**Estudo e Implementação da Otimização de Preload de Dados Usando o Processador XScale**

*Márcio Rodrigo de Oliveira*

**Dissertação de Mestrado**

Instituto de Computação
Universidade Estadual de Campinas

# Estudo e Implementação da Otimização de Preload de Dados Usando o Processador XScale

**Márcio Rodrigo de Oliveira**[1]

11 de Julho de 2005

**Banca Examinadora:**

- Prof. Dr. Guido Costa Souza Araújo (Orientador)
- Prof. Dr. André Luís de Medeiros Santos
  Centro de Informática (CIN) - UFPE
- Prof. Dr. Paulo Cesar Centoducatte
  Instituto de Computação (IC) - UNICAMP
- Prof. Dr. Sandro Rigo (Suplente)
  Instituto de Computação (IC) - UNICAMP

[1]**Apoiado financeiramente pela CAPES e Intel do Brasil.**

**Substitua pela ficha catalográfica**

# Estudo e Implementação da Otimização de Preload de Dados Usando o Processador XScale

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Márcio Rodrigo de Oliveira e aprovada pela Banca Examinadora.

Campinas, 11 de Julho de 2005.

Prof. Dr. Guido Costa Souza Araújo
(Orientador)

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

**Substitua pela folha com a assinatura da banca**

# Resumo

Atualmente existe um grande mercado para o desenvolvimento de aplicações para sistemas embutidos, pois estes estão fazendo parte crescente do cotidiano das pessoas em produtos de eletrônica de consumo como telefones celulares, palmtop's, agendas eletrônicas, etc.

Os produtos de eletrônica de consumo possuem grandes restrições de projeto, tais como custo reduzido, baixo consumo de potência e muitas vezes alto desempenho. Deste modo, o código produzido pelos compiladores para os programas executados nestes produtos, devem executar rapidamente, economizando energia de suas baterias. Estes melhoramentos são alcançados através de transformações no programa fonte chamadas de otimizações de código.

A otimização *preload* de dados consiste em mover dados de um alto nível da hierarquia de memória para um baixo nível dessa hierarquia antes deste dado ser usado. Este é um método que pode reduzir a penalidade da latência de memória.

Este trabalho mostra o desenvolvimento da otimização de *preload* de dados no compilador Xingo para a plataforma Pocket PC, cuja arquitetura possui um processador XScale. A arquitetura *XScale* possui a instrução *preload*, cujo objetivo é fazer uma pré-busca de dados para a cache. Esta otimização insere (através de previsões) a instrução *preload* no código intermediário do programa fonte, tentando prever quais dados serão usados e que darão *miss* na cache (trazendo-os para esta cache antes de seu uso). Com essa estratégia, tenta-se minimizar a porcentagem de *misses* na cache de dados, reduzindo o tempo gasto em acessos à memória.

Foram usados neste trabalho vários programas de benchmarks conhecidos para a avaliação dos resultados, dentre eles destacam-se DSPstone e o MiBench. Os resultados mostram que esta otimização de *preload* de dados para o Pocket PC produz um aumento considerável de desempenho para a maioria dos programa testados, sendo que em vários programas observou-se uma melhora de desempenho maior que 30%!

# Abstract

Nowadays, there is a big market for applications for embedded systems, in products as cellular phones, palmtops, electronic schedulers, etc.

Consumer electronics are designed under stringent design constraints, like reduced cost, low power consumption and high performance. This way, the code produced by compiling programs to execute on these products, must execute quickly, and also should save power consumption. In order to achieve that, code optimizations must be performed at compile time.

Data preload consists of moving data from a higher level of the memory hierarchy to a lower level before data is actually needed, thus reducing memory latency penalty.

This dissertation shows how data preload optimization was implemented into the Xingo compiler for the Pocket PC platform, a XScale based processor. The XScale architecture has a preload instruction, whose main objective is to prefetch program data into cache. This optimization inserts (through heuristics) preload instructions into the program source code, in order to anticipate which data will be used. This strategy minimizes cache misses, allowing to reduce the cache miss latency while running the program code.

Some benchmark programs have been used for evaluation, like DSPstone and MiBench. The results show a considerable performance improvement for almost all tested programs, subject to the preload optimization. Many of the tested programs achieved performance improvements larger than 30%.

"A grandeza não consiste em receber honras, mas em merecê-las"
Aristóteles

# Agradecimentos

Primeiramente gostaria de agradecer aos meus pais Sebastião e Derci e minha noiva Anyéle, pessoas que amo e que sempre me apoiaram nesta trajetória, e se não fosse por eles eu não teria conseguido concluir mais este desafio.

Agradeço ao professor Guido Araújo que me orientou ao longo do desenvolvimento desta pesquisa, o qual pude contar com sua experiência, sabedoria e amizade.

Aos membros da banca que se dispuseram a participar na avaliação deste trabalho, e que encontraram tempo em suas agendas lotadas de compromissos para ler o mesmo.

À CAPES e em particular a Intel do Brasil (Srs. Américo Tomé e Ruy Castro) pelo apoio no financiamento desta pesquisa.

Ao amigo Wesley Attrot, um dos pais do compilador Xingo, que me ajudou no desenvolvimento desse trabalho com idéias e soluções de alguns problemas encontrados.

Aos amigos de convivência Billo, Cleyton, Devegili, Felipe, Marcos, Matias e Patrick, e aos amigos do laboratório LSC (Laboratório de Sistemas Computacionais) do qual pude contar com suas amizades nestes dois anos de mestrado.

Ao Alex por ter implementado a otimização de *Loop Unrolling* utilizada neste trabalho.

A todos os professores com os quais fiz disciplinas, que difundiram seu vasto conhecimento para o aprendizado de assuntos interessantes e importantes na minha formação acadêmica e profissional.

Ao grupo de estudos da sala 80, onde compartilhamos quatro meses de estudos intensos para a disciplina de Teoria da Computação.

A todos os amigos que fiz durante este período, amizades valiosas que jamais me esquecerei.

Finalmente gostaria de agradecer a Deus, por me guiar em mais este caminho de sabedoria e realização.

A todos estes o meu mais sincero obrigado por fazerem parte de uma das etapas mais importantes da minha vida.

# Sumário

# Lista de Tabelas

# Lista de Figuras

# Capítulo 1

# Introdução

O crescente aumento no número de sistemas embutidos no cotidiano das pessoas deve-se ao barateamento da tecnologia de fabricação dos chips de silício usados por esses sistemas. Assim, o mercado tende a aumentar a demanda por tais dispositivos, em produtos de eletrônica de consumo como agendas eletrônicas, controle e segurança dos carros, eletrodomésticos, telefones celulares, etc.

É um equívoco pensar que a maioria dos processadores produzidos atualmente estão dentro de computadores ou PC's[1]. Estudos de mercado apontam uma fatia de 98% para o mercado de processadores embutidos e somente 2% para os PC's, estações de trabalho e servidores [39]. Como esta fatia de mercado é muito grande, a concorrência entre fabricantes de tais dispositivos e dos serviços desenvolvidos para esses dispositivos tende a ser grande também.

Um exemplo do cenário mundial na área de sistemas embutidos é o de telecomunicações. Empresas que fabricam telefones celulares estão em constante inovação de seus produtos para atender à necessidade de seus clientes, criando novas aplicações como jogos, acesso à WEB, e-mail, mensagens instantâneas, imagens de TV's, câmeras embutidas, mp3 player, etc.

Estas aplicações estão se tornando cada vez mais complexas, e o tempo de projeto para o desenvolvimento destas aplicações deve ser curto para atender a demanda de mercado, permitindo assim o lançamento de novidades à frente de empresas concorrentes.

Por outro lado, produtos de eletrônica de consumo possuem grandes restrições de projeto como custo reduzido, baixo consumo de potência e muitas vezes um alto desempenho.

Deste modo, o código produzido pelos compiladores para os programas executados nestes produtos devem rodar mais rápido, ocupar menos espaço e gastar menos energia. Estes melhoramentos são alcançados através de transformações no programa fonte chamadas de otimizações de código.

[1] Computadores Pessoais

Uma otimização deve preservar a semântica de execução dos programas, isto é, não pode alterar a saída de um programa para uma dada entrada, ou causar um erro que não estava presente na versão original do programa.

Um dos problemas enfrentados no projeto de sistemas embutidos é que eles em geral possuem memória extremamente limitada para armazenar dados e aplicações, bem como necessitam ter um baixo consumo de energia para desempenhar suas funções.

Além disso, existe uma diferença crescente entre o desempenho do processador em relação à memória DRAM. O desempenho dos processadores tem aumentado em uma taxa expressiva na última década. Essa tendência tem sido mantida por contínuas inovações de arquitetura e avanços na tecnologia de fabricação de processadores. Com o aumento nas taxas de *clock* e com o uso de paralelismo em nível de instruções, a velocidade dos processadores ainda irá aumentar expressivamente[30]. Em contraste, a memória DRAM obteve um aumento no desempenho a uma taxa muito menor como pode ser visto na figura 1.1 [53]. Podemos perceber por esta figura que a taxa de aumento em desempenho dos processadores é duplicada a cada 1.5 anos, enquanto o desempenho da memória DRAM é duplicada a cada 10 anos. A partir dessa observação, pesquisadores vem objetivando diminuir essa enorme diferença de desempenho entre estes componentes. Por isso a latência de memória é o maior assunto de pesquisa para muitos processadores modernos. Colocar a cache entre o processador e os módulos de memória é obviamente o primeiro passo para reduzir as latências médias de memória, entretanto, somente elas não são suficientes [35]. Deste modo, existe a necessidade de técnicas visando reduzir ou esconder a latência de acesso à memória principal [43].

## 1.1 Redução da Latência da Memória através da técnica de Preload de Dados

Técnicas de hardware como *caches não-bloqueantes* [24], *buffers de escrita* [37], e técnicas de software como transformações locais [44], têm sido usadas para reduzir a penalidade de latência de memória.

Uma técnica promissora para melhorar o desempenho do subsistema de memória para tentar acompanhar o alto desempenho dos processadores atuais, é a busca antecipada de dados controlada por software (*preload*).

Melhor do que esperar por um cache *miss* de uma referência, é iniciar a busca antecipada desta na memória. O algoritmo de *preload* de dados antecipa tais *misses* e despacha uma instrução de *preload* para estas referências no sistema de memória [43].

*Preload* de dados consiste em mover dados de um nível alto da hierarquia de memória para um nível baixo dessa hierarquia antes deste dado ser usado. Este é um método que

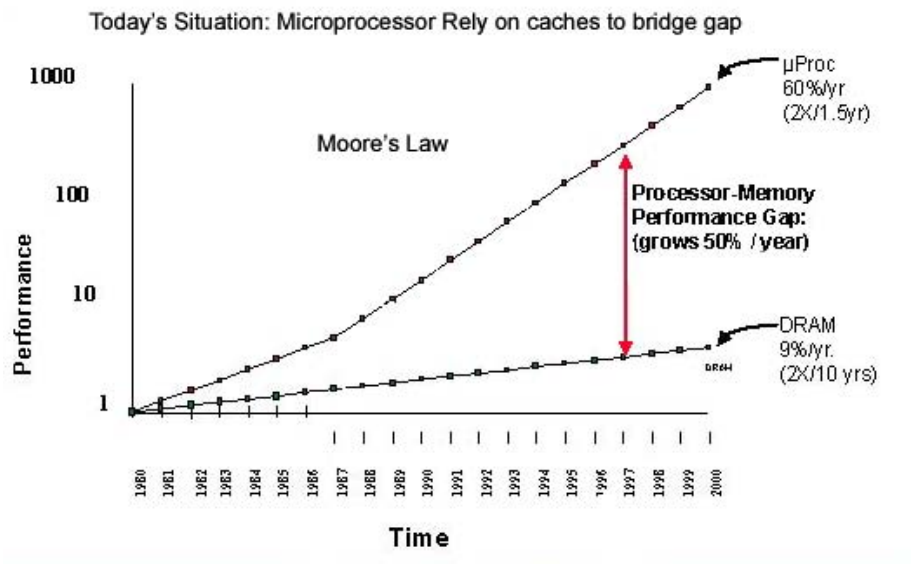



Figura 1.1: Diferenças em desempenho entre o processador e a memória DRAM [53]

pode reduzir enormemente a penalidade da latência de memória [35].

Enquanto *preload* de dados é útil para esconder essa latência de leitura de dados da memória, despachar *preloads* incorrem em *overheads* de instruções e podem aumentar o número de requisições de leitura no sistema de memória. Cuidados devem ser tomados para assegurar que tais *overheads* não excedam os benefícios [30].

Embora o uso de grandes hierarquias de caches tenha provado ser efetivo em reduzir a média de penalidade por acesso a memória para programas que mostram um alto grau de localidade em seus padrões de endereçamento, é comum estes gastarem mais do que a metade do seu tempo de execução em *stalls* de requisições de memória [42].

Este trabalho demonstra o estudo e a implementação da otimização de *preload* de dados por software no compilador Xingo, visando executar programas na arquitetura *XScale* (Intel)[51]. O *XScale* possui em seu conjunto de instruções a instrução *preload* (que faz uma pré-busca de dados na cache). Assim, um programador experiente, ou um compilador eficiente pode inserir esta instrução no código do programa fonte tentando prever quais dados serão usados e que resultarão em *miss* na cache, trazendo-os para a cache antes de seu uso. Com essa estratégia, tenta-se minimizar a porcentagem de *miss* na cache de dados, colocando estes dados em uma das linhas da cache antes de seu uso.

*Preload* é mais freqüentemente usado dentro de laços responsáveis por grandes computações em vetores. Tais laços oferecem excelentes oportunidades para o uso do *preload*, e estes são comuns em códigos científicos exibindo uma utilização pobre da cache, e freqüen-

temente possuem padrões de referências previsíveis. Achando estes padrões em tempo de compilação, instruções *preload* podem ser colocadas dentro do corpo dos laços para que os dados usados em uma futura iteração possam ser buscados durante a iteração corrente [42].

Aplicações de propósito gerais também mostram muito mais reuso de dados do que aplicações científicas, causando uma alta utilização da cache, o que diminui os benefícios do *preload* de dados caso aplicado para estas [42].

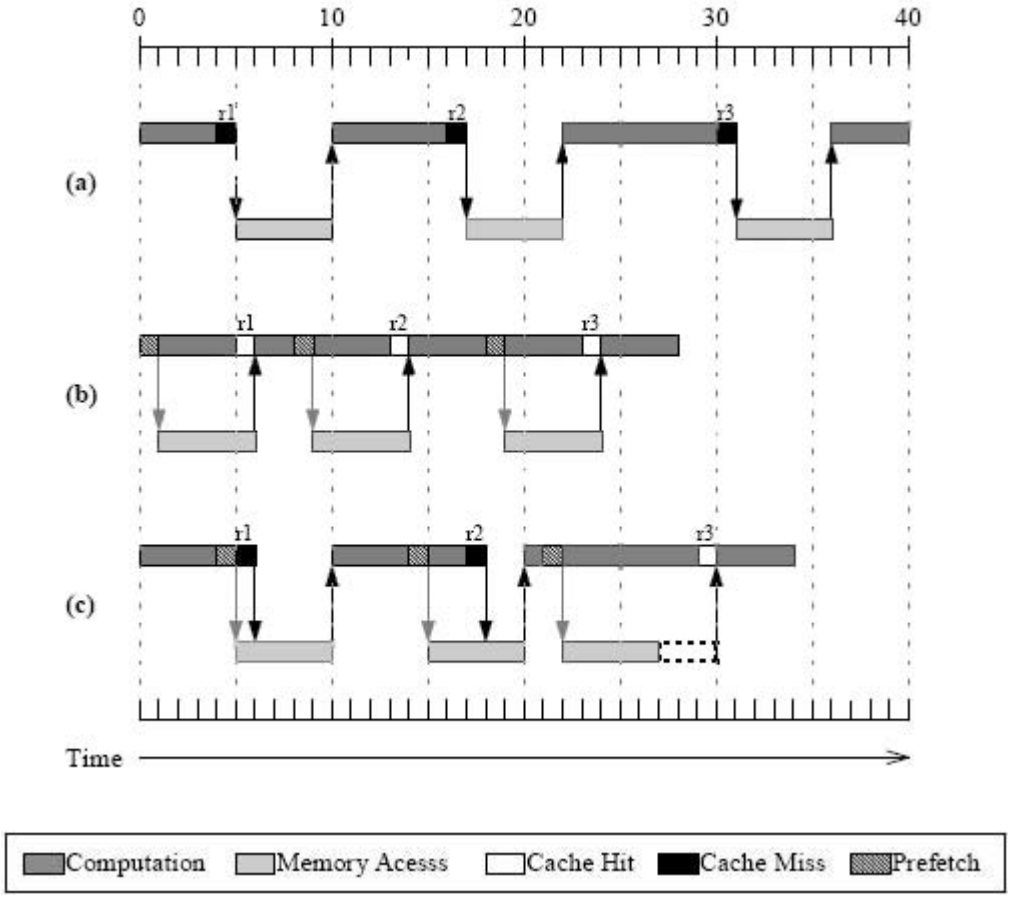



Figura 1.2: Diagrama de execução (a) - sem *preload*, com (b) - *preload* perfeito e (c) - *preload* degradado (extraído de [41])

A figura 1.2 mostra os tipos de *preload* de dados que podem ocorrer com o uso da otimização. Em cada caso as barras superiores representam a execução do programa no processador, as flechas que descem indicam as requisições do *preload* ou acessos à memória, e as que sobem o retorno do dado da memória para a execução. As barras

inferiores indicam as latências da busca do dado na memória. Os casos mais comuns são:

**(a) sem *preload*** - processador deve parar (se não houver cache não bloqueante e se houver dependência de dados) e esperar até que a referência de memória seja buscada.

**(b) *preload* perfeito** - os despachos de instruções de *preload* ocorreram de forma perfeita, pois quando o processador precisa do dado, este havia acabado de ser gravado na cache. Assim, o dado não tende a ser substituído antes de ser usado (não havendo poluição da cache), e o dado não chega depois que o processador deva usá-lo.

**(c) *preload* degradado** - neste caso o *preload* é despachado tarde o bastante para que o processador tenha que entrar no estado de espera por um período de tempo menor que o período da latência da busca dos dados.

Na figura (fig. 1.2 (a)), os blocos de dados associados como referências de memória $r1$, $r2$ e $r3$, não são encontradas na cache e devem portanto ser buscados da memória principal. Assumindo uma unidade de execução em ordem, o processador irá ser paralisado (se a cache for bloqueante, ou havendo dependência de dados) enquanto espera o bloco correspondente da cache ser buscado. A política de busca irá sempre resultar em um cache *miss* para o primeiro acesso no bloco da cache.

Melhor do que esperar por um cache *miss* para executar uma busca na memória, *preload* de dados antecipa tais *misses* e despacha uma instrução *preload* para o sistema de memória antes da atual referência de memória (fig. 1.2 (b)). Este *preload* trabalha em paralelo com a computação do processador, permitindo ao sistema de memória tempo para transferir o dado desejado da memória principal para a cache de dados. Idealmente, o *preload* irá completar esta transferência no tempo do processador acessar o dado necessário na cache sem parar o processador.

Na figura (fig. 1.2 (c)), os *preloads* para as referências $r1$ e $r2$ são despachados tão tarde que não evitam a paralisação do processador, embora o dado para $r2$ seja buscado um pouco antes para ter algum benefício. O dado de $r3$ chega cedo o bastante para esconder toda a latência de memória, mas deve ser mantido durante algum tempo na cache antes que seja usado pelo processador. Durante este tempo, este dado é exposto à política de substituição da cache, e pode ser retirado da cache antes de seu uso. Quando isto ocorre, o *preload* é inútil e tende a piorar o tempo de execução. Isso é chamado de poluição da cache.

## 1.2 Motivação

A redução dramática no custo da fabricação de *chips* de silício e os grandes melhoramentos na produtividade de ferramentas automáticas de projeto estão abrindo a possibilidade para computação de baixo custo em dispositivos portáveis e eletrônicos.

Atualmente as grandes montadoras de veículos automotivos estão cada vez mais utilizando microprocessadores para a segurança e comodidade de seus clientes. Exemplos de sistemas embutidos em veículos são: a injeção eletrônica, o sistema antiderrapagem, controle da parte elétrica, caixa-preta que registra dados importantes em um acidente, segurança do veículo com códigos criptografados nas chaves, etc. Outro segmento de mercado de sistemas embutidos é o setor de eletrônica de consumo. Neste caso, *palmtop's* e *handhelds* têm um papel central. Segundo notícia publicada pelo site WinCEBrasil [2] [48], as vendas mundiais de *handhelds* já ultrapassam 2,8 milhões de unidades.

Como o mercado de processadores embutidos está em constante expansão, as funcionalidades dos produtos e serviços oferecidos para estes dispositivos também crescem rapidamente. O resultado disto é um aumento no tamanho e na complexidade das aplicações que executam nestes dispositivos. Por outro lado, os projetos de sistemas embutidos são extremamente restritos sob o ponto de vista de custo, desempenho e consumo de potência.

Compiladores têm um papel fundamental no desenvolvimento de aplicações para sistemas embutidos. Como a complexidade destes sistemas cresce continuamente, a geração de código deve ser baseada em linguagens de alto nível, de modo a diminuir o tempo de projeto.

Como o mercado de *palmtops* e *handhelds* está em ascensão, e já que estes sistemas possuem restrições em tempo de execução e no gasto de energia, é extremamente motivante que exista um compilador que aproveite sabiamente os recursos do processador, para otimizar ao máximo as aplicações e que estas pudessem executar em menos tempo, em especial para o sistema operacional *Windows CE/ Pocket PC*, que possui uma grande fatia de mercado. O sistema operacional *Windows CE* superou o *Palm OS* pela primeira vez, revelou uma pesquisa do Gartner. O *Windows CE* respondeu por 48,1% dos PDA's vendidos no período, enquanto o *Palm OS* totalizou 29,8%. No mesmo trimestre de 2003, o sistema *Palm OS* detinha 46,9% do mercado[48].

No caso de *handhelds/palmtops* baseados no *Windows CE*, existem dois compiladores disponíveis comercialmente: o *Embedded Visual C++ 3.0/4.0 da Microsoft Corporation* [49] e o *Intel C/C++ Compiler* [50], mas ambos não implementam a otimização de *preload* de dados de forma automática [3].

Este trabalho tem como principal motivação a implementação da otimização *preload*

[2] Publicada na data de 31-01-2005 07:41:32

[3] Até a data do presente trabalho não se tinha informação sobre a existência da otimização *preload* nestes compiladores

de dados no compilador Xingo [5], e com este permitir a compilação de aplicações para o *Pocket PC* usando os compiladores mencionados acima como *backend* [4]. Assim, as aplicações desenvolvidas para estes sistemas executarão mais rapidamente (escondendo a latência de acesso à memória).

Este trabalho está organizado da seguinte forma. No capítulo 2 são apresentados os trabalhos relacionados a esta otimização. O capítulo 3 discute brevemente as ferramentas e a infra estrutura que foram utilizadas para a realização deste trabalho. Posteriormente no capítulo 4, é descrito o mecanismo de funcionamento da otimização de *preload* de dados. O capítulo 5 descreve a implementação desta otimização no compilador Xingo e no capítulo 6 são relatados os resultados experimentais. Finalmente no capítulo 7 são apresentadas as dificuldades encontradas, conclusões e trabalhos futuros.

[4]Backend é o módulo de um compilador que transforma o programa fonte em código de máquina

# Capítulo 2

# Trabalhos Relacionados

Vários trabalhos sobre *preload* de dados tanto com abordagem em software quanto em hardware, têm sido propostos na literatura a fim de melhorar o desempenho das latências de acesso à memória.

Desde meados de 1960, estudos realizados por Anaker e Wang [2] reconheceram o benefício de buscar múltiplas palavras da memória principal para a cache. Descobriram que tais blocos de memória que seriam usados poderiam ser trazidos juntos com blocos vizinhos da memória, com o intuito de levar vantagem da localidade espacial das referências de memória. Técnicas de software são mais recentes.

Técnicas de hardware como *caches não bloqueantes* [24], *buffers de escrita* [37] e técnicas de software tais como transformações locais [44], têm sido usadas para reduzir a penalidade de latência de memória.

Esta otimização de *preload* de dados já foi implementada em alguns compiladores como o compilador da IBM [7] e da HP [35], que tipicamente buscam referências da memória em laços cujos endereços possuem *strides*[1] estáticos. Estas técnicas possuem a limitação de que o *stride* deve ter tamanho fixo e conhecido em tempo de compilação.

A desvantagem de *preload* via software é que existe um *overhead* adicional da instrução necessária para despachar as requisições. Entretanto, hoje com os processadores superescalares, é possível sobrepor a latência desta instrução com outras computações [29].

Software *preload* pode freqüentemente tirar proveito de informações disponíveis em tempo de compilação (como caminhos de execução de um programa, informações de laços, cálculo da latência de um caminho de execução, número de referências à memória, etc) para escalonar *preloads* mais precisamente do que técnicas de hardware. Na prática, não é possível predizer exatamente quando escalonar um *preload* para que o dado chegue na cache exatamente quando este irá ser usado pelo processador. O tempo de execução entre

[1] Deslocamento em bytes na memória

o *preload* e o casamento com o uso deste dado na instrução *LOAD* pode variar devido à incertezas no fluxo de execução do programa, assim como às latências de acessos a memória. Estas incertezas devem ser consideradas na decisão de onde devem ser colocadas as instruções *preload* no programa [42].

Em [45] são tratadas as transformações de laços mais comuns como *fission, fusion, tiling, interchanging e loop unrolling*, que permitem melhorar a utilização da cache, o escalonamento de instruções e alocação de registradores. Neste trabalho, é mostrado como fazer decisões baseadas no conhecimento da cache de dados e detalhes do processador.

No início da década de 90 é que os primeiros trabalhos de *preload* de dados começaram a surgir, mas nessa época os trabalhos ainda eram imaturos e implementados em ambientes de simulação e em compiladores acadêmicos não completos (compiladores sem instruções de saltos por exemplo).

Os trabalhos de Vanderwiel [42][43] trazem a idéia principal no qual este trabalho foi baseado. Uma das primeiras técnicas utilizadas nestes trabalhos foi o de usar o desenrolamento de laços para aproveitar todos os usos de um vetor que resultarão em *hit* em uma linha da cache. Uma técnica mais refinada que obtém melhores resultados, é além de implementar o desenrolamento do laço, utilizar a técnica conhecida como software pipeline [31]. Esta técnica tem como objetivo evitar a ocorrência de cache *miss* no uso dos dados do vetor durante a primeira iteração do laço, e para que não haja *preloads* desnecessários despachados na última iteração do laço. Nos capítulos 4 e 5 explicaremos mais detalhadamente este algoritmo.

A aproximação de somente desenrolar o laço por um fator $N$ dependente dos usos de um vetor neste laço, foi usado nesse trabalho sem muito sucesso. A segunda técnica também implementada neste trabalho (a de software pipeline), obteve um maior êxito nos resultados. Nesta técnica, ao invés de escolher um fator de desenrolamento $N$ visando os usos no vetor de dados, o fator de desenrolamento do laço depende do cálculo da formula abaixo:

$$D = L/S$$

Onde $L$ é a média de latência de cache *miss*, medida em ciclos de processador, e $S$ é o número estimado de ciclos no caminho mais curto possível de execução em uma iteração do corpo do laço.

Vanderwiel [43] indica que o compilador deve ser confiável para predizer os padrões de acesso à memória, restringindo a laços com padrões simples de referências à vetores. Estes laços são comuns em programas científicos, e menos comuns em aplicações de propósito geral. Estratégias para as aplicações de propósito geral com estruturas de dados irregulares, têm sido pesquisadas em outros trabalhos [26],[27] obtendo sucesso limitado. Tais

aplicações não seguem um caminho de execução predizível, sendo difícil antecipar seus padrões de referência à memória.

Santhanam, Gornish e Hsu [35], implementaram o *preload* de dados em um compilador comercial da *Hewlett-Packard's* para ser usado no processador PA 8000, o primeiro processador dessa empresa a implementar a arquitetura 64-bit PA-RISC 2.0. Este processador inclui instruções definidas para *preload* de dados, além de ter uma cache não bloqueante. Ele usa a distância de *preload* como utilizada em [30], onde a latência de memória é dividida pelo tempo de execução estimado em uma iteração do laço, multiplicado pelo *stride*[2] da referência. Neste trabalho o compilador usa a estratégia de fazer somente o desenrolamento do laço explorando a localidade espacial dos *preloads* (não despacha *preloads* desnecessários para a mesma referência de memória).

Nesse mesmo trabalho, também é levado em conta o fator de incremento da variável de indução de controle do laço. Traz um estudo bastante interessante a respeito do desempenho do sistema de memória neste processador, trazendo assuntos como *misses* conflitantes e conflitos com bancos de memória (que pode aumentar significativamente o tempo de retorno da linha da cache). Mostra que muitas vezes é difícil aplicar esta otimização de *preload* de dados, pois nem sempre em tempo de compilação se conhece o volume de dados que será acessado em um laço, como exemplo, não saber as fronteiras de um laço. Demonstra que acessos à memória dentro de laços mais internos com *strides* uniformes são considerados candidatos para se fazer o *preload* de dados. Também mostra que a expressão do endereço de memória é reconstruída rastreando-se as definições das variáveis (ou registradores) das instruções no corpo do laço para trás, começando pelo registrador que a instrução *load* define. Neste trabalho só são reconstruídas expressões que forem funções lineares simples, com a variável de indução básica de controle do laço nesta expressão.

No trabalho de Mowry [30], o *preload* de dados controlado por software foi implementado no compilador otimizante SUIF (*Stanford University Intermediate Form*). Neste trabalho foi verificado que alguns programas melhoram seu desempenho em até um fator de dois. O algoritmo de *preload* de dados implementado nesse trabalho usa *software pipelining* mesmo em laços aninhados, o que aumenta consideravelmente o tamanho do código. Este algoritmo usa a análise de reuso de dados por formulações matemáticas, descobrindo se o padrão de acesso é de reuso temporal, espacial ou de grupo. Ele calcula o fator de antecipação do despacho do *preload* como sendo a latência de acesso à memória, dividido pela latência do menor caminho do corpo do laço. Os experimentos foram realizados tendo como base na arquitetura R4000 e alguns programas de benchmarks como o SPEC [40], SPLASH [36], NAS Parallel [6]. Os resultados obtidos são bons no que diz respeito à economia no tempo de acesso à memória chegando de 70 a 90% de diminuição

[2]Deslocamento em bytes na memória no uso de uma referência (vetores)

deste tempo. Dois algoritmos são propostos, um chamado de *preload* indiscriminado que busca todas as referências de acesso à memória, e o *preload* seletivo que busca somente as referências que possuem localidade espacial de diferentes linhas da cache. O *preload* seletivo apresenta um ganho bem melhor que o indiscriminado, pois o processador é paralisado se o buffer de pendências ficar cheio de requisições de acesso à memória (para maiores explicações sobre pend buffer veja a seção 5.2).

O trabalho de Porterfield entitulado *"Software Prefetching"* [8] é um dos artigos mais antigos que fala sobre esse assunto. Neste trabalho é utilizado um sistema de simulação chamado *PFC-Sim* [33], [9] no qual é possível estudar os efeitos sobre uma cache arbitrária executando qualquer programa. O benchmark RICEPS [33], [10] (usado em pesquisa de compiladores) foi utilizado neste trabalho. Este trabalho não usa o desenrolamento de laços, e simplesmente coloca a instrução *preload* logo após da referência à memória adicionado a um fator de deslocamento. Os resultados apresentados foram bons, mas em um ambiente simulado.

Tentativas em estabilizar estratégias de software *preload* para aplicações de propósito geral são impedidas por seus padrões de referenciamento de dados irregulares[12], [26], [27].

Dadas as estruturas de controle complexas típicas de aplicações de propósito geral, uma vez que um bloco de cache é acessado, existe uma chance pequena que vários sucessivos blocos da cache irão também ser requisitados quando estruturas de dados tais como grafos e listas ligadas são usadas (pois estas não possuem localidade espacial na memória). Finalmente, estas aplicações possuem características de alta localidade temporal que freqüentemente resultam na alta utilização da cache, diminuindo assim os benefícios do *preload* [43].

