
183.equake	100%	308	100%	38
187.facerec	99.3%	1391	100%	95
188.ammp	99.8%	1539	100%	125
189.lucas	100%	667	89.3%	86
191.fma3d	100%	662	94.1%	98
200.sixtrack	100%	1583	99.1%	129
301.apsi	99.2%	1627	100%	134
164.gzip	99.7%	3003	99.6%	157
175.vpr	99.9%	1454	99.9%	97
176.gcc	99.4%	2172	97.6%	203
181.mcf	99.9%	612	99.9%	48
186.crafty	99.7%	2112	95.5%	146
197.parser	100%	3607	100%	163
252.eon	97.5%	15352	90.9%	814
253.perlbmk	99.8%	4407	82.9%	253
254.gap	99.9%	2267	87.9%	111
255.vortex	99.1%	3568	99.3%	242
256.bzip2	99.8%	2168	99.9%	117
300.twolf	100%	2982	100%	181
GeoMean	99.6%	1654	97.4%	129

1.00.

The remaining five entries are all related to program execution under Pin. The column "Without Pintool" indicates the slowdown of running the benchmark under Pin without any pintool loaded. In other words, it indicates Pin's overhead alone, which turned out to be low. Column "Empty" reports the overhead to run the application with TEA with an empty set of traces. For these numbers, no traces were recorded by our Pintool at runtime.

The remaining three columns report the results for loading and replaying traces under three different scenarios. For each benchmark, every experiment use the same set of traces. Column "No Global / Local" indicates that a local cache was used to speed up transitions from one trace to another while no auxiliary data structures were used to speed up trace look up (the traces were kept in a linked list) when the local cache misses. The "Global / No Local" experiment used the global B+ tree to speed up trace look up, while no local caching scheme was employed. The last column, "Global / Local", shows the results when

Table 4.4: TEA Overhead for Various Configurations

Benchmark	Native	Under Pin				
		Without Pintool	Empty	No Global / Local	Global / No Local	Global / Local
168.wupwise	1.00	1.54	43.43	23.57	26.83	19.47
171.swim	1.00	1.04	6.33	4.61	6.15	4.44
172.mgrid	1.00	1.25	5.00	4.12	5.69	3.74
173.applu	1.00	1.09	11.73	6.70	9.90	6.40
177.mesa	1.00	1.25	31.41	29.02	18.61	12.94
178.galgel	1.00	1.06	7.97	5.45	8.33	4.80
179.art	1.00	1.22	30.28	17.05	26.93	18.30
183.equake	1.00	1.15	11.53	6.01	8.94	6.14
187.facerec	1.00	1.27	21.34	18.27	17.47	11.62
188.ammp	1.00	1.05	19.61	9.94	14.79	10.22
189.lucas	1.00	1.12	15.50	7.21	9.84	7.48
191.fma3d	1.00	1.24	10.41	6.52	7.35	5.73
200.sixtrack	1.00	1.00	11.78	6.84	11.10	5.83
301.apsi	1.00	1.11	13.56	11.50	14.44	8.31
164.gzip	1.00	1.34	45.81	22.91	34.46	22.13
175.vpr	1.00	1.18	30.44	16.64	20.72	14.80
176.gcc	1.00	3.93	81.18	278.39	64.43	43.64
181.mcf	1.00	1.04	17.55	9.69	15.65	10.14
186.crafty	1.00	2.60	56.54	51.12	48.96	32.79
197.parser	1.00	2.13	49.02	26.67	39.07	22.10
252.eon	1.00	4.17	62.48	94.77	42.65	30.96
253.perlbmk	1.00	2.97	94.68	60.21	83.72	55.55
254.gap	1.00	2.53	73.82	45.11	57.92	40.04
255.vortex	1.00	2.30	70.89	223.68	44.22	30.63
256.bzip2	1.00	1.51	37.17	20.24	27.76	18.93
300.twolf	1.00	1.15	30.34	16.98	28.10	17.49
GeoMean	1.00	1.50	25.27	18.52	20.33	13.53

both the global B+ tree and the local cache were used.

The auxiliary structures are very important in the TEA's transition function, which is the responsible for most of TEA's overhead. In fact, the first TEA implementation employed no auxiliary data structures for speeding up trace look up. The numbers for this particular experiment (which would be the "No Global / No Local" column in Table 4.4) were not collected since the slowdown was over 2 orders of magnitude from the native execution.

Our first attempt to speed up the transition function was the global B+ tree. The results were interesting, but the overhead was still very high. Later, we implemented the local cache to avoid going to the global trace container every time the system needed to search for a trace. Again, the results improved over the previous data. This configuration ("Local / Global") was used to collect all the data for the Recording / Replaying experiment.

We also investigated whether or not the global B+ tree was important to the overall performance. The experimental data shows that, while the local cache is more important than the global B+ tree, the B+ tree is important as well. A comparison between columns “No Global / Local” and “Global / Local” clearly indicates a performance improvement when using the more optimized global container. Particularly, GCC and Vortex experience a severe slowdowns without the global indexing structure.

The data on the “Empty” column report a counter intuitive result. Having no traces to simulate should be faster than having several traces. However, the numbers do make sense, as the transition function is optimized for the common case (*i.e.* executing hot code). TEA performs more work to switch states while in cold code than it does while in hot code. This run had the global B+ tree and did not have any local caches (local caches are pointless outside of traces in our implementation anyway).

4.5 Previous and Related Work

Traces are closely related to dynamic binary translation and dynamic compilation techniques. Suganuma *et al.* [18] presented a complex JIT compiler for a production level Java Virtual Machine. Unlike previous approaches, that used method boundaries for JITing, they implemented a multi-level compilation strategy and use dynamic compiler to dynamically form “regions”, which are their runtime system’s compilation unit. Zaleski *et al.* [21] presented an extensible JIT compiler which uses traces as compilation units.

Several trace recording strategies exist on the literature. MFET (Most Frequently Executed Tail) [3] instruments edges in the dynamic program execution to detect frequently executed paths. MRET (Most Recently Executed Tail) [1,5] instruments back edges only, thus posing less runtime overhead than MFET. TT [9] record traces which always end with a branch to an “anchor”, generally a loop header. CTT [15] tries to address the code duplication experienced by TTs by allowing branch targets within a path to be any loop header in that path.

Another use for traces in JIT compilers is described by Gal *et al.* [10]. They use Trace Trees [9] as compilation units for the SpiderMonkey JavaScript Virtual Machines. Besides all the complication in a JVM JIT compiler, the authors face more challenges since JavaScript is dynamically typed.

Besides those well-known uses, recently Wimmer *et al.* [20] used traces to perform phase detection. A program phase is identified when the created traces are *stable* (*i.e.*, there is a low trace exit ratio). Whenever program execution start to take side exits more often, the program is said to be in an *unstable* (*i.e.* between phases).

Several well-known optimization systems have employed traces to capture program’s code locality. Examples of these environments are Dynamo [1], FX!32 [12] and the IA-32

Execution Layer [2].

All the previously mentioned systems work with user mode code. More complicated DBT systems can translate system level code. For instance, DAISY [6, 7, 11] is a compatibility layer which translates PowerPC code to an underlying VLIW systems. The Transmeta CMS [4] is the compatibility layer on the top of the Crusoe [13] microprocessor. They both utilize some sort of trace recording to select hot code. Both system could have applied TEA as a tool for dynamic trace recording.

On the hardware side, traces have been used for high-bandwidth instruction fetch [16]. The Pentium IV processor [8] implements a trace cache. High-bandwidth instruction fetch is achieved since *logically* contiguous instructions in the instructions stream are placed adjacent to one another in the trace cache. This high-bandwidth cache was needed due to the high clock frequencies that the processor achieved [17]. TEA is different from trace caches since it does not require instructions to be contiguous on the instruction stream.

4.6 Conclusions and Future Work

This paper presents TEA, a technique that uses Deterministic Finite Automata (DFA) to map executing instructions to instructions or basic blocks in previously recorded traces. We list multiple contexts in which TEA is useful and we discuss the implementation challenges and solutions when implementing TEA on StarDBT and Pin frameworks.

Our experimental results show that the resulting TEA's transition lookup operation plays a fundamental role on TEA's performance. For this paper, we implemented the lookup operation with the help of a auxiliary look up data structures, which is searched whenever there is a transition from cold code to hot cold, or when there is a transition from one trace to another. In the future, we will investigate other techniques to optimize the transition lookup operation and amortize TEA's cost.

References

[1] Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: a transparent dynamic optimization system. SIGPLAN Not. 35(5), 1–12 (2000)

[2] Baraz, L., Devor, T., Etzion, O., Goldenberg, S., Skaletsky, A., Wang, Y., Zemach, Y.: Ia-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium®-based systems. In: MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture. p. 191 (2003)

[3] Cifuentes, C., Emmerik, M.V.: Uqbt: Adaptable binary translation at low cost. Computer 33(3), 60–66 (2000)

[4] Dehnert, J.C., Grant, B.K., Banning, J.P., Johnson, R., Kistler, T., Klaiber, A., Mattson, J.: The transmeta code morphing™software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In: CGO '03: Proceedings of the international symposium on Code generation and optimization. pp. 15–24 (2003)

[5] Duesterwald, E., Bala, V.: Software profiling for hot path prediction: less is more. In: ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems. pp. 202–211 (2000)

[6] Ebcioğlu, K., Altman, E.R.: Daisy: dynamic compilation for 100% architectural compatibility. In: ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture. pp. 26–37 (1997)

[7] Ebcioğlu, K., Altman, E.R., Gschwind, M., Sathaye, S.: Optimizations and oracle parallelism with dynamic translation. In: MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture. pp. 284–295 (1999)

[8] Friendly, D.H., Patel, S.J., Patt, Y.N.: Putting the fill unit to work: dynamic optimizations for trace cache microprocessors. In: MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture. pp. 173–181 (1998)

[9] Gal, A., Franz, M.: Incremental dynamic code generation with trace trees. Tech. Rep. 06-16, Donald Bren School of Information and Computer Science, University of California, Irvine (November 2006)

[10] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M.R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E.W., Reitmaier, R., Bebenita, M., Chang, M., Franz, M.: Trace-based just-in-time type specialization for dynamic languages. In: PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation. pp. 465–478 (2009)

[11] Gschwind, M., Ebcioğlu, K., Altman, E., Sathaye, S.: Binary translation and architecture convergence issues for ibm system/390. In: ICS '00: Proceedings of the 14th international conference on Supercomputing. pp. 336–347 (2000)

[12] Hookway, R.: Digital fx!32: Running 32-bit x86 applications on alpha nt. In: COMPCON '97: Proceedings of the 42nd IEEE International Computer Conference. p. 37 (1997)

[13] Klaiber, A.: The technology behind Crusoe^{TM} processors. Tansmeta Corporation (January 2000)

[14] Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. pp. 190–200 (2005)

[15] Porto, J.P., Araujo, G., Wu, Y., Borin, E., Wang, C.: Compact trace trees in dynamic binary translators. In: 2^{nd} workshop on architectural and micro-architectural support for binary translation (AMAS-BT'09) (2009)

[16] Rotenberg, E., Bennett, S., Smith, J.E.: Trace cache: a low latency approach to high bandwidth instruction fetching. In: MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture. pp. 24–35 (1996)

[17] Sprangle, E., Carmean, D.: Increasing processor performance by implementing deeper pipelines. SIGARCH Comput. Archit. News 30(2), 25–34 (2002)

[18] Suganuma, T., Yasue, T., Nakatani, T.: A region-based compilation technique for dynamic compilers. ACM Trans. Program. Lang. Syst. 28(1), 134–174 (2006)

[19] Wang, C., Hu, S., Kim, H., Nair, S.R., Breternitz, M., Ying, Z., Wu, Y.: Stardbt: An efficient multi-platform dynamic binary translation system. In: Asia-Pacific Computer Systems Architecture Conference. pp. 4–15 (2007)

[20] Wimmer, C., Cintra, M.S., Bebenita, M., Chang, M., Gal, A., Franz, M.: Phase detection using trace compilation. In: PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java. pp. 172–181 (2009)

[21] Zaleski, M., Brown, A.D., Stoodley, K.: Yeti: a gradually extensible trace interpreter. In: VEE '07: Proceedings of the 3rd international conference on Virtual execution environments. pp. 83–93 (2007)

Capítulo 5

Extending Decoupled Software Pipeline to Parallelize Java Programs

Prólogo

Após o trabalho inicial com *traces* o foco da pesquisa sofreu uma pequena alteração. Ao invés de detectar código quente *online*, passou-se a identificar os laços significativos para execução do programa. Esta mudança ocorreu devido ao grande *overhead* para o gerenciamento *online* de *traces* em *software*.

Como as otimizações estáticas tradicionais são dependentes do desempenho de um único núcleo, e como a indústria mudou o paradigma de desenvolvimento de processadores para múltiplos núcleos, buscou-se um tipo de otimização que utilize mais efetivamente os recursos disponibilizados por tais arquiteturas.

DSWP [41] é uma técnica de TLS com resultados promissores. Iniciaram-se os estudos desta etapa com a paralelização manual de quatro *benchmarks* do SPECjvm2008. Os resultados desta avaliação encontram-se no artigo a seguir, submetido para publicação na revista *Software Practice and Experience*.

Abstract

Programmers can no longer rely solely on micro-architectural nor technology improvements to have their programs running faster. In today's multi-core chips, parallel code need to be explicitly written in order to extract any benefits from the extra available processing power. In this paper we discuss a simple yet effective technique for program parallelization, based on Decoupled Software Pipeline (DSWP). DSWP is a recent technique proposed to parallelize program loops at the binary level, which works well only under fast inter-core communication. In this paper we propose and evaluate a software inter-core communication scheme which enables DSWP in Java applications. We analyze three memory communication queues implementations and show experimental results that reveal an average 48% speedup on some SPCjvm2008 benchmarks.

5.1 Introduction

Power and energy constraints on contemporary microprocessor designs have broken the trend of relying solely on micro-architectural optimizations and semiconductor technology advancements to improve single threaded performance. To overcome this challenge, *Chip Multi-Processors*(CMP) have been adopted as the mainstream solution. In CMPs, to better utilize the available resources, programs must be explicitly parallelized.

Parallel programming is neither easy nor intuitive [1, 2]. Since parallel software runs nondeterministically, debugging can be a challenging task, even for experienced programmers. Automatic parallelization strategies can ease multi-threaded development. Unfortunately, automatic program parallelization has been successful only in very specific cases, as in scientific computing and regular loops. Moreover, today's most ubiquitous multi-core chips (*x86*) are CMP processors with no particular support for fast inter-thread communication (a feature needed by most of the *general* parallelization techniques), which is required to be done through memory. Therefore, the implementation of fast communication mechanism through memory is a central issue in order to improve the final program performance.

Although several contributions on automatic thread extraction techniques in hardware have been made by the academia (Section 5.2.1 surveys some of them), the industry has been reluctant to adopt them due to many factors, amongst them being uncertainty about *real world* energy requirements and real benefits, *hardware* complexity and general programmability issues. This creates the need for *software*-based thread extraction techniques.

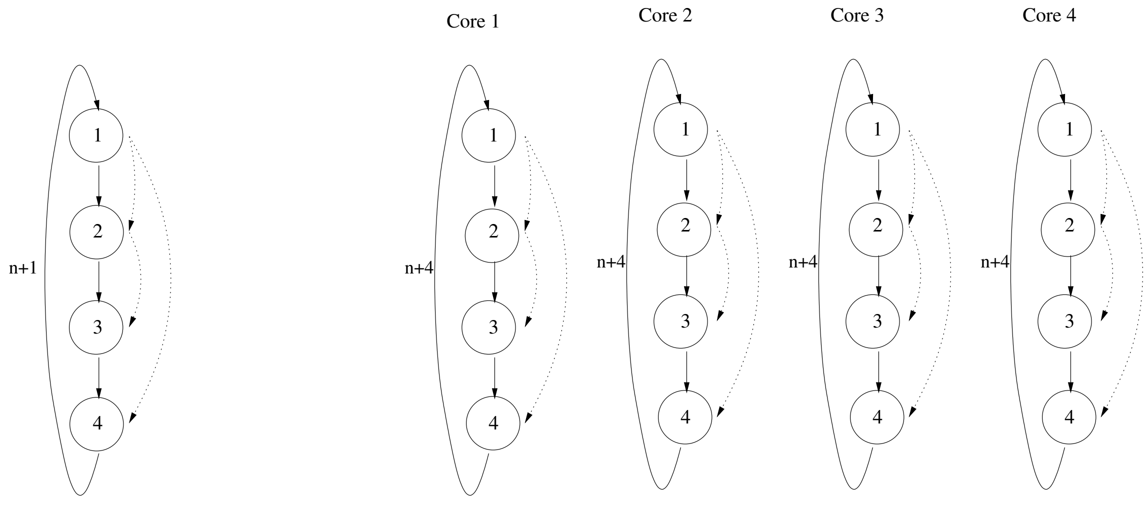


Figure 5.1: DOALL loop

5.1.1 Contributions

This paper proposes jDSWP (Java-DSWP), a simple Java parallelizing technique, based on Decoupled Software Pipelining (DSWP) [3]. It extends DSWP to source-level Java programming without the need for any special hardware nor *Java Virtual Machine* (JVM) support. We evaluated the proposed technique with four benchmarks from the SPECjvm2008 [4] suite and observed promising speedup numbers with very little added programming complexity.

This paper is organized as follows. In Section 5.2, background information about program parallelization is provided . Section 5.3 presents the parallelization strategy we used in our experiments. Section 5.4 details our inter-thread communication mechanism (on Section 5.4.3) and explains the interface of the framework built to conduct the experiments (Sections 5.4.1 and 5.4.2). Section 5.5 shows the actual parallelization work done in two SPEC jvm2008 benchmarks. Experimental results are presented in Section 5.6 while the conclusions are shown in Section 5.7.

5.2 Previous Work

Automatic loop parallelization has been well studied for a long time. There are at least three different types of loop parallelization strategies: DOALL, DOACROSS and DOPIPE. The DOALL parallelization [5] is applicable when a loop presents no loop carried dependencies. The maximum achievable speed up scales with the number of cores. Figure 5.1 (a) shows a sequential loop with no loop carried dependencies. In that Figure,

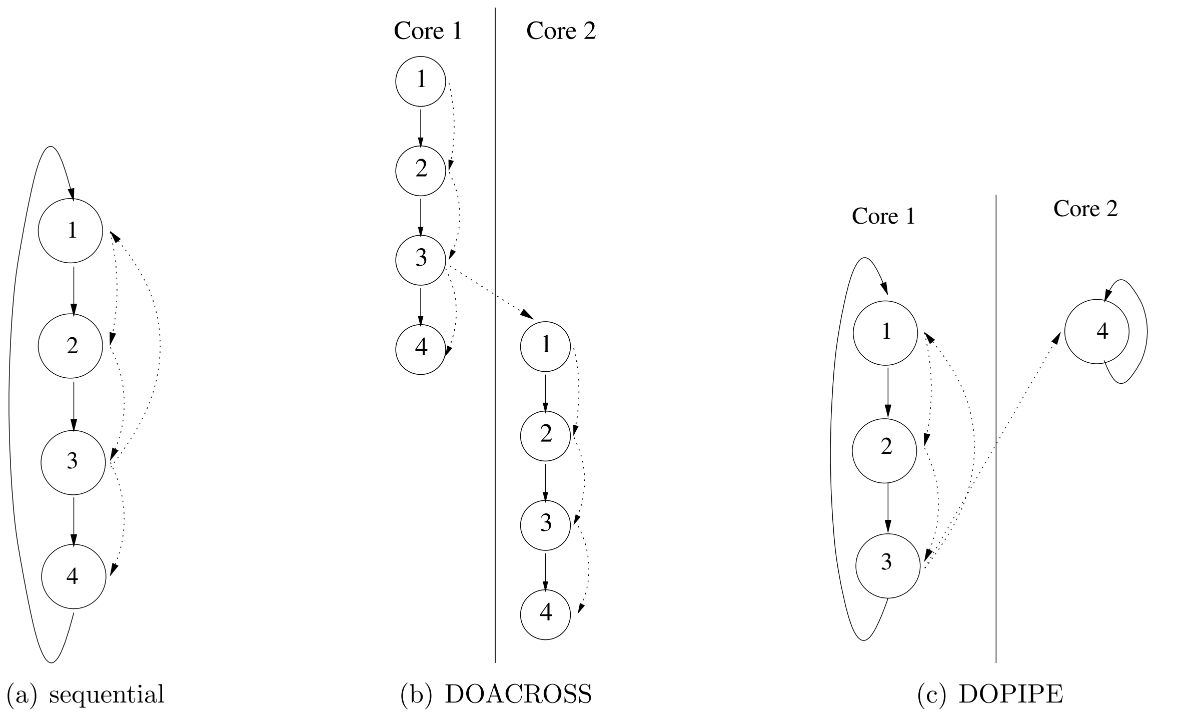


Figure 5.2: Non-DOALL loops

and (throughout the paper unless otherwise stated) solid arrows represents *control-flow* edges while dotted arrows represent *data-flow* edges. Figure 5.1 (b) shows the parallel execution of a DOALL loop using four cores. Notice how `n+1` was transformed into `n+4`. If the loop was to be executed in a processor with more cores, the loop would simply be replicated in the other cores, and the loop step would be appropriately changed. Although very efficient, this technique is not applicable to most real world applications as loops tend to have loop carried dependencies.