O trabalho [28] implementa a técnica de inserção de instruções de *preload* na plataforma Alpha, usando Pixie [38] (faz o profile do fluxo de controle e coleta informações de *strides*) e Spike [13], [55] - ferramenta otimizante para executáveis Alpha - que foi utilizada para inserir *preloads*. Nesse trabalho usou-se programas do benchmark SPEC2000 e obteve-se um *speedup* de mais de 50% para algumas aplicações. Este trabalho explorou estruturas de dados e matrizes que são alocadas dinamicamente e que possuem um *stride* constante entre seus elementos, mas que não são modificados ao longo do programa.

O trabalho em [47] propõe uma combinação de técnicas de software e hardware chamada de *relocação de dados* com *preload* de dados. Grande parte do *overhead* da cópia de dados é eliminado através do uso de um hardware especial. Na *relocação de dados*, as referências de um vetor acessado em laços aninhados são seqüencialmente mapeadas na cache antes que eles sejam acessados. As operações de relocação são invocadas por instruções explícitas que o compilador insere. O compilador também insere uma declaração dentro do código original para a alocação do buffer de relocação, reservando espaço para

o dado ser relocado na memória. Um exemplo pode ser visto na figura 2.1.

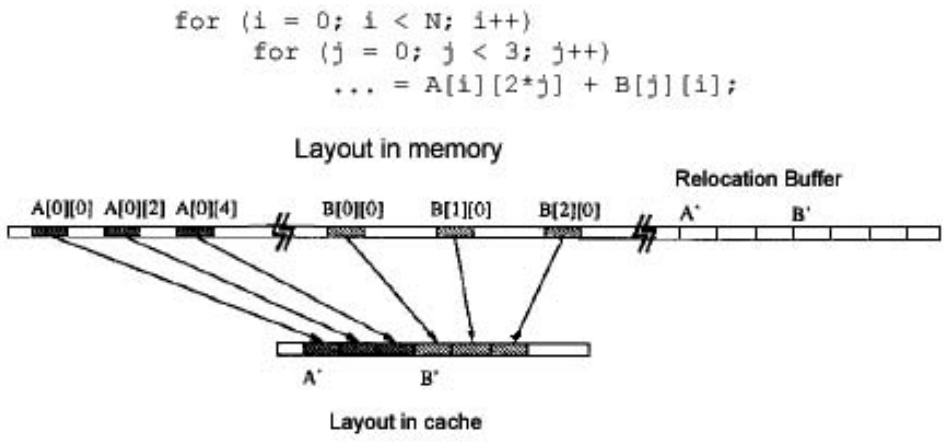



Figura 2.1: Relocação de dados [47]

Um hardware especial que mapeia e comprime os dados que serão acessados dentro do buffer virtual é anexado à unidade da cache. A relocação pode ser executada enquanto o *preload* busca o dado da memória para a cache sem paralisar a CPU. Visando acessar o dado relocado durante a computação, o compilador substitui as referências do vetor original com referências correspondentes de relocação do buffer.

Além de melhorar o acesso aos dados, esta técnica melhora a localidade espacial de acessos a vetores em um laço aninhado (diferentes áreas de um vetor - grandes *strides* - podem ser ordenados seqüencialmente). A desvantagem nesta abordagem técnica é que são necessárias novas instruções para construir o buffer de relocação, além de ocupar mais espaço na memória.

O trabalho de Luk e Mowry [29], explora aplicações contendo estrutura de dados. Identifica os problemas (como de estruturas não possuir localidade espacial na memória) de se fazer *preload* para estas aplicações, propondo alguns esquemas para a solução destes problemas. Estes esquemas foram implementados no compilador otimizante SUIF, utilizando para verificação de desempenho o benchmark Olden, que foi simulado em processadores semelhantes ao MIPS R10000. Esta estratégia explora uma otimização que potencialmente melhora a localidade espacial em estruturas de dados (lineariazação de dados), mas mesmo assim, um número significativo de cache *misses* ainda permanecem.

Este trabalho define o termo distância $d$ que deve existir entre o uso de um nodo da estrutura de dados e o despacho do *preload* para outra estrutura. O termo $d$ é definido $d = L/W$, onde $L$ é a latência de acesso à memória e $W$ o tempo de computação estimado

entre o acesso entre nodos (em ciclos).

Este trabalho especifica três possíveis algoritmos, o primeiro *"greedy preload"* faz o *preload* de um nodo de uma estrutura de dados através do ponteiro do nodo atual. A principal vantagem deste é possuir baixo *overhead* em tempo de execução, pois só é necessário a inserção das instruções de *preload*. Esta técnica tem a desvantagem de só funcionar para distâncias $d$ iguais a um (pois é despachado *preload* para a próxima estrutura). A outra técnica explorada é a de *"History-pointer preload"*, que sintetiza novos ponteiros para as estruturas de acordo com a distância $d$ calculada. Nesta técnica é construída uma fila *FIFO* que armazena os padrões de acesso das estruturas de dados. Esta técnica é útil se os padrões de acesso a estas estruturas não mudam com a execução do programa. Também possui a desvantagem de possuir um *overhead* de armazenamento e execução (construção da fila) maiores que a técnica anterior. A terceira técnica chamada *"data-linearization preload"*, mapeia as estruturas de dados em área contígua na memória *heap*, armazenando-as em um vetor de estruturas. Este esquema também possui a desvantagem de ser voltado para estruturas de dados que não são alteradas freqüentemente. Para a primeira técnica os resultados experimentais mostram que de 10 programas, existe uma melhora significativa em apenas 4 deles. Com as outras 2 técnicas os resultados são um pouco melhores, especialmente para a terceira, mas esta possui sua aplicação para um número restrito de programas.

Esquemas de *preload* de dados via hardware procuram tipicamente por padrões em acessos anteriores para predizer o comportamento futuro. Técnicas de *preload* via hardware não podem predizer o acesso caótico de padrões que ocorrem em estruturas de dados mais complexas [29].

Estas técnicas de *preload* baseadas em hardware [43][12][14] não requerem o uso explícito de uma instrução *preload*, mas empregam uma monitoração por hardware a fim de inferir oportunidades de *preload* de dados. Embora hardware *preload* não implique em se ter algum *overhead* causado por instrução, isto freqüentemente gera mais *preloads* desnecessários do que *preload* via software. Isto acontece porque eles especulam o futuro de acesso em memória sem os benefícios das informações em tempo de compilação do *preload* por software. As aproximações por hardware podem causar mais poluição da cache e consumir largura de banda do barramento da memória desnecessariamente [43].

Na técnica de *preload* por hardware, existe uma lógica especial para monitorar e detectar os padrões de referência dos endereços de vetores, detectando *strides* constantes dentro de laços.

No trabalho de Chen[12], é usada uma nova cache chamada *Reference Prediction Table* ou RPT. Esta tabela mantém as informações para as instruções de leitura de memória mais recentemente usadas. A organização da tabela RPT pode ser vista na figura 2.2.

As entradas da tabela contêm o endereço de memória (fornecido pelo *Program Counter*

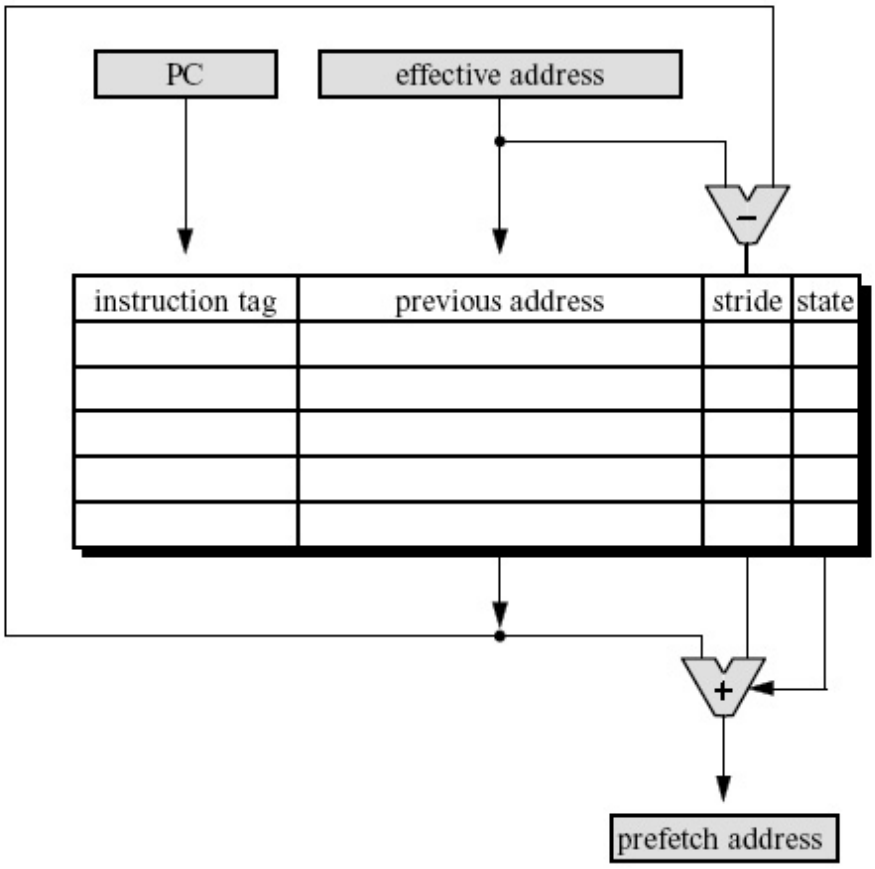



Figura 2.2: Tabela de predição de referências (RPT) [41]

*Register*), o endereço anterior acessado por esta instrução, o valor de *stride* calculado entre iterações dessa instrução, e um campo para gravar o estado corrente do diagrama de estados.

O diagrama de estado serve para ter certeza que o *stride* será constante entre as iterações do laço, e se este for o caso, será feito o *preload* para esta referência com base no *stride* calculado armazenado na RPT. Este diagrama de estados pode ser visto na figura 2.3.

Considere o exemplo da figura 2.4. Na execução da primeira iteração do laço mais interno deste exemplo, são incluídas todas as entradas dos usos dos vetores na tabela de predição com seus endereços de acesso à memória correntes no estado inicial (*initial*). Esta situação é mostrada na figura 2.4 (b).

Na segunda iteração os *strides* são computados com base no endereço atual em relação

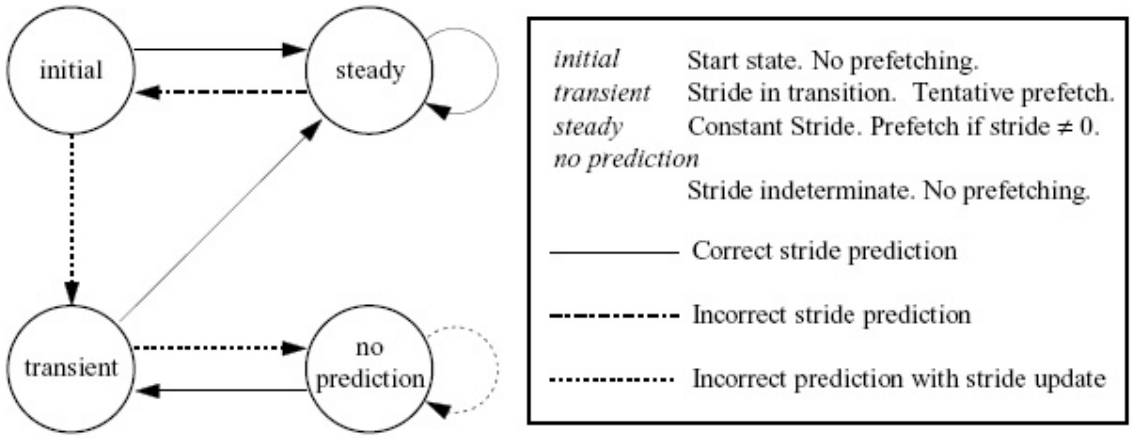



Figura 2.3: Diagrama de estados usado pelo RPT [41]

ao endereço anterior residente na RPT. Após esta computação os campos são atualizados nesta tabela. Neste caso, as referências dos vetores $b$ e $c$ são colocadas no estado *transient* pois os valores dos seus *strides* mudaram, indicando que este vetor está sendo percorrido entre as iterações. Os valores da RTP nesta iteração podem ser vistos na figura 2.4 (c).

Na próxima iteração, são novamente calculados os *strides* destas referências que estão na tabela, e verificado se os *strides* calculados pela nova iteração são os mesmos. Se forem, a entrada em seu estado é atualizada para o estado *steady*, indicando que deve ser despachado um *preload* para a referência atual mais o valor de seu *stride*, pois provavelmente estes vetores estão sendo acessados com *strides* constantes. Esta situação pode ser vista na figura 2.4 (d).

Esta aproximação tem a desvantagem de despachar o *preload* somente uma iteração à frente e que provavelmente não dará tempo para que os dados sejam colocados na cache antes de seu próximo uso.

Algumas outras técnicas tentam complementar esta deficiência, colocando um campo distância que indica quantas iterações a frente este dado deve ser buscado. Existe ainda o conceito de *lookahead program counter* (LA-PC) que tenta descobrir através de heurísticas como será a execução do programa à frente do *Program Counter*, obtendo assim a flexibilidade de despachar *preloads* bem antes do seu uso. Na técnica LA-PC pode ser usada a tabela de predição de saltos para sua melhor efetividade.

Software *preload* evita muitos dos *preloads* desnecessários que os mecanismos de hardware pode produzir. Estes *preloads* desnecessários podem substituir dados ativos dentro da cache com dados que poderiam nunca serem usados pelo processador. Além de causar

poluição da cache, *preloads* desnecessários gastam a largura de banda do barramento da memória [43].

```
float a[100][100], b[100][100], c[100][100];

...

for ( i = 0; i < 100; i++)
    for ( j = 0; j < 100; j++)
        for ( k = 0; k < 100; k++)
            a[i][j] += b[i][k] * c[k][j];
```

**(a)**

| Tag | Previous Address | Stride | State |
|---|---|---|---|
| *ld b[i] [k]* | 20,000 | 0 | initial |
| *ld c[k] [j]* | 30,000 | 0 | initial |
| *ld a[i] [j]* | 10,000 | 0 | initial |

**(b)**

| Tag | Previous Address | Stride | State |
|---|---|---|---|
| *ld b[i] [k]* | 20,004 | 4 | transient |
| *ld c[k] [j]* | 30,400 | 400 | transient |
| *ld a[i] [j]* | 10,000 | 0 | steady |

**(c)**

| Tag | Previous Address | Stride | State |
|---|---|---|---|
| *ld b[i] [k]* | 20,008 | 4 | steady |
| *ld c[k] [j]* | 30,800 | 400 | steady |
| *ld a[i] [j]* | 10,000 | 0 | steady |

**(d)**

Figura 2.4: Exemplo do funcionamento da tabela RPT [41]

# Capítulo 3

# Infra-estrutura Utilizada

Neste capítulo são discutidas as ferramentas que foram utilizadas para a realização deste trabalho.

## 3.1 Compilador Xingo (XCC)

Xingo [5] é um compilador da linguagem de programação ANSI C que está sendo desenvolvido com o objetivo de facilitar o desenvolvimento de pesquisas em otimização de código, novas técnicas em geração de código, bem como visa facilitar o desenvolvimento de compiladores para novas arquiteturas.

Xingo utiliza o *front-end* do LCC [15], que é um compilador para a linguagem ANSI C, desenvolvido por David Hanson e Christopher Fraser na Universidade de Princeton e nos laboratórios da AT&T. Este compilador transforma o programa fonte em *DAG's* que o Xingo usa para construir uma representação linear do programa fonte. O LCC está sendo usado por várias pessoas desde 1998 [5].

O Xingo possui duas representações intermediárias, uma delas composta por operações em opcodes, que recebe o nome de *XIR (Xingo Intermediate Representation)*, e tem como objetivo facilitar a implementação de otimizações independentes de máquina. Esta representação é próxima da linguagem *assembly*. A representação intermediária XDL é a mais interessante, pois é composta por operações na linguagem ANSI C que pode ser compilável por outro compilador ANSI C. Esta característica visa facilitar e tornar rápido o processo de validação de novas otimizações, pois permite acompanhar cada transformação que é realizada na representação intermediária, e faz com que este código ANSI C seja portável para qualquer arquitetura que possua um compilador para a linguagem C.

O Xingo também pode gerar código de máquina, pois ele utiliza um gerador de código para implementar seu *back-end*, de modo a poder ser rapidamente portável para várias arquiteturas.

O Xingo atualmente possui várias otimizações independentes de máquina implementadas, entre as quais destacam-se *Dead Code Elimination, Copy Propagation, CSE, Code Motion, Strength Reduction, Induction Variable Elimination, Constant Propagation, Pointer Optimization* e *PeepHole Optimization* [5].

Ele ainda oferece várias técnicas de análise de programas como *Data-Flow Analysis, Alias Analysis, Reaching Definitions, Liveness Analysis, Avaliable Expressions*, entre outras, que facilita a implementação de novas otimizações.

Atualmente o compilador Xingo encontra-se em fase de testes de desempenho e correção de erros. Para testar o compilador Xingo, estão sendo usados vários *benchmarks*, entre eles merece destaque o benchmark Nullstone [63], composto por mais de 6500 programas para testar a eficiências das otimizações desenvolvidas no compilador (são testadas mais de 40 tipos de otimizações).

Conforme os últimos dados obtidos em comparação com o compilador Linux GCC [54], o Xingo fica atrás deste em 50% na execução de todas as otimizações, sendo que em algumas delas o compilador Xingo consegue otimizar mais que o GCC. Além disso, neste benchmark existem algumas otimizações que são mais dependentes de arquitetura ao qual foi executada, e o Xingo ainda não possui um *back-end* para a arquitetura testada.

A seguir temos o exemplo de uma aplicação de ordenação de vetores escrito na linguagem C (fig. 3.1), e a representação intermediária em código ANSI C compilável gerada pelo compilador Xingo (fig 3.2).

## 3.2 A Arquitetura XScale e o Processador PXA250 - Intel

*XScale* é uma arquitetura pertencente à Intel baseada na segunda geração da família de processadores ARM. Ela é um núcleo que pode ser combinada com periféricos para fornecer *ASSP's*[1] em vários tipos de segmentos de mercado. Esta seção é baseada nas referências [18] [23] [19] e [20].

A arquitetura *XScale* é uma tecnologia *RISC superpipelined* que usa 7 estágios para operações inteiras e 8 estágios para operações de memória, e o processador PXA250 possuí um *clock* de 300MHz. O pipeline desta arquitetura pode ser visto na figura 3.3. Os primeiros estágios são comuns para busca de instruções (F1, F2). O próximo é o decodificador da instrução (ID) seguido pelo banco de registradores com a operação de deslocamento (*shift register*)(RF). Logo após encontram-se os dois estágios de execução (X1, X2), e o estado de *"write back"* (XWB). O *pipe* de memória ainda possui os estágios de acesso a cache de dados em D1 e D2 e *"write back"* na cache de dados (DWB). No

[1] *Applications Specific Standard Product*

```c
#include <stdio.h>
#include <malloc.h>

#define N 32000
int *a;

int main() {
    int i, j, maior, aux, pos = 0;

    a = (int*) malloc(sizeof(int)*N);

    for(i = 0; i < N ; i++)
    {
        maior = -99999999;
        for(j = i; j < N ; j++)
        {
            if (a[j] > maior)
            {
                maior = a[j];
                pos = j;
            }
        }
        aux = a[i];
        a[i] = maior;
        a[pos] = aux;
    }
    return 0;
}
```

Figura 3.1: Código Exemplo de uma função de ordenação escrito na linguagem C.

terceiro *pipe* se encontram os estágios que tratam as instruções de multiplicação [18].

A cache de dados possui o tamanho de 32KB, sendo uma cache 32-*way set associative*, sendo que cada *set* contém 32 *ways*. Cada *way* de um *set* contém 32 bytes (uma linha da cache) e um bit de validade. Existem também dois *"dirty bits"* para cada linha da cache, um para os 16 bytes mais acima e o outro para os 16 bytes mais abaixo [23].

Como a cache de dados possui 32 sets de 32 bytes, cada linha em um *set* alinha-se à fronteira de endereços de 1KB. A política de substituição na cache é o *"round robin"* e esta aceita as políticas de *write-back* e *write-through* dependendo da configuração da *MMU*[2] [23]. A figura 3.4 mostra a estrutura da cache em detalhes.

As caches do *XScale* são compostas de 32KB para instruções e 32KB para dados. Quando uma *cache miss* ocorre, tanto na cache de instruções quanto na de dados, gasta-se em média de 60 a 90 ciclos para buscar a linha da cache na memória, ficando em torno de 78 a 108 ciclos para o término do preenchimento desta na cache.

O processador usa quatro entradas para *fill e pend buffers*, permitindo o não bloqueamento e a operação de *"hit-under-miss"* com as caches de dados. Possui também 8 entradas para *write buffer*, fornecendo uma continua execução do processador enquanto o dado é escrito na memória.

[2] *Management Memory Unit*

```
1  static unsigned long _C7 = {128000};
2
3  char* a;
4
5  int main() {
6      int _t31;
7      int _t21;
8      char* _t24;
9      int _t75;
10     int maior;
11     int i;
12     char* _t6;
13
14 B1:
15     _t31 = 0;
16     _t6 = malloc(_C7);
17     a = ((char*) _t6 );
18     i = 0;
19     goto B2;
20 B2:
21     maior = -99999999;
22     _t75 = i;
23     goto B3;
24 B3:
25     if( _t75 < 32000 ) goto B5;
26 B4:
27     _t75 = i << 2;
28     _t24 = ((char*) _t6 ) + _t75;
29     _t21 = *((int*)_t24);
30     _t24 = ((char*) _t24 );
31     *((int*)_t24) = maior;
       _t75 = _t31 << 2;
       _t24 = ((char*) _t6 ) + _t75;
       *((int*)_t24) = _t21;
       goto B9;
   B5:
       _t21 = _t75 << 2;
       _t24 = ((char*) _t6 ) + _t21;
       _t21 = *((int*)_t24);
       if( _t21 <= maior ) goto B7;
   B6:
       _t31 = _t75 << 2;
       _t24 = ((char*) _t6 ) + _t31;
       maior = *((int*)_t24);
       _t31 = _t75;
       goto B7;
   B7:
       goto B8;
   B8:
       _t75 = _t75 + 1;
       goto B3;
   B9:
       i = i + 1;
       if( i < 32000 ) goto B2;
   B10:
       return 0;
       goto B11;
   B11:
       goto T;
   T:
       ;
   }
```

Figura 3.2: Exemplo de representação intermediária em código ANSI C compilável gerada pelo compilador Xingo.

Esta arquitetura tem uma instrução de *preload* de dados (*PLD*) em seu conjunto de instruções. O propósito dela é o pré-carregamento de dados para a cache de dados [23]. Esta instrução é como se fosse a instrução *LOAD*, com a diferença que esta é não bloqueante e para acessos o endereços de memória inexistente, não gera exceções.

Esta instrução permite esconder a latência de transferência de dados da memória, enquanto o processador continua a executar as instruções. Esta instrução é importante para o compilador, porque o uso adequado desta pode melhorar muito o desmpenho e a vazão do processador XScale. *Preload* de dados pode ser aplicado não somente a laços, mas também para qualquer referência de dados dentro de um bloco de código. *Preload* também pode ser aplicado à escrita de dados se o tipo da memória é habilitada como *"write allocate"*.

Os compiladores que não utilizam esta instrução usam instruções *load* para pré-carregar os dados da memória para cache. Esta técnica tem a desvantagem de usar um registrador para carregar os dados e requer registradores adicionais para subseqüentes *preloads*, aumentando assim a pressão nos registradores.

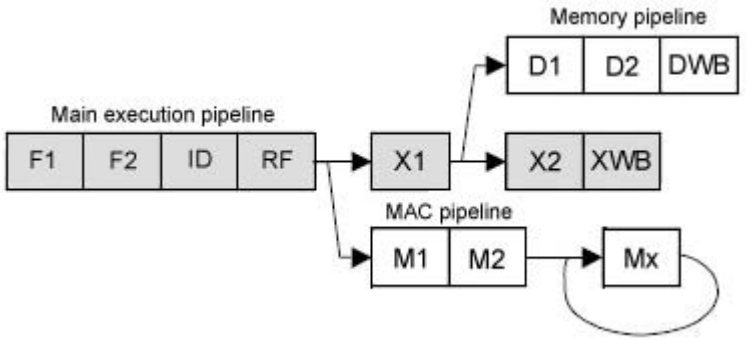



Figura 3.3: Pipeline do XScale [18]

A instrução *preload* não gera exceções, e se o chamado desta for para algum endereço nulo, o processador continuará a executar o programa normalmente.

Quando despachado um *preload* em um laço que opera sobre vetores, tem-se a vantagem de fazer este *preload* buscar os dados que serão usados em uma, duas ou mais iterações à frente. Isto dependerá do tamanho do corpo do laço e das latências das instruções deste corpo.

O uso inadequado desta instrução pode usurpar os recursos do sistema e degradar o desempenho do programa em execução. Isto acontece porque uma vez que as requisições de tráfico do barramento excedam a capacidade dos recursos do sistema, o processador pára de executar até que uma das requisições seja resolvida [23].

Os recursos de transferência de dados são:

- *4 fill buffers.*

- 4 *pending buffers.*

- 8 *write buffers* (divididas por meia linha da cache).

Os recursos da SDRAM são tipicamente:

- 4 bancos de memória.

- 1 buffer de página por banco referenciando um intervalo de endereços de 4K.

- 4 buffers de requisição de transferências.

Um *fill buffer* é alocado para cada *read miss.* Um *fill buffer* é também alocado para cada *write miss* se o espaço de memória é *"write allocate"* e mais um *pending buffer.* Uma leitura subseqüente para a mesma linha da cache não requer um novo *fill buffer*, mas

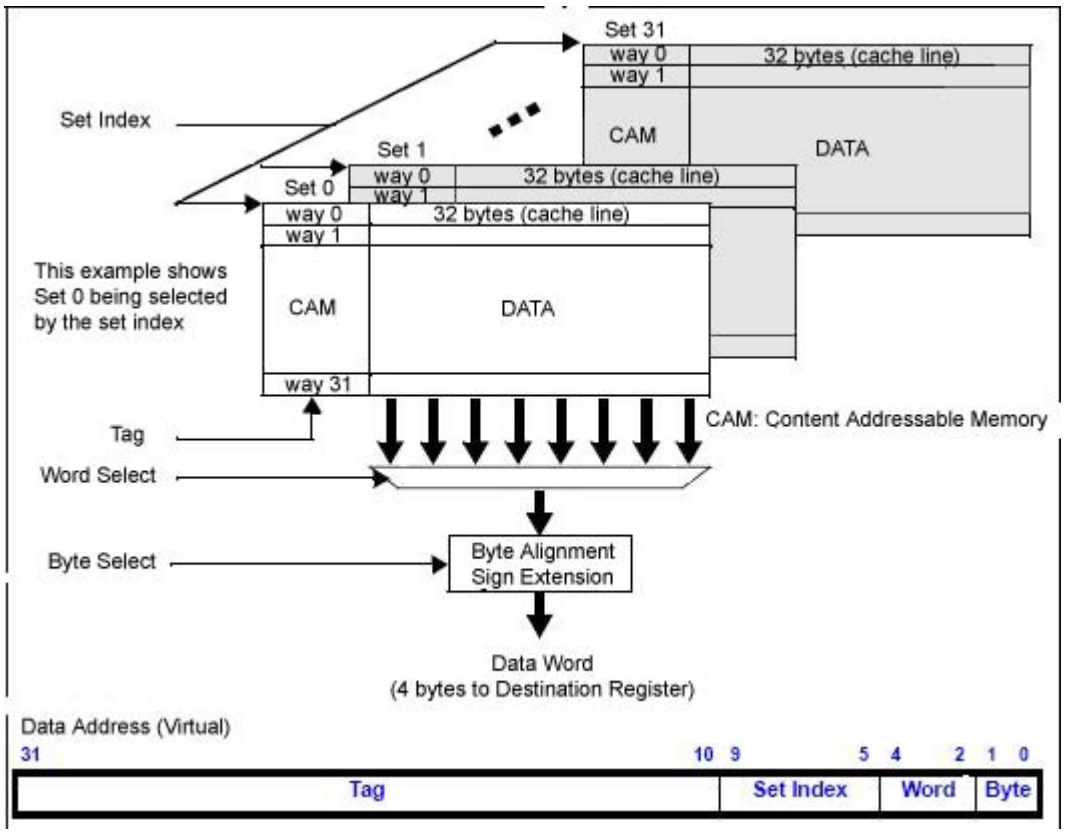



Figura 3.4: Organização da cache de dados [23]

requer um novo *pending buffer* e uma escrita subseqüente também irá requerer um novo *pending buffer*.

Quando adicionando instruções *preload*, cuidados devem ser tomados para assegurar que a combinação deste e das instruções que requisitam o barramento não excedam a capacidade de recursos do sistema descrito acima, ou o desempenho irá se degradar ao invés de melhorar.

A arquitetura *XScale* possui uma unidade de gerenciamento de energia, onde o processador pode ficar em vários estados dependendo da tarefa que estiver executando, e de acordo com cada estado gastar menos ou mais energia. Esta arquitetura também possui 128 entradas para *Branch Target Buffer*, mantendo o pipeline com a capacidade de escolhas corretas de saltos. Possui ainda uma unidade de monitoramento de desempenho (PMU), constituída de registradores contadores de eventos, e ainda uma unidade de depuração. O monitoramento dos eventos do processador que pode ser feito na execução de um programa são:

- ICache misses - Conta o número de *misses* na cache de instruções.
- ICache miss stall - Conta o número de ciclos de relógio gastos esperando a cache de instruções.
- Stall por dependências de dados - Conta o número de ciclos de relógio perdidos em dependências de dados.
- Instruction TLB miss - Conta o número de ITLB misses.
- Data TLB miss - Conta o número de DTLB misses.
- Número de instruções Branch - Conta o número de instruções branch executadas.
- Branch mispredicted - Conta o número de instruções branch que teve sua predição errada.
- Instruções executadas - Conta o número de instruções executadas no core.
- Data buffer stall duration - Conta o número de ciclos que o processador ficou paralizado pelo fato que não pode ser alocada uma requisição de leitura/escrita na memória.
- Data buffer stall count - Conta o número de vezes que o processador foi paralizado pelo fato de que não pode ser alocada uma requisição de leitura/escrita na memória.
- Data cache access - Conta o número de vezes que a cache de dados foi acessada tanto para leituras quanto para escritas.
- Data cache misses - Conta o número de vezes que um dados requisitado não estavam na cache de dados tanto para leituras quanto para escritas.
- Data cache writeback - Conta o número de vezes que dados são escritos da cache para a memória principal.

## 3.3 Pocket PC 2002

Pocket PC 2002 é uma versão do sistema operacional Windows para PDA's (Assistente Pessoal Digital). Vários fabricantes de hardware já usam este sistema operacional, pois ele possui facilidades de aplicativos multimídia que incluem funções de *streaming* para áudio e vídeo [61].

Este sistema operacional é parecido em sua utilização com o atual Windows dos computadores pessoais, com janelas e ícones e menus conforme mostra a figura 3.5.


Figura 3.5: Sistema Pocket PC

Este sistema já vem com vários aplicativos Microsoft [61] instalados, entre eles o *Outlook, Word, Excel MSN Messenger* e outros que podem ser instalados, como o *Windows Media Player*, etc.

O Pocket PC pode acessar a Internet através de comunicação sem fio com cartões *PCMCIA*, ou fazer comunicação com outro pocket através do protocolo de comunicação *Bluetooth*.

## 3.4 Intel C/C++ Compiler

Projetado para compilar programas C/C++ para a arquitetura *XScale*. Ele possui as seguintes características:

- Compilação de arquivos C/C++.
- Criação de arquivos de formatos objetos ELF/DWARF.
- Opções de linha de comando para controlar o processo de compilação.
- Geração de informação de depuração.
- Geração de posições independentes de código e dados.

- Vetorização de laços.
- Inline assembler.
- Funções intrínsecas.
- Funções ARM e em modo Thumb.

Este compilador aceita as otimizações *constant propagation, copy propagation, dead-code elimination, global register allocation, instruction scheduling, loop unrolling, loop-invariant code movement, partial redundancy elimination, strength reduction/induction variable simplification e variable renaming* [50].

Pode reconhecer funções intrínsecas, escritas como funções declaradas no código fonte, pelo qual transforma em instruções *assembly* equivalentes. As principais funções são:

```
void _WriteCoProcessor(_int64 Arg1, int Arg2);
_int64 _ReadCoProcessor(int Arg1);
void _PreLoad(unsigned long *addr);
```

As duas primeiras funções servem para escrever e ler dados em determinados co-processadores da arquitetura. A terceira função é especial, pois é a função de *preload* de dados que um programador poderia utilizar em seu código para tentar otimizar suas aplicações. Esta função foi usada neste trabalho, pois onde existir a instrução com o opcode *PLD* no código de um programa compilado pelo compilador Xingo, este imprimirá na sua representação intermediária em código C esta função intrínseca, para que o compilador da Intel (ou da Microsoft) possa compilar e gerar código de máquina para esta instrução.

O compilador da Intel pode usar a IDE de programação do Embedded Visual C++ 3.0 ou 4.0. Depois de instalar este compilador, terá uma opção no menu desta IDE para escolher entre estes qual deve ser usado para compilar as aplicações.

Apesar da arquitetura *XScale* ter hardware para permitir o *preload* de dados por software, este compilador não faz esta otimização de forma automática, ele somente oferece suporte para que um programador experiente o faça. Isto foi uma das maiores motivações para o desenvolvimento deste trabalho.

## 3.5 Microsoft eMbedded Visual C++ 3.0

A ferramenta eMbedded Visual C++ da Microsoft fornece um ambiente completo de desenvolvimento de aplicações e componentes de sistema para os dispositivos que possuem o sistema Windows CE / Pocket PC [61].

Esta ferramenta possui uma gama completa de funcionalidades para ajudar no desenvolvimento de aplicações para o *Pocket PC*, entre os quais editor de texto, editor de

recursos, gerenciador de plataformas, depurador, *Appwizard* que ajuda a criar o esqueleto de uma aplicação, *ClassView* para visualizar as classes, atributos e métodos do projeto, etc [62].

O compilador pode compilar códigos fontes C/C++ que são descriminados pela extensão dos arquivos compilados.

Entre suas opções de otimização de código estão: *disable* (desabilitar otimizações), maximização de velocidade, minimização do tamanho do código, bem como permite otimizações globais, e melhorar a consistência de pontos flutuantes dentre outras.

Este compilador também não faz a otimização de *preload* de dados de forma automática, ele somente oferece suporte (ajustando uma opção no parâmetro de compilação) para que o programador possa usar a função intrínseca *PreLoad*.

## 3.6 Server Active Sync

*Server Active Sync* é um programa que permite a sincronização direta de informações de email, calendário e contatos entre os dispositivos que rodam o Pocket PC 2002 e os servidores Exchange 2000. O Server Active Sync foi projetado para operar em conexões com ou sem fio [61].

Este aplicativo foi usado neste trabalho para transferir os programas dos benchmarks para o Pocket PC.

## 3.7 Benchmarks

Para testar a eficiência e a corretude deste trabalho, foram utilizados vários programas de alguns benchmarks que possuíam um perfil necessário para que a otimização fosse aplicada. Os benchmarks utilizados foram:

- *Livermore Loops*.
- *DSPstone*.
- *Mibench*.
- *MeuBench*.

*Livermore Loops* [57], [58] é um benchmark que contém 24 laços de kernels de aplicações reais. Estes laços foram escritos originalmente na linguagem FORTRAN [56], codificados no Lawrence Livermore National Laboratory [59], sendo usados para a verificação do desempenho aritmético de computadores e compiladores desde o início dos anos

70. Eles são uma mistura de laços vetorizáveis e não-vetorizáveis e testam as capacidades computacionais do hardware e do software de compilar um código eficiente.

Quase todos os laços desse benchmark foram passados pela otimização de *preload* de dados, e os resultados serão discutidos posteriormente no capítulo 6 (Resultados). Este benchmark foi muito útil para verificar os erros que ainda persistiam na otimização, pois os laços originavam dos mais variados tipos de aplicações, e quase todos laços possuíam o perfil necessário para que a otimização funcionasse adequadamente.

*DSPstone* [60] é um benchmark com o propósito de testar compiladores desenvolvidos para arquiteturas DSP's[3]. Este benchmark também tem na maioria dos seus programas, o perfil necessário para que a otimização funcionasse. A otimização de *preload* de dados foi executada em 10 programas de ponto fixo desse benchmark, gerando resultados satisfatórios na maioria destes.

*MiBench* [16] é um benchmark que foi desenvolvido com programas de código fonte livre, e possui a característica de poder optar por um conjunto pequeno ou grande de entrada de dados. Este benchmark consiste de seis categorias, entre as quais: *automotive, network, security, consumer devices, office automation e telecomunications*. Estas categorias oferecem diferentes tipos de programas que capacitam os pesquisadores em compiladores e arquiteturas a examinar seus projetos para um segmento particular de mercado. Os programas utilizados na otimização de *preload* de dados foram:

- Security - SHA, rijndael encriptação e decriptação.
- Telecomm - FFT, ADPCM coder e decoder.
- Network - CRC32.

Abaixo esta uma pequena descrição do que cada uma dessas aplicações faz:

- SHA - algoritmo de *hash* seguro que produz uma *message digest* de 160 bits. Pode ser usado em assinaturas digitais.
- Rijndael - cifrador de blocos com opção de chaves e blocos de 128, 192 e 256 bits.
- FFT - executa a transformada rápida de Fourier e sua transformação inversa sobre um vetor de dados. É usado em processamento de sinais digitais para encontrar as freqüências contidas em um dado sinal de entrada.
- ADPCM - significa *"Adaptative Differential Pulse Code Modulation"* que é uma variação do *"Pulse Code Modulation"*. Este programa lê *samples* de 16 bits e converte-os em samples de 4 bits, produzindo uma compressão de 4:1.

[3]Digital Signal Processing

- CRC32 - executa o 32-bit *"Cyclic Redundancy Check"* em um arquivo. É usado para detectar erros em uma transmissão de dados.

MeuBench são programas que foram desenvolvidos para este projeto para testar o perfil das otimizações em várias situações onde esta poderia ser aplicada. Os programas desse benchmark podem ser vistos no anexo A desse trabalho. Este benchmark é composto de 10 programas sendo que alguns deles são aplicações de teoria da computação.

# Capítulo 4

# Otimização de Preload de dados

Nesta seção estaremos detalhando os principais algoritmos de *preload* de dados que se tornaram a base do desenvolvimento deste trabalho.

*Preload* de dados antecipa a busca de dados do sistema de memória antes que o processador necessite destes. O trabalho do *preload* ocorre enquanto o processador continua a executar suas tarefas, dando tempo ao sistema de memória para transferir o dado desejado para a cache. Idealmente, o *preload* deveria completar à tempo para o processador acessar o dado necessitado, evitando ciclos de *stall*.

Mecanismos de *preload* também introduzem algum grau de *overhead* no hardware, incluindo lógica em hardware adicional externo à cache. Algumas técnicas requerem que lógicas também sejam adicionadas ao processador. Embora software *preload* possua exigências de hardware mínimas, esta técnica introduz uma significante quantidade de *overhead* de instrução dentro do programa do usuário [43].

Atualmente vários processadores já possuem a instrução *preload* para buscar os dados requisitados da memória antes de seu uso em determinado programa. A implementação desta instrução é semelhante a de *LOAD*, exceto que a instrução *preload* é não bloqueante (entre cache de dados e processador) e busca os dados para inserí-los na cache de dados, e não em um registrador. Assim podemos sobrepor o tempo de acesso à memória com a execução da computação anterior ao uso destes dados no processador.

Desde que *preload* esconda a latência de acesso à memória, isto pode melhorar o desempenho se uma largura de banda adicional for disponível. *Preload* não decrementa o número de acessos à memória, este simplesmente sobrepõe o tempo de busca de dados da memória com computações no processador. Entretanto, se o sistema de memória já tem uma largura de banda de memória limitada, sendo impossível o *preload* aumentar o desempenho[30].

A inserção da instrução *preload* pode ser feita tanto manualmente ou por um compilador otimizante. *Preload* é mais freqüentemente usado dentro de laços que usam vetores de

dados, fornecendo excelentes oportunidades para o uso desta otimização, pois aplicações com este perfil tendem a ter uma utilização pobre da cache, sendo comum em programas científicos.

Operações com matrizes grandes e densas que freqüentemente formam a base de tais aplicações científicas, tipicamente exibem pouco reuso de dados e assim podem anular as estratégias da cache [42].

Considere o código de programa mostrado na figura 4.1.

```
1                       for (i = 0; i < N; i++)
2                           aux += vet[i] + vet[i+1];
```

Figura 4.1: Exemplo de programa otimizável.

Suponha que os vetores são do tipo inteiro, ou seja, possuem 4 bytes em cada posição do vetor. Como uma linha da cache possui 32 bytes, então a cada 8 iterações resultará em uma cache *miss*.

A forma mais básica de despachar instruções de *preload* é inserí-las dentro do laço procurando buscar o endereço da posição no vetor da próxima iteração. Esta aproximação pode ser vista no programa 4.2.

```
1                       for (i = 0; i < N; i++)
2                       {
3                           preload(&vet[i+1]);
4                           preload(&vet[i+1+1]);
5                           aux += vet[i] + vet[i+1];
6                       }
```

Figura 4.2: Exemplo com preload

Fazendo a otimização de *preload* funcionar como no exemplo acima, provavelmente resultará em vários problemas:

1. O primeiro problema é que devido à pouca computação envolvida no corpo deste laço, não haverá tempo do *preload* buscar o dado e colocá-lo na cache.

2. O segundo problema é que a cada iteração, as instruções *preloads* são despachadas para buscar dados que já devem estar na cache na maioria das iterações. Isto resultará em um *cache hit* para a maioria dos despachos da instrução *preload*, mas a invocação da instrução gera um *overhead*, gerando um impacto no tempo de execução do programa. Esta técnica não aproveita a localidade espacial da linha da cache, ou seja, ela sempre executa o *preload* para a mesma linha da cache durante 8 iterações do laço.

3. O terceiro problema é que são despachadas instruções de *preload* desnecessárias para a mesma linha da cache pelo menos a cada 7 iterações. A instrução de *preload* traz os dados de uma linha da cache, buscando o alinhamento destes dados na memória, ou seja, o ***preload(addr)*** trará os mesmo dados para o ***preload(addr+1)*** sendo que *addr* esteja alinhada na fronteira da memória em múltiplos 32 bytes.

4. A primeira iteração do laço sempre resultará em uma *cache miss* no uso da primeira posição do vetor, pois só é iniciado o despacho da instrução *preload* para este uso da segunda iteração em diante.

5. Ocorrerão despachos desnecessário nas últimas iterações, pois são buscados dados que não serão usados na computação, e que provavelmente nem fazem parte dos vetores.

Não haveria necessidade de inserir uma instrução de *preload* a cada iteração deste laço, pois a cada chamada, a instrução de *preload* traz 32 bytes (uma linha da cache) para dentro da cache. Embora *preloads* extras não sejam ilegais, eles são desnecessários e irão degradar o desempenho. Assumindo que os vetores são alinhados na cache, o *preload* poderia ser feito somente a cada oito iterações. Uma solução seria colocar a diretiva *preload* dentro de uma condição que testasse se $i\%4 = 0$ fosse verdadeiro. Por outro lado, o *overhead* ao se incluir esta condição pode minimizar os benefícios de *preload* e, portanto deveria ser evitado. Uma solução melhor é desenrolar o laço por um fator $r$, onde $r$ é igual ao número de palavras a ser buscada pelo bloco da cache [43].

Enquanto *preload* é útil em esconder a latência do acesso à memória, despachar *preloads* incorre em um *overhead* de instrução e pode aumentar o número de acessos ao sistema de memória. Cuidados devem ser tomados para assegurar que tais *overheads* não excedam os benefícios [30].

Como vimos, esta não é a melhor técnica a ser aplicada, e se for aplicada, provavelmente deteriorará o tempo de execução.

Uma técnica com resultados melhores que a anterior [35] é a de despachar a instrução *preload* não para a referência da próxima iteração, mas sim para $x$ iterações à frente do laço (figura 4.3). Neste caso, usaríamos uma heurística calculando o tempo gasto na execução de uma iteração do laço. Então poderíamos dividir o tempo médio da latência de busca de uma linha da cache da memória por este tempo, calculado na execução de uma iteração do laço. O resultado seria o número de iterações à frente para qual a instrução *preload* deveria ser despachada. Esta aproximação garante que quando um determinado dado é usado, este já está na cache, pois o tempo gasto para a busca calculado e o despacho ocorrerá a tempo deste ser transferido para a cache.

Mesmo assim esta aproximação despacha várias instruções *preloads* desnecessárias.

```
1  for (i = 0; i < N; i++)
2  {
3      preload(&vet[i+x]);
4      preload(&vet[i+1+x]);
5      aux += vet[i] + vet[i+1];
6  }
```

Figura 4.3: Exemplo de preload para x iterações à frente

Os dados de um vetor devem ser alinhados na memória para que fiquem alinhados com as linhas da cache [30]. Portanto, o compilador não deve despachar várias requisições de acesso à referência para a mesma linha da cache (tempo desperdiçado pelo *overhead* da instrução e possível *overflow* do buffer de pendências).

A próxima técnica [35] corrige este problema. Esta técnica consiste em aproveitar a localidade espacial da linha da cache usando o desenrolamento de laços (figura 4.4).

```
1  for (i = 0; i+7 < N; i += 8)
2  {
3      preload(&vet[i+8]);
4      preload(&vet[i+1+8]);
5      aux += vet[i] + vet[i+1];
6      aux += vet[i+1] + vet[i+2];
7      aux += vet[i+2] + vet[i+3];
8      aux += vet[i+3] + vet[i+4];
9      aux += vet[i+4] + vet[i+5];
10     aux += vet[i+5] + vet[i+6];
11     aux += vet[i+6] + vet[i+7];
12     aux += vet[i+7] + vet[i+8];
13 }
14 for ( ; i < N; i += 1)
15     aux += vet[i] + vet[i+1];
```

Figura 4.4: Exemplo de preload com desenrolamento do laço

Nesta técnica ocorre o desenrolamento do laço tentando aproveitar ao máximo a localidade espacial da linha da cache que será buscada pela instrução *preload*. Com esta aproximação o algoritmo começa a ficar bem mais complexo, pois devemos desenrolar o laço tentando aproveitar ao máximo o uso dos 32 bytes trazidos pela instrução *preload*.

Se o *preload* é despachado muito cedo, existe uma chance que o dado irá substituir outro dado útil nos níveis superiores da hierarquia de memória, ou será substituído antes do seu uso (poluição da cache). Se o *preload* é despachado tarde, o dado pode não chegar antes de sua referência de memória e introduzir ciclos de *stall* no processador [43].

Se o fator de desenrolamento for muito grande, ocorre a pressão nos registradores, podendo provocar vários *spills* para a memória.

Como no caso mostrado acima, devemos introduzir um segundo laço, que executa as iterações finais do laço, para os casos onde o limite superior (neste caso $N$) do mesmo

não seja múltiplo do fator de desenrolamento (que neste caso é 8). Observa-se também que nas últimas iterações não ocorrerá o despacho de *preload* para a busca dos dados da memória que não pertencem ao vetor e que não seriam usados na computação.

Esta aproximação melhora o uso dos dados que forem trazidos para a cache, pois a cada *preload*, usam-se todos os dados trazidos, e a cada iteração nos próximos despachos, também terá seu uso sem ocorrer *cache misses* (isso se não ocorrer a substituição dessa linha ao longo da computação do laço). Note que ainda existem dois despachos para a mesma linha da cache na memória. O correto é que o compilador seja eficiente o bastante que possa reconhecer estes casos, e despachar somente uma instrução *preload* para cada linha da cache que possuir seu uso no laço.

A última aproximação é chamada de software pipeline[35] (figura 4.5). Esta técnica procura não gerar cache *misses* na primeira iteração, não despachar *preloads* para as últimas iterações cujos dados não serão usados, aproveitando a localidade espacial dos dados na linha da cache. Esta aproximação pode ser vista no código abaixo.

```
1                       preload(&vet[0]);
2                       preload(&vet[1]);
3                       for (i = 0; i+7 < N; i += 8)
4                       {
5                           preload(&vet[i+8]);
6                           preload(&vet[i+1+8]);
7                           aux += vet[i] + vet[i+1];
8                           aux += vet[i+1] + vet[i+2];
9                           aux += vet[i+2] + vet[i+3];
10                          aux += vet[i+3] + vet[i+4];
11                          aux += vet[i+4] + vet[i+5];
12                          aux += vet[i+5] + vet[i+6];
13                          aux += vet[i+6] + vet[i+7];
14                          aux += vet[i+7] + vet[i+8];
15                      }
16                      for ( ; i < N; i += 1)
17                          aux += vet[i] + vet[i+1];
```

Figura 4.5: Exemplo de preload com *software pipeline*

Se o *buffer* de requisições (estrutura que mantém as requisições de *preloads* e *loads*) ficar cheio, o processador é paralizado até que uma das entradas deste esteja disponível [30].

Esta aproximação tende a resolver os problemas citados anteriormente, e ainda busca não gerar cache *misses* na execução da primeira iteração do laço. Esta aproximação pode ser um problema para arquiteturas que possuem em seu sistema de memória um *buffer* de requisições de leitura de dados da memória, pois se forem despachados vários *preloads* consecutivos, pode ser que estas requisições extravasem o número de linhas deste *buffer*, o que implicará na paralisação da execução do processador até que alguma das requisições seja atendida.

Observa-se que existe a necessidade de se fazer uma integração das aproximações acima, balanceando os fatores decisivos para se ter um ganho ótimo na aplicação desta otimização. Assim não podemos ter um fator de desenrolamento muito grande para não provocar poluição da cache e *spills* nos registradores, e nem tão pequeno para que possa ser aproveitada a localidade espacial da cache evitando despachar *preloads* para a mesma linha da cache várias vezes. Foi com base nesta estratégia que este trabalho foi desenvolvido e será melhor explicado no próximo capítulo.

# Capítulo 5

# *Preload* de Dados no Compilador Xingo

O algoritmo de *preload* de dados parece ser algo fácil de ser implementado, mas se mostrou ao longo do desenvolvimento uma otimização desafiadora e cheia de problemas que, na medida do possível, foram resolvidos. Neste capítulo, estaremos mostrando detalhes de como funciona a otimização através de exemplos, e como cada componente desta desempenha sua tarefa. Serão também apresentadas figuras, diagramas e pseudocódigos das principais funções para uma boa compreensão do trabalho.

Existem três objetivos principais que necessitam ser buscada no projeto da otimização de *preload* de dados em um compilador:

1. Identificar as referências de memória que são candidatas para a aplicação da otimização de *preload* de dados.

2. Reduzir o *overhead* destas referências de memória candidatas, sem sacrificar a cobertura do *preload*.

3. Verificar entre as referências de memória encontradas se as mesmas possuem localidade espacial em comum entre elas.

O primeiro objetivo pode ser alcançado analisando os laços dos programas, pois são neles que as aplicações gastam maior tempo de execução. O segundo objetivo pode ser alcançado através da análise da latência das instruções no corpo dos laços, para a correta inserção das instruções *preload* visando assim maximizar a cobertura de seus usos sem que resultem *misses* na cache de dados. O terceiro e último objetivo é atingido encontrando-se, entre as referências, aquelas que possuem a mesma localidade espacial na mesma linha da cache. Neste caso somente é necessário despachar uma instrução de *preload* para satisfazer as referências que possuem a mesma localidade espacial.

A latência de escrita de dados em memória não é um problema fundamental desde que possamos "bufferizar"estas escritas e fazê-las em paralelo com a execução no processador. O desafio são as latências de leitura de dados da memória [29].

Como a arquitetura XScale da Intel possui um *buffer* de requisições de escrita na memória, o algoritmo de *preload* de dados desenvolvido não despacha instruções *preload* para as escritas, somente para leituras de dados da memória.

Todos os acessos a endereços de valores na memória encontrados no laço mais interno dos laços aninhados de um programa, sendo estes acessos com *stride* uniforme, são considerados candidatos para *preload* de dados [35]. Foi com esta visão que este trabalho começou a ser desenvolvido, procurando por acessos a memória dentro de laços internos em laços aninhados e de laços únicos.

O *PDA* usado neste trabalho foi do fabricante ViewSonic [52], com o sistema Pocket PC 2002 e o processador PXA250/XScale da Intel. Este processador possui o hardware necessário para se usar a otimização de *preload* de dados por software [18].

Um diagrama em blocos dos recursos do Pocket PC utilizados na otimização de preload pode ser visto na figura 5.1.

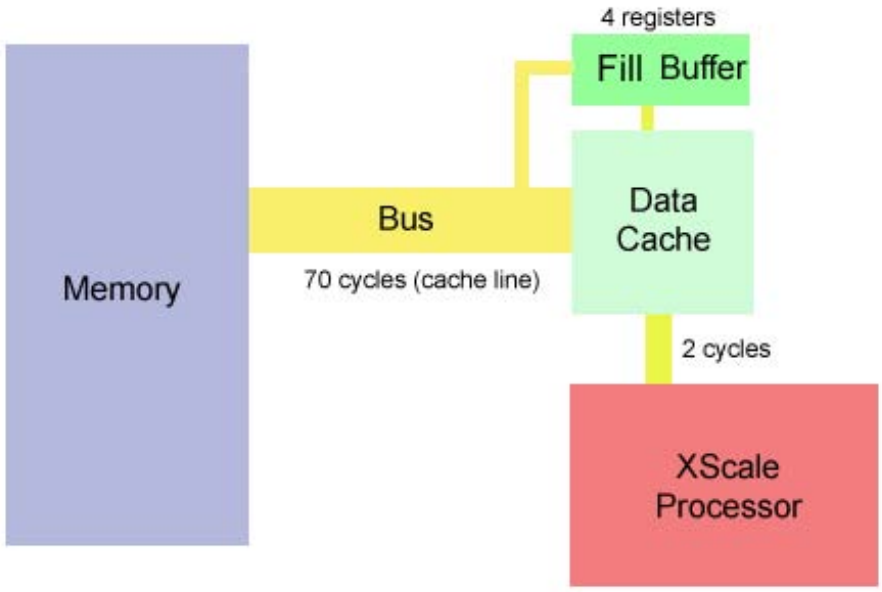



Figura 5.1: Diagrama em blocos dos recursos de hardware usados na otimização de *preload* de dados

Quando o processador XScale encontra uma instrução *preload* em um programa em execução, este despacha uma requisição para a cache de dados gastando 1 ciclo de pro-

cessador. Se os dados requisitados estão na cache, esta requisição é descartada, senão esta requisição é enfileirada no *pend buffer* para que o sistema de memória busque estes dados quando o barramento estiver disponível. A busca de dados da memória para a cache leva cerca de 70 ciclos do processador. Como visto anteriormente, o *pend buffer* suporta armazenar 4 requisições, e se o número de requisições ultrapassar este número, o processador é paralizado até que uma das requisições seja atendida.

Como as ferramentas de desenvolvimento das aplicações do *pocket pc* possuem IDE para o sistema operacional *Windows*, foi necessário portar os compiladores LCC e Xingo para este sistema. O LCC foi compilado com o compilador DJGPP [68] e o Xingo através do uso da ferramenta Visual Studio .NET 2003. Foi também com esta ferramenta que o desenvolvimento da otimização de *preload* de dados foi implementada.

Esta otimização foi desenvolvida como um módulo do compilador Xingo. Este módulo compila códigos de aplicações escrita na linguagem C, e transforma este em código de três endereços [1]. É através deste código representado por *opcodes* e operandos, que esta otimização inicia sua procura por laços que possuem em seu corpo referências à memória. Para que o compilador Xingo pudesse inserir a instrução *preload* como sendo uma instrução do código intermediário de três endereços, foi criada a instrução *PLD* com o formato *PLD addr*. O operando *addr* desta instrução é o endereço da referência à memória que será usado no laço do programa.

Encontradas as referências de leitura de dados da memória, esta instrução é inserida pela otimização *preload* no *header* do laço. Neste caso, é necessário também computar, para cada referência encontrada, uma função de acesso ao vetor. Na função de acesso ao vetor, inclui-se ainda um deslocamento desta posição com mais 32 bytes (outra linha da cache para a próxima iteração).

Quando o Xingo compila um programa, ele realiza várias otimizações (incluindo a de *preload* se especificada pela linha de comando) e gera como saída um arquivo de código também na linguagem C (trazendo o programa de entrada otimizado).

Quando a otimização de *preload* é executada, a instrução *PLD addr* é introduzida em código de três endereços na representação intermediária do Xingo. Após passar pela otimização, o Xingo (encontrando esta instrução) imprime no arquivo de saída a função intrínseca reconhecida pelos compiladores C/C++ da Intel e da Microsoft, ou seja, o compilador Xingo realiza uma conversão de C para C otimizado contendo instruções de *preload* adequadamente inseridas. O objetivo é que a otimização *preload* de dados desenvolvida no compilador Xingo seja inteligente o suficiente para inserir esta instrução de forma que a execução deste programa seja otimizada de forma eficaz.

O arquivo de saída do compilador Xingo, gerado na linguagem C e otimizado, pode ser compilado por outro compilador próprio da arquitetura em que se deseje executar o programa. No caso deste trabalho, foi utilizado o compilador C/C++ da Intel para

o Pocket PC 2002. A figura abaixo ilustra o fluxo de desenvolvimento adotado neste trabalho.

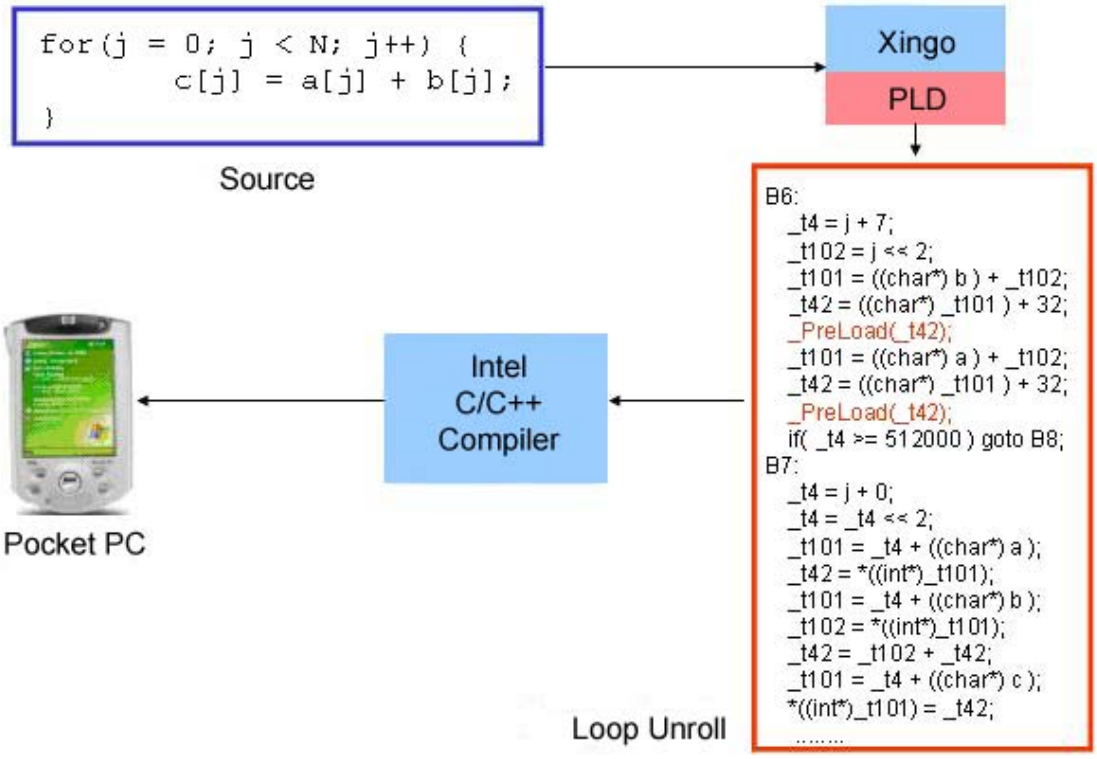



Figura 5.2: Passos do funcionamento da otimização de *preload* de dados

A otimização de *preload* no compilador Xingo ocorre somente se for especificado na linha de comando a opção *-target XScale*. Exemplo:

```
xingo -i file.xir  -target XSCALE -O1 -c file.x.c -f file.out
```

As opções usadas na linha de comando neste exemplo são as seguintes:

1. -i teste.xir - arquivo de entrada do Xingo, este arquivo é gerado pelo *front-end* LCC.
2. -target XSCALE - habilita a otimização de *preload* de dados para a arquitetura XScale.
3. -O1 - executa as otimizações de nível 1, como: *Commom Subexpression Elimination, PeepHole Optimization, Copy Propagation, Constant Propagation e Dead Code Elimination.*

4. -c file.c - nome do arquivo de saída na linguagem C como representação intermediária do Xingo.

5. -f teste.out - nome do arquivo em que são gravadas todas as informações sobre variáveis e estruturas do programa, bem como as instruções em código de três endereços. Este arquivo é muito útil para depurar as otimizações realizadas nos programas.

A otimização de *preload* de dados usa os objetos instanciados do seguinte diagrama de classes.

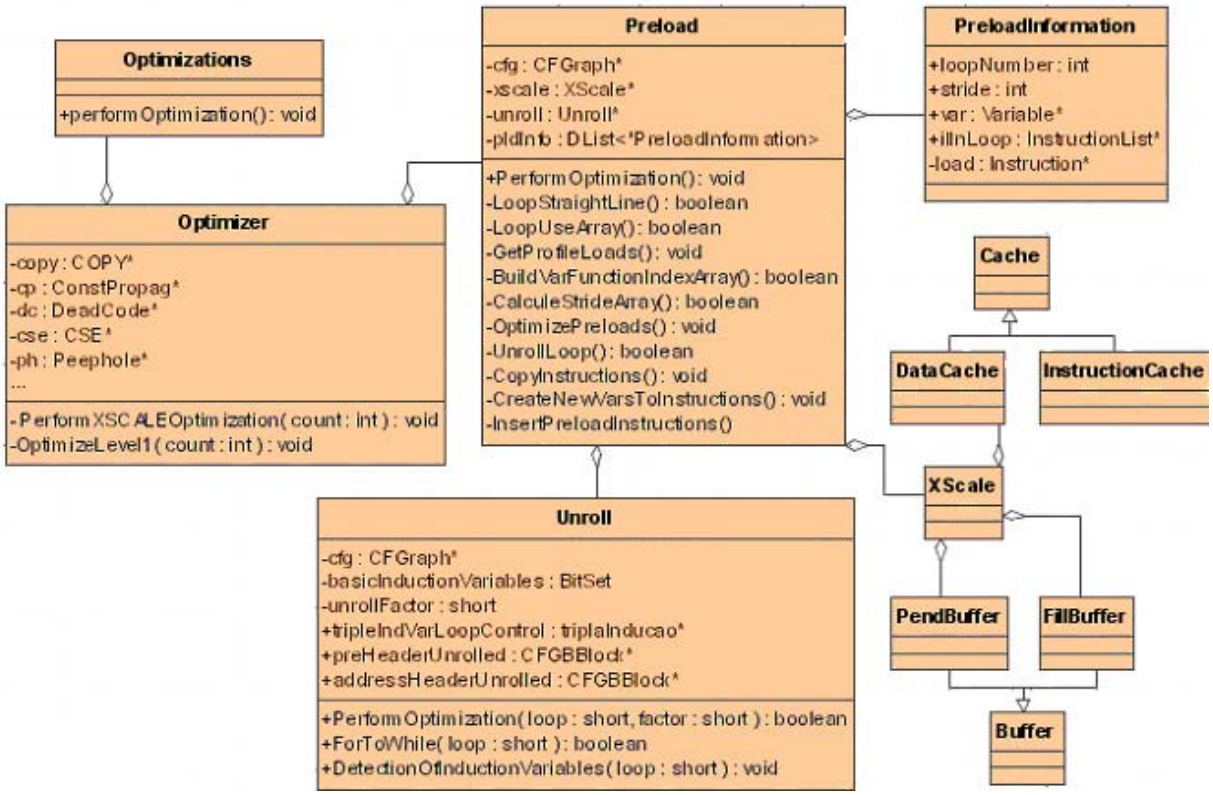



Figura 5.3: Diagrama de classes da otimização *preload* de dados

Abaixo segue uma pequena descrição do que cada objeto faz na otimização:

- Optimizer - é a classe onde são invocados os métodos das otimizações do compilador. É esta classe que decide se serão executadas as otimizações de nível 1, 2 ou 3 como especificado na linha de comando. Esta classe também decide se a otimização de

*preload* de dados será executada, verificando se o parâmetro chave para a otimização está especificado nos parâmetros da linha de comando.