Figure 5.2 (a) depicts a loop which is very similar to the loop presented in Figure 5.1 (a). A more detailed look reveals that Figure 5.2(a) contains a loop carried dependency (the dotted edge 3 $\rightarrow$ 1), which constrains the use of the DOALL parallelization. For this reason, other parallelization techniques were proposed.

DOACROSS parallelization [6] allows loops to have loop carried dependencies, which are communicated between the cores as they are computed in the loop body. Figure 5.2 (b) shows the DOACROSS parallelization of the loop in Figure 5.2 (a). The dotted edge 3 $\rightarrow$ 1 indicates that data communication between the threads is necessary. DOACROSS parallelization is effective, but it creates dependencies between the threads, which might limit the loop's runtime performance.

DOPIPE parallelization [7] (Figure 5.2 (c)) splits each loop iteration into two or more threads. The dependencies between the threads are only allowed to be communicated forward, thus avoiding cyclic dependencies. Since the code is rewritten at compile time, just as is the DOACROSS case, the generated code does not benefit from an increased number of cores as a DOALL loop would. In other words, if the code was parallelized with two threads, there will be no benefit if the program is run in systems with more than two cores.

More recently, Zhang et al. proposed Alchemist [8], which is a profiling tool that identifies potential program constructs that are amenable to parallelization. It locates program structures (*e.g.* if statements, while statements) which can be executed in parallel with one another. It does not automatically convert a sequential program into an equivalent parallel one.

Java parallelism is not new. The Java Development Kit (starting with version seven) supports the "divide and conquer" strategy. Using this strategy, a complex, large instance of a problem can be divided into two or more smaller, simpler instances (the "divide" part) which are individually solved. The individual results can then be merged to obtain the final result (the "conquer" part). In comparison, our technique works by splitting loop *bodies* across threads.

Java parallelism can also be achieved by clustering machines together. Although several different machines could be connected to the cluster, the JVM presents itself to the application as a single virtual machine. The Terracotta Infrastructure [9] is an example of such virtual machine. Although inherently parallel, this virtual machine does not add any parallel processing constructs to the Java Language Specification [10] nor to the Java Virtual Machine Specification [11]. Differently from this strategy, our experiments were executed in a regular virtual machine running on a multi-core computer.

Lithium [12] presents a skeleton based framework for writing parallel Java. One of its templates (called *Pipeline*) is very similar to the framework we used to parallelize Java applications, though it does not provide any mechanism to detect and enable DSWP parallelism.

5.2.1 A Survey on TLS

All the above mentioned approaches are software based parallelization. Parallelism may also be harvested through the use of specialized hardware. In this domain, Thread Level Speculation (TLS) has been extensively studied and has been shown to be a promising technique. In summary, TLS techniques usually *speculate* (predicts) data or control information ahead of time. All the following techniques have in common the need of some hardware support. By contrast, the technique proposed in this paper has been tested on commodity hardware widely available, not requiring any special support besides the few classes described in Section 5.4.

Ottoni *et al.* [3, 13–15] proposed DSWP, which is a type of DOPIPE parallelization. DSWP is an automatic thread extraction technique that relies on a small communication queue between cores to communicate inter-thread values. Actual program parallelization is done by a static compile. While they reported interesting numbers, the need for hardware support (the communication queue) might impair DSWP usage in today's multi-core processors (particularly *x86* CMP computers.)

Dynamic multi-threading was proposed by Akkary & Driscoll [16] as an automatic, hardware-based approach to extract threads out of sequential code. In this execution model, the hardware aggressively seeks loops and function calls on the instruction stream to generate new execution threads. Since the hardware spawns the threads aggressively in advance, the live in values for the newly spawned threads are likely unavailable, thus the hardware is responsible to speculate on them.

Checkpoint Processing and Recovery (CPR) [17] and Continual Flow Pipelines (CFP) [18] recognized that the committing instructions in program order is a severe performance restriction for Out-of-Order processors. The authors propose and evaluate a computer micro-architecture capable of out-of-order commit while preserving sequential execution semantics. They do not extract threads. CPR/CFP and TLS try to boost program performance with out-of-order instruction commit. Unfortunately, the proposed architecture proved to be very power-demanding. To mitigate that, in-order CFP (iCFP) [19] allows instructions to commit out-of-order in a simple, pipelined processor. This approach is useful to tolerate long-latency loads.

A compiler support for automatic thread extraction is described in [20]. Hybrid predictors [21] are used to speculate on unavailable *live-ins*. The authors avoid the need of more extensive hardware support by using comprehensive profile information. The lack of hardware support for miss-speculation recovery demands a conservative approach, as any violation of original program semantics will be unrecoverable.

BulkSC [22] and Bulk [23] augments the available checkpoint capabilities of modern processors with a TLS mechanism to detect memory ordering violations in threaded applications. The proposed mechanism is based on signatures of the accessed addresses. It also adds a centralized arbiter which is responsible for checking the signatures nd detecting conflicts. Upon a conflict, the hardware rolls back to the latest valid checkpoint and resumes program execution.

Hydra [24] is a CMP processor composed by four MIPS cores. The authors, recognizing the adding more cores to the processor without specialized support for speculation hardens parallelization efforts, limiting overall chip programmability. To address that they propose a coherence hardware which detects sequential violations and recover from misspeculations. The parallelized code is therefore not constrained to safe parallelization.

Kim & Yeoung [25] carry out a comprehensive study on pre-execution code. They evaluate several different pre-execution algorithms running on several different compilers. Unlike most approaches, the required hardware support is not responsible for maintaining data coherence: the authors only assume that the underlying hardware has a minimal support for thread creation.

Krishnam & Torrellas [26] proposed a TLS architecture and a binary analyzer. The later is responsible for statically analyze binary program (which makes it useful for legacy application when the source code is not available) and determine inner loop boundaries, while the former executes the annotated program. In this architecture, register dependencies are honored at runtime (rather than at compile time as in other approaches) by a synchronizing scoreboard, while memory dependencies are honored by the Memory Disambiguation table(MDT).

Marcuello & González [27] describe a TLS hardware which uses simplified heuristics for thread creation and value prediction. Thread creation focus on loops (not necessarily inner loops), and use execution traces to speculate on the dynamic behavior of the parallelized loop. Value prediction is accomplished by means of a value prediction table.

Sohi *et al.* [28] propose a hierarchical execution model, the Wisconsin Multiscalar. This model uses a specialized hardware to predict the control-flow. It is called hierarchical execution model because each thread spawns another thread, thus creating a thread hierarchy. Upon detection of miss-speculation, the offending thread and all of its descendants are squashed. Execution progress is guaranteed because of the existence of a single, non-speculative task (the oldest task) which is never be squashed.

Master/Slave Speculative Processing [29] speeds up program execution by making the common case fast. In this execution model, there is a master processor, which is responsible for running an "approximation" of the sequential program as well as to start "slave" threads. The "slave" threads run the original, sequential code and are responsible to assert that the program executed by the "master" processor is correct.

5.3 Java-DSWP Parallelization

As in DSWP, jDSWP (*Java-DSWP*) divides loops into loop-carried independent code fragments, which are then used to define a set of pipeline stages. Each stage is executed as an independent thread reading its inputs from the previous stage and sending its outputs to the next stage. In order to better describe the jDSWP parallelization strategy, we first introduce some concepts.

Definition 5.3.1. *In the Control-Flow Graph (CFG), an edge is* **interesting** *iff*

- *It has a* `call` *(function, procedure or method invocation) as its source; and*
- *The called function does not change loop-carried values.*

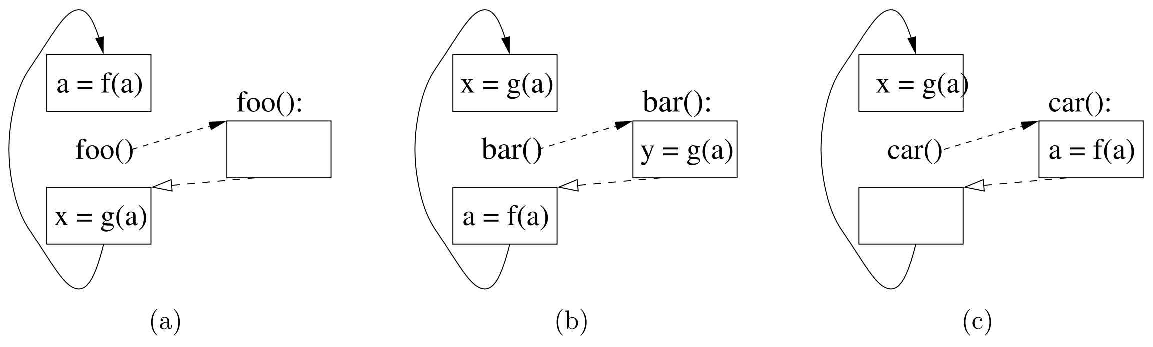


Figure 5.3: Examples for Definitions 5.3.1, 5.3.2 and 5.3.3

The *dashed* edges with solid heads in Figure 5.3 (a) is an *interesting* edge: its *source* is a `call` instruction and the *target* function is not responsible for any updates to *loop-carried* definitions. Figure 5.3 (b) shows an example of an *interesting* edge in the presence of *loop-carried* values: *bar* reads *a* but does not update it, thus not violating any condition in Definition 5.3.1. Finally, Figure 5.3 (c) shows an example where the called function assigns a new value to *a*'s *loop-carried* dependence. This violates the second condition for an edge to be considered *interesting*.

Definition 5.3.2. *A* **return** *edge is an edge in the CFG that represents the return control-flow of a function which was invoked by an interesting edge. For each interesting edge, there may be more than one return edge.*

In Figures 5.3 (a) and 5.3 (b), the *dashed* edges with a unfilled head represent *return* edges. Figure 5.3 (c) contains no *return* edges as it contains no *interesting* edges.

Definition 5.3.3. *A* **potential** *block is either*

1. the target of a **return** *edge; or*

2. dominated by a **potential** *block.*

and contains no update to loop-carried values.

The only *potential* block in the previous examples can be found in Figure 5.3 (a). Figure 5.3 (b) does not contain a *potential* block, as the *return* edge's target updates *a*, which is a loop-carried variable.

Given the previous definitions, jDSWP source level parallelization works as follows:

1. Identify the *interesting* edges on the target loop
2. For each *interesting* edge from the previous step, identify the corresponding *return* edges.
3. For each *return* edge, identify the *potential* blocks.
4. Let F be the set of every function f which is the target of an *interesting* edge.
5. For each function f in F, remove f from F if its outputs either:

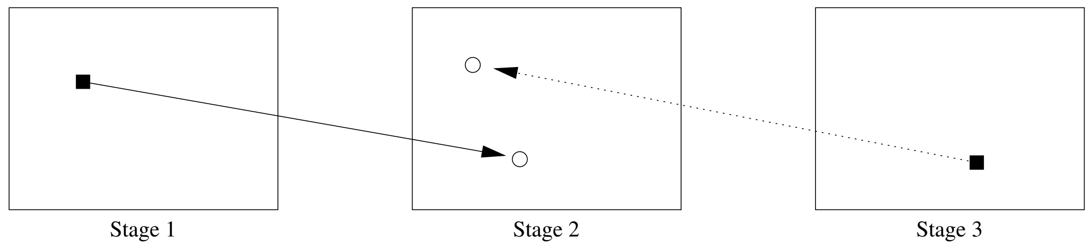


Figure 5.4: Forward and Backward Communication Example

(a) define a loop-carried dependency; or

(b) are used after the *return* edge in a non-potential block.

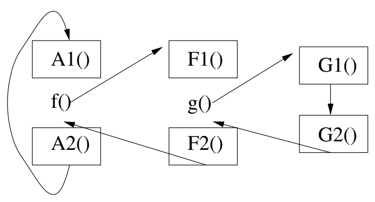
Each function *f* selected as amenable to parallelization can be extracted from the loop body and moved into its own pipeline stage. *f*'s potential blocks (*i.e.*, its continuation) can also be moved to *f*'s stage, or to any stage following *f*, as long as *f*'s potential blocks do not update any loop carried dependency. Marking a function as *not amenable* to parallelization in step 5 is important to avoid **backward** communication in the pipeline. **Backward** communication is likely to "re-couple" the (decoupled) pipeline stages, thus increasing the communication overhead and possibly creating circular dependencies among threads. Only **forward** communication is allowed in the proposed parallelization strategy. To illustrate these two types of communication direction, see Figure 5.4: each *stage* is executed in a different processor; the black squares represent *producer* statements (*i.e.*, statements that generate some value) and the circles represent *consumer* statements. The solid edge represents **forward** communication while the dotted edge represents **backward** communication.

Figure 5.5 illustrates our approach with three (non-exhaustive) examples. In Figure 5.5 (a) there is an outer loop which invokes *f()*, which then invokes *g()*.

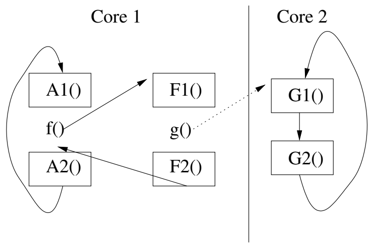
Figure 5.5 (b) illustrates an example where *g()* was *independent* from the rest of the loop. This is the case when, for example, *g()* is a log function. In this case, removing the log output from the application's loop might yield a significant speedup, as logs are usually written to disk.

Figure 5.5 (c) depicts a scenario where *f*'s tail (the code *f* executes when *g* returns) is not part of any loop-carried dependencies in the main loop. In such case, the parallelized application can execute *f*'s tail outside the main loop. This is beneficial for the parallelized application since it is likely to increase the maximum speedup.

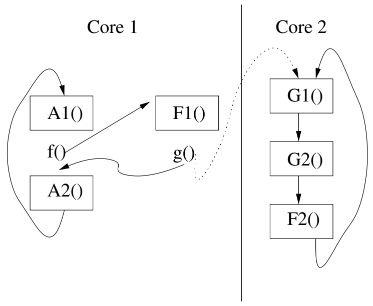
Figure 5.5 (d) shows the result of applying our technique to the DOALL loop of Figure 5.5 (a). Since DOALL loops can not have *loop-carried* dependencies, every call edge becomes an *interesting* edge.



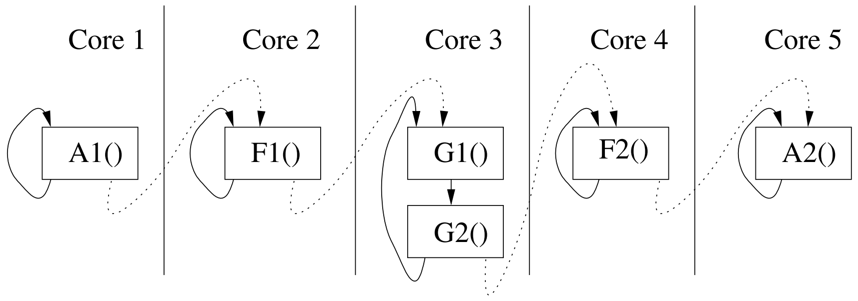
(a) Sequential loop



(b) g's Invocation Pipelined



(c) g's Invocation and a *Potential* Block Pipelined



(d) A DOALL Loop Pipelined

Figure 5.5: Sample Loop

5.4 The `funpipe` Package

As explained above, jDSWP parallelizes an application by creating a pipeline of loop-carried independent stages. Given the pipeline logic does not change from application to application (although each stage *logic* changes), a package called `funpipe` was created to handle the application-independent features of the pipeline.

This package contains several classes, but only two of them are visible: the `funpipe.Pipeline` class and the `funpipe.Pipeline.Stage` class. The later is an abstract class which must be specialized by the programmer to define each stage in the pipeline, and is described in Section 5.4.1, while the former implements a pipeline of `funpipe.Stage`s and is explained in Section 5.4.2.

Section 5.4.3 exposes some of the inner details of the class `funpipe.Communication-`

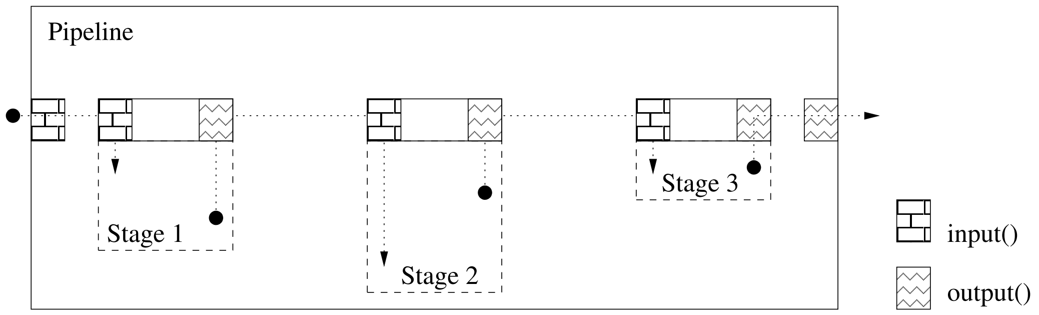


Figure 5.6: A Sample 3-Stage Pipeline

`Queue` which, as its name suggests, implements a communication queue. This queue is used by the `funpipe.Pipeline` class to create communication channels between stages. As an analogy with pipelines in computer architectures, the communication queues can be seen as FIFO pipeline registers.

A high-level overview of a three-stage pipeline is shown on Figure 5.6. The variable logic in at is stage is contained within the dashed squares in the figure (labeled Stages 1 through 3). The motif-decorated squares represent the communication queues between the stages. The dotted arrows depict the data flow between the stages.

```
package funpipe;

public final class Pipeline
{
    public static abstract class Stage
    {
        public abstract void stage()  throws Exception;
        protected void  finish();
        protected Object input();
        protected Object output();
    }

    public Object input(Object data);
    public Object output();
    public void shutdown();
    public static Pipeline createPipeline
          (final Pipeline.stage... stages);
}
```

Figure 5.7: The `funpipe` Package Interface

5.4.1 The `funpipe.Pipeline.Stage` Abstract Class

Each application demands its own set of pipeline stages. Nevertheless, stages do share some characteristics like communication and thread management. For this reason, the `funpipe` package provides the `abstract` class `funpipe.Pipeline.Stage` which already implements these common features, thus easing the parallelization process. Its interface is shown in Figure 5.7.

To communicate to its adjacent stages the `funpipe.Pipeline.Stage` class provides to communication channels. The `input()` channel receives data from the previous `Stage` (or from outside the `Pipeline`). The `output()` channel provides the channel to send data to the next `Stage` (or outside the `Pipeline`). These members are `protected`, and should not be made available outside the stage.

To implement a `Stage`, the programmer needs to override the `funpipe.Pipeline.Stage.stage()` abstract method. This method does not receive any parameters, thus any information necessary for the stage should be received either by constructors or (preferably) using the stage's `input()` channel. Furthermore, the `stage()` method is repeatedly invoked by the pipeline if the pipeline's execution is not completed, so the programmer does not need to implement a non-stopping loop. If any post-processing action is required, the programmer can override the `protected` method `funpipe.Pipeline.Stage.finish()`.

5.4.2 The `funpipe.Pipeline` Class

This class isolates the logic related to the pipeline, and its interface is also shown in Figure 5.7. It was designed so the programmer cannot tamper with any of the inner details once a pipeline is created. *E.g.*, the programmer has no power to change the stage ordering after the `funpipe.Pipeline.createPipeline()` method is invoked. The pipeline class automatically sets up the necessary communication queues so that the stages can communicate.

If the pipeline needs any inputs the programmer can use the `funpipe.Pipeline.input()` method. This channel binds to the input of the first stage in the pipeline. Also, if the pipeline generates any output, the programmer can retrieve it using the `output()` method.

Once the pipeline is no longer needed, the programmer must invoke the `funpipe.Pipeline.shutdown()` method to relinquish the `Pipeline`'s resources. This methods sends a special token to the first pipeline stage, which propagates the signal to the second stage and so on. The `shutdown()` method waits until every stage has finished its execution (which includes the execution of the possibly overridden `finish()` method) before returning to its caller.

The pipeline implemented for this experiment has a very simple communication scheme implicitly shown in Figure 5.6: each stage can only write to the next stage, and can only read data from the previous stage. There is no support for data forwarding (move data backwards in the pipeline) or bypassing (move data between non-adjacent stages.)

The lack of support for bypassing is intentional: implementing a complex communication network connecting all the cores can become expensive (specially when the number of cores increase.) In this simple implementation, the values need to go through each pipeline stage until it reaches the destination stage. For example, a value generated in Stage 1 that will be used by Stage 3 must go through Stage 2 before reaching its final destination. By carefully implementing the communication protocol between the stages, it is possible to mitigate the penalty for the lacking of bypass mechanisms.

Data forwarding moves the data *backwards* in the pipeline (*i.e*, *forward* in time.) jDSWP carefully generates the stages to avoid this kind of dependency since they are not supported.

5.4.3 The `funpipe.CommunicationQueue` Class

Although it is a known fact that memory is much slower than the processor, the lack of fast communication mechanisms between cores in modern x86 processors leaves memory as the fastest communication alternative between threads. Hence, one of the goals of our work was to identify efficient mechanisms for queue communication between pipeline stages.

Our framework included three different types of communication queue. The first implementation (called "Busy-waiting" throughout Section 5.6) makes no use of Java's `synchronized` mechanism, thus avoiding the overhead of suspending/resuming threads, as well as avoiding switching back and forth to/from the operating system.

The second implementation ("Lock-free") utilizes a lock-free FIFO queue. It relies on Java's synchronization methods (`synchronized`, `Object.wait()` and `Object.notify()`) to suspend the thread when appropriate (reading from an empty queue or writing to a full queue.) We named this implementation "Lock-free" since in the regular scenarios (reading a non-empty or writing to a non-full queue) `read`s and `write`s can occur concurrently and safely.

The last implementation ("Lock") marks all methods in the *funpipe.CommunicationQueue* class as `synchronized`, thus serializing all accesses to the communication channels.

To select between the three different types of communication queues without the need to recompile the source code, we implemented a CommunicationQueue factory which, based on a property specified in the command line, instantiates the correct queue.

These methods were implemented to evaluate the impact of the queue operation in the overall pipeline performance. Our assumptions were that the "Busy-waiting" implementation would outperform the other two implementations, and that the "Synchronized" implementation would be the slowest of the three. Contrary to our assumptions, all the implementations were roughly equivalent. For a detailed analysis refer to Section 5.6.5.

Table 5.1: The Selected Benchmarks

Benchmark	Maximum Speedup	Parallelization Changes
compress	1.73	Split the main loop so that there is a "compress" and a "decompress" stage
crypto.rsa	1.06	Split the main loop in three stages: "encrypt", "decrypt" and "check"; *System.array_copy()* statements were needed between stages
derby	1.56	Split the main loop into three pipeline stages, where the second and the third stages are parts of a inner loop from the main loop
mpegaudio	1.04	Removed the checksumming from the main loop; added an *Object.clone()* statement

5.5 Experimental Parallelization

We picked a subset of the SPECjvm2008 suite [4] programs to evaluate the potential of jDSWP. Programs were selected based on the availability of loop-carried free code fragments within its most significant loop.

Table 5.1 shows the benchmarks we used in our experiments. The column *Maximum Speedup* indicates the maximum achievable speedup for each benchmark. In other words, it is an upper bound for the achievable speedup due solely to parallelization. This number was obtained separately for each benchmark by first measuring the dynamic size of the candidate loop. Afterwards, we delimited the stages in the sequential source code and measured the dynamic size of the each stage (see Table 5.2). Since the dynamic size of the longest stage dominates the overall execution time of the parallelized loop, the maximum speedup is calculated by Equation (5.1) (Amdhal's Law), where $ParallelSize$ is the dynamic size of the longest stage and $SequentialSize$ is the dynamic size of the loop to be parallelized.

$$MaximumSpeedup_{parallelization} = \frac{1}{1 - \frac{ParallelSize}{SequentialSize}}$$

$$MaximumSpeedup_{parallelization} = \frac{SequentialSize}{SequentialSize - ParallelSize} \tag{5.1}$$

It is worthwhile to notice that Section 5.6 do present some performance numbers above this theoretical upper bound. Although at first glance this may seem odd, it is perfectly explainable: the upper bound calculated by Equation 5.1 refers to theoretical performance limit achievable using *exclusively* the parallelization strategy (thus the

Table 5.2: Size of the Pipeline Stages Compared to the Sequential Loop

Benchmark	Stages		
	1	2	3
compress	<1%	58%	42%
crypto.rsa	5%	95%	<1%
derby	1%	64%	35%
mpegaudio	96%	4%	N/A

name $MaximumSpeedup_{parallelization}$). Modern JVMs, however, optimize the execution with runtime optimizations like *Just-in-Time* (JIT) Compilation. Therefore, when running parallelized applications with JIT enable, the overall runtime performance speedup included benefits from both JITing and parallelization.

Next two subsections briefly explain how we parallelize two out of the four studied applications. The other two benchmarks were parallelized in a similar fashion.

5.5.1 Parallelizing the `mpegaudio` benchmark

As it can be seen in Figure 5.8 (a), mpegaudio's main loop is very simple. First, a chunk of data is read from the input stream into a buffer (an array of fixed size). Then, the data is converted into an MPEG audio format. Finally, the program calculates the CRC of the converted data.

The parallelized loop is shown in Figure 5.8 (b). The first stage is responsible for reading the data as well as converting it to the MPEG format. The second stage is responsible for calculating the CRC (the `updateCRC32()` method invocation) of the encoded data, which is communicated through the available communication channel (the communication statements are underlined in Figure 5.8 (b)). Although very simple, this parallelization was effective, yielding up to 4% speedup in non-JITted executions (which is the $MaximumSpeedup_{parallelization}$.)

Mpegaudio was parallelized in just a few minutes. The only needed change was the invocation of method `Object.clone()` (in italic in Figure 5.8 (b)) to clone the buffer returned by `SampleBuffer.getBuffer()` so that it could be used by the second stage of the pipeline without data hazards.

The need to duplicate buffers was a recurrent issue. Programmers still think very carefully about memory management, even in garbage collected environments. This decision makes sense when memory is a scarce resource. In today's world, however, many systems over 2 GB of memory. Programmers need to learn to carefully *waste*[1] memory on some program spots to avoid false dependencies between unrelated code and ease the

[1]carefully waste, not leak

```
 while (decodedFrames < FRAME_LIMIT &&
        (h = stream.readFrame()) != null)
{
    decodedFrames++;
    updateCRC32(
      crc,
      ((SampleBuffer)
         decoder.decodeFrame(h,stream)
      ).getBuffer()
    );
    stream.closeFrame();
}
```

(a) Sequential

```
Pipeline.Stage s1 = new Pipeline.Stage() {
    @Override
    public void stage() throws Exception {
        CRC32 crc = (CRC32) input();
        short[] buffer = (short[]) input();
        updateCRC32(crc, buffer);
    }
};

Pipeline p = Pipeline.createPipeline(s1);

 while (decodedFrames < FRAMES_LIMIT &&
        (h = stream.readFrame()) != null) {
    decodedFrames++;
    p.input(crc);
    p.input(
        ((SampleBuffer) decoder.
              decodeFrame(h, stream))
        .getBuffer().clone());
    stream.closeFrame();
}

p.shutdown();
stream.close();
```

(b) Parallel

Figure 5.8: mpegaudio

```
for ( int i = 0; i < LOOP_COUNT; ++i )
{
    for ( int j = 0; k < FILES_NUMBER; ++j )
    {
        Source source = SOURCES[j]
        Buffer comprBuffer, decomprBuffer;
        comprBuffer = compress(source);
        decomprBuffer =
            uncompress(comprBuffer);
        Context.getOut().print(...);
        Context.getOut().print(...);
        Context.getOut().print(...);
    }
}
```

(a) Sequential

```
Pipeline.Stage s1 = new Pipeline.Stage() {
    @Override
    public void stage() throws Exception {
        int j = (Integer)this.input();
        PrintStream out = (PrintStream) this.input();

        Source source = SOURCES[j];
        Buffer comprBuffer = compress(source);
        this.output(comprBuffer);
        this.output(out)
    }
};

Pipeline.Stage s2 = new Pipeline.Stage() {
    @Override
    public void stage() throws Exception {
        Buffer comprBuffer = (Buffer) input();
        PrintStream out = (PrintStream) input();

        Buffer decomprBufer =decompress(comprBuffer);

        out.print(...);
        out.print(...);
        out.println(...);
    }
};

Pipeline p = Pipeline.createPipeline(s1, s2);

for ( int i = 0; i < LOOP_COUNT; i++) {
    for ( int j = 0; j < FILES_NUMBER; j++) {
        p.input(j);
        p.input(Context.getOut());
    }
}

p.shutdown();
```

(b) Parallel

Figure 5.9: compress

Table 5.3: Setup Configurations

CPU	RAM	Operating System	HT?
Core i7 975 3.33GHz	12 GB	Windows Seven	Yes
			No
		Ubuntu 9.10	Yes
			No
Core2 Duo P8600 2.4 GHz	2 GB	Windows Seven	No
		Ubuntu 9.10	No
Atom N260 1.6GHz	1 GB	Windows Seven	Yes
		Ubuntu 9.10	Yes

parallelization effort.

5.5.2 Parallelizing the `compress` benchmark

Figure 5.9 (a) shows compress' main loop's structure. It works by first compressing the data in a temporary buffer ($comprBuffer$), uncompressing it to yet another temporary buffer ($decomprBuffer$), and outputting details about them.

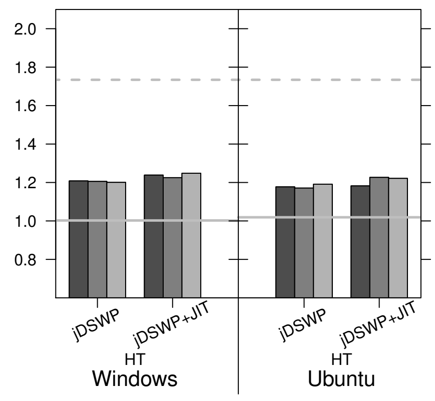
This simple loop led to a three stage pipeline (see Figure 5.9 (b)), with the first stage being responsible to iterate the original loops. The second stage performs the actual data compression, with the third stage being responsible to decompress the data.

This benchmark would be amenable to DOALL parallelization if the inputs were not read from a file (this read operation is not shown on Figure 5.9 (a)). Compress is a good example of why DOPIPE parallelization is powerful: it allows loops with IO operations to be parallelized since the IO operations will be deterministically invoked. Nevertheless, parallel loops should not perform IO operations if squashes are possible due to the difficulty (and sometimes impossibility) of rolling back IO operations.

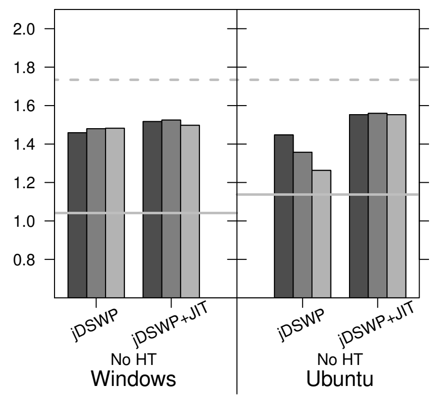
5.6 Experimental Results

The different hardware setups used to collect the experimental data are listed on Table 5.3. The experiments were run using similar software configurations. All the operating systems were 64-bit versions *except* on the Atom processor, as the available system was not 64-bit capable. All data were collected using Sun's 32-bit JDK 1.6.0_18. By using the 32-bit JVM, we ensured that all experiments were subject to the same software limitations.

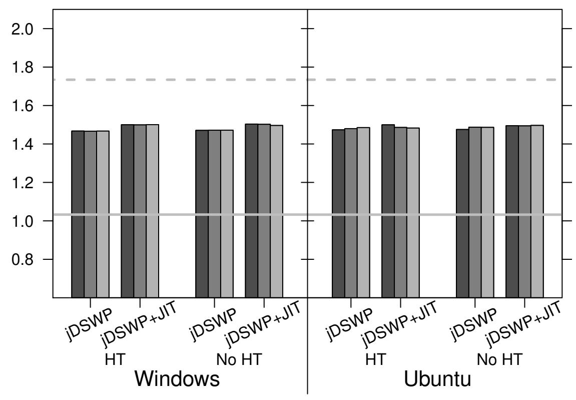
Sections 5.6.1, 5.6.2, 5.6.3 and 5.6.4 analyze each benchmark individually. The speedup given in all charts are always compared to a sequential, non-JITted execution of the



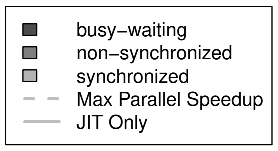
(a) Atom



(b) Core2



(c) Core i7



(d) Legend

Figure 5.10: compress

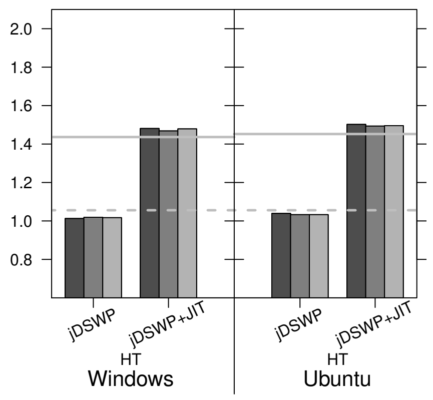
benchmark. Section 5.6.5 analyzes the role that the different communication queue implementations played on the overall parallel program performance.

5.6.1 `Compress` Evaluation

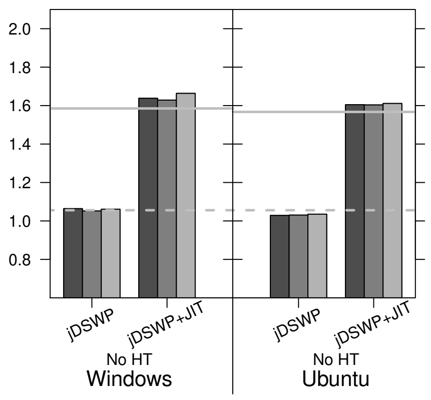
The compress benchmark parallelization is previously explained in Section 5.5.1. Figure 5.10 shows the results we collected for this benchmark.

The first interesting result we collected is the low performance speedup due to JIT compilation (the solid grey line on the charts.) On the other hand, the parallel speedup upper bound is high, which means that, for this benchmark, jDSWP parallelization should be preferred over JITing if this was a choice the programmer had to make.

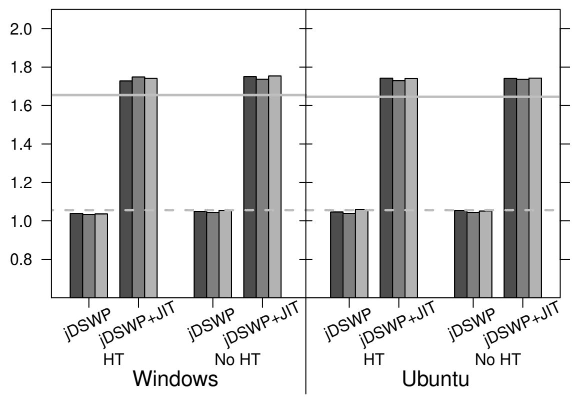
For the Atom platform (Figure 5.13 (a)), the actual program speedup shown at runtime



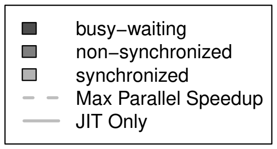
(a) Atom



(b) Core2



(c) Core i7



(d) Legend

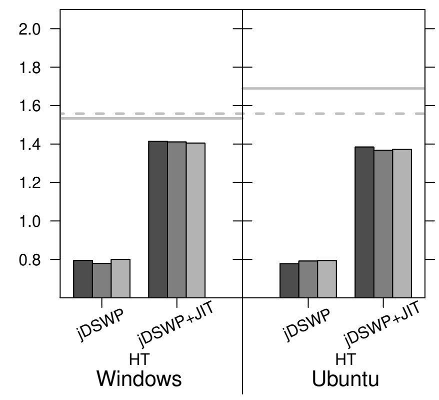
Figure 5.11: crypto.rsa

is much lower than the potential speedup. This is expected as this benchmark employs a three-stage pipeline and the system only have two processing elements. Actually, we expected that the runtime performance of the parallelized application would be slower than the sequential version, thus the speedup is somewhat unexpected.

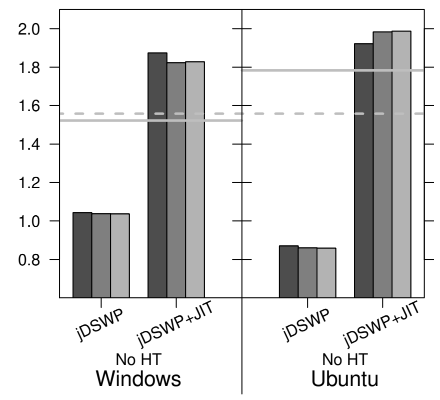
The Core2 (Figure 5.13 (a)) and Core i7 (Figure 5.13 (a)) performance figures, while being better than observed in the Atom platform, are still below the maximum potential speedup. This is possibly due to communication overhead. Nevertheless, the parallel performance outperforms JIT compilation.

5.6.2 `Crypto.rsa` Evaluation

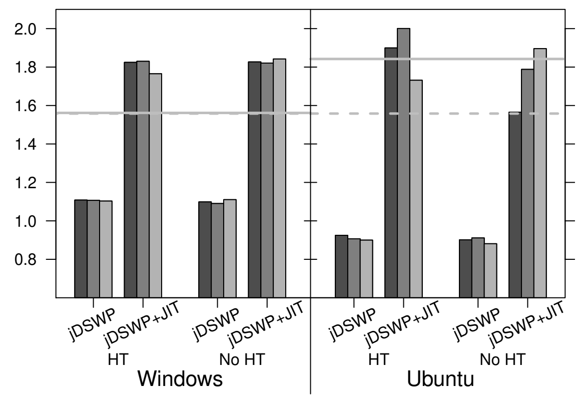
`Crypto.rsa` exhibits a low parallel speedup upper bound (see Figure 5.11.) This arises



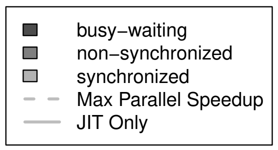
(a) Atom



(b) Core2



(c) Core i7



(d) Legend

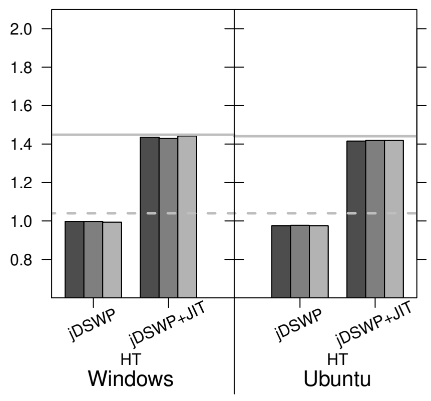
Figure 5.12: derby

from the low dynamic coverage of the selected loop. This benchmark contains long sequential steps that are not amenable to jDSWP parallelization strategy. In this benchmark, the potential benefits from JITing are higher.