- Preload - Esta é a classe que executa a otimização (juntamente com a classe *Unroll*). É nesta classe que são identificados os laços a serem otimizados, que busca e guarda a lista dos *preloads* juntamente com as instruções que formam o acesso aos vetores. Esta classe também faz cortes de *preloads* desnecessários que possua localidade espacial em comum. Esta fase será melhor detalhada mais a frente nesta seção.

- Unroll - Esta classe é muito importante para a otimização de *preload*, pois é ela que faz o desenrolamento do laço. Ela também tem a função de descobrir as variáveis de indução dos laços e transformá-los colocando a condição de saída (*header*) no início do laço quando estes estiverem no final do mesmo.

- PreloadInformation - Classe que guarda informações do número do laço que se encontrou uma referência, *stride* do vetor, a variável de referência ao vetor, as instruções da função de formação do acesso a este vetor, e a instrução de leitura ($LOAD$) encontrada no laço que dará lugar a uma instrução *preload*. A classe *Preload* contém ainda uma lista de objetos que são usados para realizar a inserção de instruções *preload* pela otimização.

- XScale - Guarda informações sobre a arquitetura XScale, como parâmetros da cache de dados, número de linhas do *Pend Buffer e Fill Buffer*, e informações sobre a memória. Esta classe recebe informações estáticas declaradas no arquivo *"XScaleConfig.h"*. Se algum dia algum fator determinante para a otimização mudar, como o número de *pend buffers* da arquitetura, a otimização poderá acomodar as novas mudanças simplesmente alterando este arquivo. Este arquivo também pode ser modificado de modo a acomodar esta otimização em outras arquiteturas.

- CFGraph - Possui todas informações do grafo de fluxo de controle, além de conter várias outras informações, como informações de *DFA (Data Flow Analysis)*, de alias, de laços, etc.

## 5.1 O Algoritmo de Preload

A figura 5.4 mostra o diagrama de seqüência simplificado indicando como os métodos são invocados.

Para que a otimização de *preload* de dados funcione adequadamente, é necessário o cálculo correto do *stride* de acessos consecutivos em um vetor. Para que este cálculo do *stride* seja correto, o compilador Xingo deve encontrar as variáveis de indução do laço em

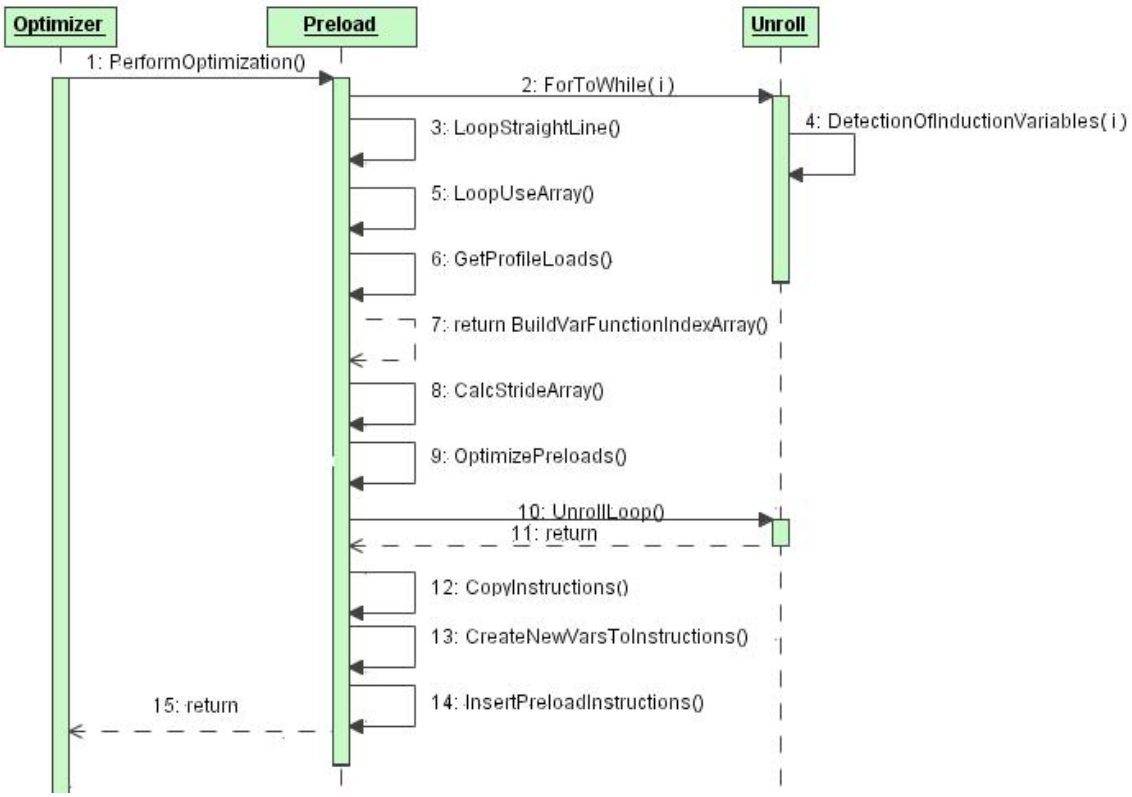



Figura 5.4: Diagrama de seqüência da otimização *preload* de dados

questão. De modo que o Xingo encontre estas variáveis, é necessário que se executem primeiramente as otimizações *Commom Subexpression Elimination, PeepHole Optimization, Copy Propagation, Constant Propagation e Dead Code Elimination*, e posteriormente procura por sentenças do tipo $x = i + c$, $x = i - c$, $x = i * c$, que normalmente definem funções de variáveis de indução. Usando algoritmos de *Induction Variable Detection* [1][31] nestas sentenças, conseguimos identificar as famílias de variáveis de indução.

Para saber qual variável de indução é a que controla o laço (ou seja, a variável de indução básica), é verificada entre as variáveis de indução encontradas, aquela que é operando da instrução que leva a execução do programa para fora do laço. Identifica-se então o bloco de saída do laço (o compilador Xingo já possui esta informação), e então é verificado se a instrução possui um salto para fora deste laço (geralmente uma instrução condicional). Os operandos desta instrução são comparados com as variáveis de indução encontradas pelo Xingo, encontrado assim a variável de indução básica que controla o laço.

Os métodos utilizados pela otimização são explicados logo a frente da apresentação do

pseudo-algoritmo da otimização de *preload* de dados da fig. 5.5.

```
1  function PreLoad()
2  {
3      for each "natural inner loop" i do
4      {
5          unroll->ForToWhile(i);
6          LoopStraightLine();
7          LoopUseArray();
8          GetProfileLoads();
9          if (BuildVarFunctionIndexArray())
10         {
11             CalcStrideArray();
12             OptimizePreloads();
13             if (UnrollLoop())
14             {
15                 CopyInstructions();
16                 CreateNewVarsToInstructions();
17                 InsertPreloadInstructions();
18             }
19         }
20     }
21 }
```

Figura 5.5: Passos da otimização de *preload* de dados desenvolvido neste trabalho.

**ForToWhile**

A otimização de *preload* de dados começa verificando os laços mais internos de cada função de cada arquivo fonte do programa que esta sendo compilado. São nestes laços que estão as maiores oportunidades para otimização. Estes laços, no entanto, devem possuir os pré-requisitos necessários para que esta otimização possa ser aplicada. O compilador Xingo fornece a informação de laços aninhados do programa através de conjuntos *BitSets do CFG* do programa fonte.

Encontrando um laço do programa no qual pode ser aplicada a otimização, este é convertido (se necessário) para que a condição de saída ocorra no início do laço. Isto é feito pela função *ForToWhile(i)* do algoritmo mostrado na figura 5.5 na linha 5. Essa transformação é muito importante, pois as instruções *preload* devem ser invocadas antes do início da execução do corpo do laço, e não no final do mesmo. Isso se deve ao fato de que se fossem inseridas no final da execução do corpo do laço, não daria tempo das instruções *preload* buscarem os dados da memória, para serem usados na próxima iteração que estaria próxima de ser executada. A figura 5.6 ilustra essa situação.

A função *ForToWhile(i)* é um método pertencente à classe *Unroll* como visto no diagrama de classes. É na classe *Unroll* que são guardadas várias informações do laço, como variáveis de indução, a variável de controle do laço, *header* do laço que está sendo otimizado, entre outros. Esta classe será melhor detalhada a seguir. Após a transformação do laço por esta função, o *header* do mesmo fica sendo no começo e não no final do laço como geralmente acontece com laços *"for"* no Xingo.

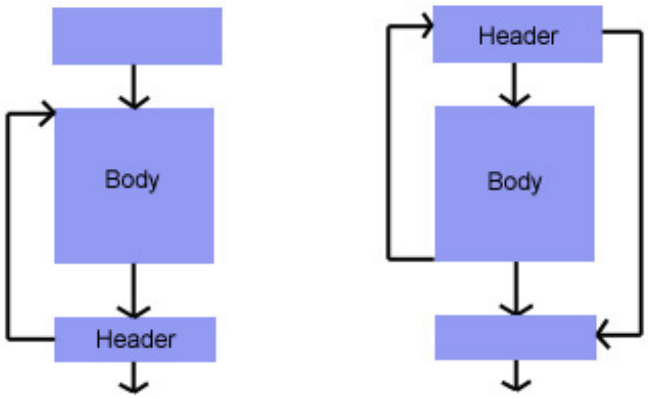



Figura 5.6: Bloco *header* no final do laço e header no inicio do laço

**LoopStraightLine**

A função *LoopStraightLine()* do algoritmo mostrado na figura 5.5, linha 6, verifica se o laço não possui *branches* (estruturas condicionais) em seu corpo como mostra a figura 5.7. Isso é necessário, pois a otimização *loop unrolling* usado neste trabalho não desenrola laços que possuem *branches*. Mesmo assim, nestes casos a otimização de *preload* é aplicada (só que o laço não é desenrolado), e ainda assim pode gerar bons resultados.

O pseudocódigo desta função pode ser visto na figura 5.8.

Como mostrado no pseudo-código da figura 5.8, a função *LoopStraightLine()* procura em determinado laço $i$ se os blocos básicos mais internos deste possuem mais de um caminho de execução. Não é verificado no *header* do laço nem no bloco de saída, pois estes não fazem parte do corpo do laço. Se for encontrado um caminho de execução maior que um, isto significa que este laço possui *branches* em seu interior, não é possível a execução do algoritmo de desenrolamento de laços.

**LoopUseArray**

A função *LoopUseArray()* da figura 5.5 na linha 7, percorre todos os blocos básicos do corpo do laço à procura de variáveis do tipo $\_ARRAY$ (tipo primário de uma variável no compilador Xingo). Este tipo de informação é fornecido pelo compilador Xingo através de sua lista de símbolos (variáveis) quando há vetores declarados estaticamente no programa fonte. Se for encontrada uma instrução no corpo do laço que use uma variável do tipo $\_ARRAY$, então é incluída uma nova *PreloadInformation* na lista de possíveis *preloads* a serem despachados. O pseudo-código desta função pode ser visto na fig. 5.9.

Como mostrado no algoritmo da fig. 5.9, o *CFG* do compilador Xingo retorna todas as variáveis do tipo $\_ARRAY$ que foram declaradas no programa. Então cada bloco do laço é percorrido e verificado para cada instrução destes blocos se é do tipo $LOAD$ e se possui sua

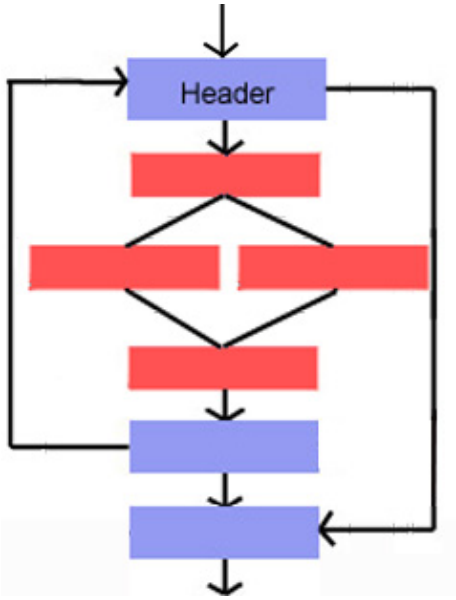



Figura 5.7: Exemplo de laço com branch

variável destino na lista de variáveis que são do tipo *_ARRAY*. Se estas condições forem satisfeitas, um objeto *PreloadInformation* é alocado para guardar os dados referentes a esta referência. A estrutura *PldInfo* é uma lista dos possíveis *preloads* que serão inseridos pela otimização.

Como estas variáveis de *arrays* são declaradas estaticamente no programa fonte, podemos tentar alinhar estes vetores na fronteira de 32 bytes da memória, para que a otimização de *preload* funcione adequadamente. Isto é possível imprimindo na declaração desta variável no arquivo de saída do Xingo a macro mostrada abaixo:

```
#define CACHE_LINE  32
#define CACHE_ALIGN __declspec(align(CACHE_LINE))

CACHE_ALIGN int vector[N];
```

Macros para alinhamento de variáveis arrays

O alinhamento pode ser necessário pois a instrução *preload* busca 32 bytes da memória para a cache, sendo estes bytes em fronteira de múltiplas linhas da cache mapeadas na memória. Por exemplo, se tivermos no inicio da memória linhas da cache mapeadas em multiplos de 32 bytes nos endereços 0, 31, 63, etc. Se chamarmos *preload* para o endereço 12 da memória *PLD 12* (que poderia ser o início de um vetor), esta instrução traria os

```
1   function LoopStraightLine (int loop)
2   {
3       BBlock blocks_loop = cfg->NaturalLoop(loop);
4       for each blocks_loop i do
5       {
6           if((blocks_loop[i]->NumFanins() > 1) and
7               (blocks_loop[i]->IsNotHeader())    and
8               (blocks_loop[i]->IsNotOutBlock()))
9               return FALSE;
10      }
11      return TRUE;
12  }
```

Figura 5.8: Pseudocódigo da função *LoopStraightLine()*.

```
1   function LoopUseArray (int loop)
2   {
3       VarList varList = cfg->Syms->PrimType(_ARRAY);
4       for each blocks_loop (block) do
5       {
6           for each Instruction (instr) in block do
7           {
8               if ((varList->Contains(instr->Vars(DST))) and
9                   (instr->Opcode() = LOAD))
10              {
11                  PreloadInformation *pli = new PreloadInformation();
12                  pli->instr = instr;
13                  pli->loop  = loop;
14                  PldInfo.Append(pli);
15              }
16          }
17      }
18  }
```

Figura 5.9: Pseudo-código da função *LoopUseArray()*.

bytes de 0 a 31 (trata-se da primeira linha da cache mapeada em memória). Assim, esta instrução estaria trazendo bytes desnecessários se o *stride* de acesso a este vetor fosse constante e crescente (os bytes de 0 a 11 não seriam usados). Com o alinhamento do vetor, podemos aproveitar todos os bytes trazidos, pois este estaria alinhado no endereço 0 da memória.

**GetProfileLoads**

A função *GetProfileLoads()* da figura 5.5 na linha 8, percorre os blocos básicos do corpo do laço à procura de instruções *LOAD* que podem referenciar vetores declarados dinamicamente. Nos vetores declarados dinamicamente, as variáveis de referência das instruções *LOAD* não são classificadas pelo LCC (*front-end* do Xingo) como do tipo *_ARRAY* (mesmo sendo um vetor). Neste caso, somente após o cálculo do *stride* desta referência é que verificamos se este *preload* pode ser despachado, pois este *stride* deve ser constante e menor que 32 bytes. O pseudo-código desta função pode ser visto na fig 5.10.

```
1  function GetProfileLoads (int loop) {
2      for each blocks_loop (block) do
3      {
4          for each Instruction (instr) in block do
5          {
6              if (instr->Opcode() == LOAD)
7              {
8                  PreloadInformation *pli = new PreloadInformation();
9                  pli->instr = instr;
10                 pli->loop  = loop;
11                 PldInfo.Append(pli);
12             }
13         }
14     }
15 }
```

Figura 5.10: Pseudo-código da função *GetProfileLoads(loop)*

**BuildVarFunctionIndexArray**

A função *BuildVarFunctionIndexArray()* da figura 5.5 na linha 9 é muito importante para a otimização de *preload*, pois é nela que é remontada a função de acesso aos vetores. Isto é feito através de uma busca por instruções que estão dentro dos blocos básicos do laço, e que definem as variáveis usadas pela instrução *LOAD* encontrada. Se as instruções encontradas diferirem do padrão de acesso dos valores deste vetor, a otimização irá chamar *preload* para uma posição de memória que talvez não exista, ou que não faça parte do acesso deste vetor, ou seja, ao invés de melhorar o desempenho do programa, ele pode piorar. Isso ocorre pois cada vez que uma instrução *preload* é invocada, gasta-se um ciclo de *overhead* para processar esta no processador, e mais uma linha do recurso de *pend buffer*. Se este último recurso tiver um uso demasiado (ultrapassando sua capacidade máxima), ele pode paralisar o processador até que uma das pendências de acesso à memória seja resolvida.

A técnica utilizada para encontrar a função de formação do acesso do array é baseada em *Reaching Definition* disponível no compilador Xingo. Quando uma instrução *LOAD* está disponível, verifica-se a informação de "uso-definicão"(*UDChain*) da variável usada nesta instrução. O compilador Xingo possui o método

$$BitSet\ cfg->ReachDefInfo()->UDChain(instrDefVar,\ var)$$

que retorna um conjunto *BitSet*. Neste conjunto, os bits em 1 representam instruções que definem a variável *var*, e que chegam na instrução *instrDefVar*. Quando temos mais de uma instrução neste *BitSet* resultante, então queremos saber a instrução que define aquela variável e que esteja dentro do laço. Assim, construímos um outro *BitSet* colocando em 1 todos os identificadores das instruções que estão dentro do laço. Entre estes dois *BitSets*

é feita uma operação de interseção, e o *BitSet* resultante conterá somente uma instrução (conjunto unitário), sendo a que estávamos procurando.

Depois de encontrada a instrução que define o operando da instrução $LOAD$, é verificado o número de operandos que esta nova instrução possui e inicia-se uma nova procura pelas definições desses novos operandos. Quando uma instrução possui mais de um operando, deve-se se ter o cuidado de encontrar as duas instruções que definem estas variáveis, e verificar qual das instruções é executada primeiro, pois se estas forem invertidas na lista das instruções que serão inseridas com o *preload*, o endereço de acesso ao vetor será diferente do procurado. A ordem destas instruções é preservada verificando as informações do bloco básico das mesmas.

A figura 5.11 mostra o pseudo-código desta função.

Como explicado anteriormente, a função começa a procurar a instrução que define a variável do operando da instrução $LOAD$ que chegam na instrução $pld \rightarrow instr$. Depois de encontrar esta instrução, coloca-se esta na lista de procura e inicia-se uma nova procura olhando agora para os operandos da nova instrução. Se esta nova instrução possuir dois operandos, teremos uma instrução definindo cada um dos operandos. Neste caso, temos que verificar a ordem de precedência de uma dessas instruções em relação à outra antes de inserir na lista de reconstrução da função de acesso ao vetor. Ao final quando acabarem as instruções de procura, teremos dentro do objeto *PreloadInformation* a lista completa das instruções que reconstrói a função de acesso ao vetor.

Existem *front-end* de compiladores que já podem fornecer expressões completas sobre a função de acesso ao vetor. O *front-end* do Xingo é o LCC, e este não fornece este tipo de informação. Por isso o propósito do algoritmo 5.11 é o de reconstruir esta função de acesso ao vetor.

Note que com a função desenvolvida, a reconstrução da função de acesso ao vetor é mais confiável do que as informações de alto nível do compilador. Como exemplo considere o programa mostrado na figura 5.12.

No caso do compilador fornecer a expressão que faz o acesso ao vetor, no exemplo acima este retornaria a expressão $i + Y * 2$ e este não verificaria que para a próxima iteração o valor da variável $Y$ será multiplicado pela variável $X$, gerando neste caso uma função de acesso ao vetor que não corresponde ao endereço que será acessado na próxima iteração do laço. Já no caso da função desenvolvida nesta otimização, esta instrução que modifica esta variável fará parte das instruções que formam a função de acesso a este vetor, calculando o endereço correto para o despacho da instrução *preload*.

**CalcStrideArray**

O padrão de referências a vetores com grande *strides* resulta em uma pobre utilização da cache [43].

A análise para se determinar as instruções *preloads* que são necessárias para um *stride*

```
1  function BuildVarFunctionIndexArray(PreloadInformation pld)
2  {
3      Variable varSrc1, varSrc2;
4      BitSet bs1, bs2;
5      InstructionList instrList;
6      Instruction instr, instr1, instr2;
7
8      instrList.Append(pld->instr); // first looking for load var definition
9
10     while (!instrList.isEmpty())
11     {
12         instr = instrList.Head();
13         if (instr->Operands() == 1)
14         {
15             varSrc1 = instr->Operand(SRC_1);
16             bs1  = cfg->ReachDefInfo()->UDChain(instr, varSrc1);
17             if (bs1.size == 1)
18             {
19                 instrList.Append(cfg->Instruction(bs1.first()));
20                 pld->InsList->Append(cfg->Instruction(bs1.first()));
21             }
22             else if (bs1.size > 1)
23             {
24                 instrList.Append(InstructionInLoop(bs1));
25                 pld->InsList->Append(InstructionInLoop(bs1));
26             }
27         }
28         else if (instr->Operands() == 2) // verify precedence
29         {
30             varSrc1 = instr->Operand(SRC_1);
31             varSrc2 = instr->Operand(SRC_2);
32             bs1  = cfg->ReachDefInfo()->UDChain(instr, varSrc1);
33             bs2  = cfg->ReachDefInfo()->UDChain(instr, varSrc2);
34             if (bs1.size == 1)
35                 instr1 = cfg->Instruction(bs1.first());
36             else if (bs1.size > 1)
37                 instr1 = InstructionInLoop(bs1);
38
39             if (bs2.size == 1)
40                 instr2 = cfg->Instruction(bs2.first());
41             else if (bs2.size > 1)
42                 instr2 = InstructionInLoop(bs2);
43
44             if (instr1->Position() > instr2->Position())
45             {
46                 instrList.Append(instr1);
47                 instrList.Append(instr2);
48                 pld->InsList->Append(instr1);
49                 pld->InsList->Append(instr2);
50             }
51             else
52             {
53                 instrList.Append(instr2);
54                 instrList.Append(instr1);
55                 pld->InsList->Append(instr2);
56                 pld->InsList->Append(instr1);
57             }
58         }
59     }
60 }
```

Figura 5.11: Pseudo-código da função *BuildVarFunctionIndexArray()*

que é maior do que o tamanho da linha cache é complicado pela necessidade de se contar com um alinhamento pobre da cache dessas referências de dados [35].

É devido as afirmações acima que a função *CalcStrideArray()* (da figura 5.5 linha 11) é muito importante como uma forma de melhorar o desempenho dessa otimização. Esta função recebe a função de acesso (as instruções que formam o endereço do acesso à memória) e calcula seu deslocamento na memória entre algumas iterações do laço. Se os deslocamentos nos usos de um vetor forem maiores do que a linha da cache (no caso do XScale maior que 32 bytes), provavelmente nem compensará despachar a instrução *preload* para esta referência, pois não haverá proveito da localidade espacial que esta otimização possa oferecer. Por isso é importante o correto cálculo do *stride* desta função, pois assim teremos como decidir se despachamos ou não o *preload* para esta referência.

Outra característica importante que a função *CalcStrideArray()* verifica, é se a função de acesso à referência encontrada é uma progressão aritmética. Isso é feito verificando se o *stride* é constante e o *preload* possa ser aplicável. Caso a função não seja linear, então esta referência será excluída dos possíveis *preloads* que serão despachados. Isso se deve ao fato que não haverá proveito da localidade espacial porque o *preload* trará uma linha inteira da cache que não será usada.

Por isso que o *stride* da função é calculado entre algumas iterações do laço, pois é através dos resultados nestas iterações que podemos verificar se este é constante para cada uma das iterações calculadas.

A função do cálculo do *stride* pode ser vista no pseudo-código mostrado na figura 5.13.

Neste algoritmo é calculado para cada *preload* encontrado o *stride* sofrido em cada uma de $N$ iterações do laço ($N$ deve ser no mínimo 3), pois assim poderemos comparar os *strides* entre estas iteração e verificar se os mesmos são constantes. A função *CalcInstruction(instr, hash)* executa a instrução *instr* com base em seu *opcode* e nos operandos (valores das variáveis armazenadas na *hash*) que este utiliza. Se a instrução usa uma variável que não está na tabela *hash*, é atribuido o valor 0 (zero) para esta variável não definida. Se a instrução usa uma variável de indução, devemos saber o valor da mesma com base na iteração corrente que estamos calculando e ainda saber de quanto esta variável de indução é incrementada de uma iteração do laço para outra. É com esta função

```
1  function foo (int Y, int X)
2  {
3      for i in 0 to N do
4      {
5          A[i] = B[i+Y*2];
6          Y *= X;
7      }
8  }
```

Figura 5.12: Programa exemplo.

```
1   function CalcStrideArray()
2   {
3       int stride[N], varValue;
4       HashTable<var, value> hash; // store variable values
5
6       for each pld = pldInfo->Next() do
7       {
8           for i in 0 to N do // calcule to N iterations
9           {
10              while (instr = pld->InsList->Head()) do
11              {
12                  varValue = CalcInstruction(instr, hash);
13                  hash->Append(instr->Operand(DST), varValue);
14              }
15              stride[i] = hash->Get(pld->instr->Operand(DST));
16          }
17          VerifyStride(stride); // verifify if stride is arithmetic progress
18      }
19  }
```

Figura 5.13: Pseudo-código da função *CalcStrideArray()*

que temos uma base do deslocamento real dos acessos aos vetores. A função *VerifyStride(stride)* vista na figura 5.13 na linha 17, verifica se os *strides* entre cada iteração são incrementos constantes, indicando se a função de acesso ao vetor é linear. O *preload* pode ser aplicado se este valor do *stride* não for maior que os 32 bytes da cache.

**OptimizePreloads**

A função *OptimizePreloads()* do algoritmo mostrado na figura 5.5 linha 12 não deixa de ser tão importante quanto as demais. Ela é composta por várias funções cuja finalidade é a de excluir os *preloads* desnecessários da lista dos possíveis *preloads* a serem despachados. O pseudo-código pode ser visto na figura 5.14.

```
1   function OptimizePreloads()
2   {
3       DeletePldStridesGreater32OrEqualZero();
4       PointerPreloads();
5       LeadingReference();
6       CutPreloads();
7   }
```

Figura 5.14: Pseudo-código da função *OptimizePreloads*

Esta função é extremamente necessária, pois como o *pend buffer* pode enfileirar somente quatro requisições de acesso à memória, se forem despachados 5 *preloads* seguidos, o processador irá paralisar até que uma das requisições seja resolvida. Como os programas fazem vários acessos à memória, resolvi ao contrário de despachar no máximo 4 *preloads*, despachar somente 3. Isso se deve ao fato de já poder haver alguma requisição pendente neste *buffer*, extravazando com estes despachos o mesmo, e parando o processador até que

uma das referências seja resolvida.

A função *DeletePldStridesGreater32OrEqualZero()* na linha 3 da figura 5.14, como o próprio nome já sugere, exclui da lista de *preloads* os que possuem $stride > 32$ bytes, ou que possuem $stride = 0$ bytes. Se $stride > 32$, como explicado anteriormente, não temos proveito espacial nos 32 bytes que a instrução *preload* traz para a cache. Se o $stride = 0$, então esta referência de memória será invariante ao laço, e por isso resultará em cache *miss* somente na primeira vez de seu uso. Nos usos posteriores provavelmente resultará em cache *hit*. Se for chamado *preload* para esta referência, sempre gastará desnecessariamente 1 ciclo de processador a cada iteração.

A função *PointerPreloads()* na linha 4 do algoritmo mostrado na figura 5.14 verifica se os *preloads* que são ponteiros para vetores (alocados dinamicamente) são formados pelas mesmas variáveis. Se forem formados pelas mesmas variáveis e se possuírem $strides < 32$, podemos despachar somente um *preload* para uma dessas referências, pois a outra referência se beneficiará deste mesmo despacho. Esta política é chamada de *"Leading Reference"*.

A próxima função na linha 5 do algoritmo mostrado na figura 5.14 *LeadingReference()* possui a mesma função explicada acima, mas neste caso ela é utilizada para as variáveis que são declaradas estaticamente, e que conhecemos quem é a variável declarada. Já no caso de variáveis que são alocadas dinamicamente, estas poderão diferir entre os *preloads*, mas devem pertencer à mesma área de memória.

Se após passar pelas funções explicadas anteriormente e ainda tivermos uma lista de $preloads > 3$, a otimização executa a função *CutPreloads()* da linha 4 do algoritmo mostrado na figura 5.14, que tem como objetivo cortar os *preloads* em excesso a fim de não ultrapassar o *pend buffer*. A política de corte usada nesta função é a de excluir primeiro os que possuírem os maiores *strides*, pois estes possuem um menor proveito na localidade espacial do vetor.

**UnrollLoop**

Continuando com as funções principais da otimização, a função *UnrollLoop()* desenrola o laço de acordo com o fator de desenrolamento calculado por esta mesma função.

## 5.2 Heurísticas Utilizadas no Desenrolamento de Laços

Ao longo do trabalho, foram desenvolvidas duas heurísticas principais para encontrar o fator de desenrolamento ótimo. A primeira heurística foi a de tentar aproveitar ao máximo todos os 32 bytes trazidos pela instrução de *preload*. Nesta heurística, não se leva em conta a distância desde a chamada do *preload* até seu uso no corpo do laço. Assim pode acontecer o problema de se chamar o *preload* muito antes de seu uso, de modo que o dado seja substituído antes de ser usado. Ou ainda, despachar a instrução

*preload* tarde o bastante para que quando chegar seu uso o dado ainda não esteja na cache.

A adição de instruções *preload* aumenta a pressão por registradores se o *backend* do compilador reservar um registrador para armazenar o cálculo do endereço do *preload*. O problema é aumentado quando existem vários *preloads*, já que isso implica em manter vários registradores de endereços para manter múltiplos endereços *preloads*, ou armazenar estes endereços na memória se o número de registradores requisitados não estiver disponível [43].

A primeira heurística trouxe alguns bons resultados, mas não ótimos. Um dos problemas é que quando houver no corpo do laço muita computação, o fator de desenrolamento tende a ser muito grande. Por exemplo, se tivermos um vetor do tipo inteiro e o incremento do laço for de 1, então o fator de desenrolamento seria 8 (32 bytes trazidos pelo preload / *stride*). Em alguns testes executados, ocorreram problemas com esta aproximação, pois o compilador que funciona como *backend* do Xingo, gera muitas variáveis que são alocadas na memória (*spills*). Desta forma, muitos *loads e stores* ocorrem no corpo deste laço. Deste modo, quanto maior o fator de desenrolamento, maior será a pressão nos registradores. Assim, o compilador aloca muitas variáveis à memória, e o ganho em desempenho que se teria com o *preload*, desaparece nos acessos das variáveis temporárias alocadas à memória.