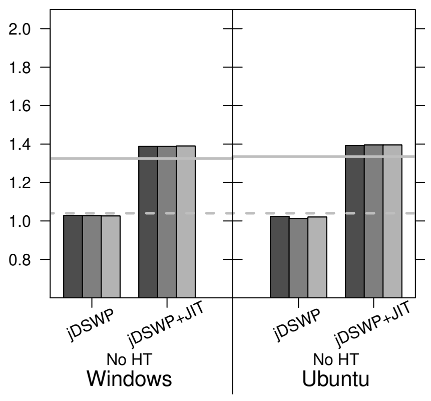
Running the parallelized benchmarks without JIT compilation led to no observable benefit or, even worse, execution slowdowns in all the platforms. Nevertheless, the combination of JIT and parallelization overcame the benefits of JITing alone. Moreover, JIT potentialized the benefits from parallelization, specially on the more powerful Core i7 platform.

5.6.3 `Derby` Evaluation

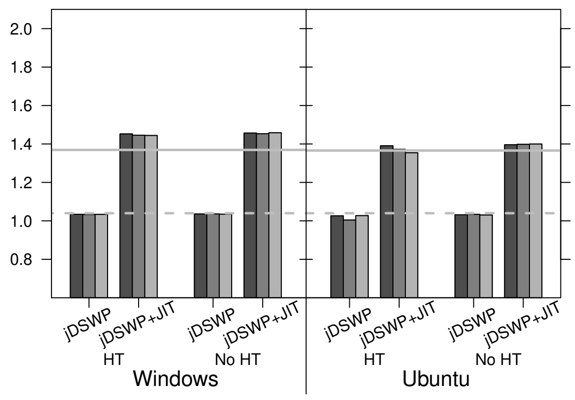
`Derby` is yet another benchmark that was parallelized using a three-stage pipeline, as



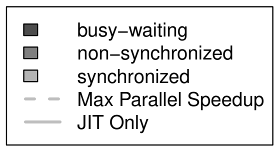
(a) Atom



(b) Core2



(c) Core i7



(d) Legend

Figure 5.13: mpegaudio

`compress` was. This time, however, the predicted slowdown on the Atom platform (Figure 5.12 (a)) was visible. `Derby` features a more unbalanced pipeline when compared to `compress` (which benefited from the three-stage parallelization). Besides, `derby`'s smallest stage accounts for at least 1% of the loop's total workload. Therefore, it is likely that the effects of task switching are more apparent, thus significantly degrading performance.

The Core2 system (Figure 5.12 (b)), like the Atom platform, contains two processing elements. Differently from the Atom, the Core2 features two independent execution cores. Besides, the Core2 system is not designed for low-power computing, thus performing faster than the power-efficient (but slower) Atom. When run under Windows 7, the parallel, non-JITed `derby` performance was similar to the baseline execution. Under Ubuntu, however, the parallel, interpreted execution performed much worse than the baseline.

The Core i7 (Figure 5.12 (c)) platform experienced the same issues experienced by the core2 system. Nevertheless, the more powerful execution cores delivered more performance at runtime than its predecessor.

5.6.4 `Mpegaudio` Evaluation

The `mpegaudio` parallelization was discussed in Section 5.5.1. The experimental results are shown in Figure 5.13.

As previously discussed, the$MaximumSpeedup_{parallel}$ upper bound is small as the only parallelization opportunity observed was moving the CRC checksumming from the application's most significant loop to its own pipeline stage. A similar effect would be achieved by invoking the `updateCRC32()` method asynchronously.

The parallelization was effective on every system but the Atom. On the Core2 system, as well as on the Core i7 system, the parallelization effects were increased when used with JIT compilation. Without JIT compilation, the performance on those systems were roughly equivalent to the sequential execution with JIT enabled.

5.6.5 Evaluating the Different Communication Queues Implementations

This paper evaluated three different communication queues implementation. As described in Section 5.4.3, our assumptions were that a communication queue which does not rely on expensive synchronization mechanisms (*e.g.*, suspending a thread upon necessity) would perform faster than other implementations relying on those heavy supports.

A thorough review of the results presented in this Section, however, pinpoints that the communication queue implementation is nearly irrelevant to the overall parallel performance.

It is possible that the thread imbalance (see Table 5.2) actually helped to mitigate the effects of the communication queue implementation: a less loaded thread is suffers most communication overhead, while the more heavily loaded threads are performing useful computations. Therefore, the less loaded threads hide the communication latency from other (busier) threads.

5.7 Conclusions

In this paper, we have proposed jDSWP, a simple yet effective source-level parallelization technique for Java applications. Our experimental data shows that, even in the absence of hardware support for inter-thread communication, program parallelization is possible.

The experimental data we collected also shows that the communication channel's implementation does not play a big role on the parallel application's overall runtime performance. Nevertheless, to avoid unnecessary communication overheads, the communication channels should be carefully and efficiently implemented.

We also observed the need for *Synergy* between the parallel application and the runtime environment executing it. The processor is not the only parameter that needs to be evaluated when a program is to be parallelized. Other variables (*e.g.*, the operating system and the Java Virtual Machine) are just as important. Therefore, static parallelization (as done in the paper) is not the best solution for program parallelization if the resulting application is to be deployed across multiple different environments.

References

[1] Bocchino Jr RL, Adve VS, Adve SV, Snir M. Parallel Programming Must Be Deterministic by Default. *HotPar 2009: First USENIX Workshop on Hot Topics in Parallelism*, 2009; 836–884.

[2] Lee EA. The Problem with Threads. *Technical Report*, University of California, Berkeley 2006.

[3] Ottoni G, Rangan R, Stoler A, August DI. Automatic Thread Extraction with Decoupled Software Pipelining. *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, 2005; 105–118, doi:http://dx.doi.org/10.1109/MICRO.2005.13.

[4] Standard Performance Evaluation Corporation. SPECjvm2008. http://www.spec.org/jvm2008 2008.

[5] Lundstorm SF, Barnes GH. A Controlable MIMD Architecture. *ICPP '80: International Conference on Parallel Processing*, 1980; 19–27.

[6] Cytron R. DOACROSS: Beyond Vectorization for Multiprocessors. *ICPP '86: International Conference on Parallel Processing*, 1986; 836–884.

[7] Padua DA. Multiprocessors: Discussion of Some Theoretical and Practical Problems. *Technical Report*, Department of Computer Science, University of Illinois, Urbana, IL 1979.

[8] Zhang X, Navabi A, Jagannathan S. Alchemist: A Transparent Dependence Distance Profiling Infrastructure. *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, 2009; 47–58, doi:http://dx.doi.org/10.1109/CGO.2009.15.

[9] Terracotta, Inc. *The Definitive Guide to Terracotta*. Apress, 2008.

[10] Gosling J, Joy B, Steele G, Bracha G. *The Java^{TM} Language Specification.* 3rd edn., Addison Wesley, 2005.

[11] Lindholm T, Yellin F. *The Java^{TM} Virtual Machine Specification.* 2nd edn., Prentice Hall, 1999.

[12] Danelutto M, Teti P. *Computational Science – ICCS 2002*, chap. Lithium: A Structured Parallel Programming Environment in Java. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2002; 844–853.

[13] Raman E, Ottoni G, Raman A, Bridges MJ, August DI. Parallel-stage Decoupled Software Pipelining. *CGO '08: Proceedings of the 6th Annual IEEE/ACM International Aymposium on Code Generation and Optimization*, 2008; 114–123, doi: http://doi.acm.org/10.1145/1356058.1356074.

[14] Rangan R, Vachharajani N, Ottoni G, August DI. Performance Scalability of Decoupled Software Pipelining. *ACM Transactions on Architecture and Code Optimization* 2008; **5**(2):1–25, doi:http://doi.acm.org/10.1145/1400112.1400113.

[15] Vachharajani N, Rangan R, Raman E, Bridges MJ, Ottoni G, August DI. Speculative Decoupled Software Pipelining. *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007; 49–59, doi: http://dx.doi.org/10.1109/PACT.2007.66.

[16] Akkary H, Driscoll MA. A Dynamic Multithreading Processor. *MICRO 31: Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998; 226–236.

[17] Akkary H, Rajwar R, Srinivasan ST. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003; 423.

[18] Srinivasan ST, Rajwar R, Akkary H, Gandhi A, Upton M. Continual Flow Pipelines. *SIGPLAN Notes* 2004; **39**(11):107–119, doi:http://doi.acm.org/10.1145/1037187.1024407.

[19] Hilton AD, Nagarakatte S, Roth A. iCFP: Tolerating All-level Cache Misses in In-Order Processors. *HPCA 2009: IEEE 15th International Symposium on High Performance Computer Architecture*, 2009; 431–442.

[20] Bhowmik A, Franklin M. A General Compiler Framework for Speculative Multithreading. *SPAA '02: Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, 2002; 99–108, doi:http://doi.acm.org/10.1145/564870.564885.

[21] Wang K, Franklin M. Highly Accurate Data Value Prediction Using Hybrid Predictors. *MICRO 30: Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997; 281–290.

[22] Ceze L, Tuck J, Montesinos P, Torrellas J. BulkSC: Bulk Enforcement of Sequential Consistency. *SIGARCH Computer Architecture News* 2007; **35**(2):278–289, doi:http://doi.acm.org/10.1145/1273440.1250697.

[23] Torrellas J, Ceze L, Tuck J, Cascaval C, Montesinos P, Ahn W, Prvulovic M. The Bulk Multicore Architecture for Improved Programmability. *Communications of the ACM* 2009; **52**(12):58–65, doi:http://doi.acm.org/10.1145/1610252.1610271.

[24] Hammond L, Hubbert BA, Siu M, Prabhu MK, Chen M, Olukotun K. The Stanford Hydra CMP. *IEEE Micro* 2000; **20**(2):71–84.

[25] Kim D, Yeung D. A Study of Source-level Compiler Algorithms for Automatic Construction of Pre-Execution Code. *ACM Transactions on Compututer Systems* 2004; **22**(3):326–379, doi:http://doi.acm.org/10.1145/1012268.1012270.

[26] Krishnan V, Torrellas J. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers* 1999; **48**(9):866–880.

[27] Marcuello P, González A. Clustered Speculative Multithreaded Processors. *ICS '99: Proceedings of the 13th International Conference on Supercomputing*, 1999; 365–372, doi:http://doi.acm.org/10.1145/305138.305214.

[28] Sohi GS, Breach SE, Vijaykumar TN. Multiscalar Processors. *SIGARCH Computer Architecture News* 1995; **23**(2):414–425, doi:http://doi.acm.org/10.1145/225830.224451.

[29] Zilles C, Sohi G. Master/Slave Speculative Parallelization. *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society Press: Los Alamitos, CA, USA, 2002; 85–96.

Capítulo 6

Cache-Based Cross-Iteration Coherence for Loop Parallelization

Prólogo

Depois dos estudos de *traces* (Capítulos 3 e 4) e da avaliação dos potenciais benefícios que TLS pode propiciar (Capítulo 5), este Capítulo apresenta uma arquitetura que possibilita o uso seguro de técnicas automáticas de paralelização, oferecendo o suporte necessário para a detecção de violações à semântica do programa original (*sequencial*), bem como automaticamente fazendo a comunicação de dados de memória entre os processadores.

O artigo abaixo foi submetido ao *Symposium on Parallelism in Algorithms and Architectures* (SPAA) para apresentação em 2011.

Abstract

This paper proposes an architecture model that enables the parallel execution of loop iterations in multicore architectures, using DOPIPE based compilation techniques (e.g. DSWP). The proposed architecture supports runtime detection of sequential consistency violations across parallel loop iterations, while allowing for light-weight commit and squash operations. This is achieved by adding extra tag bits to the cache, and a small separate logic. No changes on pre-existing cache-coherence protocols are required. The impact of the extra cache bits is fairly small, and can be amortized as transistor count continues to increase. To evaluate such impact, cache size requirements per iteration have been measured for SPEC CINT 2000 loops, revealing a dynamic footprint compatible with the caches found in modern processors.

6.1 Introduction

Multicore chips have been adopted as a solution to the power-wall limits encountered in modern microprocessor clock rates. From the chip manufacturers' perspective, multicores are an effective solution, as they offer the possibility of improving transistor count, while maintaining power density/consumption under control. They are now used in a range of devices from mid-level laptops to servers, a trend which seems to be here to stay.

For servers, multicore are a natural solution, as they enable more tasks to be handled at a single time, thus increasing the overall system throughput. Moreover, through virtualization they allow a single physical machine to host multiple virtual servers, thus saving space and decreasing data-centers' energy requirements.

In desktops, multicore helps to improve the overall system usability. Users can have some heavy-weight background tasks, such as powerful anti-virus and intelligent Internet firewalls running alongside with productivity applications without any perceivable slowdowns due to processor overload. Unfortunately, there is a limit on the number of ready process in a desktop system, a side effect of how many tasks one user can simultaneously manage. After this limit has been reached, increasing the core count will simply not benefit desktop users at all. This can, of course, can be overcome by program parallelization, either manually or automatically.

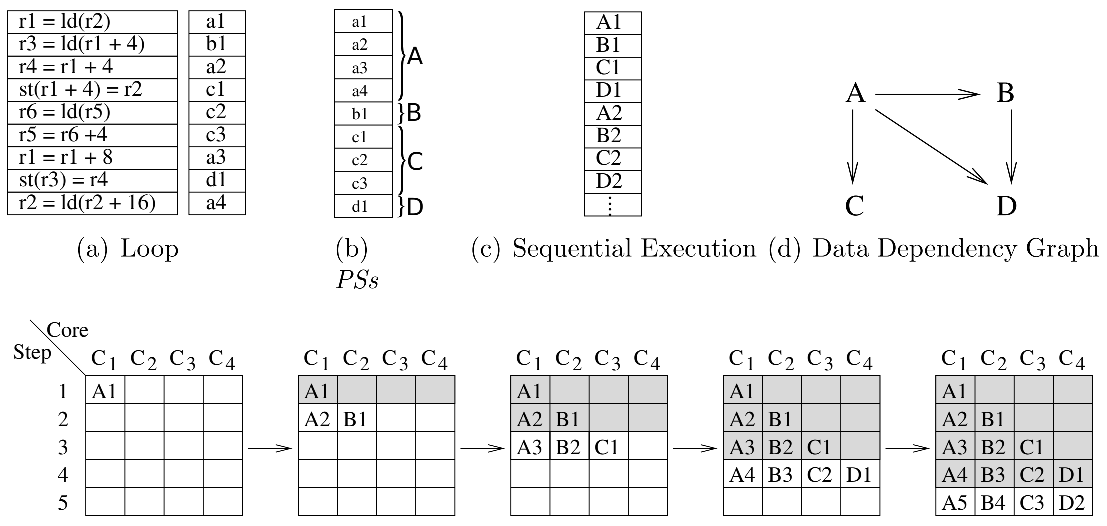
It is widely known that manual program parallelization, or thread programming, is hard, error-prone and difficult to debug [3, 11]. Despite of libraries that aid program parallelization [7, 21] and debugging [12], the average computer programmer is just not ready for the challenge. On the other hand, decades of parallel compiler research, although offered good solutions to scientific data-intensive programs, has not been able to effectively parallelize general-purpose programs.

Many techniques which combine parallel compiling techniques with architecture support have been proposed to enable general-purpose program parallelization. DSWP [16] aims at extracting parallel loop fragments and spreading them into cores using communicating queues. Although it has shown very good performance numbers, it is somewhat bounded by the ability of the technique to detect aliasing at compile/run time. Hardware support for Thread-Level Speculation (TLS) has also been proposed [5, 8, 24] in order to enable compilers to extract more parallelism by aggressively reordering memory operations. This hardware supports detecting memory ordering violations at run-time and squashing the offending threads. However, none of these approaches is suitable for DOPIPE [17]-based techniques, as they can either yield incorrect results or be very inefficient. Moreover the above cited techniques require expensive changes in the cache or the cache-coherence protocol to be effective.

In this paper we propose an architecture model that enables the execution of parallel loop code, generated using any DOPIPE [17]-based technique (*e.g.*, DSWP). Our model supports the runtime detection of sequential consistency violations in the parallelized code, while allowing efficient commit and squash operations, without the need to traverse any auxiliary structures or increase memory burst traffic. This is achieved without changing the hardware of the cache controller or the cache-coherence logic.

Our technique basically adds a few extra tags into the program cache lines, requiring a fairly small logic to support its operation. This hardware does not affect the coherence logic. The added tags are used for three major purposes: (i) keeping track of the status of speculative data; (ii) allowing for loop-carried data versioning; (iii) enforcing memory data dependencies. The impact of the extra bits is fairly small, and will be certainly amortized as transistor counts continue to increase with technology. The proposed scheme allows efficient detection of runtime memory ordering violations, while allowing true memory dependencies (read after write) to be honored. Moreover, this scheme can be built on top of most invalidation-based cache coherence protocols without affecting it. In addition, we do have a formal proof of the mechanisms' correctness, which can be found on Appendix 6.C. To evaluate the impact of multiversioning, we measured the dynamic cache size pressure resulting from relevant SPEC CINT 2000 loops, and show that they are smaller then the cache sizes found in modern processors.

This paper is divided as follows. Section 6.2 describes a simple, DOPIPE-based parallelization technique, and Section 6.3 describes an architecture with support for correct execution of the parallelized loops. In Section 6.4, some loops of the SPEC CINT 2000 were analyzed in order to determine if there would be any pressure on the cache size. Section 6.5 presents previous work that is related to this paper, and Section 6.6 presents our conclusions and research directions. Appendix 6.A provides examples that show how the memory data forwarding support works, and how miss-speculation is detected, and



(a) Loop (b) *PSs* (c) Sequential Execution (d) Data Dependency Graph

(e) Parallel Execution

Figure 6.1: A loop, its *PSs* and the pipelined execution.

Appendix 6.B presents our strategy to detect and select loops. Appendix 6.C presents the versioning mechanism's proof of correctness.

6.2 Loop Parallelization and Execution

DOPIPE loop parallelization works by identifying sequences of *parallel sections* (*PS*), as depicted in Figure 6.1. *PSs* need not to be contiguous instructions in the loop's instruction stream. Finding these sections is usually done by identifying Strongly Connected Components (SCC) [25] of the Control/Data-Flow Graphs, but any other heuristic can be used.

The loop in Figure 6.1(a) is divided into *PSs* which were computed using SCCs. Our model assumes that executing the original sequential loop is equivalent to executing the reordered loop shown in Figure 6.1(b), in which the instructions were rescheduled so as to keep instructions belonging to each *PS* together. For example, instructions a1, a2 and a3 are grouped together into a node labeled A. Notice on Figure 6.1(b) instructions belonging to each *PS* grouped together.

Figure 6.1(c) depicts the sequential execution of the loop. In that Figure, Ai represents the execution of *PS* A at iteration i. The same applies for Bi, Ci and Di.

The reordered loop scheduling is not arbitrary. Figure 6.1(d) shows the Data Dependence Graph (DDG) for the loop's *PSs*' static (*i.e.*, register) dependencies. An edge $X \rightarrow Y$ in the DDG means that node X produces a value consumed by Y. Notice that the DDG contains no cycles. As DSWP, our model assumes the DDG is acyclic, meaning that it can be topologically sorted [6]. Any topological sort for the DDG represents a valid scheduling for the loop. In the example, there are 3 different topological sorts; in theory any of them is a valid scheduling (see Section 6.3.3).

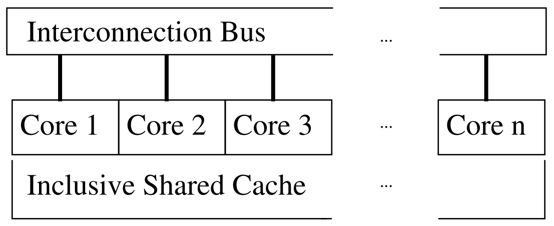
Figure 6.1(e) illustrates how the loop's execution evolves on a quad-core system implementing our execution model. Initially, it executes the first step of the parallel execution, which only contains A1 (*i.e.*, iteration 1's *PS* A). After step one completes, step two, comprised of A2 and B1, starts. This is followed by step three (A3, B2, C1). The fourth step (A4, B3, C2, D1) is the first one to fill all the cores with one loop stride. If this step terminates, all the *PSs* for iteration 1 (of the sequential loop) will have been (successfully) executed, meaning that iteration 1 has completed (*i.e.*, iteration 1 commits). Iteration 2 will commit in the following step. This process continues until the last *PS* of the last iteration completes its execution and commits.

This example assumes that the speculation (*i.e.*, the parallel execution) was successful. For this specific example, using the topological sort $A \rightarrow B \rightarrow C \rightarrow D$ might actually work: since the DDG is built with static dependencies, there are no guarantees that the parallel code will preserve the memory ordering which occurs in the sequential execution.

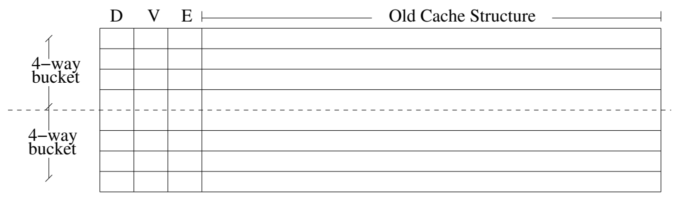
In principle, dynamic (*i.e.*, memory) dependencies could be identified with profiling. Perfect profiling information (*i.e.*, information about every runtime dependency observed by the program) requires heavy-weight profiling, and is generally not applicable to non-academic benchmarks and test-cases. As noted by [5,24], conservative code parallelization is not a solution either, as it will miss several optimization opportunities due to false dependencies. Moreover, if the conservative analysis misses a dependency, the resulting parallel program would run fast but incorrectly.

The ability to recover from miss-speculations is central to architectures intended for parallel execution of sequential code. Despite the existence of previous work in this area (see Section 6.5), most proposed solutions have either (overly) complicated hardware support for miss-speculation handling (not good if project complexity is to remain low) or assume that perfect profiling information is available.

To bridge this gap, we propose the architecture described in the Section 6.3. There, we propose simple additions to the cache that allow a processor to efficiently identify miss-speculations due to memory ordering violations. The same mechanism allows the processor to identify runtime dependencies that do not violate the loop's sequential memory ordering. The proposed mechanism is light enough to allow support for efficient committing and squashing while not introducing any significant overhead to the cache controller. Moreover, it can be built on top of most invalidation-based cache coherence protocols.



(a) Architecture



(b) New Cache Line

Figure 6.2: Multicore System

6.3 Architecture

Figure 6.2(a) depicts a hardware design implementing our execution model. It features n cores connected by an Interconnect Bus (IB) similar to current multicore architecture implementations, and an inclusive, coherent, cache hierarchy employing an invalidation-based cache coherence protocol.

Besides the coherence messages (*e.g.*, downgrading), the IB has another role on our execution model. As indicated in Section 6.2, each step comprises *PSs* from different iterations executing on distinct cores. Our model assumes the hardware is capable of synchronizing the start of the *PSs*'s execution on every core, contrasted to DSWP [16] which assumes decoupled execution of the *PSs*. Since core synchronization might cause performance degradation, we are currently investigating an asynchronous model, which is left for future work (see Section 6.6).

Our model assumes that register dependencies are honored statically by code generation, so it only needs to assert memory dependencies, which is accomplished by augmenting the cache lines with Cross Iteration Coherence Tags. The following Sections describe the architecture in details.

6.3.1 The Iteration ID

Our execution model runs different *PSs* from different loops at the same time, thus requiring the ability to separate iterations apart. The simplest scheme to achieve that is to generate an unique ID for each iteration that starts execution. This approach may not be the best solution as it requires an upper bound on the iteration ID so that the ID can be represented in the hardware. Moreover, to check for conflicts, the system will need several comparators, incurring in a potentially high area (and power) overhead, and possibly of affecting the cache circuit's critical path.

Due to its synchronous nature, our execution model presents an upper bound on the number of *active* iterations at any step of an n-core system: there may be at most n different active loop iterations at any given step. Because of this feature, it is possible to use a $log_2(n)$-bit wide iteration ID. We chose an n bit tagging scheme which represents the iteration ID with a single bit on. As we will explain in Section 6.3.3, this decision simplifies the implementation of commit and squash operations.

Revisiting Section 6.2's example (Figure 6.1(e)), the first iteration (which starts at step 1) is labeled 1000. Iteration 2 (starting at step 2) is labeled 0100. Similarly, iteration 3 is labeled 0010. After iteration 4 (whose ID is 0001), the IDs are recycled, thus iteration 5 is labeled 1000 etc. This recycling is possible since there are at most 4 iterations active per step, and thus iterations 1 (ID 1000) and 5 (ID 1000) will never be active simultaneously. For an n-core machine implementing our model, the ID for any iteration i at any loop can be calculated by Equation 6.1. In practice, each core just need to know its current iteration ID as the next ID is deterministically produced by a rotate right.

$$ID(i) = 1000_b >> ((i-1) \bmod n) \tag{6.1}$$

Recycling IDs has a drawback: it creates ordering issues among the IDs. In the example, is iteration 0001 logically before or after iteration 1000? The answer depends on the current meaning of the IDs. If 1000 represents iteration 1 and 0001 represents iteration 4, then $1000 < 0001$. However, if 1000 represents iteration 5, then obviously $1000 > 0001$. To address this issue, we rely on the fact that, in our execution model, a core's ID determines the core's relative position related to the other cores.

For instance, Core 1 is aware that there can be no on-the-fly iteration logically *after* its current iteration. Likewise, Core 2 knows that there is at most one on-the-fly iteration logically after its current iteration, and so on, up to core 4, which is aware that every on-the-fly iteration is logically *after* its current iteration. With that in mind, it is always possible to determine, for each core, the set of *previous* (and *later*) on-the-fly iterations.

For example, assume that Core 3 is currently running iteration 0010 and it needs to decide if iteration 0100 happened before or after 0010. Core 3 knows that two iterations *after* 0010 have already started, so, since every ID is unique at any given point in time, it knows that iterations 0001 and 1000 are logically *later* than its current (0010) iteration. Since iteration 0100 is neither 0001 nor 1000, than it must be logically *earlier* than 0010.

V				ADDR
0	1	0	0	X
1	0	0	0	Y
0	0	1	0	Y

Figure 6.3: Core 0's Written Data

6.3.2 The Coherence Tags

Figure 6.2(b) shows two buckets of a 4-way set associative cache. The Figure represents the Coherence Tags as three n-bit tags named D, V and E. Each tag plays a distinct role in our system explained in Sections 6.3.2, 6.3.2 and 6.3.2.

V: Iteration ID

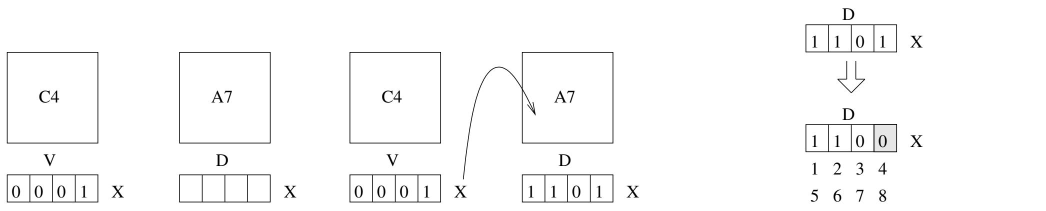
V is used by the system to indicate that a data has been speculatively written to. This field is filled with the iteration ID of the writing core.

Figure 6.3 shows three examples of cache entries and their respective V tag. Assume, for example, that Figure 6.3 represents the state of Core 3's cache of Figure 6.1 after executing D1, D2 and D3. From Figure 6.3, it is possible to tell that both iterations 1 (1000) and 3 (0010) stored into Y, while iteration 2 (0100) stored into address X. The example shows that the same address (Y) can now be present in more than one cache line. While they have the same address, their *speculative* address is different, since one data was produced at iteration 1 (1000) and the other is at iteration 3 (0010). We will use the notation Y_{1000} to represent the speculative address Y in iteration 1000. Y_{0010} represents the speculative address Y in iteration 0010.

Any value at V other than 0000 indicates a speculative cache line which can be neither evicted from the cache, nor written back to memory. The latter should be clear: no speculative data should be allowed to drain to memory, while the former cannot happen because it would require a write back to memory.

D: Speculative Dependency

D is used to detect memory ordering violations. The system uses D to hold information about the interval in which a cache line is speculative. For example, Figure 6.4(a) shows the V tag for data X after C4 writes it. A7 reads X, which creates the speculative address X_{0010}. The D tag for X_{0010} is shown on Figure 6.4(b), and it represents the speculation interval for X_{0010} (in the example, the speculation interval is $[4, 6)$). Any modification to X at any iteration in the speculation interval triggers a miss-speculation recovery. Memory ordering violations are detected by Equation 6.2.



(a) Write at C4 (b) Read at A7 (c) After Iteration 4 Commits

Figure 6.4: Keeping Track of Dependencies

$$D \quad \textbf{AND} \quad ID(i) \neq 0 \tag{6.2}$$

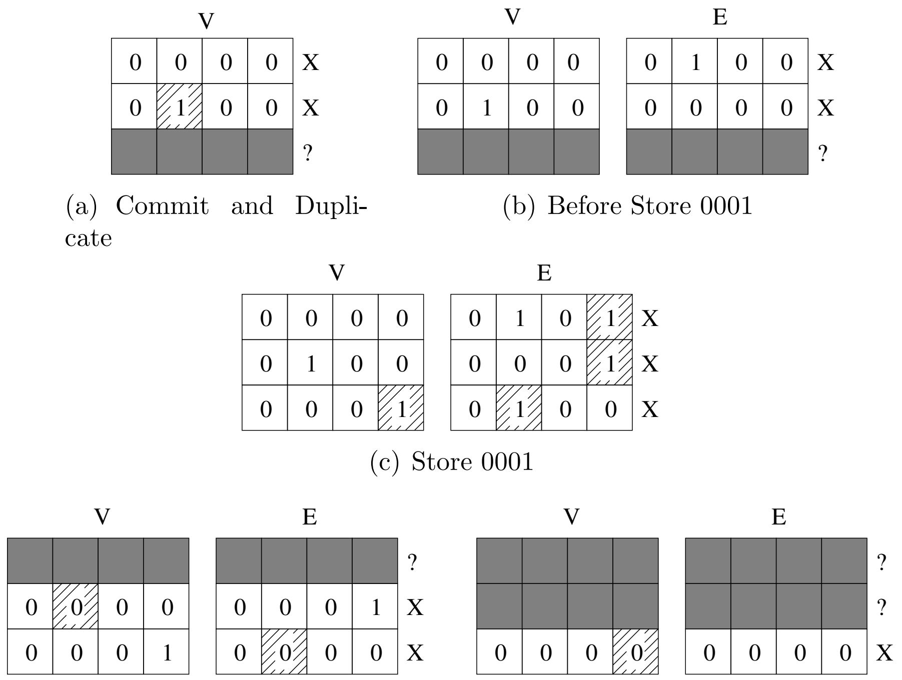
Recycling IDs might create an issue for D though. After the bits are set, the hardware has no notion of whether ID 0001 now represents iteration 8, or if it still represents iteration 4. If it now represents iteration 8, any stores to X_{0001} occurred logically *after* the read of X_{0010} and should not cause the hardware to signal a memory ordering violation.

To overcome this, the hardware *clears* the D's bits upon committing an iteration. For example, if iteration 0001 commits, the hardware *must* clear the bits in D that represent the committing iteration. By assuring this is done *before* the next step starts, the ID can be safely recycled.

E: Duplicated Cache Entries Tag

Keeping multiple versions of speculatively written addresses avoids anti-dependencies among different iterations just like register renaming does for out-of-order processors. It does, however, create an additional issue for an architecture implementing our execution model: prior to the commit of an iteration that stored to location X, there may be a previous, non-speculative version of X in the cache (see Figure 6.5(a) for an example). The hardware must address this situation to avoid duplicated non-speculative data in the cache and other issues that might arise from this.

One alternative solution is to burst-invalidate every line that becomes non-speculative. This naive solution imposes a large burden on the memory controller, as it must be able to handle (possibly) large burst write requests, as every evicted dirty line must be written back to memory. In addition, burst invalidated, non-speculative data might be requested by any processor, which would require the system to fetch the data from memory, thus affecting performance negatively. Another alternative would be to look up, upon commit, for any non-speculative data already in the cache and invalidate the lines which could cause a conflict. This solution is also not desirable since it would be hard to efficiently implement it (both performance- and power-wise).



(a) Commit and Duplicate

(b) Before Store 0001

(c) Store 0001

(d) Commit 0100

(e) Commit 0001

Figure 6.5: Avoiding Duplicated Cache Lines

To tackle this problem, we propose the E tag. Every bit on E is used to represent the presence, in the cache, of another data with the same address but with a different speculative address. For example, Figure 6.5(b) shows the state of a cache which contains two copies of location X: one non-speculative version (*i.e.*, with its V set to 0000) and one speculative version for iteration 0100. Note the E on the same Figure.

Figure 6.5(c) shows what happens on the cache after iteration 0001 stores into location X. Notice that the non-speculative entry for X contains the bitmask 0101, which indicates the existence of X_{0100} and X_{0001} on the cache. X_{0100} entry indicates that X_{0001} exists on the cache, and likewise for X_{0001}.

Figure 6.5(d) shows what happens to the cache upon iteration 0100's commit. The non-speculative entry is invalidated and, if necessary, written-back to memory. X_{0001}'s E had its bit 2 set, but it was neither invalidated nor written-back to memory since it is still speculative and it may not drain to memory. Finally, Figure 6.5(e) shows iteration 0001 committing, thus invalidating the previous non-speculative line. This scheme only writes-back to memory when not doing so would create duplicated lines on the cache, thus avoiding bursty traffic and preserving spacial locality. To completely eliminate bursty traffic, the hardware could, upon commit of the speculative line X_i, propagate the dirty state of a non-speculative line X (that is to be evicted because of X_i's commit) to X_i.

6.3.3 Committing and Squashing

Committing in our execution model means that the speculation of a loop iteration was successful (*i.e.*, had the same outcome as if it had been executed sequentially). As shown in the preceding Sections, committing an iteration is just a matter of clearing the right bits on the D, V and E tags.

Squashing occurs in response to a miss-speculation, which can occur due to:

- Cache overflow: the running (parallel) loop requested data from memory and there is no cache entry available for that data. This might occur if every entry that is a fit for the new data is populated with speculative (non-evictable) data. Section 6.4.1 provides experimental data which shows that the cache size requirements are within reasonable sizes;
- A memory ordering violation: the parallelized code did not honor the memory dependencies of the sequential code. In this case, the runtime system should invoke a recovery scheme;
- Hardware / software exceptions: since invoking interrupt (or exception) handlers means exiting the current loop, it makes no sense to continue the parallel execution; and
- False Sharing: this occurs because of the per-line management versioning. Although compiler techniques can mitigate false sharing [9], like [5, 8], we do have a solution for this problem which uses two bits per word. However, as this involves additional design complexity, we omit these per-word bits in this paper for simplicity.

Squashing is simply discarding the (wrongfully executed) speculative code. To accomplish this, the architecture must restore the sequential register state, and it must invalidate every speculative cache line (*i.e.*, lines with a non-zero Vtag) without writing them back. After this sequential state is restored, some policy must be employed. For example, the processor might just restart the parallel execution. It could also jump to a user-defined *after-squash* handler which would decide if parallel execution is to resume, or if parallel execution is not moving forward, execute the code sequentially. All these policies are hardware independent and their evaluation is left for future work.

Our execution model assumes the compiler generates correct sequential code. Using the example from Figure 6.1, if the compiler schedules $A \to C \to B \to D$ (a valid DDG's topological sort), the generated code will be incorrect because the compiler moved the store to `r1 + 4` (*PS* C) before a load of that same address on *PS* B.

Since miss-speculations are possible in our execution model, at least one core should be capable of performing checkpoints (*e.g.*, the core that commits the iterations). Since the memory state is kept consistent by the cache, the core only needs to checkpoint the sequential register state so that, upon miss-speculation, the sequential state of the sequential application can be fully restored.

After this overview of our execution model and an architecture implementing it, Section 6.A will provide some examples of the data forwarding (and memory ordering violations). While not a formal proof of correctness, we expect that the examples explore all the possible situations in a parallel system like the one described in the paper.

6.3.4 On L1 Pressure

Modern high-end processors heavily rely on a fast cache subsystem to quickly provide needed data upon request. Increasing L1 size would potentially change its access time, which could negatively impact its performance. For example, Intel's latest high end processor employs a 3-level cache hierarchy, with a secondary cache level between L1 (which is fast and small) and the third cache level (L3, slow but large). Any technique that increases L1 usage and imposes more pressure on it is likely to hinder other benefits. Furthermore, usually L1 access path is one of the critical determinants of a processor's cycle time [14].

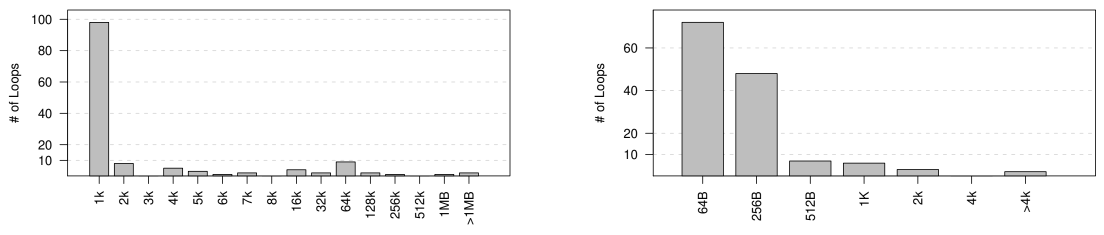
Keeping multiple versions of the same cache line would, in theory, increase the pressure on L1. However, with our execution model, only one version of the speculative data is necessary at the L1 cache level: the line "belonging" to the iteration currently running on its processor. Other versions can safely be evicted from L1, and lower levels, as long as it does not drain from the last cache level into memory.

6.4 Experimental Evaluation

This Section evaluates two different yet important aspects of this project. Section Section 6.4.1 evaluates the pressure on the cache size, which proved to be fairly small for the bast majority of the experiments. Section Section 6.4.2 evaluates the performance impact of synchronizing the cores.

6.4.1 Cache Size Requirements

Before venturing any further on this particular project, we needed some sort of upper bound on the requirements on the cache size. Obviously, if the size requirements were unrealistic, the whole project would be undesirable as it depends on the ability to hold (probably) large amounts of data on the cache. To evaluate this need, we used the SPEC2000 benchmark compiled with GCC version 4.4.1-4ubuntu9 with -O3 optimization level under Ubuntu 9.10 running the Linux Kernel 2.6.31-6 on a Core i7 with 12GB of DDR3 RAM.