Além disso, como citado em [43], *preload* também resulta em uma significativa expansão no tamanho do código, devido ao desenrolamento do laço e inserção de novas instruções (preloads e da função de acesso ao vetor), podendo assim degradar o desempenho da cache de instruções. O fator de desenrolamento para esta heurística pode ser calculado pela seguinte fórmula:

```
unroll_factor = 32 bytes / stride                (1)
```

Por exemplo, temos um vetor sendo acessado com *stride* de 4 bytes (cada posição de um vetor do tipo inteiro tem $stride = 4$, assumindo que os acessos serão incrementados em uma posição no vetor), então o fator de desenrolamento deverá ser 8, ou seja, a cada oito iterações serão consumidos todos os dados trazidos por uma única chamada de *preload*.

A outra heurística que foi implementada consiste em calcular a latência gasta para percorrer o menor caminho de execução no corpo do laço. Com base neste cálculo, podemos nos aproximar do tempo que levará para buscar os dados na memória durante o despacho da instrução *preload*. Por exemplo, a arquitetura XScale tem uma latência de acesso a memória entre 50-70 ciclos de processador. Como geralmente além desta estatística envolve outros itens a serem levados em consideração que influem no desempenho do gerenciador de memória, assumiremos o número de 70 ciclos. Desta forma, o fator de desenrolamento pode ser calculado pela fórmula.

```
unroll factor = memory latency / shortest way loop latency          (2)
```

Se por exemplo, a latência do menor caminho do corpo do laço for de 20 ciclos, o fator de desenrolamento seria 3. Assim, podemos estimar que o tempo levado para completar uma iteração do laço desenrolado 3 vezes, é suficiente para que a chamada do *preload* disponibilize na cache os dados que serão usados na próxima iteração. Além de dar tempo para que o *preload* possa trazer os dados, aproveita-se o resto dos bytes da linha da cache trazida por este *preload*.

Para o cálculo da latência do corpo do laço de menor caminho, foi utilizada um mapeamento das instruções de 3 endereços do compilador Xingo com as latências das instruções da arquitetura XScale. Além de contar com estas latências de instruções, a cada variável global das instruções dentro do corpo do laço, somava-se ao cálculo total da latência mais 2 ciclos, pois estas variáveis geram *loads* na memória. Com isso temos um tempo de execução aproximado pela soma das latências das instruções do corpo de um laço quando forem executadas no processador.

Conflitos de bancos de memória aumentam a latência de transferência de dados da memória [35]. Este problema resulta diretamente em um aumento na latência de busca de dados da memória e, assim, o algoritmo encarregado do cálculo da latência média de acesso à memória não pode ser tão confiável. Fatores como este fazem com que seja complicada a análise e aplicação da otimização de forma ótima para determinados programas.

Esta otimização não prevê *preloads* para estruturas de dados, pois estas possuem *strides* que geralmente são maiores que o tamanho de uma linha da cache. Além disso, geralmente estas não contêm padrões regulares de acesso à memória. Nestes casos, a otimização não seria tão benéfica quanto para vetores. Maiores explicações serão descritas no capítulo 7 entitulado "Conclusões e Trabalhos Futuros".

A figura 5.15 nos mostra o funcionamento da otimização. Neste exemplo o código fonte é compilado com o compilador Xingo e otimizado com a otimização de *preload* de dados. Vemos por esta figura que a instrução *preload* foi inserida no *header* do laço juntamente com as instruções que formam a função de acesso ao vetor, e ainda a instrução de deslocamento para o efetivo cálculo do endereço deste vetor para a próxima interação (bloco B10). Percebe-se também que o laço foi desenrolado 8 vezes para que a latência de busca destes dados na memória possa ser sobreposta.

## 5.3 Um Exemplo Simples

Agora vamos ver um exemplo simples do funcionamento da otimização, e para isso vamos utilizar o programa mostrado na figura 5.16. Este programa preenche dois vetores alocados dinamicamente e faz o produto escalar dos mesmos, guardando o resultado em

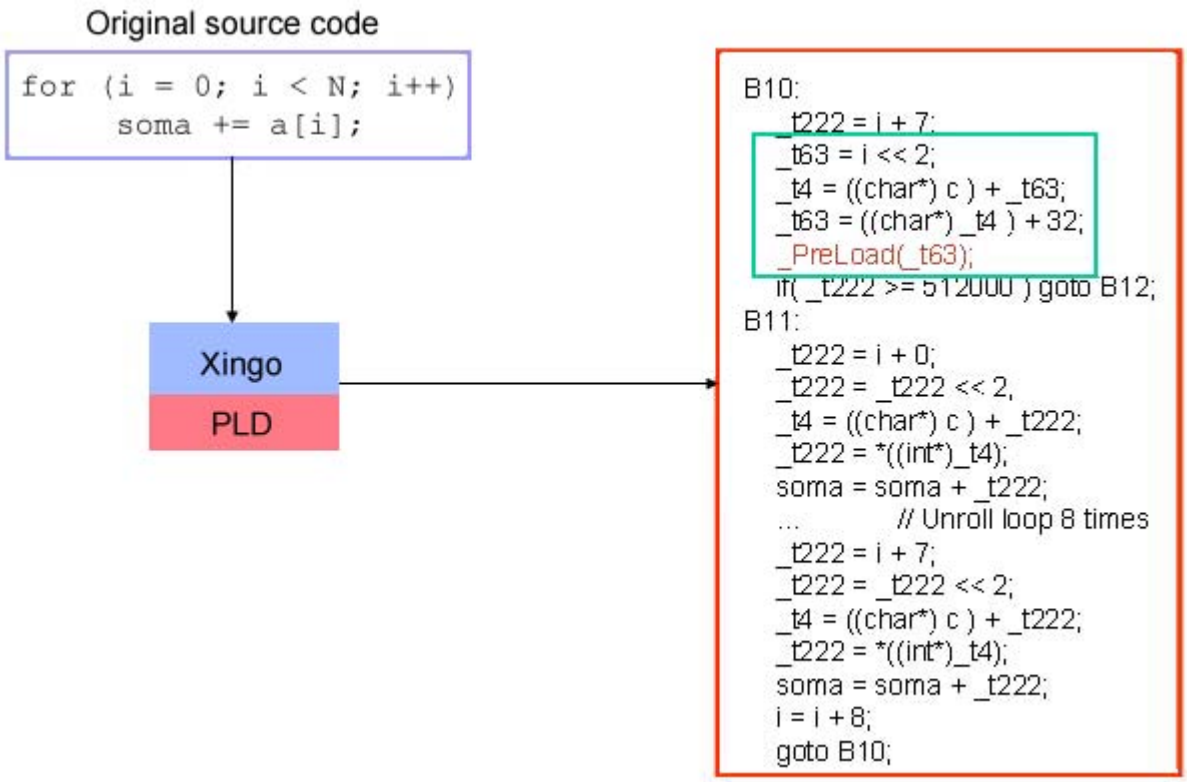



Figura 5.15: Exemplo de funcionamento da otimização

uma variável acumuladora. O tamanho dos vetores são de 512KB, o que garante que a cache de dados irá ter uma baixa utilização, pois este tamanho dos vetores é maior que o tamanho da cache do *Pocket PC*.

```
1  #include <stdio.h>
2  #define N 512000
3  int *a, *b;
4  int soma = 0;
5
6  int entryMain() {
7      int j;
8
9      a = (int*)malloc(sizeof(int)*N);
10     b = (int*)malloc(sizeof(int)*N);
11     for(j = 1; j < N ; j++) {
12        a[j] = j*2;
13        b[j] = j*2;
14     }
15     for(j = 0; j < N; j++)
16         soma += a[j] * b[j];
17     return 0;
18 }
```

Figura 5.16: Programa Exemplo

O código gerado pela otimização pode ser visto na figura 5.17.

Neste exemplo podemos perceber que o laço otimizado foi o segundo do programa original (no código iniciando no bloco *B6*), pois é neste laço que são usados os valores dos vetores no programa. Este foi desenrolado 8 vezes visto que o corpo do laço original é muito pequeno, fazendo com que se tenha que executar as 8 iterações desenroladas para que os dados das próximas 8 iterações já estejam na cache antes de seu uso.

Observa-se também que existe um segundo laço começando no bloco *B8* que serve para executar as iterações finais, caso o limite superior do laço não seja múltiplo do fator de desenrolamento. Isto é muito bom pois não é necessário despachar a instrução *preload* para as últimas iterações (*software pipelining*), economizando alguns ciclos de *overhead* da instrução (como o laço está acabando, não desejamos despachar mais a instrução de *preload* pois não necessitaremos mais trazer os dados para cache).

Vemos que antes de incluir as instruções *preload* no *header* do laço, temos que montar a função de acesso a este vetor. Isto é feito colocando as instruções que montam esta função, e trocando-se as variáveis das mesmas por variáveis não usadas no programa. O objetivo com isso é de garantir que estas novas instruções inseridas não modifiquem a semântica do programa original.

Este programa obteve um *speedup* de *19,1%* em relação ao programa original podendo ser visto no capítulo 6 chamado "Resultados Experimentais"na seção 6.1 do *benchmark* MeuBench do programa *Arrays 1*.

```
1  static unsigned long _C4 = {2048000};
2  char* b;
3  char* a;
4  int soma = {0};
5
6  int entryMain() {
7      int _t54;
8      char* _t32;
9      int _t48;
10     unsigned int _t17;
11     char* _t18;
12     int j;
13
14     _t54 = malloc(_C4);
15     _t17 = (unsigned int) _t54;
16     _t32 = (char*) _t17;
17     a = ((char*) _t32 );
18     _t54 = malloc(_C4);
19     _t17 = (unsigned int) _t54;
20     _t18 = (char*) _t17;
21     b = ((char*) _t18 );
22     j = 1;
23     goto B2;
24 B2:
25     if( j >= 512000 ) goto B4;
26 B3:
27     _t54 = j << 1;
28     _t48 = j << 2;
29     _t32 = ((char*) a ) + _t48;
30     *((int*)_t32) = _t54;
31     _t32 = ((char*) _t18 ) + _t48;
32     *((int*)_t32) = _t54;
33     goto B15;
34 B4:
35     j = 0;
36 B6:
37     _t48 = j + 8;
38     _t54 = j << 2;
39     _t32 = ((char*) _t18 ) + _t54;
40     _t32 = ((char*) _t32 ) + 32;
41     _PreLoad(_t32);
42     _t32 = ((char*) a ) + _t54;
43     _t32 = ((char*) _t32 ) + 32;
44     _PreLoad(_t32);
45     if( _t48 >= 512000 ) goto B8;
46 B7:
47     _t54 = j + 0;
48     _t54 = _t54 << 2;
49     _t32 = ((char*) a ) + _t54;
50     _t48 = *((int*)_t32);
51     _t32 = ((char*) _t18 ) + _t54;
52     _t54 = *((int*)_t32);
53     _t54 = _t48 * _t54;
54     soma = soma + _t54;
55     _t54 = j + 1;
56     _t54 = _t54 << 2;
57     _t32 = ((char*) a ) + _t54;
58     _t48 = *((int*)_t32);
59     _t32 = ((char*) _t18 ) + _t54;
60     _t54 = *((int*)_t32);
61     _t54 = _t48 * _t54;
62     soma = soma + _t54;
63     _t54 = j + 2;
64     _t54 = _t54 << 2;
65     _t32 = ((char*) a ) + _t54;
66     _t48 = *((int*)_t32);
67     _t32 = ((char*) _t18 ) + _t54;
68     _t54 = *((int*)_t32);
69     _t54 = _t48 * _t54;
70     soma = soma + _t54;
       _t54 = _t54 << 2;
       _t32 = ((char*) a ) + _t54;
       _t48 = *((int*)_t32);
       _t32 = ((char*) _t18 ) + _t54;
       _t54 = *((int*)_t32);
       _t54 = _t48 * _t54;
       soma = soma + _t54;
       _t54 = j + 4;
       _t54 = _t54 << 2;
       _t32 = ((char*) a ) + _t54;
       _t48 = *((int*)_t32);
       _t32 = ((char*) _t18 ) + _t54;
       _t54 = *((int*)_t32);
       _t54 = _t48 * _t54;
       soma = soma + _t54;
       _t54 = j + 5;
       _t54 = _t54 << 2;
       _t32 = ((char*) a ) + _t54;
       _t48 = *((int*)_t32);
       _t32 = ((char*) _t18 ) + _t54;
       _t54 = *((int*)_t32);
       _t54 = _t48 * _t54;
       soma = soma + _t54;
       _t54 = j + 6;
       _t54 = _t54 << 2;
       _t32 = ((char*) a ) + _t54;
       _t48 = *((int*)_t32);
       _t32 = ((char*) _t18 ) + _t54;
       _t54 = *((int*)_t32);
       _t54 = _t48 * _t54;
       soma = soma + _t54;
       _t54 = j + 7;
       _t54 = _t54 << 2;
       _t32 = ((char*) a ) + _t54;
       _t48 = *((int*)_t32);
       _t32 = ((char*) _t18 ) + _t54;
       _t54 = *((int*)_t32);
       _t54 = _t48 * _t54;
       soma = soma + _t54;
       goto B14;
B8:
       if( j >= 512000 ) goto B11;
B9:
       _t54 = j << 2;
       _t32 = ((char*) a ) + _t54;
       _t48 = *((int*)_t32);
       _t32 = ((char*) _t18 ) + _t54;
       _t54 = *((int*)_t32);
       _t54 = _t48 * _t54;
       soma = soma + _t54;
       goto B10;
B10:
       j = j + 1;
       goto B8;
B11:
       return 0;
B14:
       j = j + 8;
       goto B6;
B15:
       j = j + 1;
       goto B2;

}
```

Figura 5.17: Programa em C gerado pelo compilador Xingo executando o algoritmo de *preload* de dados

# Capítulo 6

# Resultados Experimentais

Abaixo seguem os resultados encontrados para cada um dos *benchmarks*. Alguns destes *benchmarks* tiveram alguns programas modificados para possibilitar os testes da otimização. Uma das modificações que foi necessária em alguns programas, principalmente *Livermore e DSPstone* foi o de aumentar o tamanho dos vetores usados por estes programas, pois se os vetores usados no programa são muito pequenos (menores que o tamanho da cache) a cache poderia conter todo o vetor e não seria necessário a chamada do *preload* novamente em seu reuso.

Outra modificação feita foi a conversão de vetores do tipo ponto flutuante para o tipo inteiro. Isso foi feito porque o LCC não está estável o suficiente para representações em pontos flutuantes, ficando complicado atestar os resultados obtidos pela otimização e garantir seu correto funcionamento.

Foi usado o compilador Intel C/C++ versão 1.1 para compilar os programas dos benchmarks para a plataforma Pocket PC 2002. A escolha desse compilador foi pelo fato de o mesmo reconhecer funções instrínsecas (principalmente a de *preload* usada neste trabalho) de forma automática no código fonte. Além disso, a representação em linguagem *assembly* do programa fonte é bem mais compreensível do que a representação gerada pelo compilador da Microsoft. Para alguns programas, houve a necessidade da análise de sua representação em *assembly*, a fim de justificar alguns resultados.

Inicialmente alguns programas também foram compilados com o compilador da Microsoft, resultando em tempos de execução muito próximos comparado ao compilador da Intel. Assim, pelas justificativas acima é que o compilador da Intel foi o escolhido para ser usado neste trabalho.

Para várias execuções de um mesmo programa no *pocket pc*, a diferença entre estes tempos variava muito pouco (poucos milisegundos). Mesmo assim, todos os programas foram executados mais de uma vez, a fim de garantir que a variação destes tempos fosse insignificante para os resultados.

A referência [21] diz que para que uma aplicação do *pocket pc* obtenha uma melhor performance, é necessário compilar as interfaces gráficas com o compilador da Microsoft (por possuir bibliotecas gráficas mais otimizadas), e o resto da aplicação com o compilador da Intel (por possuir melhores otimizações dependentes da arquitetura e outras bibliotecas melhores otimizadas). Como o objetivo deste trabalho foi o de compilar programas sem interfaces gráficas, o compilador da Intel foi o mais adequado para tal.

Os programas foram compilados primeiramente pelo compilador Xingo, passando por algumas otimizações de código como: *Commom Subexpression Elimination, PeepHole Optimization, Copy Propagation, Constant Propagation e Dead Code Elimination*. Os arquivos de saída do Xingo (programa em C), foram incluídos em um projeto no Embedded Visual C++ e compilados com o compilador da Intel, usando a opção de otimização de deixar o programa mais rápido.

## 6.1 MeuBench

O *benchmark* sintético *MeuBench* foi desenvolvido neste projeto com o propósito de tentar cobrir vários casos de programas reais no qual a otimização poderia funcionar. Os programas deste *benchmark* também foram incluídos no apêndice A.

Estes programas foram construídos para testar a eficiência da otimização em vários casos, como vetores declarados dinamicamente e estaticamente dentro de laços simples, laços aninhados, laços com saltos, etc.

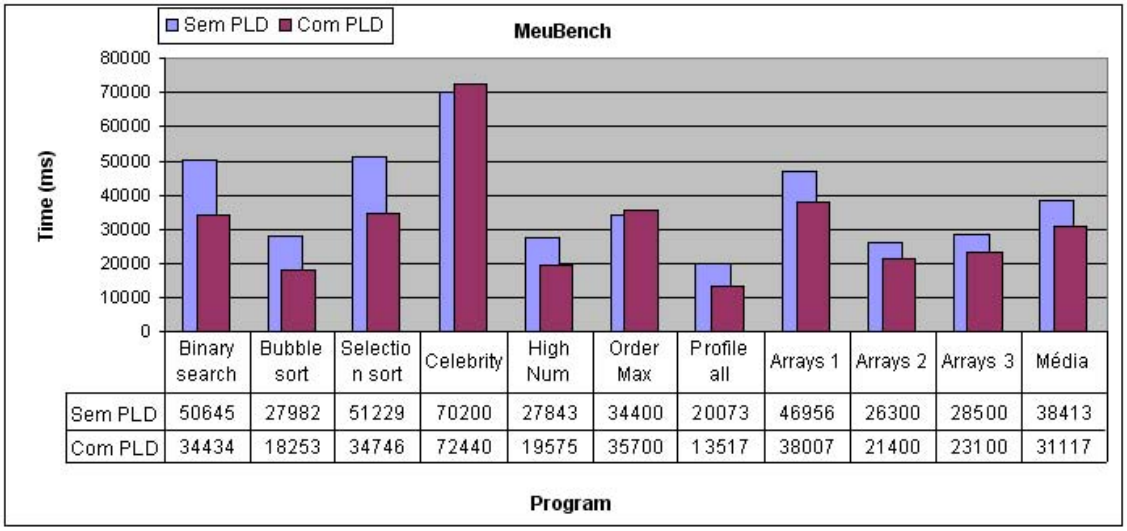
Apesar de tentar cobrir vários possíveis casos de programas reais, eles são bastante simples e serviram no início do projeto para testar a eficiência e a corretude da otimização.

Alguns destes programas são baseados em problemas da teoria da computação, e foram os primeiros programas mais realísticos a serem testados com a otimização. O desempenho medido em tempo e *speedup* para este benchmark pode ser visto nos gráficos mostrados nas figuras 6.1 e 6.2.

Um fato importante observado nos testes feitos com os programas deste *benchmark*, é que se a complexidade de um programa for ditada pelo aninhamento de seus laços, quanto maior o aninhamento, melhor pode ser seu desempenho quando utilizando a otimização (para vetores grandes).

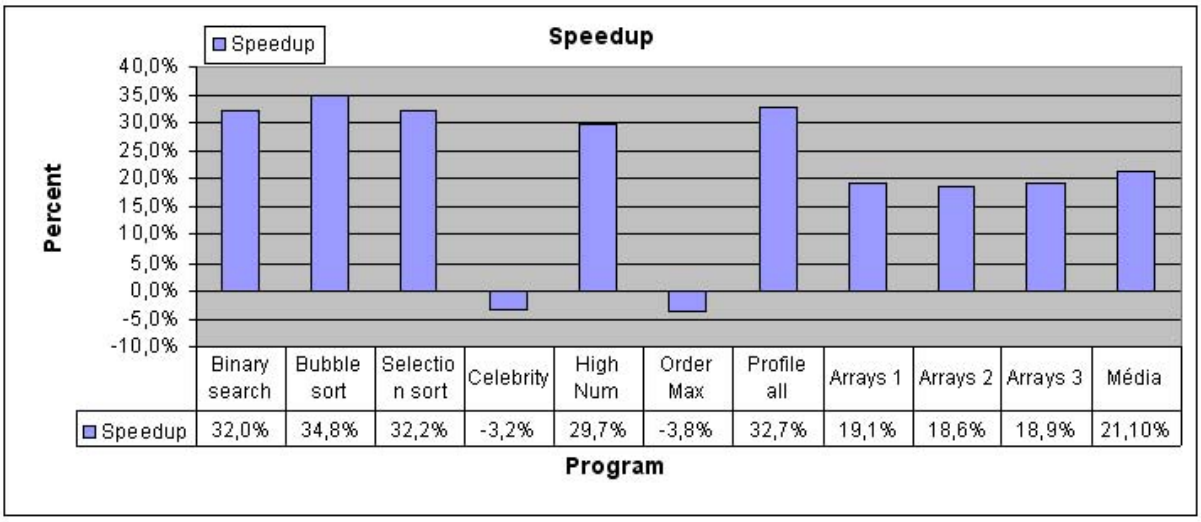
Isso pode ser notado nos programas do *benchmark MeuBench*. Os programas arrays 1, 2 e 3 possuem um único laço em cada um desses programas. Assim, a melhora em performance chega a aproximadamente 19%. Com os outros programas que tem uma complexidade $n^2$, a melhora passa dos 30% em sua maioria.

Isso deve-se ao fato de que como os vetores são grandes, quando houver o reuso das suas primeiras posições pela execução do laço mais interno, o gerenciador de memória já terá substituído na cache todos os dados que foram usados inicialmente resultando em




| | Binary search | Bubble sort | Selection sort | Celebrity | High Num | Order Max | Profile all | Arrays 1 | Arrays 2 | Arrays 3 | Média |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Sem PLD | 50645 | 27982 | 51229 | 70200 | 27843 | 34400 | 20073 | 46956 | 26300 | 28500 | 38413 |
| Com PLD | 34434 | 18253 | 34746 | 72440 | 19575 | 35700 | 13517 | 38007 | 21400 | 23100 | 31117 |

Figura 6.1: Tempo gasto nos laços (em *ms*) para programas do MeuBench




| | Binary search | Bubble sort | Selection sort | Celebrity | High Num | Order Max | Profile all | Arrays 1 | Arrays 2 | Arrays 3 | Média |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Speedup | 32,0% | 34,8% | 32,2% | -3,2% | 29,7% | -3,8% | 32,7% | 19,1% | 18,6% | 18,9% | 21,10% |

Figura 6.2: Speedup com a otimização *preload* (em %) para programas do MeuBench

vários *misses* na cache. Com a otimização, estes *misses* não acontecem, pois o *preload* é efetivo no laço mais interno a cada iteração dos laços mais externos. Se o vetor fosse pequeno o bastante para permanecer por inteiro na cache, as chamadas à instrução de *preload* poderiam prejudicar o desempenho, pois os dados já poderiam estar na cache.

O programa *binary search* obteve um bom resultado, pois para se fazer a busca binária de um valor em um vetor, primeiramente é usada uma função para ordenar o vetor, e é nesta função que a otimização *preload* age, otimizando-a e fazendo com que este programa executar 32% mais rápido.

Os programas *selection sort e bubble sort* obtiveram uma melhora significativa (34,8% e 32,2%) sendo métodos diferentes de ordenar os vetores. O método *bubble sort* é um pouco mais inteligente, e executa em menos tempo do que o método *selection sort* (veja a tabela de tempos de execução), mas ambos são métodos de ordenação gulosa de complexidade $n^2$.

O método *selection sort* sai varrendo o vetor procurando a posição que possui o menor valor, colocando-a na primeira posição. Depois faz isso novamente começando pela posição 2 em diante.

O método *bubble sort* varre o vetor comparando os valores em posições vizinhas. Se a posição acima for maior, há uma troca de posições entre estes valores. Isso é feito $N$ vezes garantindo que um vetor de tamanho $N$ seja ordenado. Este método é um pouco mais inteligente, pois se ele não fez uma troca de posição nas $N$ vezes que procurou em uma das $N$ iterações, este saí fora do laço pois sabe que o vetor já esta ordenado.

No programa *Celebrity* existe uma matriz que representa a relação de conhecimento entre pessoas. Esta relação de conhecimento de uma pessoa $i$ por uma pessoa $j$ é representada pelo número 1 na linha $i$ e na coluna $j$ desta matriz. Assim uma pessoa é celebridade se ela é conhecida por todos e ela não conhece ninguém, ou seja, sua linha $i$ está com o número um em todas as posições (todos a conhecem) e na coluna $i$ ela não conhece ninguém (todas as posições dessa coluna são 0). Um exemplo de uma celebridade pode ser vista na matrix abaixo.

$$\mathbf{X} = \begin{pmatrix} X & 1 & 0 & 1 \\ 0 & X & 0 & 1 \\ 1 & 1 & X & 1 \\ 0 & 1 & 0 & X \end{pmatrix}$$

No caso da matriz acima, a celebridade seria aquela que está na $3^a$ linha da matriz, ou seja, a pessoa número 3.

Neste programa houve uma leve piora no tempo de execução pois a matriz é percorrida pelo laço mais interno a cada iteração com acessos à memória com localidade temporal.

Também existe o aspecto de como se trata de acessos em uma matriz, os acessos à memória são maiores, ainda mais com o despacho da instrução *preload* para a memória. É possível que no programa original não tenha ocorrido *overflow* no *pend buffer* e com o despacho do *preload* ocorra a paralisação do processador por algum tempo. Nota-se ainda que este programa possui saltos no corpo do laço mais interno, não podendo desenrolar o mesmo e não fazendo proveito da localidade espacial que o *preload* oferece. Se o programa fosse codificado de outra forma, percorrendo cada linha primeiro e depois sua coluna, o resultado poderia ser positivo (isto também é verdade para o desempenho da cache mesmo sem a otimização de *preload* de dados).

Isso mostra que a forma de como um programa é codificado infere no resultado do despacho de instruções *preloads*. Isso é muito comum no uso de matrizes. De acordo com a forma que uma matriz é armazenada na memória, e de acordo com a forma como esta é acessada (com localidade espacial e não temporal), pode ser que o desempenho melhore com a aplicação do algoritmo de *preload*. Um estudo interessante sobre otimizações de memória pode ser visto em [44][45].

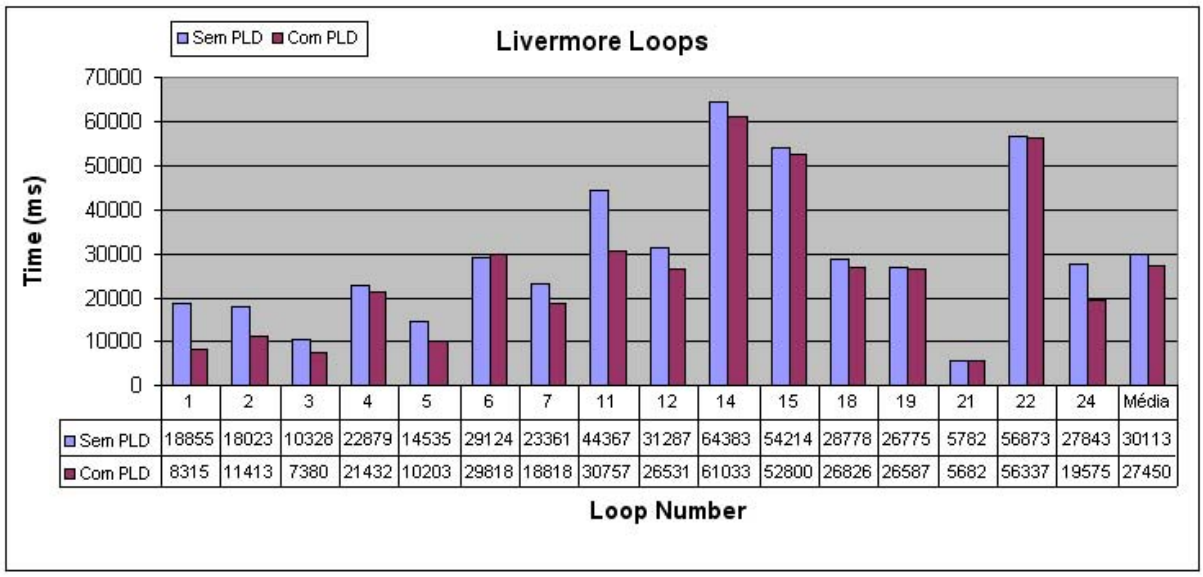
O programa *high number* obteve uma boa melhora (29,7%). Este programa procura pelo maior valor em um vetor.

O programa *order max* faz uma ordenação sabendo qual o valor máximo que pode ter no vetor de entrada. São criados dois vetores e um deles (vetor de quantidades) é zerado em todas as posições, pois este guarda o número de vezes que determinado valor $n$ ocorre no vetor de entrada (complexidade $N$), fazendo um incremento na posição $n$ do vetor. Neste programa houve uma leve piora, pois o vetor de quantidades é acessado de forma aleatória de acordo com os valores lidos no vetor de entrada.

O programa *profile all* obteve uma ótima melhora em relação ao programa original (32,7%). O objetivo deste foi investigar os possíveis casos em que a otimização de *preload* de dados poderia ser aplicada e verificar os resultados. Os casos cobertos com este programa foram:

- uso de vetores dinâmicos em laço sem saltos.
- uso de vetores dinâmicos em laço com saltos.
- uso de vetores estáticos em laço sem saltos.
- uso de vetores estáticos em laço com saltos.
- uso de vetores em laços aninhados.

O uso destes programas deste *benchmark* foi muito útil, pois permitiu as primeiras verificações de erros no algoritmo de *preload*.Também foi através destes que foram obtidos




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 11 | 12 | 14 | 15 | 18 | 19 | 21 | 22 | 24 | Média |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sem PLD | 18855 | 18023 | 10328 | 22879 | 14535 | 29124 | 23361 | 44367 | 31287 | 64383 | 54214 | 28778 | 26775 | 5782 | 56873 | 27843 | 30113 |
| Com PLD | 8315 | 11413 | 7380 | 21432 | 10203 | 29818 | 18818 | 30757 | 26531 | 61033 | 52800 | 26826 | 26587 | 5682 | 56337 | 19575 | 27450 |

Figura 6.3: Tempo gasto nos laços (em *ms*)

os primeiros resultados da otimização. Os resultados foram satisfatórios, mas devemos levar em consideração que estes são programas sintéticos voltados para o perfil da otimização de *preload* de dados. Os próximos benchmarks serviram como exemplo prático da aplicação desta otimização para programas de propósito geral.

## 6.2 Livermore Loops

Este *benchmark* é composto por 24 laços de kernels de aplicações numéricas e foi originalmente escrito na linguagem de programação *FORTRAN* [56]. Este *benchmark* vem sendo usado para a verificação do desempenho aritmético de computadores e compiladores desde o início dos anos 70.

A figura 6.3 lista os resultados encontrados para o *benchmark Livermore Loops*. Esta figura mostra os tempos de execução com e sem a otimização de *preload* executados com o compilador Xingo. Podemos ver que na maioria dos laços houve uma melhora significativa. A diferença dos tempos é bastante variada, pois isso depende do tipo do laço em questão e do tamanho do vetor ao qual o laço foi submetido. Os programas utilizados para os testes podem ser encontrados no apêndice A.

O gráfico da figura 6.4 mostra o *speedup* que se obteve na execução dos laços quando a otimização de *preload* é utilizada. Este gráfico nos dá uma visão melhor dos ganhos em desempenho.