(a) Maximum (b) Average

Figure 6.6: Cache Size Requirements

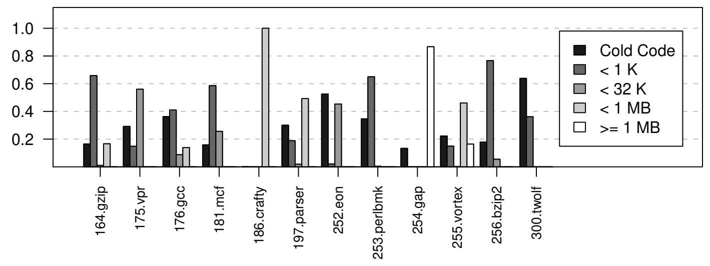


Figure 6.7: Cache Size Requirements per Application

We implemented a pintool [13] which is capable of collecting the runtime memory requirement for the selected loops' iterations. Loop selection is described in Appendix 6.B.

Figure 6.6 shows the cache size requirements for the loops' iterations, with Figure 6.6(a) showing the maximum and Figure 6.6(b) showing the average size requirement. Each bar on the charts represent one different cache size requirement. For example, the first bar on Figure 6.6(a) shows that 98 of the analyzed loops accessed (read or wrote) at most 1K per iteration. The second bar of the same graph report that 9 loops accessed more than 1K but less than 2k. The meaning for Figure 6.6(b) is the same, but the scale is different.

The largest memory footprint we observed was 7MB per iteration for `254.gap`. This means that, in order to allow that benchmark to be parallelized and executed in a quad-core processor implementing our execution model, 28MB are necessary. This requirement varies if different loops are profiled. So, if the 28MB are not available, the programmer/compiler can choose a different loop to parallelize.

Nevertheless, most loops we analyzed require a fairly small amount of memory (less than 32KB per iteration) that already fits into commercially available processors' cache.

Figure 6.7 shows the analyzed code's coverage for each benchmark. We also separated that information in four different cache size requirements: less than 1 K, more than 1K but less than 32K, less than 1 MB and more than 1 MB. We also show the "Cold Code" size (*i.e.*, code executed outside the profiled loops) for completeness. This chart shows that we selected loops that are meaningful for the benchmark's execution. It also shows that most of dynamic code run with mild memory footprint (32 KB or less), which might be used as an indicator that extremely large caches are not necessary for the average case.

6.4.2 Synchronization Overhead

Perhaps the most unique feature of our approach, namely core synchronization, is also the most controversial. Several issues arise from core synchronization. The first is "How hard is it to synchronize the cores?" Yet another important question is "How much performance is lost due to synchronization?" While the former question is hard to answer without a HDL model, the later can be assessed with a simulator. Therefore, we augmented the SESC simulator [23] with synchronization capabilities. We also implemented communication channels to enable the execution of pipeline-like parallelization. We then implemented a synthetic application with a parameterizable workload. The idea was to have a simple program which could have been used to collect experimental data regarding core synchronization. Table 6.1 contains this experimental data.

Four different workloads were used for the simulation. The first workload, labeled *"Balanced Stages"* in Table 6.1 is composed of 4 *PSs* with the exactly same dynamic workload. *"2x-Stage"* is a workload composed of three equally-sized *PSs* and a different *PS* that is twice as big as the other three *PSs*. Likewise, *"4x-Stage"* contains one stage four-times bigger than the other three *PSs*. Finally, *"Unbalanced 40x-Stage"* contains one *PS* that is forty times bigger than the other *PSs*. This last workload was intended to assess the synchronization overhead with highly unbalanced *PSs*. Columns "Synch" and "Asynch" shows the speedups relative to the sequential execution in a the same processor type (OoO or in-order). Column "'Impact' shows the core synchronization impact.

As Table 6.1 summarizes, the effect of synchronizing the core wears off with higher workloads. Nevertheless, the performance impact is negligible even for the small workloads since it is no greater than 0.9%. The boldface entry in the Table pinpoints a scenario where the core synchronization was beneficial (even though by a negligible margin).

6.5 Related Work

The Decoupled Software Pipeline (DSWP) [10, 19] execution model, which is a kind of DOPIPE [17] parallelization strategy, has been shown to be a promising [20] technique towards irregular loop automatic parallelization. Despite the promising results, the authors

Table 6.1: Synchronization Overhead in a Quad-Core Processor

	Workload	OoO			InOrder		
		Synch	Asynch	Impact	Synch	Asynch	Impact
Balanced Stages	Small	3.7510	3.7567	0.001523	3.4593	3.4608	0.000425
	Medium	3.9822	3.9823	0.000024	3.6001	3.6002	0.000113
	Large	3.9938	3.9939	0.000012	3.6099	3.6099	0.000010
2x-Stage	Small	2.4710	2.4734	0.000966	2.2130	2.2134	0.000182
	Medium	2.4926	2.4926	0.000013	2.4957	2.4957	0.000008
	Large	2.5412	2.5412	**1.000002**	2.2566	2.2566	0.000005
4x- and 2x-Stage	Small	2.0020	2.0029	0.000409	1.7844	1.7846	0.000110
	Medium	1.9951	1.9951	0.000006	1.9978	1.9978	0.000006
	Large	1.9968	1.9969	0.000032	1.9997	1.9997	0.000059
Unbalanced 40x-Stage	Small	1.0718	1.0723	0.000495	0.9525	0.9528	0.000240
	Medium	1.0726	1.0726	0.000006	0.9524	0.9524	0.000004
	Large	1.0726	1.0727	0.000004	0.9524	0.9524	0.000003

do not provide any support for memory aliasing, which impairs its wide adoption.

The interest in Thread Level Speculation (TLS) has grown in the past years since the multicore architecture became mainstream, so the literature is plentiful. For example, BulkSC [4] and Bulk [26] employ a light-weight, signature-based support for memory aliasing detection and no support for memory versioning is provided, which requires task squashes upon conflict detection.

Bhowmik *et al.* [2] describes a general framework to allow TLS execution. Differently from our approach, they allow tasks to be created at any point and out-of-order. They also require profiling, which is not necessary for correct program execution in our proposed hardware, although it can improve the parallel code generation.

An architectural support for TLS was proposed by Steffan *et al.* [24]. While this approach can potentially scale up to large machine sizes, it modifies the cache coherence protocol by introducing new states, transitions and coherence messages, making it less attractive. Furthermore, it generates bursty traffic on commits and additional coherence messages upon speculative stores to dirty cache lines. This proposed design also relies on a special hardware module that, upon each task commit, sequentially requests ownership for a group of cache lines whose addresses are stored in a buffer. Thus, besides generating bursty traffic, because in the meantime the processor stalls, execution is slower and consumes more energy [22]. Our approach does not suffer of any of these problems.

Another scalable approach, proposed by Cintra *et al.* [5], also modifies the cache coherence protocol. In addition, squashes and commits perform many potentially slow operations; some of them involving different execution nodes. Commits also involve bursty traffic generation between L1, L2 and memory, and walks through a whole group of cache lines. Squashes requires both walks through a whole group of cache lines and a synchronization of all the nodes in a barrier. Moreover, the whole system relies on

Memory Disambiguation Tables (MDT) that not only must me accessed and searched on every speculative cache misses and thread squashes, but also can cause thread stalls when it is full and a new entry must be allocated. Furthermore, a MDT must be informed on every load issued by a thread, whether it hits or misses in L1. Finally, this scheme does not allow dirty data to remain in L1 or L2 across speculative thread initiations, what makes each new thread start with a cold cache (except for non-speculative data). Again, our proposed architecture does not present any of these problems.

The hardware support for TLS proposed by Gopal *et al.* [8], though not scalable to large machine sizes, generates neither a bursty traffic on commits, nor additional coherence messages upon speculative stores to dirty cache lines. However, it still not only modifies the cache coherence scheme, but also makes use of a control unit to predict the tasks to be assigned to each processor, introducing task squashes when mispredicted tasks are detected. In addition to that, this proposal introduces a centralized hardware mechanism responsible for managing all the speculative execution, which represents a single point of failure. Finally, upon *every* cache miss and even reads that *hit* a committed cache line, this introduced centralized hardware must traverse a linked list of versioned cache lines spread over all the processor's caches. None of these issues are present in our design.

Moreover, in all of these proposals, each processor's cache retains only the last value written by this processor to a cache line, what is not suitable for executing applications parallelized for pipelined execution models, possibly yielding incorrect results.

One issue against TLS support on chip multiprocessors (CMP) is its low energy efficiency. Renau *et al.* [22] showed that with some energy-saving optimizations this TLS support may become rather energy-efficient. These optimizations comprehend both the compiler and the architecture. Although our focus here is not energy-centric, we may see that almost all the energy-consumption problems addressed by the proposed architectural optimizations are naturally solved by our proposed design. One of the optimizations aims to avoid walking through a group of cache lines changing their states in task squashing and commits, and on finding an available tag (ID) to version data. Another proposed optimization is making this tag size short, in order to reduce the energy required in comparisons. Finally, Renau *et al.* [22] points the multiple cache line versions provided when a processor requests a line and the invalidation of all the older versions of a committed line when it is evicted as major problems affecting energy efficiency. Our approach does not have any of these problems, what tends to make it rather energy-efficient.

Recently, Raman *et al.* [18] proposed STMX, a novel scheme that utilizes unix processes and x86 virtual memory protection to enable DSWP on commodity hardware. While presenting promising numbers, this approach heavily relies on profile information for correct operation. Moreover, STMX lacks a mechanism to forward data across different *PSs* in different iterations, therefore narrowing the range of applications amenable to parallelization.

6.6 Conclusions and Future Work

This paper introduced some techniques to aid automatic, safe program parallelization. In particular, it proposed a data versioning scheme that allows the hardware to dynamically identify memory ordering violations.

The proposed model expects the cores to be able to synchronize their execution on stage boundaries, which can be challenging to implement. We already have an envisaged asynchronous model which does not require this expensive core synchronization mechanism, and we are in the process of proving its correctness.

We provided experimental data to evaluate how large a cache would have to be in this kind of system, and we concluded that most meaningful loops actually require just a few bytes per iteration on the cache. We also concluded that the memory requirements are directly related to the quality of the selected loops. Regarding synchronization, we assessed that the *runtime* overhead is negligible, but we did not assessed the *design* complexities.

The proposed architecture is based on the addition of extra tags into the cache and very simple logic which does not require any changes to the underlying cache coherence protocol or the cache controller.

References

[1] A.M.D. *AMD64 architecture programmer's manual volume 2: System programming.* A.M.D., 2010.

[2] Anasua Bhowmik and Manoj Franklin. A general compiler framework for speculative multithreading. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 99–108, 2002.

[3] R. Bocchino, V. Adve, S. Adve, and M. Snir. Parallel programming must be deterministic by default. In *First USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2009.

[4] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: bulk enforcement of sequential consistency. *SIGARCH Comput. Archit. News*, 35(2):278–289, 2007.

[5] M. Cintra, J.F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 13–24, 2000.

[6] T.H. Cormen. *Introduction to algorithms*. The MIT press, 2001.

[7] L. Dagum and R. Menon. Open MP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[8] S. Gopal, TN Vijaykumar, J.E. Smith, and G.S. Sohi. Speculative versioning cache. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 195–205, 1998.

[9] Hwansoo Han and Chau-Wen Tseng. Improving compiler and run-time support for irregular reductions using local writes. In *Languages and Compilers for Parallel Computing*, volume 1656 of *Lecture Notes in Computer Science*, pages 181–196. 1999.

[10] J. Huang, A. Raman, T.B. Jablin, Y. Zhang, T.H. Hung, and D.I. August. Decoupled software pipelining creates parallelization opportunities. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 121–130, 2010.

[11] E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[12] B. Lucia, L. Ceze, and K. Strauss. ColorSafe: Architectural Support for Debugging and Dynamically Avoiding Multi-variable Atomicity Violations. In *ISCA '10: Proceedings of The 37th International Symposium on Computer Architecture*, 2010.

[13] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[14] S.L. Min, J. Kim, C.S. Kim, H. Shin, and D. Jeong. V-P cache: a storage efficient virtual cache organization. *Microprocessors and Microsystems*, 17(9):537–546, 1993.

[15] A.N. Moudgal and B.M. Kuttanna. Apparatus and method to prevent overwriting of modified cache entries prior to write back, 2001.

[16] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, 2005.

[17] D. A. Padua. Multiprocessors: Discussion of some theoretical and practical problems. Technical report, Department of Computer Science, University of Illinois, Urbana, IL, 1979.

[18] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 65–76, 2010.

[19] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 114–123, 2008.

[20] Ram Rangan, Neil Vachharajani, Guilherme Ottoni, and David I. August. Performance scalability of decoupled software pipelining. *ACM Trans. Archit. Code Optim.*, 5(2):1–25, 2008.

[21] J. Reinders. *Intel threading building blocks.* O'Reilly, 2007.

[22] J. Renau, K. Strauss, L. Ceze, W. Liu, S.R. Sarangi, J. Tuck, and J. Torrellas. Energy-efficient thread-level speculation. *IEEE Micro*, 26(1):80–91, 2006.

[23] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[24] J.G. Steffan, C.B. Colohan, A. Zhai, and T.C. Mowry. A scalable approach to thread-level speculation. *ACM SIGARCH Computer Architecture News*, 28(2):1–12, 2000.

[25] R. Tarjan. Depth-first search and linear graph algorithms. In *Conference Record 1971 12th Annual Symposium on Switching and Automata Theory*, pages 114–121. IEEE, 1971.

[26] Josep Torrellas, Luis Ceze, James Tuck, Calin Cascaval, Pablo Montesinos, Wonsun Ahn, and Milos Prvulovic. The bulk multicore architecture for improved programmability. *Commun. ACM*, 52(12):58–65, 2009.

6.A Examples

For the cases of this section we use the MOSI [15] invalidation-based cache coherence protocol for simplicity. The extension for the MOESI [1] protocol, which is used by the AMD multiprocessors, is straightforward.

The cases of Figure 6.8 show all the possible situations (see Figure 6.8(a)) which may occur when B6 reads a data X. There are 5 possible cases corresponding to when/where the data provided to B6 was produced, and in all of these cases, we show that the system causes a rollback if a read-after-write (RAW) dependence is violated, what means that our execution model is correct. The bit vector shown below the grid of each case in Figure 6.8 corresponds to the D tag of B_{X_6} (X_6 cache line at core B) after core B got the requested data.

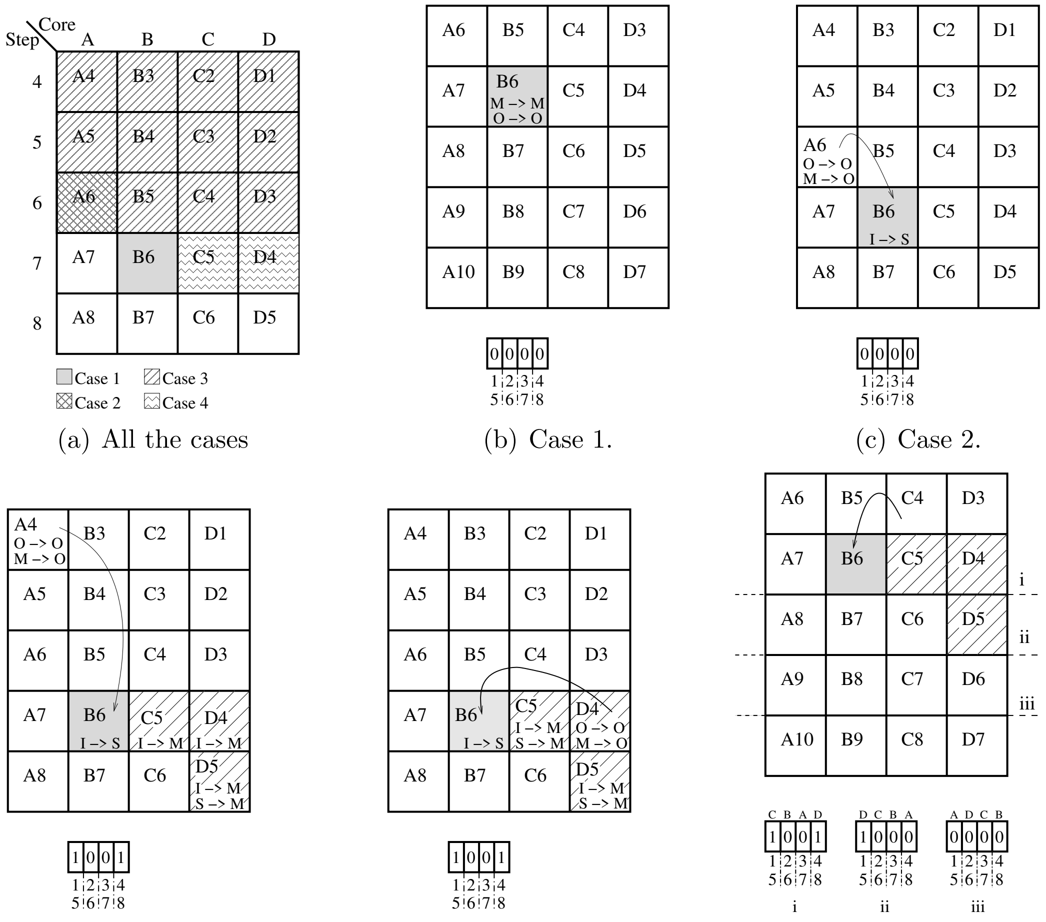
Case 1. *Data was produced at the same iteration and core.*

Figure 6.8(b) shows what happens when a read operation requests a value that was written by the same core at the same iteration of the read. In Figure 6.8(b) B6 reads X, but X has already been written at B6. Notice that when this write took place, B_{X_6} went to state M. Now, when B6 reads X, this cache line can only be either in M or O (in case another core asked for this modified value). B_{X_6} cannot be in S or I because only B is at iteration 6 in this step and thus no other core could have written to X at iteration 6 (this would change the state of B_{X_6}) after B6 wrote to X. Therefore, this read will be satisfied by a cache hit, and this is obviously the correct value for this read. Also notice that since no bit is set in the D tag of B_{X_6} (D is shown below the grid), there will be no rollbacks because of this read.

It is important to notice that a read will never "think" that a data was produced at the same iteration when it was not. For example, one may naively think that, since iterations 2 and 6 share that same address (both B_{X_2} and B_{X_6} are represented by $B_{X_{0100}}$) and we implement lazy commits (when iteration 2 commits, B_{X_2} is neither written back immediately, nor invalidated), a read at B6 could be satisfied by a cache hit because iteration 2 left the cache line $B_{X_{0100}}$ in M or O states (again, both B_{X_2} and B_{X_6} are represented by $B_{X_{0100}}$). However, when iteration 2 committed, B_{X_2} was changed into B_{X_0} and until core B reads or writes to X at iteration 6 (i.e., at B6), cache line B_{X_6} does not even exist. Therefore, if there are any reads preceding a write at B6, the first of these reads will not find an X_6 cache line at core B, causing a cache miss and then being treated by the following cases.

Case 2. *Data was produced at the same iteration at another core.*

Figure 6.8(c) shows the same read, but this time assuming that the most appropriate value (notice that the "most appropriate" value is the closest value in sequential execution order that has already been produced by the time of the read) comes from iteration 6



(a) All the cases (b) Case 1. (c) Case 2.

(d) Case 3. (e) Case 4. (f) Detailed execution.

Figure 6.8: All possible cases.

(the core does not matter). If core A provides X_6, A must have this cache line either in M or O, by the coherence protocol. Since the provided X comes from iteration 6, no bit in the D tag of B_{X_6} will be set, and therefore there will be no rollbacks because of this read. This is the expected behavior every time the most appropriate value for a read has been produced at the same iteration, since this value must be the correct one for this read. Also, in such cases, the coherence protocol *naturally* guarantees that the most recent value produced at this iteration will always be provided because the cache line with the most recent value will be the only one (at iteration 6) in M or O states. For example, if C6 reads X and X was written at both A6 and B6, the value written at B6 will be provided because when B6 wrote to X, the coherence protocol invalidated A_{X_6}. Notice that we are not modifying the coherence protocol as these actions are already part of it.

Case 3. *Data was produced at another iteration in a previous execution step.*

This case (Figure 6.8(d)) illustrates a read operation executed at B6, where the most appropriate value was produced at A4. As in case 2, A_{X_4} must be either in M or O , i.e., it is the most recent value produced at iteration 4. Notice that bits 4 and 5 were set in the D tag of B_{X_6} because of this read, as the read value came from iteration 4. A rollback of iteration 6 should occur due to this read if X is written at either iteration 4 or 5, and that's exactly what happens: iteration 6 will only be rolled back when core B detects a write to X performed at either iteration 4 or 5 because these bits are set in B_{X_6}'s D tag. In addition, one should notice the following:

- Bits 4 and 5 will only be unset when iterations 4 and 5 (respectively) are committed and that's exactly when these iterations cannot write anything else to X, so X_6 is "safe" now.
- Iterations that share the same bits in D tags (for example, the same bit is shared by iterations 1, 5, 9 etc.) will never interfere to each other because when iteration 5 is started, iteration 1 has already committed and when iteration 9 starts, iteration 5 has also committed. A similar reasoning can be used for bit 4 as well.
- After B6 reads X, the first eventual write to X performed at either iteration 4 or 5 *will* appear on the bus, so B will detect it. Take for example iteration 5: the first time C5 writes to X (C5 has not written to X yet, otherwise this value would be more appropriate than the one that came from iteration 4), if C5 has already read X, C_{X_5} must be in state S (as this is the first write to X at C5, C_{X_5} couldn't have gone to M); otherwise, C_{X_5} does not even exist yet. Therefore, the first write to X at C5 will generate a bus transaction.

Case 4. *Data was produced at another iteration in the current execution step.*

This case, illustrated by Figure 6.8(e), is similar to case 4. Here, when B6 reads X, D4 has already written to X, and this is the most appropriate value for B6's read. After getting the value, bits 4 and 5 are set in the D tag of B_{X_6}. Therefore, writes to X at iterations 4 or 5 should cause a rollback. For iteration 5, an eventual first write to X at C5 (notice that C5 has not written to X yet, otherwise this value would be more appropriate than the one written at D4) will go on the bus by the same reason outlined in the last bullet of case 3. The same reasoning can be used for D5 and D_{X_5}. For iteration 4, core D is the only one that can still write to X; but since bit 4 is set in the D tag of B_{X_6}, and core D must generate a bus transaction in order to perform another write to X (when D4 sent D_{X_4} to B6, this cache line went to O), the system will behave correctly.

Finally, Figure 6.8(f) shows the time evolution of the system. Initially, B6 reads X, and the most appropriate value comes from iteration 4. Obviously, by the time X is requested, neither C5 nor D4 have written to X, because both would be more appropriate than the value provided. In step (i), only cores C and D can cause a rollback because of

B6's read; notice that these bits are correctly set in the D tag of B_{X_6} (the evolution of this D tag is shown below the grid in Figure 6.8(f)). In step (ii), since iteration 4 has just committed, only D5 can cause a rollback because it's the only one that still may write to X at iteration 5. In step (iii) iterations 4 and 5 have already committed and the bits in the D tag of B_{X_6} are all in zero; but now there are no more possible conflicts because of B6's read. Again, notice that the bit corresponding to iteration 5, for example, is not influenced by iterations 1 or 9 because when iteration 5 started, iteration 1 had just committed, and when iteration 9 starts, 5 has already committed.

6.B Loop Detection and Selection

This Appendix describes how loops were identified (Section 6.B) and selected (Section 6.B) for the experiment in Section 6.4.1. The experimental setup is the same presented in Section 6.4.1.

Identifying the Loops

We did not modify the compiler to output the loop structure, so we had to collect this some other way. We implemented a pintool [13] that dynamically identifies the program's loop structure using Algorithm 6.B.1. *BB* (Basic Block) is a single-entry, single-exit sequence of instructions.

The idea behind Algorithm 6.B.1 is to keep a stack of *BBs* that are currently not identified to be part of a loop. Whenever the program is about to execute a *BB* that is currently on the stack, Algorithm 6.B.1, Line 4 identifies that this *BB* is in *S*, thus being a loop header. The stack is popped (Algorithm 6.B.1, Line 4) and the popped blocks are considered part of the loop. This is true because no duplicated *BB* is kept on the stack. The first occurrence of a duplicated *BB* pops the stack up to the duplicated *BB*, which is accomplished by the loop starting at Algorithm 6.B.1, Line 4. Algorithm 6.B.1, Line 16 ensures that *BB* would be considered a loop header in the future. The simple case (*BB* not in the stack) is handled by Algorithm 6.B.1, Line 1.

Selecting Loops

After identifying the loops, we profiled every loop in order to gather execution statistics. The issue faced here is that, in our execution model, once a loop is selected for parallelization, all of its inner loops will be part of the parallelized code. This may not be the ideal scenario.

Figure 6.9 shows simple examples of what we called a *Loop Graph* (LG). LG is a graph where nodes represent loops in the program and edges represent the "is outer loop with

Algoritmo 6.B.1: Loop Detection Algorithm

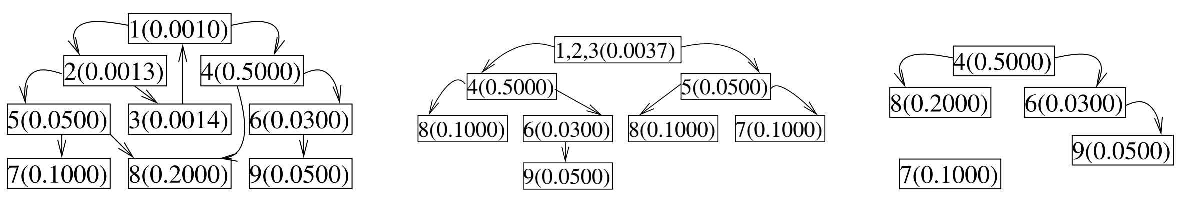
```
Input: BB: BB that is about to be executed
Input: S: A Stack of previously executed BBs
Output: Loop: The identified loop
1  if $BB \notin S$ then
2      S.push(BB)
3      return $LoopNotIdentified$
4  else
5      Last ← Nil First ← Nil
6      repeat
7          Current ← S.pop
8          Loop ← Loop $\bigcup$ {Current}
9          if $Last = Nil$ then
10             First ← Current
11         else
12             Add edge from Current to Last
13         Last ← Current
14     until $Current = BB$;
15 Add Edge from Last to First
16 S.push(BB)
17 return $LoopIdentified$
```

respect to" relation (e.g., an edge from node A to node B indicates that A is an outer loop with respect to B). On Figure 6.9, the nodes' labels are "A(N)", where A is the loop ID and N is the loop dynamic coverage of the entire program without considering its inner loops coverages.

Figure 6.9(a) shows the LG as collected by our pintool. This graph contains an SCC with loops 1, 2 and 3. It also contains a join node (*i.e.*, a node with multiple predecessors). To analyze this data, and to select the loops for the next phase, we implemented a tool that reads the identified loops, merges all the nodes in an SCC and finally splits all the join nodes. The resulting graph is a DAG (Direct Acyclic Graph),

Next, our tool looks for the DAG's roots in order to determine whether or not that root should be selected for profiling. Here, we use a simple heuristic: if the root's coverage is smaller than the sum of its successors' coverage, the root is not a good candidate for profiling. Its immediate successors are then considered for profiling.

For example, on Figure 6.9(b), root node (1,2,3) (i.e., the resulting node after collapsing the SCC) has a very low coverage, so it is not considered as a candidate for profiling. As node (1,2,3) is discarded, nodes 4 and 5 become potential candidates for profiling, since they are now roots of their on trees. Node 4 will eventually be selected for profiling



(a) Before Analysis (b) During Analysis (c) After Analysis

Figure 6.9: Loop analysis and selection.

since its coverage (50%) is greater than its inner loops' coverage summed up. Node 5, on the other hand, will not be selected since its inner loops' coverage (20%) is greater than its own coverage (5%). The selected loops to be profiled are depicted in Figure 6.9(c).

6.C Proof of Correctness

In this Appendix, we present a formal proof of correctness for our idea. Throughout the Appendix, n is the number of cores.

6.C.1 Lemma 1

The first write to location x performed by core c at iteration i generates a bus transaction.

Proof: When iteration i starts in the system, V's bit ($i \bmod n$) is set to 0 for every location x in the cache of all the cores. This happens because iteration $(i-n) \bmod n$ has just committed. But now, when core c writes for the first time to x at i, the bit $i \bmod n$ is still set to 0, since it would be set to 1 only if this core had already written to x at an iteration y such that $i - n < y < i$ and $y \bmod n = i \bmod n$. Obviously such y doesn't exist.

6.C.2 Write After Write (WAW)

All we need to prove in this section is that when a location x is written at iteration i:

- The value written is invalidated by a posterior write to the same location at the same iteration.
- The value written is not invalidated by any other posterior write to same location at another iteration.

For the first item, when core c writes to x for the last time at iteration i, it must put c's cache line $\langle x, i \bmod n\rangle$ in M state (if it's not already in M). If c has to send this value to satisfy any posterior read in the system, this line goes to O state. It will only go to I state if there is a write on the bus to location $\langle x, i \bmod n\rangle$ before i commits. But if another core writes to x at iteration i, it will have to generate a bus transaction for a write to $\langle x, i \bmod n\rangle$, since this core has to put this line in M state and c has the same line in M, O or S states. Since this write occurs at iteration i, obviously it's before i commits, and c's cache line $\langle x, i \bmod n\rangle$ will be invalidated. Alternatively, one may notice that any posterior write to the same location must occur in another core in a posterior step. Since any core executes iteration i just once, by this time, Lemma 1 guarantees that c's cache line $\langle x, i \bmod n\rangle$ will be invalidated.

The second item is easy. A cache line $\langle x, i \bmod n\rangle$ is invalidated only when there's a write on the bus to the same location of this line. But after i, the next iteration that could write to same line is $i + n$, and when this iteration is put in the "pipeline", i has just committed.

6.C.3 Write After Read (WAR)

If core c reads location x at iteration i, but the correct value for this location (the one that should be read by this iteration) is written in a posterior moment somewhere in the system, we show that iteration i won't be committed. Therefore, we need to show two things:

1. When the correct (or a more appropriate) value of x for i is written at iteration j, the bit $(j \bmod n)$ is set to 1 in D of c's cache line $\langle x, i \bmod n\rangle$, and since $j < i$, i won't commit.
2. When the value of item 1 is written by a core, this core generates a bus transaction.

Proof 1: First of all, we'll prove that the value read may not come from iteration i and therefore may not be satisfied by a cache hit. Thus the correct bits for conflict detection will be set to 1.

Notice that this read must occur before any write to the same location x at iteration i in core c, otherwise the location would already have the correct value for the read (no other value could be more appropriate than this one), contradicting the hypothesis that the correct value has not been written yet.

Now, we show that this read operation is not satisfied by a cache hit, thus generating a bus transaction. Such transaction would not occur only if c's cache line $\langle x, i \bmod n\rangle$ were in M, O or S states **and** V's $i \bmod n$ bit of the same cache line in the same core were set to 1. However, when iteration $i - n$ committed (this already happened since i is in execution), this bit was set to 0. Thus, this bit would be set to 1 only if in core c

at some iteration y, $i - n < y \leq i$ and $y \bmod n = i \bmod n$, x was read or written. But since $y \bmod n \neq i \bmod n \; \forall \; y$ in the range $(i - n, i)$, we conclude that $y = i$. We have just proved that c didn't write to location x at iteration i before the read. So, the only lasting possibility is that c has already read x at i; this possibility does exist for all the read operations, except for the first. Therefore, the first read operation in c at i must have generated a bus transaction, and that's all we need to prove because all the reads to location x executed in core c at i before any write operation to the same location in c at i must get the same value of the first read. So, the values got by these subsequent reads are right if the first read got the right value, and are wrong otherwise.

Furthermore, one should notice that the most appropriate value will not come from iteration i because otherwise this would be the correct value, contradicting the hypothesis that the correct value has not been written yet.

Now that the most most appropriate (but incorrect!) value for location x has been read by c at i, suppose that this value was written somewhere in the system at iteration k (which is the next iteration to commit if the value came from memory, i.e. it's not a speculative value). Notice that D's bits in the range $[k \bmod n, i \bmod n)$ of c's cache line $\langle x, i \bmod n \rangle$ would not be set to 1 only if $k \bmod n = i \bmod n$. But since k is at least the next iteration to commit (in the case when x's correct value comes from memory) and $i - n$ has already committed, $k > i - n$. Also, because $i + n$ has not entered the "pipeline" yet, $k < i + n$. We have just proved that $k \neq i$; therefore $k \bmod n \neq i \bmod n$, and the referred bits are set.

Now c's cache line $\langle x, i \bmod n \rangle$ has its D's bits in the range $[k \bmod n, i \bmod n)$ set to 1. Thus, i will cause a rollback because of this read only when a write to location x appears on the bus, and this write comes from the execution of a iteration w such that $w \bmod n$ is in the range of these bits set to 1.

Suppose that when x is written at j, the bit $(j \bmod n)$ is set to 0 in D of c's cache line $\langle x, i \bmod n \rangle$. It means that sometime between the read executed at i and this write at j, some iteration y such that $y \bmod n = j \bmod n$ was committed. Let's suppose, for contradiction, that such y iteration does exist.

Obviously, $y < j$, since j is being executed now. In addition, when the read operation was executed at i, $i - n$ had already been committed, which gives us $y > i - n$. But $i > j \Rightarrow i - n > j - n$, and since $y > i - n$, these give us $y > j - n$. However, $\nexists \; y$ such that $j - n < y < j$ and $y \bmod n = j \bmod n$, and the item 1 is proved.

Proof 2: Let's say that the incorrect value came from core q, iteration k, and the correct (or a more appropriate) value will be written by core p at iteration j. Now we have two possibilities:

- The correct (or more appropriate) value will be written in a future step.
- The correct (or more appropriate) value will be written in the current step.

The fist case is trivially satisfied by Lemma 1. The second case may be further divided into two subcases:

1. The incorrect value read by i had been written in some step behind.
2. The incorrect value read by i had been written in the current step.

Again, the first subcase is trivially satisfied by Lemma 1. For the second subcase, notice that $q \neq c$, since otherwise the correct value would have already been written, contradicting the hypothesis that the correct value had not been written yet. Now, if $k = j$, $p = q$ and this core's cache line $\langle x, k \bmod n \rangle$ went to O state by the time this core sent the incorrect value to i. When this core writes the same location at iteration k again, there will be a bus transaction since this line (now in O state) must go to the M state before the write may be performed. Finally, if $k \neq j$, $p \neq q$. But notice that when c asked for the value of x, if $\langle x, k \rangle$ was chosen, it's because $\langle x, j \rangle$ hadn't been written yet (otherwise the last would be the more appropriate). Therefore, when $\langle x, j \rangle$ is written, Lemma 1 guarantees that a bus transaction will be generated.

6.C.4 Read After Write (RAW)

In this case, core c reads location x at iteration i, and the correct value has already been written. The proof is divided in the following three exclusive cases:

- If core c has already written x at i (then this is the correct value).
- Else if another core has already written x at i (then this is the correct value).
- Else, the correct value was written by some core at an iteration k such that $k < i$

In the first case, c's cache line $\langle x, i \bmod n \rangle$ has the most appropriate value and this is the correct one (by the WAW proof). Thus, we need to prove:

1. The read will be a cache hit (therefore there will be no bus transaction) and this read will get the correct value.
2. There will be no rollback of i because of this read.

Proof 1: When c's cache line $\langle x, i \bmod n \rangle$ was written by c at i, it was sent to M state, and after that, it would go to I state only if another core $p \neq c$ had written x at $i \bmod n$. But since in this step only c is at i and there is no other core in an iteration y such that $y \bmod n = i \bmod n$, that's not possible, and c's cache line $\langle x, i \bmod n \rangle$ is not in I.

Proof 2: There will be no rollback because of this read because no D's bit of c's cache line $\langle x, i \bmod n \rangle$ will be set to 1.

For the second case, the last core that wrote to x at i has the most appropriate value and this is the correct one (by the WAW proof). So we need to prove:

1. The first read will ask for the most appropriate value on the bus (subsequent reads will use the same value through cache hits).

2. The core that wrote to x at i the last has its cache line $\langle x, i \bmod n\rangle$ in a valid state and thus it will be sent (and this line has the correct value by the WAW proof).
3. There will be no rollback of i because of this read.

Proof 1: There won't be a cache hit because when $i - n$ committed V's bit $i \bmod n$ of c's cache line $\langle x, i \bmod n\rangle$ was set to 1, and since c didn't write to x at i, this bit is still set to 1 when this read is executed in c at i (this bit would be set to 0 only if some iteration y such that $i - n < y < i$ and $y \bmod n = i \bmod n$ wrote to x before this read in c at i. But since such y doesn't exist, the bit is set to 1). Therefore, there will be a bus transaction asking for x' value.

Proof 2: The last core that wrote to x (and therefore this is the correct value for c's read at i) is the only one that has cache line $\langle x, i \bmod n\rangle$ in M state if someone does, and if no one has this line in M state, it's the only one that has it in O state. Thus, this core will send the last value that was written to x (which is the correct one for the c's read at i).

Proof 3: i will not be rolled back because of this read because no D's bit of c's cache line $\langle x, i \bmod n\rangle$ will be set to 1.

For the last case, we need to prove:

1. The first read will ask for the most appropriate value on the bus (subsequent reads will use the same value through cache hits).
2. The core that wrote to x at k the last has its cache line $\langle x, k \bmod n\rangle$ in either M or O states and thus it will be sent (and this line has the correct value by the WAW proof).
3. There will be no rollback of i because of this read.

Proof 1: There won't be a cache hit by the same reason outlined in proof 1 of the second case above. So, core c will generate a bus transaction asking for x's value and since the correct value has already been written, this value is going to be the chosen one.

Proof 2: To show this, suppose that the correct value was written at iteration k by core p. This core's cache line $\langle x, k \bmod n\rangle$ would be in I state only if another core q wrote to x at some iteration y such that $y \bmod n = k \bmod n$. Also, y is obviously one of the n uncommitted iterations (there are only n uncommitted iterations at anytime), and so is k (k is at least the next iteration to commit if x's correct value comes from memory). It follows that $y = k$. Notice then that if $p = q$, p's cache line $\langle x, k \bmod n\rangle$ is in either M or O states. On the other hand, $p \neq q$ is impossible because this contradicts the assumption that the correct value was written at iteration k by core p.

Proof 3: Upon receiving the (correct) value for x, c's cache line $\langle x, i \bmod n\rangle$ has its D's bits in the range $[k \bmod n, i \bmod n)$ set to 1. Therefore, i will be rolled back because of this read only if there is a write to x at a iteration y such that $y \bmod n \in [k \bmod n, i \bmod n)$ before i commits. Suppose, for contradiction, that such y does exist.

Notice that $y > i - n$ because the D's bits (in the range $[k \bmod n, i \bmod n))$ are set to 1 when x is read at i and by this time $i - n$ has already been committed. Since the value came from iteration k, $k > i - n$ (even if the value comes from memory, k would be at least $i - n + 1$). But if $y > i - n$, $y \bmod n \in [k \bmod n, i \bmod n)$ and $k \leq i$, then $y \geq k$. In addition, one should notice that when iteration $i + n$ begins (is put in the "pipeline"), i will have already been committed by the end of the preceding step. Since the write at y must be performed before i commits, $y < i + n$. Thus, as we have $k \leq y < i$ and $y \bmod n \in [k \bmod n, i \bmod n)$, there are two possibilities:

- $k \leq y < i$
- $k + n \leq y < i + n$

In the first case, the value written to x at y is more appropriate for i. But since this write occurs after the value of x is asked at i, there's a contradiction with the hypothesis that the correct value had been written before c asked for the value at i.