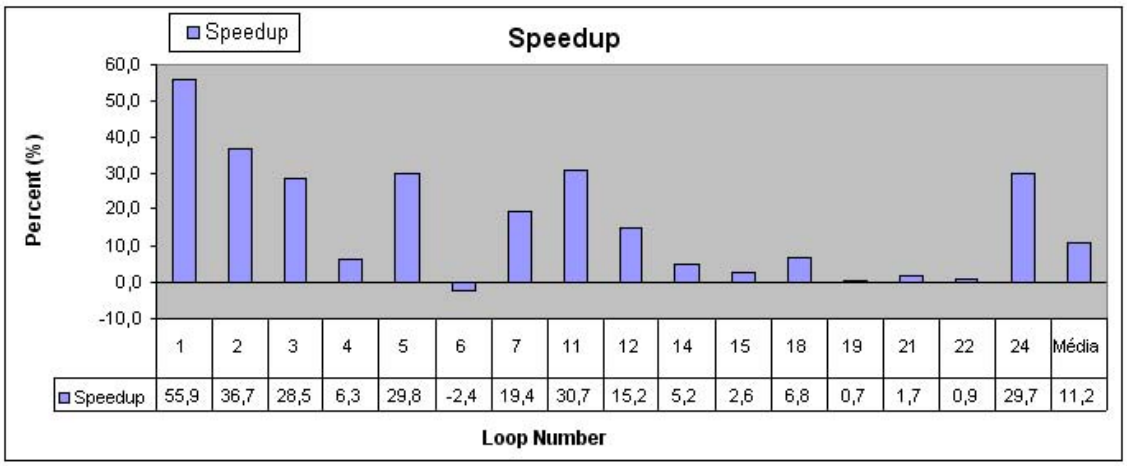



Figura 6.4: Speedup com a otimização *preload* (em %)

A otimização pode ser aplicada em 16 dos 24 laços deste benchmark, pois estes eram adequados ao perfil para a aplicação da otimização.

A média geral do *speedup* de todos os laços Livermore foi de 11,15%. Esta média poderia ser muito melhor devido ao fato de que em alguns destes laços a otimização não foi aplicada devido à sua grande complexidade, e em alguns outros por não possuírem o perfil necessário para a aplicação da otimização. A média do *speedup* nos laços em que foi aplicada a otimização foi 16,73%, um bom resultado na média.

Podemos ver que nos laços 1, 2, 3, 5, 11 e 24 houve uma melhora em desempenho de mais do que 28% em relação ao programa original, resultando em uma ótima média de aproximados 35,2%. Estes laços foram bem sucedidos na otimização, pois possuem o perfil desejado para se executar esta otimização, ou seja, uso de vetores no laço mais interno com corpo pequeno e sem *branches*.

O laço mais interno do segundo laço Livermore possui a seguinte operação.

```
x[i] = x[k] - v[k] * x[k-1] - v[k+1] * x[k+1];
```

Podemos ver que existem vários usos de várias posições dos vetores, e o *stride* entre estes é pequeno ($< 32$ bytes). Se fosse chamado *preload* para cada uso desses vetores, teriam que ser despachadas 5 instruções *preloads*, o que paralisaria o processador ocorrendo um *"pend buffer overflow"*. Com a execução de nossa otimização isso não ocorre pois esta verifica que não são necessários todos estes *preloads*, e despacha somente 2, o que é o correto (uma para o vetor $x$ e outro para o vetor $v$).

No laço 4 (6,3%), a otimização não foi tão bem sucedida pois o *stride* de acesso aos vetores é muito grande. A variável de indução de controle do laço é incrementada em 5

posições a cada iteração, ou seja, se o vetor é de um tipo de 4 bytes, o *stride* será de 20 bytes (5 x 4). Neste caso, não está sendo aproveitada a linha da cache que o *preload* busca para a cache.

No laço 6 houve uma leve piora na performance (-2,4%) pelo fato do padrão de acesso aos vetores ser mais complexo. Neste laço foi utlizada uma matriz cujo padrão de acesso é temporal, e por isso não pode ser melhorada pela otimização. O outro vetor também tem o problema de possuir na sua função de formação duas variáveis de indução que controlam os dois laços mais internos, e o valor máximo de uma delas depende da outra. Neste caso, este vetor é reusado várias vezes nas primeiras iterações do segundo laço mais interno, tornando desnecessário o *preload* nas primeiras iterações do laço mais interno. Estes valores estão na cache e o *preload* neste caso é desnecessário.

No laço 7 a melhora do desempenho foi boa (19,4%) e só não pode ser melhorado porque este laço usa muitos vetores. Com a política de despachar somente 3 *preloads*, puderam ser despachados somente para três destes vetores. Além disso, as operações entre estes vetores são mais complexas o que aumenta consideravelmente o tamanho do corpo do laço. Um dos vetores foi usado 7 vezes, com *strides* variados, do qual somente poderiam ser despachados 2 *preloads* para 2 destes usos. Como existem outros dois vetores diferentes destes, não se pode despachar mais instruções *preloads*. Assim, ocorre pelo menos um *miss* na cache de dados para alguns dos usos deste buffer (como este vetor usa 4 referências no corpo do laço, o *pend buffer* provavelmente resultou em *overflow* o que degradou um pouco o desempenho do resultado).

O laço 8 é muito complexo como pode ser visto no apêndice A. Este laço possui o corpo muito grande com matrizes tridimensionais e vários vetores com padrões de acesso variados. Por isso a otimização não pode ser executada neste e nos próximos laços (9, 10, 13, 16, 17, 20, 23)

Os laços 9, 10, 13 além de serem complexos, contém um corpo extenso com vários acessos a matrizes, estes não são acessados com localidade espacial, e sim temporal (*strides* muito grandes).

No laço 14 não houve uma grande melhora devido ao fato de vetores serem usados como índices de outros vetores, gerando referências para referências (como acontece em estruturas de dados tal como listas ligadas). Nestes casos, o acesso aos vetores é não determinístico e por isso não se pode aplicar a otimização.

O laço 15 possui um corpo muito extenso com vários *branches* e uso de matrizes, e por isso não houve quase nenhuma melhora.

O laço 18 não houve uma melhora significativa pois os laços possuiam grande quantidade de computação sobre matrizes, sobrecarregando os recursos do sistema, (*pend e fill buffers*, transferências de dados no barramento). Neste caso fica difícil fazer com que a otimização funcione adequadamente.

O laço 21 não obteve um bom resultado pois este possui usos de matrizes com acessos temporais.

O laço 22 possui vários usos de vetores diferentes, impossibilitando o despacho de *preload* para todos eles. Além disso, faz-se uso de uma função externa no corpo do laço o que dificulta o cálculo da latência de execução do laço, pois não podemos inferir em quantos ciclos esta função irá executar. Por isso, o fator de desenrolamento do laço pode ser calculado de forma errônea o que dificulta a aplicação correta da otimização.

Os laços que obtiveram pouca ou quase nenhuma melhora em desempenho foram os laços que possuem seus corpos extensos, fazendo com que não fosse aplicado o desenrolamento de laço (pois o cálculo da latência do corpo do laço ultrapassava os 70 ciclos de acesso à memória) não permitindo o aproveitamento da localidade espacial que o *preload* oferece. Alguns destes laços também possuem muitos saltos em seu corpo, e alguns possuem mais de um ponto de saída do laço, o que minimiza as chances de melhora de desempenho.

Em todos estes laços foram testados os resultados armazenados nos vetores. Foram gravados em arquivo os valores de saída dos vetores do programa original, e posteriormente na execução do programa com a otimização *preload* no *Pocket PC*. Estes arquivos de saída foram comparados para ver se os resultados eram os mesmos, verificando se a otimização não estava alterando a semântica do programa. Foi desta forma que foram corrigidos vários erros e problemas que a otimização continha, tonando-a mais confiável para uma gama variada de aplicações.

Este benchmark foi utilizado para realizar uma calibração do cálculo da latência de instruções no corpo do laço, atribuindo a cada instrução do Xingo uma latência de acordo com sua latência como especificada no manual do XScale.

## 6.3 DSPstone

DSPstone [60] é um *benchmark* com o propósito de testar compiladores desenvolvidos para arquiteturas DSP's.

Vários programas deste *benchmark* possuem o perfil da otimização de *preload* de dados para que esta seja efetiva e possa ser executada. Na maioria dos programas testados foi aumentado o tamanho dos vetores para a verificação da eficiência da otimização. Isso foi útil até mesmo para que o tempo de execução fosse maior (geralmente estes programas na sua forma original executam muito rapidamente - em torno de 1 ms no *Pocket PC*). Se um programa é executado muito rápido, então não podemos precisar o resultado obtido com a otimização de *preload* de dados, pois a diferença de tempo obtida com e sem a otimização pode ser mínima.

O desempenho destes programas é mostrado nos gráficos de tempo de execução e de

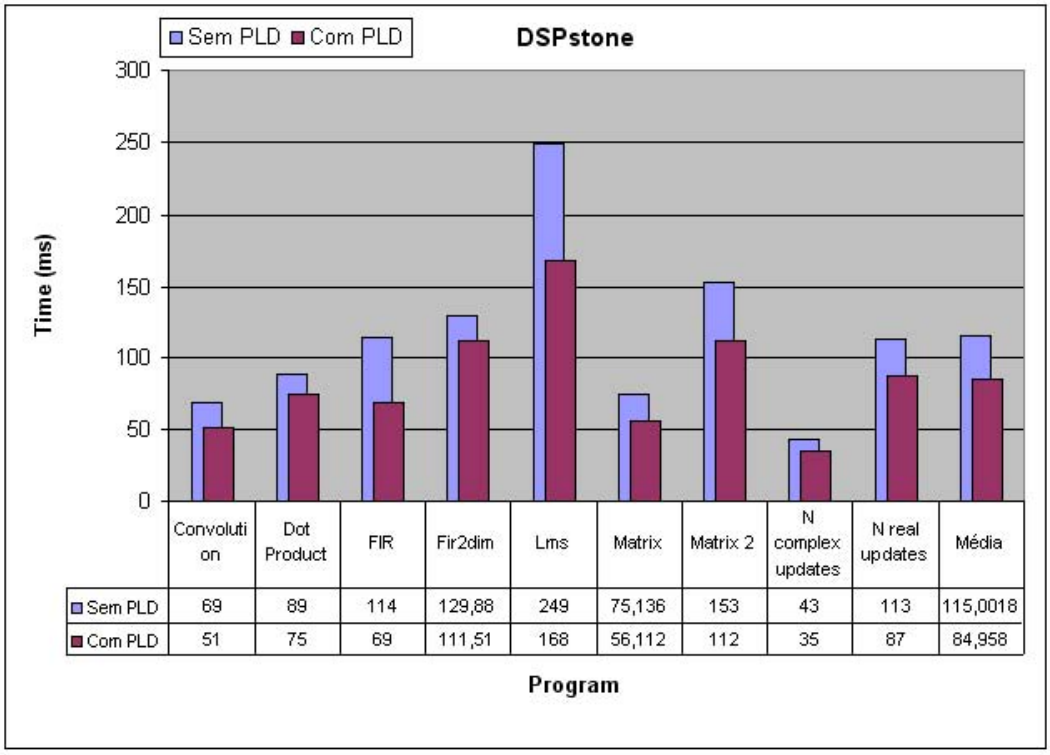



Figura 6.5: Tempo gasto nos laços (em *ms*) para programas do DSPstone

*speedup* (figuras 6.5 e 6.6).

Na maioria dos programas deste *benchmark* os resultados obtidos foram satisfatórios. A média geral foi de 19,47%.

Os programas *convolution, dot product, FIR, FIR2DIM, lms, Matrix, Matrix 2, N-complex, N-real updates* obtiveram uma melhora significativa, mais de 23% em 6 deles.

Em geral os resultados foram muito bons, com destaque para o programa de filtro *FIR* que obteve uma melhora de aproximadamente 40%.

## 6.4 MiBench

MiBench [16] é um *benchmark* que possui várias categorias de aplicações, oferecendo diferentes tipos de ambientes de execução, e capacitando os pesquisadores em compiladores e arquiteturas a testar seus projetos para vários programas de alguns segmentos de mercado.

Como Vanderwiel mostra em seu trabalho [41], *preload* para aplicações de propósito geral são mais difíceis de aplicar, pois estes programas geralmente fazem uso de estruturas

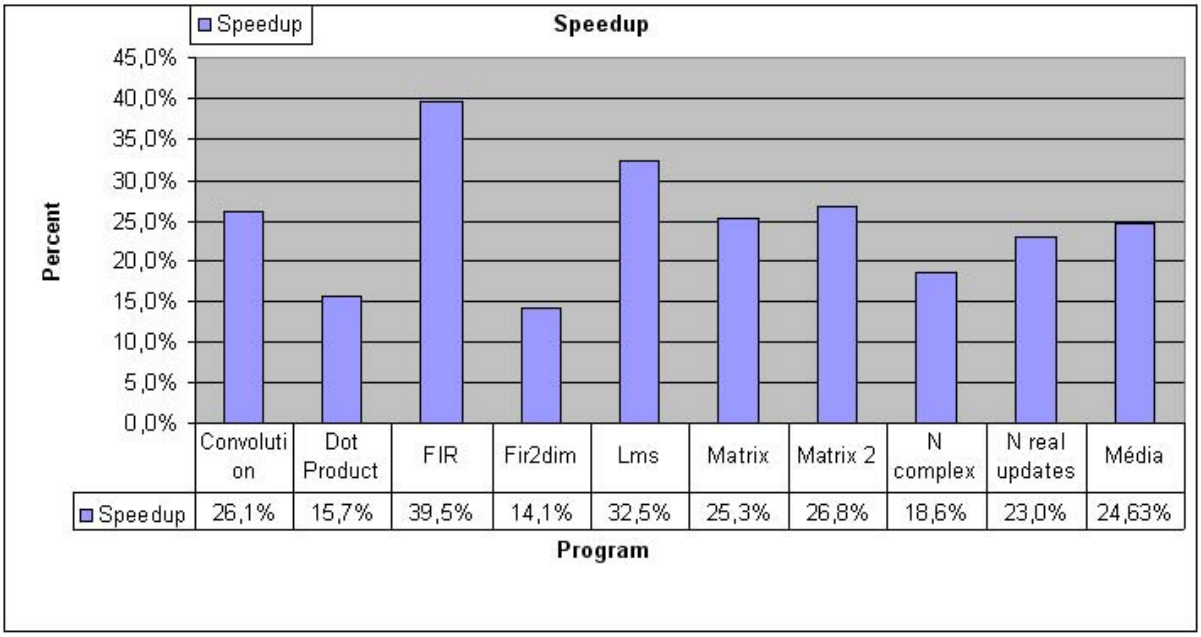



Figura 6.6: Speedup com a otimização *preload* (em %) para programas do DSPstone

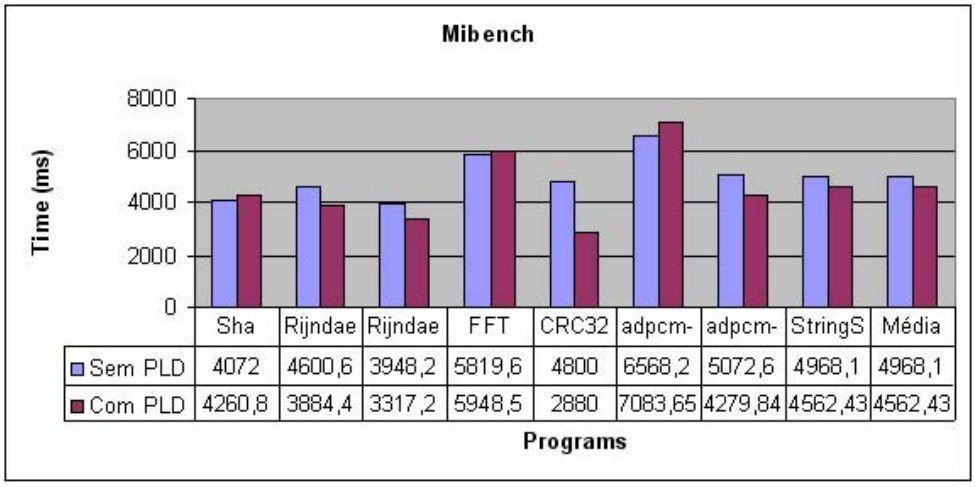
de dados mais complexas, tentado abstrair os aspectos de programação com a realidade.

Vários artigos [26][27][29][47] também mostram que o trabalho de aplicar *preload* para estes tipos de aplicação é muito mais difícil, pois pode ser que estas estruturas de dados sejam alocadas em áreas diferentes da memória, perdendo a propriedade da localidade espacial do *preload* (linha da cache). Além disso, nestes casos existem referências de referência para dados na memória e é mais complicado descobrir para qual referência o *preload* deveria despachar.

Estas aplicações geralmente já exibem um bom desempenho na cache de dados e produzem padrões de referências à memória altamente irregulares, não se beneficiando muito da otimização de *preload* de dados [42].

Como podemos observar, fazer a otimização de *preload* de dados funcionar com aplicações de propósito geral é bem mais complicado que para aplicações científicas. Várias técnicas têm sido discutidas para que a otimização resulte em um melhor desempenho para estes casos (algumas destas técnicas foram discutidas na seção 4). Estas novas técnicas são baseadas em outras técnicas existentes, havendo um incremento de novas idéias, como a utilização de um buffer de referências.

Os programas do MiBench foram escolhidos de acordo com o perfil desta otimização de *preload* de dados, pois não haveria a possibilidade desta otimização ser executada em




| | Sha | Rijndae | Rijndae | FFT | CRC32 | adpcm- | adpcm- | StringS | Média |
|---|---|---|---|---|---|---|---|---|---|
| Sem PLD | 4072 | 4600,6 | 3948,2 | 5819,6 | 4800 | 6568,2 | 5072,6 | 4968,1 | 4968,1 |
| Com PLD | 4260,8 | 3884,4 | 3317,2 | 5948,5 | 2880 | 7083,65 | 4279,84 | 4562,43 | 4562,43 |

Figura 6.7: Tempo gasto nos laços (em *ms*) para programas do Mibench

programas com estruturas de dados complexas.

Os resultados referentes a este benchmark podem ser visualizados nos gráficos mostrados nas figuras 6.7 e 6.8.

*SHA* (*Secure Hash Algorithm*) é um algoritmo de condensação de dados empregado em uma grande variedade de aplicações de segurança e protocolos, como *TLS, SSL, PGP, SSH, S/MIME*, e *IPSec* [64].

O programa *Sha* neste benchmark não possui o perfil completo para a aplicação da otimização de *preload* de dados. Os laços otimizados possuem vetores de tamanho pequeno e este é sempre reusado para uma nova cópia de dados do buffer de entrada. É por isso que esta aplicação obteve uma leve piora no desempenho (-4,6%). Um outro teste feito com esta aplicação é que o código gerado pelo compilador Xingo sem passar pela otimização de *preload* de dados, também executa no mesmo tempo ou até um pouco pior do que a passada pela otimização.

Foi feito um teste com tentativas de inserção das instruções *preloads* para alguns laços no código fonte original desta aplicação, e este mostrou uma piora do desempenho, pior até que o código gerado pelo compilador Xingo. Como o código gerado pelo compilador Xingo otimizou mais laços do que este teste, talvez alguns laços tenham sido otimizados mas outros piorados. Na média o desempenho piorou um pouco como pode ser visto nos resultados.

O programa *Rijndael* é uma implementação do algoritmo de encriptação *AES* desenvolvido por Joan Daemen e Vincent Rijmen. Este algoritmo foi implementado para blocos e tamanho de chaves de 128, 192 e 256 bits (16, 24 e 32 bytes) por Brian Gladman [16].

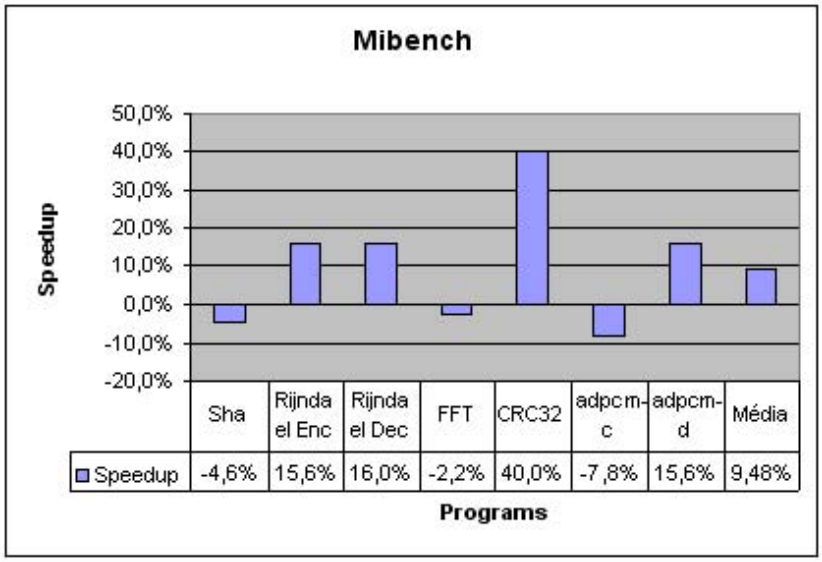



Figura 6.8: Speedup com a otimização *preload* (em %) para programas do Mibench

*Rijndael Enc* obteve uma boa melhora de 15,6% enquanto *Rijndael Dec* obteve uma melhora de 16%. Estes programas são compostos de vários laços que usam vetores, e algum deles puderam ser otimizados eficientemente.

*FFT* (*fast Fourier transform*) é um algoritmo computacionalmente eficiente que implementa a transformada discreta de Fourier. Ele é usado para transformar dados discretos no domínio do tempo e espaço para o domínio da freqüência [66]. *FFT* obteve uma piora de -2,2% devido ao fato da otimização do laço mais interno da função *fft_float()* ter um corpo muito extenso e possuir vários vetores. As requisições de memória para este programa são várias. Despachando as instruções *preload* pode deteriorar o tempo de execução devido à pressão no buffer de pendências. Neste caso, ocorre *overflow* do buffer que paralisa o processador.

*CRC* (*Cyclic Redundancy Check*) é uma técnica usada para se prover confiabilidade em transmissões de dados. *CRC* mantém a informação redundante dos dados transmitidos e pode ajudar a detectar erros nestes dados [65]. O programa CRC32 do Mibench, obteve uma ótima melhora de 40%, pois é uma aplicação adequada para esta otimização.

*ADPCM* (*Adaptive Differential Pulse Code Modulation*) é um método de codificação de sons e dados, que requer menos espaço de armazenamento (*menor bit-rate*) do que o formato *PCM* usado por arquivos *WAV* [67].

O *ADPCM coder* teve uma leve piora de -7,8% em seu desempenho, enquanto seu decoder melhorou em 15,6%. Estas aplicações possuem um laço muito extenso e cheio

de saltos. Em experimentos, a aplicação *ADPCM coder* executou em 19,5 segundos no programa original, enquanto que passando pelo compilador Xingo sem executar a otimização de *preload* de dados, executou em 20,7 segundos. Observamos que neste cenário houve uma piora de -5,8%. O código transformado pelo compilador Xingo neste caso não melhorou o desempenho em relação ao código original compilado pelo compilador do XScale (compilador Intel). Com a execução da otimização de *preload* de dados pelo compilador Xingo o tempo de execução passou de 20,7 segundos para 20,9 segundos.

Como visto no início desta subseção, aplicações de propósito geral apresentam comportamentos variados para a otimização de *preload* de dados. De acordo com os resultados apresentados, essas são mais difíceis de se obter um ganho de desempenho.

# Capítulo 7

# Conclusões e Trabalhos Futuros

De acordo com os resultados apresentados, conclui-se que esta otimização de software *preload* obteve resultados satisfatórios onde várias aplicações obtiveram um fator de *speedup* de aproximadamente 40%, com uma média de 18,20% dos programas testados.

Esta implementação será de grande utilidade para compilar aplicações já desenvolvidas na linguagem de programação C, que forem portadas para o sistema Pocket PC. Como o mercado de PDA's com o sistema Pocket PC está em ascensão, utilizar esta otimização pode ser um fator determinante no quesito desempenho para o desenvolvimento de aplicações para estes dispositivos.

É necessário precauções quanto ao tipo da aplicação que se deseje otimizar, pois esta deve possuir o perfil de usos de vetores de tamanho grande (pelo menos metade do tamanho da cache) em laços de um programa, cuja forma de acesso a estes seja linear e espacial. Existem várias aplicações, principalmente as numéricas, as quais esta otimização pode ser aplicada.

## 7.1 Justificativas do Algoritmo de *Preload* de Dados Utilizado

Como vimos anteriormente, optamos por escolher uma heurística que não despachasse mais de três *preloads* por iteração do laço, com o objetivo de não sobrecarregar o buffer de pendências (e de preenchimento) de requisições à memória.

O algoritmo de *preload* de dados não faz depachos de instruções *preload* para cada referência de memória encontrada em um programa, o algoritmo faz um estudo destas referências e despacha somente as necessárias, ou seja, o algoritmo não faz despachos indiscriminados.

Para justificar o não uso da técnica de *Software Pipeline* e o uso da política de *Leading*

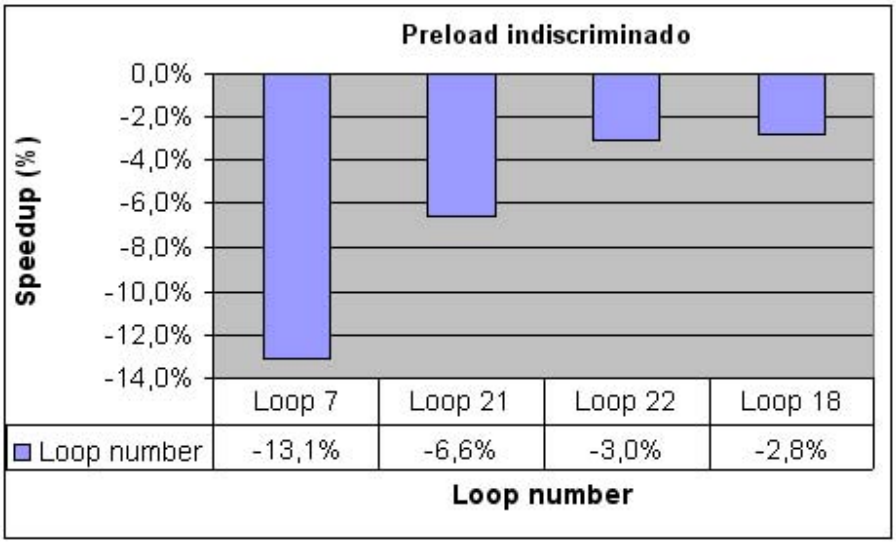



Figura 7.1: Preload indiscriminado e Software Pipeline em relação ao algoritmos de Preload utilizado

*Reference*, compilamos alguns programas do benchmark *Livermore Loops* com o compilador Xingo e com a otimização de *preload* de dados habilitada. Os resultados podem ser observados na figura 7.1.

Este teste foi feito com laços que possuíam várias referências à memória em seu corpo, para que houvesse propositalmente vários despachos da instrução *preload*. Percebemos por este gráfico que em todos os programas testados (com destaque para o *loop* 7 (-13,1%)), houve uma deterioração da performance em relação ao algoritmo usado. Com o aumento de *preloads* despachados devido ao *Software Pipeline* e ao despacho a todas as referências de memória encontrada, houve uma degradação de performance devido ao *overflow* dos buffers de pendências e/ou de preenchimento que paralisou o processador até que uma das referências fosse resolvida. Também devemos levar em consideração que a cada instrução *preload* despachada sem necessidade houve gasto de 1 ciclo extra do processador.

## 7.2 Tamanhos dos Arquivos Objetos Otimizados

Como os dispositivos eletrônicos portáteis chamados de sistemas embutidos possuem limitação no uso da memória, também devemos analisar o quanto a otimização de *preload* aumenta o tamanho do programa.

Como a otimização de *preload* de dados faz desenrolamento de laços e inserção de

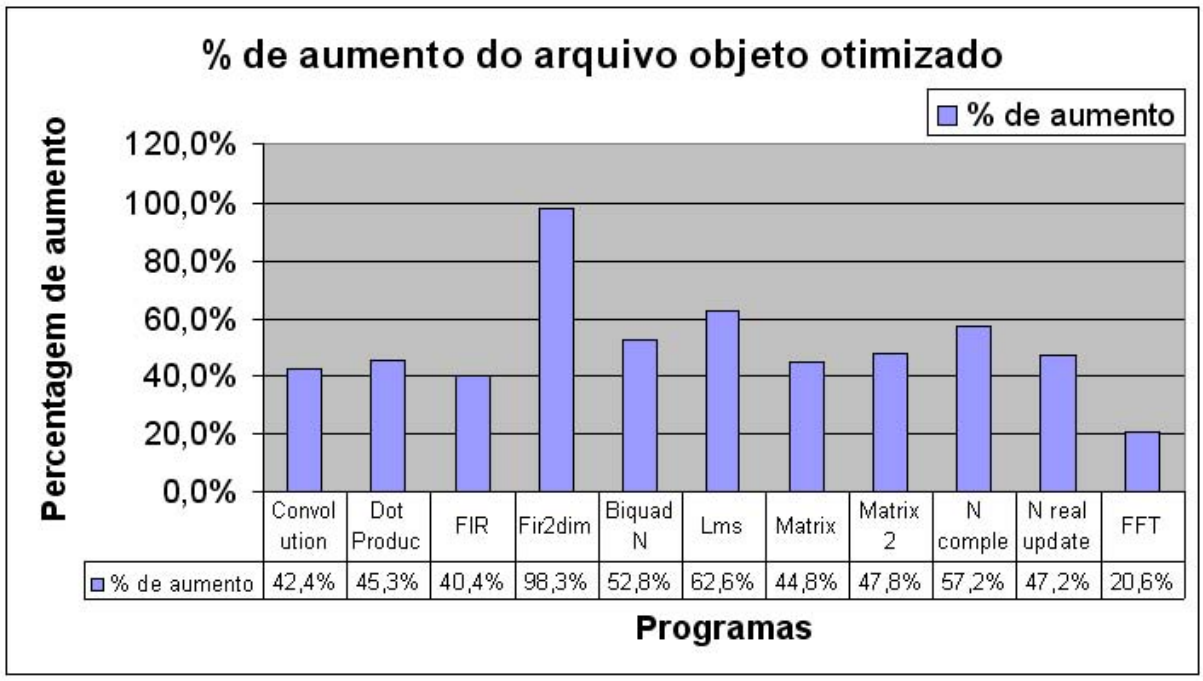



Figura 7.2: Aumento no tamanho dos arquivos objetos (em %) com e sem a otimização de *preload* de dados

novas instruções, é natural que esta otimização aumente o tamanho do código objeto do programa compilado. Para demostrar isto, compilamos alguns programas do *benchmark DSPstone* e avaliamos o tamanho de seus arquivos objetos sem e com a otimização de *preload* de dados. Os resultados podem ser visto na figura 7.2.

O tamanho do código gerado com a otimização tem influência principalmente pelos fatores: desenrolamento do laço, tamanho do corpo do laço, e do número de laços que podem ser otimizados. Como os programas deste benchmark são muito pequenos e possuem um laço único, o aumento do código objeto foi considerável (em média de 50%) como pode ser visto na figura 7.2.

No caso do programa *fir2dim*, o aumento foi de 98,3% pois foram otimizados 3 laços que tinha corpos pequenos, assim o fator de desenrolamento tende a ser grande.

Entretanto, cada programa deve ser analisado à parte, bem como a quantidade de memória disponível para executá-lo. Programas pequenos que possuem poucos laços e todos otimizáveis por *preload* de dados, tendem a aumentar seu código consideravelmente, (como são pequenos, talvez não tenhamos problemas com memória). Programas maiores geralmente não possuem todos os laços otimizáveis e com isso a percetagem do aumento do código diminui.

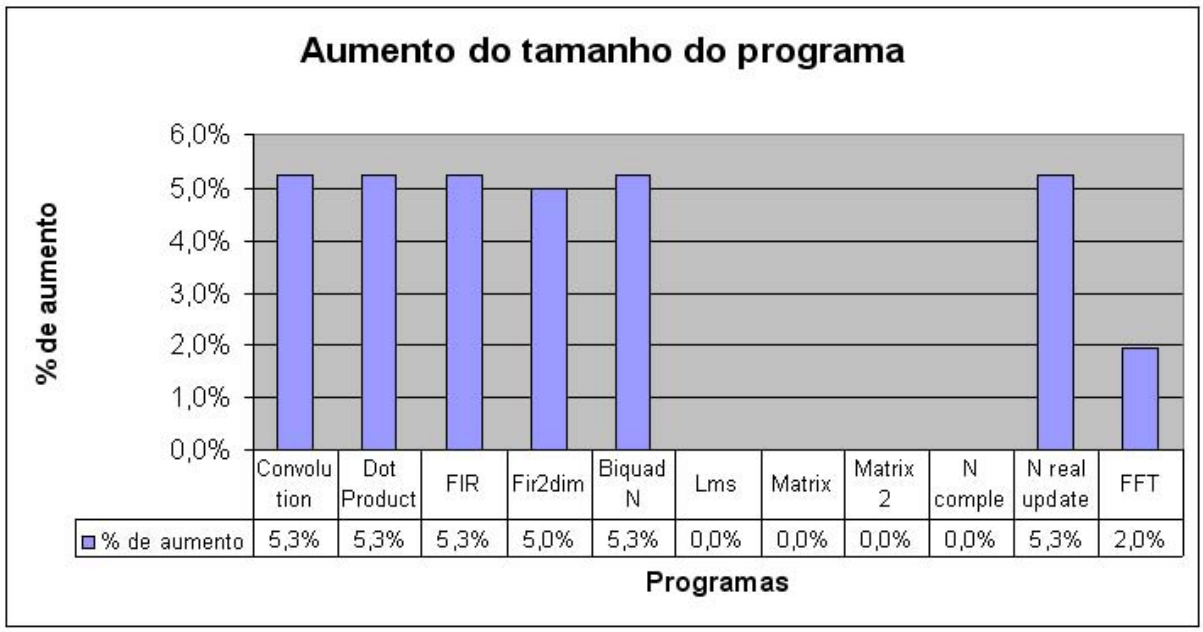



Figura 7.3: Aumento no tamanho dos arquivos executáveis (em %) com e sem a otimização de *preload* de dados

Para o *Pocket PC 2002* utilizado neste trabalho, não houveram problemas nas limitações ao uso de memória. Primeiro, o *linker* [1] do compilador *Microsoft eMbedded C++* reserva blocos fixos de memória para os programas, e o tamanho do código objeto dos programas compilados não teve grande influência na geração do programa executável final. Segundo, tive que utilizar interface gráfica com controles para verificar o tempo de execução de cada laço, e por isso o tamanho dos programas compilados não influenciaram no executável final em relação ao código objeto da interface gráfica que é maior. O aumento no tamanho do programa executável foi em média 5% como pode ser visto na figura 7.3.

## 7.3 Dificuldades Encontradas

A implementação da otimização *preload* de dados no compilador Xingo foi desafiante em vários aspectos que serão discutidas nesta seção.

O primeiro problema enfrentado foi o porte do LCC, que é o *front-end* do compilador Xingo, para o sistema operacional Windows. O LCC foi compilado com o compilador

[1] *Programa que pega os arquivos objetos e as bibliotecas estáticas e gera o programa executável final*

DJGPP [68] ocorrendo alguns problemas na geração de seu executável (erros de compilação). Após algumas tentativas e modificações em alguns arquivos, obteve-se sucesso na geração deste executável.

Inicialmente a otimização funcionava somente para vetores declarados estaticamente. Isto não gerava resultados satisfatórios para uma gama maior de aplicações, pois a maioria dos programas possuem vetores declarados dinamicamente dependendo de alguma entrada de dados (podendo ser variável).

Quando um vetor é declarado estaticamente, o LCC gera informação deste vetor para o uso da otimização como: variável base do vetor, tamanho do vetor, tipo do vetor, etc. Isso facilita muito a aplicação da otimização para estes casos. Quando um vetor é declarado dinamicamente, o LCC não sabe que aquela variável base atribuída pela função de alocação de memória (*malloc*) faz parte de um vetor, somente que esta é um ponteiro para algum tipo de estrutura de dados.

Então como a otimização procede para otimizar estes casos onde um vetor é alocado dinamicamente? Depois da otimização guardar informações dos usos dos vetores estáticos em um laço, esta começa a procurar pelas instruções com opcode *LOAD* dentro deste laço, guardando estas informações no objeto *PreloadInformation*. Note que esta função pega todas as instruções *LOAD* (mesmo sendo de estruturas de dados). Posteriormente é que a otimização elimina estas *PreloadInformation* de estruturas de dados pela análise do valor calculado de seu *stride*.

Um problema associado à análise de vetores dinâmicos é que a instrução *LOAD* pode não trazer em seus operandos (valor da referência de memória) a variável atribuída pela instrução de alocação de memória. Assim fica mais complicada a análise se dois usos de vetores dentro de um laço são da mesma alocação (mesma área de memória). Nestes casos, geralmente o Xingo faz conversões dessa variável atribuida pela função de alocação de memória para outra variável antes do início do laço, e com a execução de outras otimizações, o compilador Xingo pode usar uma variável diferente para ambos *LOAD*s.

A estratégia adotada neste caso é a de procurar nos operandos das instruções que formam o padrão de acesso das informações de *PreloadInformation*, aquelas que são ponteiros. Achando um ponteiro, verificamos se esta pode ser *alias* de outras variáveis e compara com cada outra variável ponteiro encontrada em outras *PreloadInformation's* para ver se é a mesma variável. Se forem (da mesma alocação), é verificado se a diferença entre os *strides* das duas informações não é maior que 32 bytes, eliminando a que contiver o maior *stride*. Se forem maiores que 32 bytes, a otimização despacha *preloads* para ambas *PreloadInformation's*.

As *PreloadInformation's* de instruções *LOAD's* dentro dos laços investigados que pertencem a estruturas de dados, são eliminadas verificando se na sua lista de instruções que formam a função de acesso a esta referência de memória possui duas instruções *LOAD's*

(referência para referência).

Não foram despachadas *preloads* para instruções *STORE* pois devido a política de escrita na memória usada pelo sistema de memória do XScale não traria benefícios, visto que este usa um buffer de escrita e escreve quando o barramento estiver disponível.

Um aspecto observado nos testes feitos é que quando haviam instruções $i+1$ na função de formação do acesso ao vetor, e quando se fazia o desenrolamento do laço, o compilador *backend* do Xingo colocava as variáveis definidas pelas novas instruções $i+1$, $i+2$, ..., $i+n$, (onde $n$ é o número de vezes que o laço foi desenrolado) todas na memória, fazendo vários *stores e loads* a cada uso das mesmas. Nestes casos, a probabilidade de resultar em uma cache *miss* aumenta bastante dependendo do programa em execução e do fator de desenrolamento usado.

No início foi cogitado em se executar a otimização *Address Optimization*[31] a fim de armazenar as variáveis globais de mesmo tipo e que não seriam vetores, em um vetor único para serem armazenadas em uma área contígua de memória. Isso poderia ser útil para despachar *preloads* para estas variáveis, mas pode haver alguns problemas que provavelmente piorem o desempenho ao invés de melhorar. Um destes problemas é que se estas variáveis globais forem armazenadas em um vetor, este vetor seria de tamanho do número destas usadas no programa, geralmente muito pequeno para despachar *preloads*. Além disto, se a função de acesso a um vetor usar algumas dessas variáveis, teríamos que acessar um vetor através de outro vetor (referência de referência), e assim piorando o desempenho da aplicação, correndo ainda o risco de aumentar o número de instruções (por cálculos de referências ao vetor), e os acessos à memória.

Um outro problema que ocorreria com a aplicação da otimização *Address Optimization* no compilador Xingo, é que este não poderia descobrir a variável de indução que controla um laço se esta estivesse referenciada por uma posição do vetor.

Existem algumas restrições de execução de outras otimizações no compilador Xingo para que a otimização de *preload* de dados funcione corretamente. Não se deve usar a otimização de *Block Merging* e *Branch Elimination* depois de passar pela otimização de *preload* de dados, pois estas colocam o *header* do laço (com as intruções *preload*) no final deste. Isso faz com que a otimização não seja efetiva, pois logo após chamar as instruções *preload*, a execução do programa volta para o início do laço para executar a próxima iteração que usaria os dados que o *preload* acabou de buscar (talvez com isso ultrapassando o *pend buffer* com mais requisições de referências de memória).

Um outro problema encontrado é quando estamos otimizando um laço e temos funções externas sendo executadas no corpo deste laço. Fica complicado neste cenário inferir quanto tempo esta função consome, e conseqüentemente para o cálculo da latência de execução do corpo do laço. Dessa forma, o compilador Xingo disponibiliza uma opção na linha de comando para que se possa escolher se esta otimização será ou não executada

neste caso.

Uma característica importante nesta otimização, é que ela possui um arquivo de configuração do hardware para o qual esta será executada. Assim podemos configurar a otimização para ser executada em outras arquiteturas facilmente, bastando editar este arquivo informando o tamanho da cache, do *pend buffer*, o tamanho da linha da cache, etc.

Um outro problema encontrado, ocorre quando a variável de indução que controla o laço é de um tipo diferente de inteiro. O Xingo neste caso não consegue verificar que esta variável é de indução. Isso ocorre pois o LCC faz conversões de outros tipos para o tipo inteiro. Dessa forma, o algoritmo que descobre a variável de indução que controla o laço não funciona.

Foram corrigidos vários erros no algoritmo de desenrolamento de laços do Xingo, pois dependendo do tipo do laço, a otimização não funcionava corretamente.

A maioria dos estudos de *preload* concentram-se em aplicações numéricas baseadas em vetores. Estes programas tendem a gerar padrões de acesso à memória que, embora previsíveis, produzem uma baixa utilização da cache, podendo então se beneficiar mais do *preload* do que as aplicações gerais. Técnicas automáticas que são efetivas para aplicações gerais têm recebido pouca atenção [43].

Estruturas de dados incluem objetos familiares tais como listas ligadas, árvores, grafos, etc, onde nodos individuais são dinamicamente alocados na memória *heap*, e estes são interligados através de ponteiros. Existe pouca localidade espacial entre nodos acessados consecutivamente em estruturas de dados, uma vez que estes são alocados dinamicamente na memória *heap* e podem possuir endereços arbitrários.

Exemplos de uma árvore e uma lista ligada podem ser vistos nas figuras 7.8 e 7.12 respectivamente.

O endereço de um ponteiro ($*p$) é desconhecido em tempo de compilação (a menos que o valor armazenado nele seja lido). Por isso a otimização visando os casos de estruturas de dados é mais desafiadora do que em vetores. Em geral, analisar os endereços de objetos alocados na memória *heap* é um problema muito difícil para o compilador [29].

Esta otimização de *preload* de dados não despacha instruções *preload* para referências de estruturas de dados. Alguns pesquisadores [29] estão tentando desenvolver heurísticas e algoritmos mais avançados para estes tipos de estruturas que geralmente são usados em aplicações de propósito gerais. Nestes casos, o padrão de acesso a estas estruturas é bem mais complexo, pois envolve referência para referência de endereços da memória (ponteiros) e o *preload* não pode ser aproveitado pela localidade espacial da linha da cache que este traz, pois estas estruturas podem ser armazenadas na memória de forma caótica.

No início deste projeto, tentou-se fazer *preloads* para estruturas, mas como explicado anteriormente, não foi possível continuar pois os resultados não se mostraram satisfatórios.

Uma das principais dificuldades encontradas nestes casos é que quando existem estruturas no corpo dos laços, não se pode desenrolar os mesmos, pois haveria mais referências a serem resolvidas e despachadas pelo *preload*. Para estruturas de dados, deve-se despachar um *preload* para cada referência no uso de uma estrutura. Se esta for maior que 32 bytes (que o *preload* traz da memória), haverá *misses* para o acesso ao restante da estrutura. Por isso não se deve desenrolar o laço, pois as referências aumentarão na mesma proporção do fator de desenrolamento. Sendo assim, pode ser que se fosse despachado um *preload* para uma estrutura de dados em um laço que não fosse desenrolado, o tempo gasto entre o despacho deste *preload* e o uso do mesmo na próxima iteração não fosse o bastante para que este colocasse a estrutura de dados na cache.

Por estas dificuldades com *preload* de dados para estruturas é que não desenvolvemos a otimização visando estes casos. Otimização de *preload* de dados para estruturas de dados pode ser um trabalho futuro a ser desenvolvido.

Além de reduzir o tempo de execução de aplicações, o mecanismo de *preload* tende a aumentar a latência média da busca de dados do sistema de memória, pois isto efetivamente aumenta a taxa de requisições de referências de memória no processador, que de volta, pode introduzir congestionamento dentro do sistema de memória [43].

Quando as fronteiras do intervalo de execução de um laço não são conhecidas em tempo de compilação, esconde-se a predição do volume de dados que está sendo acessado dentro do laço, tornando-se difícil a identificação das referências de memória que são candidatas ao *preload* [35]. Este foi outro problema encontrado, em programas que possuem vetores com volume de dados menor que o tamanho da cache de dados, e estes são constantemente reutilizados em laços aninhados. Neste caso, o despacho da instrução *preload* pode arruinar o desempenho, pois estes vetores poderão sempre estar na cache e haverá o *overhead* da chamada da instrução de *preload*. Além disso, gasta-se uma posição no buffer de pendências, que conseqüentemente pode piorar o desempenho deste programa paralisando várias vezes o processador.

O uso de *preload* pode trazer efeitos secundários se não usado adequadamente, tal como poluição da cache, e o aumento das requisições na largura de banda do sistema de memória [43].

Se um laço possui muitos acessos à memória, pode ser que a otimização não obtenha um resultado satisfatório, pois nestes casos o buffer de pendências é constantemente ultrapassado gerando paradas no processador. Se entre estas pendências houver instruções de *preload*, esta pode não cumprir seu objetivo de chegar a tempo com seus dados, devido ao grande número de requisições à memória.

Não se deve despachar um número maior de *preloads* que o buffer de pendências possui, pois a ultrapassagem do tamanho deste buffer faz com que o processador paralize suas funções até que uma das requisições seja atendida. Por isso o número máximo de despachos

de *preloads* neste trabalho foi de 3, apesar da arquitetura de memória do XScale possuir 4 entradas para este buffer de pendências. Se forem despachados *preloads* e ainda houver pendências não resolvidas neste buffer, os despachos destas podem gerar a paralisação do processador, arruinando assim o desempenho de execução (a cada iteração o processador pára a fim de que seja atendida uma das requisições). O exemplo abaixo descreve um programa que usa dois vetores (A e B) e dados de elementos de uma lista ligada. *Element* é uma estrutura que tem como campos o próximo elemento, e um número. Esta lista é do tamanho dos vetores ($N$). O Código fonte do programa pode ser visto na figura 7.4.

```
1                       int soma = 0;
2                       Element *e = header->begin;
3                       for (int i = 0; i < N; i++)
4                       {
5                           soma += (A[i]* B[i]) + e->number;
6                           e = e->next;
7                       }
```

Figura 7.4: Programa exemplo dos usos do buffer de pendências.

O programa acima tem pelo menos 4 acessos à memória por iteração (um para carregar valor do vetor $A[i]$, outro para o valor do vetor $B[i]$, outro para carregar o valor de *element->number* e um último para *element->next*).

Vamos supor a seguinte situação de despachos dessas requisições de memória, preenchendo o buffer de pendências mostrado na figura 7.5.

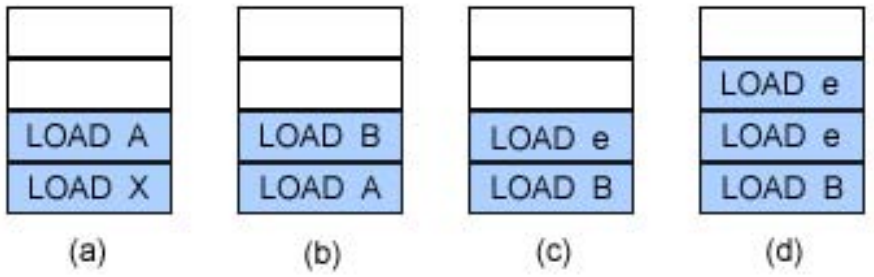



Figura 7.5: Estados do buffer de pendências na execução do programa

Vamos supor que estejamos executando a primeira iteração do laço, e quando carregamos $A[0]$, este não esteja na cache de dados provocando um cache *miss*. O gerenciador de memória reserva uma posição no buffer de preenchimento e uma posição no buffer de pendências para esta requisição. Supondo que o gerenciador de memória já esteja buscando uma requisição feita antes da entrada do laço, a instrução *LOAD A[0]* é enfileirada

no buffer de pendências como mostra a figura 7.5 (a).

Agora temos que carregar o valor de $B[0]$ e como este está sendo usado pela primeira vez, também provocará cache *miss*. Nesta etapa de execução, o gerenciador de memória está acabando de atender a requisição anterior e remove esta requisição do buffer de pendências e insere a requisição *LOAD A[0]*. Esta situação é ilustrada na figura 7.5 (b).

A expressão da estrutura *element->number* também provocará cache *miss* de seu uso, resultando em reservar mais uma posição para esta requisição no buffer de pendências. Nesta etapa de execução, o gerenciador de memória estará acabando de atender a requisição anterior (*LOAD A[0]*), desenfileirando esta do buffer de pendências. Esta situação é mostrada na figura 7.5 (c).

Finalmente, temos um cache *miss* da instrução *element->next*. Como a requisição anterior ainda não foi atendida, esta é enfileirada no buffer de pendências, mesmo sendo uma referência para uma requisição que está no buffer. Esta situação é ilustrada na figura 7.5 (c).

Agora vamos supor que despachemos instruções *preload* para os vetores deste laço. A figura 7.6 ilustra as situações do buffer de pendências neste caso.

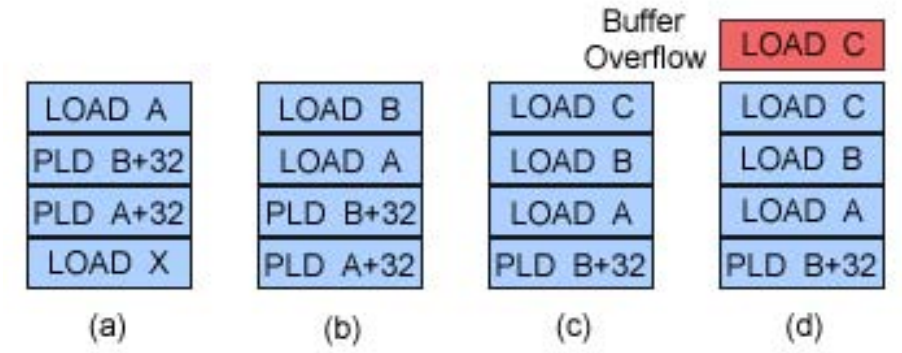



Figura 7.6: Estados do buffer de pendências na execução do programa com a otimização de *preload* de dados

Supondo que as etapas de atendimento às requisições sejam as mesmas que as do exemplo anterior (fig. 7.6), este buffer irá ultrapassar e paralisar o processador, pois com a inclusão das instruções *preloads* neste buffer, aumentará a pressão neste recurso.

Tirando a redução no tempo de execução de aplicações, o mecanismo de *preload* tende a aumentar a latência média de memória por remover ciclos de parada do processador. Isto efetivamente aumenta a taxa de requisições de referências de memória do processador que, de volta, pode introduzir congestionamento dentro do sistema de memória [43].

Por isso o algoritmo para eliminar os *preloads* desnecessários é de extrema importância nesta otimização, pois é este que garante o número de despachos necessários sem ultrapassar o número de entradas do buffer de pendências, além de garantir que não existam despachos para 2 referências de memória que mapeiam para a mesma linha da cache (mesmo sendo para a mesma linha, são necessárias 2 entradas no buffer de pendências).

Instruções *preload* adicionam *overhead* ao processador não somente porque elas requerem ciclos extras de execução, mas também porque o endereço fonte do *preload* deve ser calculado e armazenado no processador [43]. A cada instrução *preload* que for escolhida para ser despachada, existe a inserção de novas instruções que calculam a função de acesso ao vetor no *header* do laço a ser otimizado, diminuindo um pouco o desempenho conquistado com a otimização. Um exemplo de instruções inseridas no header do laço pode ser visto na figura 7.7.

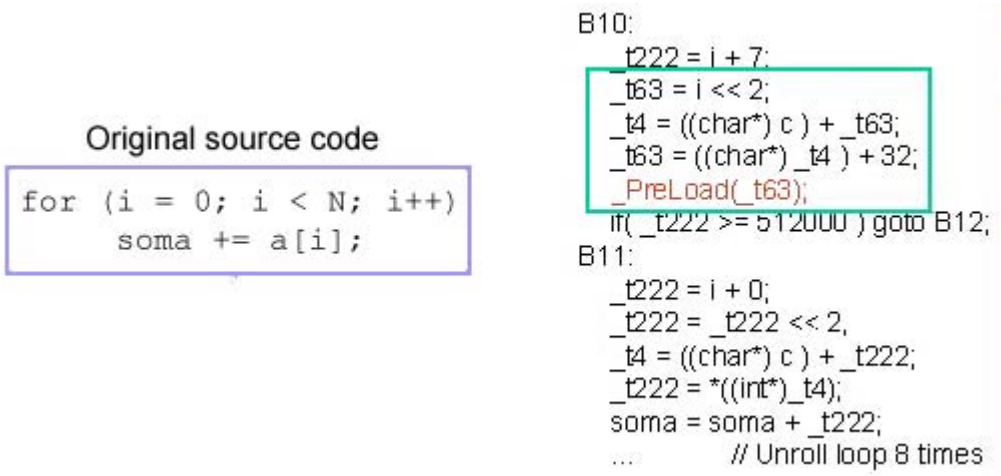



Figura 7.7: Instruções inseridas no *header* do laço exemplo

*Preload* também resulta em uma significante expansão de código pelo desenrolamento do laço, podendo assim degradar o desempenho da cache de instruções [43].

Mecanismos de *preload* também introduzem algum grau de *overhead* de hardware, pois adiciona lógica à cache e requer hardware adicional que é externo à cache. Algumas técnicas requerem que lógicas também sejam adicionadas ao processador. Embora software *preload* possua exigências de hardware mínimas, esta técnica introduz uma significante quantidade de *overhead* de instrução dentro do programa do usuário [43].

O uso desta otimização para aplicações que possuem acessos a vetores com grandes *strides*, ou que tenham uma forma aleatória de acesso não é indicada, pois a otimização não consegue aproveitar a localidade espacial da linha da cache que é trazida a cada

despacho desta instrução.

Não é recomendado o uso desta otimização em aplicações que possuem o corpo do laço muito grande, pois neste caso o fator de desenrolamento tende a ser pequeno, o que gera muitos despachos de instruções *preloads* desnecessárias (despachar *preloads* para a próxima iteração, mas a linha já estaria na cache).

Não compensa aplicar esta otimização para vetores que possuem uma função não linear de acesso, pois o *stride* será variável e de difícil previsão.

Seja o exemplo de uma aplicação que use uma árvore de dados, e esta for representada por um vetor. Vamos supor que esta aplicação precise percorrer esta árvore pela esquerda. A função de acesso para percorrer a árvore até a folha mais a esquerda é $2 * i + 1$ como mostra a figura 7.8

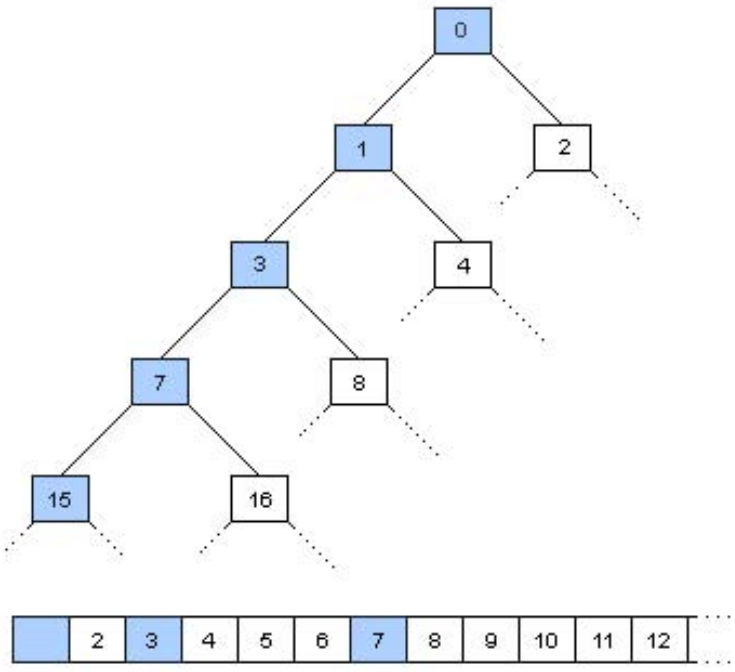



Figura 7.8: Árvore representada por um vetor

Para uma aplicação que use esta árvore, não compensa aplicar a otimização de *preload* de dados, pois apesar desta ter funções de acesso ao vetor bem definidas ($2 * i + 1$ para percorrê-la pela esquerda), não possui acessos com *strides* constantes.

A aplicação desta otimização para matrizes pode gerar bons resultados. Isso dependeria de como esta matriz é acessada no laço a ser otimizado (com localidade espacial ou temporal) e de como esta é armazenada na memória (em área contígua por linhas ou

colunas). Se a matriz é armazenada em área contígua da memória por linhas e o acesso a esta matriz tende a ser por localidade espacial (acesso por colunas), então a otimização poderia ser aplicada garantindo bons resultados. A aplicação para matrizes com acesso com localidade temporal não gera bons resultados, pois o *stride* neste caso seria muito grande (o *stride* seria de fator $N$, onde $N$ é o número de colunas).

Esta otimização pode ter um resultado inesperado se for aplicada em um laço que contenha uma ou mais funções, porque é difícil saber a latência que esta instrução gastará no corpo deste laço. Isto dificulta o cálculo da latência do corpo do laço, interferindo no fator de desenrolamento do mesmo.

Um fato importante a ser levado em consideração é que se a complexidade de um programa for ditada pelo aninhamento de seus laços, quanto maior o aninhamento, melhor será seu desempenho quando passado pela otimização (para vetores grandes). Isto se deve ao fato de que a otimização funciona para todos os acessos feitos, inclusive no reuso deste mesmo vetor (nas outras iterações pelo laço mais externo). Otimizações de memória feitas em laços podem ser vistas em [44][45].

## 7.4 Trabalhos Futuros

Futuras pesquisas podem ser feitas tomando como base este trabalho. Um exemplo é implementar "relocação de dados"(semelhante ao trabalho [47]) para despachar *preloads* para estruturas de dados.

A decisão de despachar a instrução *preload* para determinado vetor ainda poderia ser melhorada, verificando quando possível, o tamanho do vetor e se existe recorrência no seu uso (geralmente acontece em laços aninhados). Se o tamanho do vetor é pequeno (menor que o tamanho da cache de dados) e existe recorrência de seu uso, então o *preload* não deveria ser despachado, pois como este é pequeno, talvez ainda esteja na cache quando for reusado, gerando *overhead* no despacho da instrução e no uso dos buffers de requisição de acesso à memória.

Uma característica que poderia trazer melhores resultados para programas que houvessem variáveis vetores declaradas dinamicamente no programa fonte, seria o de se fazer o alinhamento destas variáveis na fronteira de 32 bytes na memória (na alocação do vetor fazer o compilador iniciar em fronteiras de 32 bytes da memória). Isso faria com que a otimização aproveitasse toda a localidade espacial de uma linha completa da cache.

Um outro ganho de desempenho que poderia ser adquirido é o de implementar o desenrolamento de laços para aqueles que tenham saltos em seu corpo (instruções condicionais). A atual otimização não desenrola laços que possuem saltos, e se isso acontecesse, o resultado poderia ser melhorado.

A forma como um programa é codificado pode interferir nos resultados, principalmente

em se tratando de matrizes. Por exemplo, vamos assumir que devemos ler os valores de uma matriz e somá-los, como mostra o código 7.9:

```
1   int soma = 0;                          int soma = 0;
2   for (i = 0; i < N; i++)                for (i = 0; i < N; i++)
3       for (j = 0; j < N; j++)                for (j = 0; j < N; j++)
4           soma += A[i][j];                       soma += A[j][i];
5           (a)                                    (b)
```

Figura 7.9: Vetor sendo acessado por a)colunas b)linhas no laço mais interno.

Vamos supor que esta matriz seja armazenada por linhas na memória como mostra a figura 7.10.

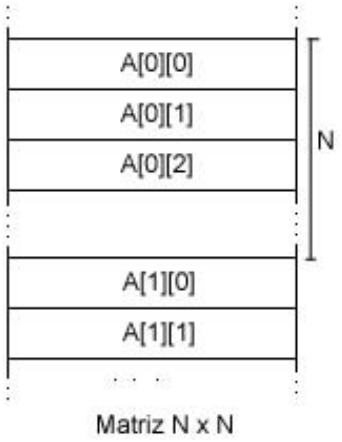



Figura 7.10: Matriz $A[N][N]$ armazenada na memória

Neste caso, a otimização de *preload* de dados seria bem empregada no caso (a), pois poderia haver um despacho da instrução *preload* para esta linha, pois esta é acessada com localidade espacial (pelas colunas).

No exemplo, a matriz se comporta como se tivesse $N$ vetores de mesmo tamanho, e o *preload* funcionaria para estes $N$ vetores, acessados um de cada vez pelo laço mais externo.

Quando o Xingo possuir um gerador de código para a arquitetura XScale, isso dará à otimização uma maior precisão no cálculo das latências das instruções, gerando assim melhores resultados e aumentando a confiabilidade desta otimização. A confiabilidade pode ser aumentada pois a saída que é gerada pela otimização de *preload* no compilador Xingo, pode ser modificada nos outros compiladores que geram o código executável para

```
1   int soma = 0;
2   for (i = 0; i < N; i++)
3   {
4       for (j = 0; j < N; j+=8)
5       {
6           _Preload(&A[i][j+8]);
7           soma += A[i][j];
8           soma += A[i][j+1];
9           soma += A[i][j+2];
10          soma += A[i][j+3];
11          soma += A[i][j+4];
12          soma += A[i][j+5];
13          soma += A[i][j+6];
14          soma += A[i][j+7];
15      }
16      for (; j < N; j+=8)
17      {
18          soma += A[i][j];
19      }
20  }
```

Figura 7.11: Aplicando a otimização para o exemplo (b).

o *Pocket PC*. Como hoje temos outro compilador que gera o código objeto, este pode re-escalonar as instruções geradas pelo Xingo e diminuir sua efetividade no desempenho.

Em aplicações com características de alocação dinâmica de memória, como ponteiros e referências indiretas, existe um número não trivial de cache *misses* causado por referências com *strides* que não são capturados pelo compilador [28].

Aplicações numéricas freqüentemente contêm grandes vetores de dados em laços aninhados. A execução destes vetores em laços de um programa, freqüentemente produz padrões de referências à memória que utilizam ineficientemente a cache de dados [47].

Como também pode ser visto em [26][27][29][47], aplicar *preload* em aplicações de propósito geral é bem mais complexo, pois existe a possibilidade de que estas estruturas de dados sejam alocadas em áreas diferentes da memória (se for por alocação dinâmica na memória *Heap*), não apresentando o princípio da localidade espacial que o *preload* oferece. Além disso, nestes casos o padrão de acessos aos dados possui *strides* variados, impossibilitando a previsão de utilização das próximas estruturas. Sem contar que nestes casos não faz sentido usar o desenrolamento do laço (sem ter a localidade espacial de acesso aos dados).

As aplicações de propósito geral exibem comumente um bom desempenho na cache de dados, e produzem padrões de referências à memória altamente irregulares. Assim a otimização de *preload* de dados perde sua efetividade para estas aplicações[42].

O trabalho [21] mostra um exemplo de *preload* de dados funcionando para estruturas de dados. Neste exemplo ele cria um novo campo (um ponteiro para a n-ésima próxima estrutura) na estrutura de dados ao qual ele quer fazer *preload*. Na alocação destas

estruturas de dados, este ponteiro aponta para a n-ésima próxima estrutura. Quando estas forem usadas em um laço (acessos da lista), a chamada de *preload* é despachada para o endereço deste novo ponteiro criado, tentando buscar os dados da n-ésima próxima estrutura apontada por esta. Esta situação pode ser visualizada na figura 7.12.