For the second case, if $k + n \leq y < i + n$, when y writes, $y - n$ will have already been committed. Therefore, D's bit $y \bmod n$ will represent y. But, as was shown above, $k > i - n$ and we have $k + n > i$. Since $y \geq k + n > i$, i won't be rolled back because of this write.

Capítulo 7

Considerações Finais

Inegavelmente, *La Microarchitecture est Morte*. Os desafios enfrentados pelos micro-arquitetos para aumentar o desempenho de aplicações sequenciais claramente apontam para a necessidade de soluções diferentes das adotadas atualmente pela Indústria.

Apesar de inúmeras propostas, tanto pela Indústria quanto pela Academia, não se visualiza claramente uma solução definitiva de longo prazo. Existem apostas em vários modelos distintos de execução: processadores assimétricos, aceleradores, *chips* reconfiguráveis, múltiplos núcleos dentre vários.

TLS é mais um modelo. As contribuições da Academia neste campo foram muito extensas mas, no momento da escrita desta Tese, nenhuma solução conseguiu convencer a Indústria a apostar neste modelo de computação. A grande dificuldade em adotar TLS como solução *de facto* para continuidade no desenvolvimento de processadores é garantir a eficiência das soluções propostas que, em sua maioria, apostam pesadamente em especulação de dados e controle (o que pode não ser adequado para para garantir benefícios reais em aplicações no mundo real), requerendo usualmente suporte específico e complexo em *hardware*.

O trabalho apresentado nesta Tese, em particular a arquitetura proposta no Capítulo 6, difere da ampla maioria da bibliografia disponível em TLS: ao abandonar sofisticados *hardware* de especulação e detecção de violações, espera-se que a dificuldade em uma eventual transição Academia – Indústria seja amenizada. Conforme explicado no Capítulo 6, os mecanismos propostos são leves e de simples implementação.

Qualquer que seja o novo (ou novos) modelo de desenvolvimento adotado, novas técnicas serão necessárias. Apesar de ainda não estar claro quais serão estas técnicas, tudo indica que serão, essencialmente, arquiteturais. Assim, *Longue Vie à la Architecture*!

7.1 Trabalhos Futuros

Desde que arquitetura de computadores tornou-se quantitativa [24], todo projeto ou proposta na área precisa, invariavelmente, de um simulador. Apesar de ser impossível, apenas com o simulador, *provar* a corretude de uma proposta, mostrar números que *validem* uma idéia é praticamente essencial para a pesquisa em arquitetura de computadores. Assim, o primeiro e mais óbvio trabalho futuro é a implementação do modelo proposto no Capítulo 6 em um simulador.

Na verdade, este projeto já está, no momento da escrita desta Tese, sendo implementado como projeto de mestrado de um aluno do Instituto de Computação da Unicamp. Espera-se que este simulador esteja pronto até o final do primeiro semestre de 2011.

Outro trabalho futuro é implementar o suporte em compilador para geração de código paralelo. Ter um compilador capaz de gerar o código paralelo é fundamental para uma melhor avaliação do modelo por permitir que uma maior quantidade de instâncias de testes sejam feitas em um menor espaço de tempo. Além disso, a disponibilidade de um compilador possibilita a avaliação experimental de diversos algoritmos de paralelização.

A pesquisa realizada supôs que todos os núcleos no processador são homogêneos. Uma extensão das idéias aqui apresentadas pode ser a avaliação de um sistema heterogêneo. Por exemplo, além de núcleos de propósito geral, o processador pode contar com núcleos especializados (*e.g.*, aceleradores físicos, criptográficos, FPGAs ...). Esta arquitetura heterogênea pode ser utilizada em sistemas específicos (*e.g.*, *video games*, *home theaters* e sistemas embarcados). Outra variação interessante seria o uso de processadores com suporte a SMT. Ter todos os estágios do programa paralelizado executando no mesmo núcleo pode diminuir o tráfego de *snooping* na arquitetura do Capítulo 6, bem como simplificar a lógica para sincronização.

O modelo proposto não avalia o desempenho do sistema na presença de multitarefa e interrupções. Por exemplo, se uma interrupção ocorrer durante a sincronização dos núcleos, não é claro qual a melhor política deve ser empregada. A interface do sistema operacional com processadores operando em conjunto deve ser estudada.

Como a implementação do modelo descrito nesta tese adiciona *bits* à *cache* do processador, seria interessante avaliar a possibilidade de utilizar estes *bits* extras para outras técnicas. Por exemplo, seria interessante identificar potenciais usos do modelo proposto com memória transacional.

Conforme a tecnologia avançar, o custo do gerenciamento *online* de *traces* pode diminuir drasticamente. Por exemplo, o automato descrito no Capítulo 4 poderia ser utilizado pelo processador para a identificação dos *traces*. Isto ocorrendo, seria interessante ver como um compilador dinâmico seria capaz de paralelizar *traces* de execução. Comparar este otimizador dinâmico com o estático seria um resultado natural deste projeto.

Na área de confiabilidade, a arquitetura proposta pode ser integrada à TEA. Neste sistema, conforme o programa avança na computação, o processador pode "anotar" os acessos à memória no TEA do programa. Quando uma transição de estado ocorrer, um núcleo verificador reexecuta o código do estágio anterior, comparando os seus resultados com as anotações no autômato. Uma divergência indicaria a existência de algum problema, e alguma ação de correção pode ser tomada. O versionamento dos dados na *cache* garantiria que ambas as *threads* teriam acesso ao mesmo estado da memória.

Referências Bibliográficas

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. Compilers: Principles, Techniques, and Tools, 2006.

[2] H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *MICRO 31: Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–236, 1998.

[3] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–, 2003.

[4] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W.-m. W. Hwu. Integrated Predicated and Speculative Execution in the IMPACT EPIC architecture. In *ISCA '98: Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 227–237, 1998.

[5] S. Balakrishnan and G. S. Sohi. Program Demultiplexing: Data-flow based Speculative Parallelization of Methods in Sequential Programs. *SIGARCH Computer Architecture News*, 34(2):302–313, 2006.

[6] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skalesky, Y. Wang, and Y. Zemach. IA-32 Execution Layer: A Two Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium®-based Systems. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, pages 191–202, 2003.

[7] A. Bhowmik and M. Franklin. A General Compiler Framework for Speculative Multithreaded Processors. *IEEE Transactions on Parallel Distributed Systems*, 15(8):713–724, 2004.

[8] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August. Revisiting the Sequential Programming Model for the Multicore Era. *IEEE Micro*, 28(1):12–20, 2008.

[9] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. *SIGARCH Computer Architecture News*, 35(2):278–289, 2007.

[10] C. Cifuentes and M. V. Emmerik. UQBT: Adaptable Binary Translation at Low Cost. *Computer*, 33:60–66, March 2000.

[11] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-memory Multiprocessors. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 13–24, 2000.

[12] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. *SIGARCH Computer Architure News*, 29(2):14–25, 2001.

[13] R. Cytron. DOACROSS: Beyond Vectorization for Multiprocessors. In *ICPP '86: International Conference on Parallel Processing*, pages 836–844, 1986.

[14] L. Dagum and R. Menon. Open MP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[15] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing^{tm} Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 15–24, 2003.

[16] E. Duesterwald and V. Bala. Software Profiling for Hot Path Prediction: Less is More. In *ASPLOS 9: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 202–211, 2000.

[17] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 465–478, 2009.

[18] A. Gal and M. Franz. Incremental Dynamic Code Generation with Trace Trees. Technical Report 06-16, Donald Bren School of Information and Computer Science, University of California, Irvine, November 2006.

[19] G. Gerosa, S. Curtis, M. D'Addeo, B. Jiang, B. Kuttanna, F. Merchant, B. Patel, M. Taufique, and H. Samarchi. A Sub-1W to 2W Low-Power IA Processor for Mobile Internet Devices and Ultra-Mobile PCs in 45nm Hi-κ Metal Gate CMOS. In *ISSCC '08: The IEEE International Solid-State Circuits Conference 2008. Digest of Technical Papers.*, pages 256 –611, 2008.

[20] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative Versioning Cache. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architectures*, pages 195–205, 1998.

[21] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[22] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. *SIGOPS Operating Systems Review*, 32(5):58–69, 1998.

[23] H. Han and C. Tseng. Improving Compiler and Tun-time Support for Irregular Reductions using Local Writes. *Languages and Compilers for Parallel Computing*, pages 181–196, 1999.

[24] J. Hennessy, D. Patterson, and D. Goldberg. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann, 2003.

[25] A. Hilton, S. Nagarakatte, and A. Roth. iCFP: Tolerating all-level Cache Misses in in-order Processors. In *HPCA 2009: IEEE 15th International Symposium on High Performance Computer Architecture*, pages 431–442, 2009.

[26] D. Hiniker, K. Hazelwood, and M. D. Smith. Improving Region Selection in Dynamic Optimization Systems. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 141–154, 2005.

[27] R. Hookway. Digital FX!32: Running 32-bit x86 Applications on Alpha NT. In *COMPCON '96: Proceedings of the 42nd IEEE Computer Society International Conference*, pages 37–42, 1997.

[28] J. Huang, A. Raman, T. Jablin, Y. Zhang, T. Hung, and D. August. Decoupled Software Pipelining Creates Parallelization Opportunities. In *CGO '10: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 121–130, 2010.

[29] Intel Corporation. Intel®Pentium® IV Processor 6xx Sequence and Intel®Pentium® IV Processor Extreme Edition. http://www.intel.com/Assets/PDF/datasheet/306382.pdf. Acessado em 23 de Novembro de 2010.

[30] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut Program Decomposition for Thread-level Speculation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 59–70, 2004.

[31] K. Kennedy and J. Allen. *Optimizing Compilers for Modern Architectures: a Dependence-based Approach*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2001.

[32] D. Kim and D. Yeung. A Study of Source-level Compiler Algorithms for Automatic Construction of Pre-execution Code. *ACM Transactions on Computer Systems*, 22(3):326–379, 2004.

[33] H. Kim, A. Raman, F. Liu, J. W. Lee, and D. I. August. Scalable Speculative Parallelization on Commodity Clusters. In *MICRO 43: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–12, 2010.

[34] V. Krishnan and J. Torrellas. Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor. In *ICS '98: Proceedings of the 12th International Conference on Supercomputing*, pages 85–92, 1998.

[35] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[36] C. Madriles, P. López, J. M. Codina, E. Gibert, F. Latorre, A. Martinez, R. Martinez, and A. Gonzalez. Boosting Single-thread Performance in Multi-core Systems Through Fine-grain Multi-threading. *SIGARCH Computer Architecture News*, 37(3):474–483, 2009.

[37] P. Marcuello and A. González. Clustered Speculative Multithreaded Processors. In *ICS '99: Proceedings of the 13th International Conference on Supercomputing*, pages 365–372, 1999.

[38] P. Marcuello and A. Gonzalez. Thread-Spawning Schemes for Speculative Multithreading. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 55–64, 2002.

[39] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[40] T. Ohsawa, M. Takagi, S. Kawahara, and S. Matsushita. Pinot: Speculative Multithreading Processor Architecture Exploiting Parallelism over a Wide Range of Granularities. In *MICRO 38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, 2005.

[41] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *MICRO 38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, 2005.

[42] D. A. Padua. Multiprocessors: Discussion of Some Theoretical and Practical Problems. Technical report, Department of Computer Science, University of Illinois, Urbana, IL, 1979.

[43] D. Patterson and J. Hennessy. *Computer Organization and Design: the Hardware/Software Interface.* Morgan Kaufmann Pub, 2009.

[44] D. A. Penry, D. I. August, and M. Vachharajani. Rapid Development of a Flexible Validated Processor Model. In *In Proceedings of the 2005 Workshop on Modeling, Benchmarking, and Simulation*, 2005.

[45] J. P. Porto, G. Araujo, Y. Wu, and E. Borin. Trace execution automata in dynamic binary translation. In *3^{rd} workshop on architectural and micro-architectural support for binary translation (AMAS-BT'10)*, June 2010.

[46] J. P. Porto, G. Araujo, Y. Wu, E. Borin, and C. Wang. Compact trace trees in dynamic binary translators. In *2^{nd} workshop on architectural and micro-architectural support for binary translation (AMAS-BT'09)*, June 2009.

[47] M. Prvulovic and M. J. Garzarán. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *ISCA '01: In proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 204–215, 2001.

[48] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis Compiler: an Infrastructure for Speculative Threading based on Precomputation Slices. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 269–279, 2005.

[49] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative Parallelization Using Software Multi-threaded Transactions. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pages 65–76, 2010.

[50] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage Decoupled Software Pipelining. In *CGO '08: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 114–123, 2008.

[51] R. Rangan, N. Vachharajani, G. Ottoni, and D. I. August. Performance Scalability of Decoupled Software Pipelining. *ACM Transactions on Architecture and Code Optimization*, 5(2):1–25, 2008.

[52] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-time Parallelization of Loops with Privatization and Reduction Parallelization. *SIGPLAN Not.*, 30(6):218–232, 1995.

[53] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.

[54] J. Renau, K. Strauss, L. Ceze, W. Liu, S. R. Sarangi, J. Tuck, and J. Torrellas. Energy-Efficient Thread-Level Speculation. *IEEE Micro*, 26(1):80–91, 2006.

[55] P. Salverda and C. Zilles. Fundamental Performance Constraints in Horizontal Fusion of In-Order Cores. In *HPCA '08: The 14th IEEE International Symposium on High Performance Computer Architecture*, pages 252–263, 2008.

[56] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. *SIGARCH Computer Architure News*, 23(2):414–425, 1995.

[57] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, 2004.

[58] J. Steffan and T. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, 1998.

[59] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-level Speculation. *SIGARCH Comput. Archit. News*, 28(2):1–12, 2000.

[60] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or Discard Execution Model for Speculative Parallelization on Multicores. In *MICRO 41: Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–341, 2008.

[61] R. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[62] J. Torrellas, L. Ceze, J. Tuck, C. Cascaval, P. Montesinos, W. Ahn, and M. Prvulovic. The Bulk Multicore Architecture for Improved Programmability. *Communications of the ACM*, 52(12):58–65, 2009.

[63] D. Tullsen. Simulation and Modeling of a Simultaneous Multithreading Processor. In *22nd Annual Computer Measurement Group Conference*, pages 819–828, 1996.

[64] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative Decoupled Software Pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 49–59, 2007.

[65] Y. Watanabe, J. D. Davis, and D. A. Wood. WiDGET: Wisconsin Decoupled Grid Execution Tiles. *SIGARCH Computer Architecture News*, 38(3):2–13, 2010.

[66] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

[67] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A Transparent Dependence Distance Profiling Infrastructure. In *CGO '09: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 47–58, 2009.

[68] H. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 25–36, 2007.

[69] C. Zilles and G. Sohi. Execution-based Prediction using Speculative Slices. In *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, 2001.

[70] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *MICRO 35: Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 85–96, 2002.

Apêndice A

Prova de Corretude – Consistência e Versionamento de Dados

Suponha um sistema com n processadores e *caches* privativas cuja coerência é mantida pelo protocolo *Modified – Owner – Shared – Invalid* (MOSI). Suponha também que haja ao menos um nível de memória *cache* compartilhado. Suponha, ainda, que a hierarquia de *cache* deste sistema implementa os vetores T_1, T_2 e T_3.

Lema A.0.1. *Uma leitura de uma linha de cache L não causa* squash*.*

Na ausência de escritas, leituras são sempre **seguras**, de modo que elas sempre podem executar e nunca falham devido ao estado atual da *cache*. Desta forma, a detecção de violação da semântica sequencial é feita quando da escrita de dados na memória.

Lema A.0.2. *A* cache *do sistema comporta todos os dados de iterações pendentes.*

Isto é necessário para garantir que dados especulativos fiquem contidos dentro do processador, de modo a garantir que a memória esteja sempre com os valores corretos para a última iteração executada com sucesso.

Proposição A.0.1. *A leitura da linha L durante a iteração i, denotada L_i, satisfeita pelo encaminhamento de L_j, com $j < i$, define T_1 de L_i como $T_1 = \{x : j \leq x \wedge x < i\}$.*

Teorema A.0.1. *Os elementos do conjunto T_1 da Proposição A.0.1 contém os elementos suficientes e necessários para detecção de conflitos RAW.*

Demonstração. Necessidade: Suponha que T_1 contém um elemento k que não é necessário para a detecção de conflitos. Considere o conjunto $T_1' = T_1 - \{k\}$. Desta forma, uma atualização da linha L_k não será identificada como violação. Entretanto, pela definição de T_1, a iteração k ocorre no intervalo $[j, i)$ e, portanto, indicam a utilização errônea de L_j quando da leitura de L_i. Assim, não pode haver tal T_1'. $\square$

Demonstração. Suficiência: Suponha que exista a linha L_k no sistema e que $k \notin T_1$. Se $k < j$, então escritas em L_k não afetam L_i pois são anteriores à L_j. Se $i < k$, então k ocorre logicamente após i, de modo que escritas em L_k não afetam L_i. Se $j \leq k \wedge k < i$, então, obrigatoriamente, $k \in T_1$. Se $k = i$, nenhum conflito ocorre pois acessos à memória dentro de uma mesma iteração ocorrem na ordem do laço sequencial. □

Teorema A.0.2. *Ao ler uma linha L_i que não está presente na* cache*, é suficiente e necessário que T_1 de L_i seja $T_1 = \{x : o \leq x \bigwedge x < i\}$, onde o é o identificador da iteração mais antiga pendente.*

Demonstração. Uma leitura de uma linha da memória é equivalente a ler um dado da última iteração que terminou. Neste caso, toda escrita ao dado em iterações anteriores à iteração que requisitou a linha indicam uma violação da semântica sequencial. A prova de suficiência e necessidade são idênticas aos do Teorema A.0.1. □

Corolário A.0.1. *Dependências do tipo RAW são corretamente satisfeitas ou um conflito é detectado.*

Demonstração. Uma dependência não ser satisfeita implica T_1 não contém todos os elementos necessários para detecção de dependências, o que é absurdo. □

Corolário A.0.2. *Dependências do tipo RAW não são espuriamente detectadas.*

Demonstração. Um conflito espúrio detectado indica a existência de elementos excedentes em T_1, o que é absurdo, já que T_1 é mínimo. □

Lema A.0.3. *Por ser uma* cache *versionada, anti-dependências WAW e WAR são naturalmente evitadas.*

O uso de uma *cache* versionada é semelhante à renomeação de registradores. Da mesma forma, anti-dependências de memória são resolvidas alocando-se múltiplas entradas para a mesma linha de *cache* como se endereços idênticos de iterações distintas fossem distintos.

Proposição A.0.2. *Uma escrita à linha L_i é segura, ou o protocolo de coerência enviará um pedido de acesso exclusivo.*

Demonstração. Suponha que um processador p escreve em L_i. Dois cenários são possíveis. Ou p tem acesso exclusivo à L_i ou p não tem acesso exclusivo à L_i.

Se p não tem acesso exclusivo à L_i ele o requisitará enviando uma mensagem aos outros processadores. Esta mensagem, existente no protocolo original, será utilizada para detecção de conflitos.

Se p tem acesso exclusivo à L_i, a operação de escrita é segura e nenhuma mensagem de coerência é necessária.

Suponha, agora, que L_i pode ser exclusivamente acessada e que existe uma linha L_j que é afetadas por escritas em L_i. Tal linha só pode existir se L_i foi utilizada na leitura de L_j ou se existe uma linha L_k tal que $k < i$ e $i < j$. Se tal L_k de fato existe, então, obrigatoriamente, quando do pedido de leitura de L_j, L_i não exista. Se existisse, L_i seria mais apropriada para a leitura de L_j e teria sido usada, saindo do estado de acesso exclusivo. Como ela não existia, quando L_i for escrita o sistema, obrigatoriamente, o protocolo enviaria uma mensagem de coerência para obtenção do acesso exclusivo. $\square$