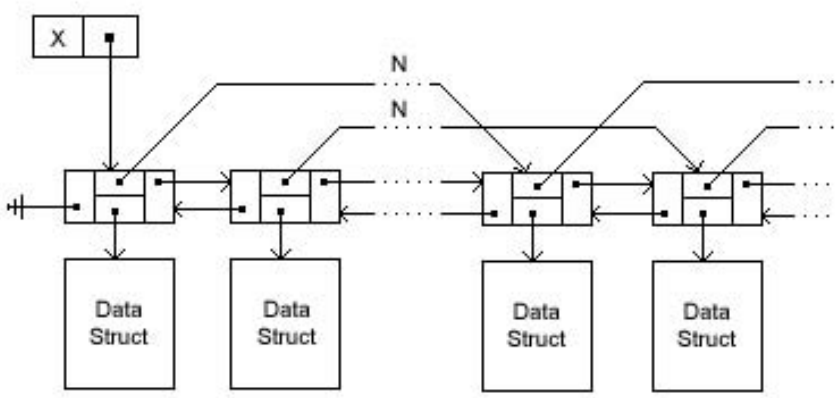



Figura 7.12: Estruturas de dados em uma lista de $X$ elementos. Cada elemento aponta para o N-ésimo próximo elemento

Seja o exemplo do programa da figura 7.13. Este percorre uma lista de estruturas de dados zerando os dados destas estruturas.

```
1   void startup()
2   {
3       Element *element;
4
5       element = head->begin;
6       if (element != NULL)
7       {
8           do
9           {
10              element->struct.data = 0;
11          } while ((element = element->next) != NULL);
12      }
13  }
```

Figura 7.13: Aplicando a otimização para o exemplo (b).

*Element* é a estrutura que tem os campos de apontar para o próximo/anterior elemento e para a estrutura de dados.

Como é mostrado na figura 7.13, se for criado um novo campo (chamado *preload*) na estrutura *Element* que aponta para o N-ésimo elemento à frente da lista, poderíamos aplicar a otimização de *preload* de dados no programa anterior (fig. 7.14).

Isso pode dar tempo para a requisição do *preload* ser atendida pelo gerênciador de memória, pois este terá o tempo de execução de $N$ iterações para trazer os dados da estrutura.

Não foi mostrado neste exemplo, mas a inclusão destas estruturas de dados nesta lista também deve ser modificada, aumentando o tempo de geração desta lista de estruturas de dados.

Implementar isto em um compilador é extremamente complexo, e é difícil ter a confiabilidade na escolha de um fator $N$ ao qual a estutura deverá apontar (a não ser tentando calcular a latência das instruções no corpo do laço). Neste exemplo, o autor deixa em aberto como achar o melhor fator $N$.

Enfim, implementar esta otimização de *preload* de dados foi ao mesmo tempo desafiante e gratificante, pois conseguimos contornar vários obstáculos que foram aparecendo, gerando ótimos resultados para várias aplicações otimizadas.

```
1  void startup()
2  {
3      Element *element;
4
5      element = head->begin;
6      if (element != NULL)
7      {
8          do
9          {
10             Preload(element->preload);
11             element->struct.data = 0;
12         } while ((element = element->next) != NULL);
13     }
14 }
```

Figura 7.14: Aplicando a otimização para o exemplo (b).

# Referências Bibliográficas


[1] Aho, Alfred V. and Sethi, Ravi and Ullman, Jefrey D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] Anaker, W. and Wang, C. P. *Performance Evaluation of Computing Systems with Memory Hierarquies*. IEEE Trans. Comput. 16, 6, 764-773, 1967.

[3] Appel, Andrew W. *Modern Compiler Implementation in Java*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, United Kingdom, 1998.

[4] Appel, Andrew W. and Davidson, J. and Ramsey, N. *The Zephyr Compiler Infrastructure*. Technical report, http://www.cs.virginia.edu/zephyr, University of Virginia, 1998.

[5] Attrot, Wesley. *Xingo - Compilação para uma Representação Intermediária Executável*, dissertação de mestrado, UNICAMP, Abril de 2004.

[6] Bailey, D., Barton, J., Lasinski, T. and Simon, H. *The NAS Parallel Benchmarks*. Technical Report RNR-91-002, NASA Ames Research Center, August 1991.

[7] Bernstein, D., Cohen, D., Freud, A. and Maydan, D. E.*Compiler techniques for data preloading on the PowerPC* in Proc. Of the 1995 Int'l Conference on Parallel Architectures and Compilation Techniques, pp. 19-26, June 1995.

[8] Callahan, D., Kennedy, Ken and Porterfield, Allan, *Software Preloading*, ACM, pp. 40-52, 1991.9

[9] Callahan, David and Kennedy, Ken and Porterfield, Allan.*Analyzing and visualizing performance of memory heirarchies*. In Margaret Simmons and Rebecca Koskela,editors, Instrumentation for Visualization, Frontier Series, pages 1-26. ACM Press, 1990.

[10] Callahan, David and Porterfield, Allan.*Data Cache Performance of Supercomputer Applications*. In Super-computer 90, 1990.

[11] Carro, Luigi and Wagner, Flávio R. *Sistemas Computacionais embarcados*, Anais do XXII Congresso da Sociedade Brasileira de Computação - JAI, Assunto 2, pag 45-94.

[12] Chen, T. F., Bear, J. L.*Effective Hardware-based data prefeching for high performance processors*. IEEE Trans. Computing, 44, 5 (May), 609-623, 1995.

[13] Cohn, R., Goodwin, D. and Lowney, P. G. *Optimizing Alpha executables on Windows NT with Spike*. Digital Technical Journal, 9(4):3-20, 1997.

[14] Dahlgren, F. and Stenstrom, P.*Effectiveness of Hardware-based stride and sequential prefeching in shared memory multiprocessors*. In Proceedings of 1st IEEE Symposium on High-Performance Computer Architecture. IEEE Press, Piscataway, NJ, 68-77, 1995.

[15] Fraser, Christopher W. and Hanson, D.*A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, Menlo Park, California, 1995.

[16] Guthaus, Matthew R. and et al. *MiBench: A free, commercially representative embedded benchmark suite*, The University of Michigan.

[17] Henessy, J. L. and Patterson, D. A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[18] *Intel XScale Microarchitecture*, Programmers Reference Manual, February 2001.

[19] *Intel XScale Core, Developer's Manual*, December 2000.

[20] *Intel PXA250 and PXA210 Optimization Guide, for Application Developers*, December 2002.

[21] *Intel PXA250 and PXA210, Application Processors Optimization Guide, for Application Developers*, February 2003.

[22] *Intel C++ Compiler, for Intel XScale Microarchitecture* - For Platform Builder for Microsoft* Windows* CE .NET and eMbedded* Visual C++, User's Manual, February 2003, Revision 1.2.2

[23] *Intel XScale Core* - Developer's Manual, december 2000.

[24] Kroft, D. *Lockup-free Instruction Fetch/Preload Cache Organization*, International Symposium on Computer Architecture, pp. 81-87, 1981.

[25] Leupers, R. *LANCE: A C Compiler Platform for Embedded DSPs. Embedded Systems/ Embedded Intelligence*, Feb. 2001.

[26] Lipasti, M. H. et al.*SPAID: Software Preloading in Pointer and Call-Intensive Enviroments* Proc. 28th Int'l Symp. Microarchitecture, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 231-236.

[27] Luk, C.K. and Mowry, T. C.*Compiler-Based Preloading for Recursive Data Structures* Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems, ACM Press, New Your, 1996, pp. 222-233.

[28] Luk, Chi-Keung et al. *Profile-Guided Post-Link Stride Preloading*, ACM ICS, 22-26, pp 167-177, June 2002.

[29] Luk, C., Mowry, T. C.*Compiler-Based Preloading for Recursive Data Structures*. Department of Computer Science, University of Toronto, ACM, pp. 222-233, October 1996.

[30] Mowry, Todd C., Lam, Monica S. and Gupta, Anoop, *Design and Evaluation of a Compiler Algorithm for Preload*, ACM, pp. 62-73, 1992.

[31] Muchnick, Steven S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA, 1997.

[32] Paulin, P., Cornero, M., Liem, C. et al. *Trends in Embedded Systems Technology*. In: M.G. Sami, G. De Micheli (eds):

[33] Porterfield, Allan. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, 1989. Technical Report Number Rice COMP TR89-93.

[34] Pegoraro, Renê. *Técnicas de Otimização*, Intel PCA, WCN Unesp - Bauru.

[35] Santhanam, V., Gornish, Edward H. and Hsu, Wei-Chung, *Data Preload on the HP PA-8000*, ACM, pp. 264-273, 1997.

[36] Singh, J. P., Weber, W-D. and Gupta, A. *Splash: Stanford parallel applications for shared memory*. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[37] Smith, A. *Cache Memories*, Computing Surveys, vol. 14, pp. 473-530, September 1982.

[38] Smith, M. D. *Tracing with pixie*. Technical Report CSL-TR-91-497, Stanford University, November 1991.

[39] Tennenhouse, D.*Proactive Computing*, ACM Press, Vol. 43, Nº 5, pp. 43-50, 2000.

[40] *The SPEC Benchmark Report*. Waterside Associates, Fremont, CA, January 1990.

[41] VanderWiel, Steven P.*Masking Memory Access Latency with a Compiler-Assisted Data Preload Controler*, University of Minnesota, Phd thesis, September 1998.

[42] VanderWiel, Steven P. and Lilja, David J, *When Caches Aren't Enough: Data Preload Techniques*, Computer - IEEE, pp. 23-30, July 1997.

[43] VanderWiel, Steven P. and Lilja, David J, *Data Prefetch Mechanisms*, ACM Computing Surveys, Vol. 32, Nº 2, June 2000.

[44] Wolf, M. E. and Lam, Mônica. *A Data Locality Optimizing Problem*, Proceedings of the SIGPLAN - 91 Conference on Programming Language Design and Implementation, pp. 33-44, June 1991.

[45] Wolf, M. E., Maydan, Dror E. and Chen, Ding-Kai. *Combining Loop Transformations Considering Caches and Scheduling*, Computer - IEEE, pp. 274-286, 1996.

[46] Wolfe, M. J. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.

[47] Yamada, Y., Gyllenhall, J., Haab, G. and Hwu, W.*Data Relocation and Preloading for Programs with Large Data Sets*. University of Illinois, ACM, pp. 118-127, November 1994.

[48] Página WinCEBrasil - http://www.wince.com.br/cgi-bin/flashinfo/03.idc?IDFlashInfoDiario=885 (2005-01-31 07:41:32)

[49] http://msdn.microsoft.com/mobility/othertech/eVisualc/default.aspx - Embedded Visual C++ - Microsoft.

[50] http://www.intel.com/software/products/ - Intel Compiler.

[51] http://www.intel.com/ - Intel Portal.

[52] http://www.viewsonic.com/ - Viewsonic palm top

[53] Extraído do Sítio www.ece.eps.hw.ac.uk/.../Performance/sld017.htm

[54] http://gcc.gnu.org/ - GCC Home Page - GNU Project - Free Software Foundation (FSF).

[55] Compaq Computer Corporation. Spike for Thru64 UNIX. http://www.tru64unix.compaq.com/spike

[56] http://www.fortran.com - Fortran Company, acessado em 14/02/2005.

[57] http://www.top500.org/reports/1994/benrep3/node10.html - Top 500 Supercomputer sites, acessado em 14/02/2005.

[58] http://www.netlib.org/benchmark/livermorec - código em C, acessado em 14/02/2005.

[59] http://www.llnl.gov/ - Lawrence Labs.

[60] http://www.ert.rwth-aachen.de/Projekte/Tools/DSPSTONE/dspstone.html - DSP stone site, acessado em 14/02/2005.

[61] http://www.microsoft.com/brasil/pr/2001/pocketpc.htm - Site da Microsoft do Brasil, acessado em 14/02/2005.

[62] Help do Microsoft eMbedded Visual C++ 3.0, feito em 22/03/2000.

[63] http://www.nullstone.com/ - NULLSTONE Automated Compiler Performance Analysis.

[64] http://en.wikipedia.org/wiki/SHA-1 - SHA hash functions, acessado em 22/05/2005.

[65] http://www2.rad.com/networks/1994/err_con/crc.htm - Cyclic Redundancy Check (CRC), acessado em 22/05/2005.

[66] http://www.auditmypc.com/acronym/FFT.asp - Fast Fourier Transform, acessado em 22/05/2005.

[67] http://www.webopedia.com/TERM/A/ADPCM.html - Webopedia, the #1 online encyclopedia dedicated to computer technology, acessado em 22/05/2005.

[68] http://www.delorie.com/djgpp/ - DJGPP sítio, acessado em 18/07/2005.

# Apêndice A

## MeuBench

### Binary Search

```
#include <stdio.h>
#include <malloc.h>

#define N 32000 int *a;

int tempoTotal, loop1, loop2, loop3;

void fillArray();
void orderArray();
void binarySearch(int num);

void fillArray() {
    int i;

    srand( GetTickCount() );
    for(i = 0; i < N ; i++)
        a[i] = rand() % N;
}

void orderArray() {
    int i, j, k, l, menor, aux, pos = 0;

    for(i = 0; i < N ; i++) {
        menor = N;
        for(j = i; j < N ; j++) {
            if (a[j] < menor)
            {
                menor = a[j];
                pos = j;
            }
        }
```

```
        aux = a[i];
        a[i] = menor;
        a[pos] = aux;
    }
}

void binarySearch(int num) {
    int pos, begin, end;

    begin = 0;
    end   = N - 1;
    pos   = (end - begin) / 2;

    while ((a[pos] != num) && ((end - begin) > 1))
    {
        if (num > a[pos])
        {
            begin = pos;
            pos   = (end - begin) / 2 + begin;
        }
        else if (num < a[pos])
        {
            end = pos;
            pos = (end - begin) / 2 + begin;
        }
    }
}

int entryMain()
{
    int inicioTotal, inicio, fim, num;

    a = (int*) malloc(sizeof(int)*N);
    num = N/256;
    inicio = GetTickCount();
    inicioTotal = inicio;
    fillArray();
    loop1 = GetTickCount() - inicio;

    inicio = GetTickCount();
    orderArray();
    loop2 = GetTickCount() - inicio;

    inicio = GetTickCount();
    binarySearch(num);
    loop3 = GetTickCount() - inicio;
```

```
    tempoTotal = GetTickCount() - inicioTotal;
    free(a);
    return 0;
}
```

### Bubble Sort 2

```
#include <stdio.h>
#include <malloc.h>

#define N 32000
#define TRUE  1
#define FALSE 0

int *a;

int tempoTotal, loop1, loop2;

void fillArray();
void orderArray();

void fillArray() {
    int i;

    srand( GetTickCount() );
    for(i = 0; i < N ; i++)
    {
        a[i] = rand() % N;
    }
}

void orderArray() {
    int i, aux;
    char continua = TRUE;

    while(continua)
    {
        continua = FALSE;
        for(i = 0; i < N ; i++)
        {
            if(a[i] > a[i+1])
            {
                aux = a[i];
                a[i] = a[i+1];
```

```
                a[i+1] = aux;
                continua = TRUE;
            }
        }
    }
}

int entryMain() {
    int i, j, maior;
    int inicioTotal, inicio, fim;

    a = (int*) malloc(sizeof(int)*N);

    inicio = GetTickCount();
    inicioTotal = inicio;
    fillArray();
    loop1 = GetTickCount() - inicio;

    inicio = GetTickCount();
    orderArray();
    loop2 = GetTickCount() - inicio;

    tempoTotal = GetTickCount() - inicioTotal;
    return 0;
}
```

### Bubble Sort 1

```
#include <stdio.h>
#include <malloc.h>

#define N 32000 int *a;

int tempoTotal, loop1, loop2; int soma = 0;

int entryMain() {
    int i, j, maior, aux, pos = 0;
    int inicioTotal, inicio, fim;

    a = (int*) malloc(sizeof(int)*N);

    inicio = GetTickCount();
    inicioTotal = inicio;
    srand( GetTickCount() );
    for(i = 0; i < N ; i++)
```

```
        a[j] = rand() % N;
    loop1 = GetTickCount() - inicio;

    inicio = GetTickCount();
    for(i = 0; i < N ; i++) {
        maior = -99999999;
        for(j = i; j < N ; j++) {
            if (a[j] > maior)
            {
                maior = a[j];
                pos = j;
            }
        }
        aux = a[i];
        a[i] = maior;
        a[pos] = aux;
    }
    loop2 = GetTickCount() - inicio;

    tempoTotal = GetTickCount() - inicioTotal;

    return 0;
}
```

### Celebrity

```
#include <stdio.h>
#include <malloc.h>

#define N 1000

#define TRUE  1 #define FALSE 0

int *a[N];

int tempoTotal, loop1, loop2;

void fillMatrix();
void findCelebrity();
void printMatrix();

void fillMatrix() {
    int i, j;

    srand( GetTickCount() );
```

```
    for(i = 0; i < N ; i++)
        for(j = 0; j < N ; j++)
            a[i][j] = rand() % 2;
}

void findCelebrity() {
    int i, j, aux;
    char continua = TRUE;

    j = 0;
    while ((continua)&&(j < N))
    {
        continua = FALSE;
        for(i = 0; i < N ; i++)
        {
            if (j != i)
            {
                if((a[j][i] != 0) || (a[i][j] != 1))
                {
                    continua = TRUE;
                    break;
                }
            }
        }
        j++;
    }
    if (j < N)
        printf ("Celebridade eh o %d\n", j);
}

int entryMain()
{
    int i, j, maior;
    int inicioTotal, inicio, fim;
    int *temp;

    for(i = 0; i < N ; i++)
        a[i] = (int*) malloc(sizeof(int)*N);

    inicio = GetTickCount();
    inicioTotal = inicio;
    fillMatrix();
    loop1 = GetTickCount() - inicio;

    inicio = GetTickCount();
    for (i = 0; i < N; i++)
```

```
    findCelebrity();
    loop2 = GetTickCount() - inicio;

    tempoTotal = GetTickCount() - inicioTotal;

    for(i = 0; i < N ; i++)
    {
        temp = a[i];
        free (temp);
    }

    return 0;
}
```

### High Number

```
#include <stdio.h>
#include <malloc.h>

#define N 128000

int *a;
int tempoTotal, loop1, loop2; int soma = 0;

void fillArray();
void high_number();

void fillArray() {
    int i;

    srand( GetTickCount() );
    for(i = 0; i < N ; i++)
        a[i] = rand() % N;
}

void high_number() {
    int i, high;

    high = -99999999;
    for(i = 0; i < N ; i++)
    {
        if (a[i] > high)
            high = a[i];
    }
}
```

```
int entryMain()
{
    int inicioTotal, inicio, fim;
    int i;

    a = (int*) malloc(sizeof(int)*N);

    inicio = GetTickCount();
    inicioTotal = inicio;
    fillArray();
    loop1 = GetTickCount() - inicio;

    inicio = GetTickCount();
    for (i = 0; i < N*10; i++)
    high_number();
    loop2 = GetTickCount() - inicio;

    tempoTotal = GetTickCount() - inicioTotal;
    free(a);
    return 0;
}
```

### Order Max

```
#include <stdio.h>
#include <malloc.h>

#define N 128000

#define TRUE  1
#define FALSE 0

int *a, *b;

int tempoTotal, loop1, loop2, loop3;

void fillArrayA();
void fillArrayB();
void orderArray();

void fillArrayA() {
    int i;
```

```
    srand( GetTickCount() );
    for(i = 0; i < N ; i++)
        a[i] = 0;
}

void fillArrayB() {
    int i;

    srand( GetTickCount() );
    for(i = 0; i < N ; i++)
        b[i] = rand() % N;
}

void orderArray() {
    int i, aux;
    char continua = TRUE;

    for(i = 0; i < N ; i++)
    {
        a[b[i]]++;
    }
}

int entryMain()
{
    int inicioTotal, inicio, fim, i;

    a = (int*) malloc(sizeof(int)*N);
    b = (int*) malloc(sizeof(int)*N);

    inicio = GetTickCount();
    inicioTotal = inicio;
    fillArrayA();
    loop1 = GetTickCount() - inicio;

    inicio = GetTickCount();
    fillArrayB();
    loop2 = GetTickCount() - inicio;

    inicio = GetTickCount();
    orderArray();
    loop3 = GetTickCount() - inicio;

    tempoTotal = GetTickCount() - inicioTotal;
    free(a);
    free(b);
```

```
    return 0;
}
```

### Profile All

```
#include <stdio.h>

#define N 128000

int loop1, loop2, loop3, loop4, loop5;
int soma = 0;
int *a, b[N];

/*investigar os possiveis casos de preload e verificar resultados
feito na mao
 * 1 - uso de arrays dinamicos em loop sem branchs
 * 2 - uso de arrays dinamicos em loop com branchs
 * 3 - uso de arrays estaticos em loop sem branchs
 * 4 - uso de arrays estaticos em loop com branchs
 * 5 - uso de arrays em loops aninhados
 */

int main () {
    int i, j, k;
    int fim, inicio;
    struct Teste *p;
    int maior = -99999999;

    a = (int*) malloc(sizeof(int) * N);

    for (i = 0; i < N; i++)
        b[i] = a[i] = i;

    inicio = GetTickCount();
    for (i = 0; i < N; i++)
        soma += a[i];
    loop1 = GetTickCount() - inicio;

    inicio = GetTickCount();
    for (i = 0; i < N; i++)
    {
        if (a[i] > N/2)
            soma += a[i];
    }
    loop2 = GetTickCount() - inicio;
```

```
    inicio = GetTickCount();
    for (i = 0; i < N; i++)
        soma += b[i];
    loop3 = GetTickCount() - inicio;

    inicio = GetTickCount();
    for (i = 0; i < N; i++)
    {
        if (b[i] > N/2)
            soma -= b[i];
    }
    loop4 = GetTickCount() - inicio;

    inicio = GetTickCount();
    for (i = 0; i < 1000; i++) // acha o maior valor
    {
        for (j = 0; j < N; j++)
            a[j] = b[j] + soma;
    }
    loop5 = GetTickCount() - inicio;

    return 0;
}
```

### Arrays 1

```
#include <stdio.h>

#define N 512000

int *a, *b;

int tempoTotal, loop1, loop2; int soma = 0;

int entryMain() {
    int j;
    int inicioTotal, inicio, fim;

    a = (int*) malloc(sizeof(int)*N);
    b = (int*) malloc(sizeof(int)*N);

    inicio = GetTickCount();
    inicioTotal = inicio;
    for(j = 1; j < N ; j++) {
```

```
        a[j] = j*2;
        b[j] = j*2;
    }
    fim = GetTickCount();
    loop1 = fim - inicio;

    inicio = GetTickCount();
    for(j = 0; j < N; j++) {
        soma += a[j] * b[j];
    }
    fim = GetTickCount();
    loop2 = fim - inicio;
    tempoTotal = fim - inicioTotal;

    return 0;
}
```

### Arrays 2

```
#define N 20000

int tempoTotal, loop1, loop2, loop3; int soma = 0;
int *a, *b;

int entryMain() {
    int j,i;
    int inicioTotal, inicio, fim;

    a = (int*) malloc(sizeof(int)*N);
    b = (int*) malloc(sizeof(int)*N);

    inicio = GetTickCount();
    inicioTotal = inicio;
    for(j = 0; j < N ; j++) {
        a[j] = j;
        b[j] = j;
    }
    fim = GetTickCount();
    loop1 = fim - inicio;

    inicio = GetTickCount();
    for(j = 0; j < N; j++)
    {
        soma += a[j] + b[j];
    }
```

```
    fim = GetTickCount();
    loop2 = fim - inicio;

    inicio = GetTickCount();

    for(i = 0; i < N/2; i++)
    {
        for(j = 0; j < N; j++)
        {
            a[j] = a[i] * b[j];
            b[j] = 2 * a[j] + b[i];
        }
    }
    fim = GetTickCount();
    loop3 = fim - inicio;

    tempoTotal = fim - inicioTotal;

    return 0;
}
```

### Arrays 3

```
#include <stdio.h>
#include <malloc.h>

#define N 512000

struct Teste {
    int num;
    struct Teste *next;
};

struct Head {
    int num;
    struct Teste *first;
    struct Teste *last;
};

int loop1, loop2, loop3; int soma = 0; struct Head *head; int *a;

void crieList() {
    struct Teste *p, *aux;
    int i;
```

```
    for (i = 0; i < N; i++)
    {
        p = (struct Teste*)malloc(sizeof(struct Teste));
        srand( GetTickCount() );
        p->num = rand() % N;

        if (head->num == 0)
        {
            head->first = head->last = p;
            head->num++;
        }else
        {
            aux = head->last;
            head->last = aux->next = p;
            head->num++;
        }
    }
}

int entryMain () {
    int i;
    int fim, inicio;
    struct Teste *p;

    a = (int*) malloc(sizeof(int) * N);
    head = (struct Head*)malloc(sizeof(struct Head));

    inicio = GetTickCount();
    crieList();
    fim = GetTickCount();
    loop1 = fim - inicio;

    p = head->first;
    inicio = GetTickCount();
    for (i = 0; i < N; i++)
    {
        a[i] = p->num;
        p = p->next;
    }
    fim = GetTickCount();
    loop2 = fim - inicio;

    inicio = GetTickCount();
    for (i = 0; i < N; i++)
        soma += a[i];
```

```
    fim = GetTickCount();
    loop3 = fim - inicio;
    return 0;
}
```

## Livermore Loops

### Loops 1-4

```
#include "livermore.h"

int loop1, loop2, loop3, loop4, time_load;

void startup() {
    int i;

    loop = N;
    n = N/2;
    q = 1;
    r = 2;
    t = 3;
    v = (int*)malloc(sizeof(int)*N);
    x = (int*)malloc(sizeof(int)*N);
    //y = (int*)malloc(sizeof(int)*N);
    //z = (int*)malloc(sizeof(int)*N);
    for (i = 0; i < N; i++)
    {
        x[i] = i;
    //  y[i] = i;
    //  z[i] = i;
        v[i] = i;
    }
}

void garbage() {
    free(v);
    free(x);
    //free(y);
    //free(z);
}

int main() {
    int k , l , ipnt , ipntp , i;
    int lw , j, ii;
    int temp;
```

```
int inicio;

/*
 *   Prologue
 */

inicio = GetTickCount();
startup();
time_load = GetTickCount() - inicio;

/*
 ********************************************************************
 *   Kernel 1 -- hydro fragment
 ********************************************************************
 */

inicio = GetTickCount();
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=0 ; k<n ; k++ ) {
        x[k] = q + y[k]*( r*z[k+10] + t*z[k+11] );
    }
}
loop1 = GetTickCount() - inicio;
/*
 ********************************************************************
 *   Kernel 2 -- ICCG excerpt (Incomplete Cholesky Conjugate Gradient)
 ********************************************************************
 */

inicio = GetTickCount();
for ( l=1 ; l<=loop ; l++ ) {
    ii = n;
    ipntp = 0;
    do {
        ipnt = ipntp;
        ipntp += ii;
        ii /= 2;
        i = ipntp - 1;
        for ( k=ipnt+1 ; k<ipntp ; k=k+2 ) {
            i++;
            x[i] = x[k] - v[k] * x[k-1] - v[k+1] * x[k+1];
            //printf("x[%d] = %d\n", i, x[i]);
        }
    } while ( ii > 0 );
}
loop2 = GetTickCount() - inicio;
```

```
    /*
     ********************************************************************
     *   Kernel 3 -- inner product
     ********************************************************************
     */

    inicio = GetTickCount();
    for ( l=1 ; l<=loop ; l++ ) {
        q = 0;
        for ( k=0 ; k<n ; k++ ) {
            q += z[k]*x[k];
        }
    }
    loop3 = GetTickCount() - inicio;

    /*
     ********************************************************************
     *   Kernel 4 -- banded linear equations
     ********************************************************************
     */

    inicio = GetTickCount();
    m = ( 1001-7 )/2;
    for ( l=1 ; l<=loop ; l++ ) {
        for ( k=6 ; k<N; k=k+m ) {
            lw = k - 6;
            temp = x[k-1];
            for ( j=4 ; j<n ; j=j+5 ) {
                temp -= x[lw]*y[j];
                lw++;
            }
            x[k-1] = y[4]*temp;
        }
    }
    loop4 = GetTickCount() - inicio;

    garbage();

    return 0;
}
```

### Loops 5-8

```
#include "livermore.h"
```

```
int loop1, loop2, loop3, loop4, time_load;

void startup() {
    int i, j;

    loop = N/2;
    n = N-10;
    q = 1;
    r = 2;
    t = 3;
    x = (int*)malloc(sizeof(int)*N);
    y = (int*)malloc(sizeof(int)*N);
    z = (int*)malloc(sizeof(int)*N);
    w = (int*)malloc(sizeof(int)*N);
    u = (int*)malloc(sizeof(int)*(N+6));
    du1 = (int*)malloc(sizeof(int)*N);
    du2 = (int*)malloc(sizeof(int)*N);
    du3 = (int*)malloc(sizeof(int)*N);
    for (i = 0; i < N; i++)
    {
        x[i] = y[i]   = z[i]   = w[i] = i;
        u[i] = du1[i] = du2[i] = du3[i] = i;
        for (j = 0; j < N ; j++)
            b[i][j] = j;
        for (j = 0; j < 5; j++)
        {
            u1[0][i][j] = i;
            u2[0][i][j] = i;
            u3[0][i][j] = i;
            u1[1][i][j] = i;
            u2[1][i][j] = i;
            u3[1][i][j] = i;
        }
    }
}

void garbage() {
    int i;

    free(x);
    free(y);
    free(z);
    free(w);
    free(u);
    free(du1);
```

```
    free(du2);
    free(du3);
}

int main() {

    int k , l , ipnt , ipntp , i;
    int lw , j, ii, nl1, nl2, kx, ky;
    int inicio;

    /*
     *   Prologue
     */

    inicio = GetTickCount();
    startup();
    time_load = GetTickCount() - inicio;

    /*
     ****************************************************************
     *   Kernel 5 -- tri-diagonal elimination, below diagonal
     ****************************************************************
     */

    inicio = GetTickCount();
    for ( l=1 ; l<=loop ; l++ ) {
        for ( i=1 ; i<n ; i++ ) {
            x[i] = z[i]*( y[i] - x[i-1] );
        }
    }
    loop1 = GetTickCount() - inicio;

    /*
     ****************************************************************
     *   Kernel 6 -- general linear recurrence equations
     ****************************************************************
     */

    inicio = GetTickCount();
    for ( l=1 ; l<=loop ; l++ ) {
        for ( i=1 ; i<n ; i++ ) {
            for ( k=0 ; k<i ; k++ ) {
                w[i] += b[k][i] * w[(i-k)-1];
            }
        }
    }
```

```
loop2 = GetTickCount() - inicio;

/*
 ********************************************************************
 *   Kernel 7 -- equation of state fragment
 ********************************************************************
 */

inicio = GetTickCount();
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=0 ; k<n ; k++ ) {
        x[k] = u[k] + r*( z[k] + r*y[k] ) +
               t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
                  t*( u[k+6] + r*( u[k+5] + r*u[k+4] ) ) );
    }
}
loop3 = GetTickCount() - inicio;

/*
 ********************************************************************
 *   Kernel 8 -- ADI integration
 ********************************************************************
 */

inicio = GetTickCount();
for ( l=1 ; l<=loop ; l++ ) {
    nl1 = 0;
    nl2 = 1;
    for ( kx=1 ; kx<3 ; kx++ ){
        for ( ky=1 ; ky<n ; ky++ ) {
           du1[ky] = u1[nl1][ky+1][kx] - u1[nl1][ky-1][kx];
           du2[ky] = u2[nl1][ky+1][kx] - u2[nl1][ky-1][kx];
           du3[ky] = u3[nl1][ky+1][kx] - u3[nl1][ky-1][kx];
           u1[nl2][ky][kx]=
              u1[nl1][ky][kx]+a11*du1[ky]+a12*du2[ky]+a13*du3[ky] + sig*
                 (u1[nl1][ky][kx+1]-2*u1[nl1][ky][kx]+u1[nl1][ky][kx-1]);
           u2[nl2][ky][kx]=
              u2[nl1][ky][kx]+a21*du1[ky]+a22*du2[ky]+a23*du3[ky] + sig*
                 (u2[nl1][ky][kx+1]-2*u2[nl1][ky][kx]+u2[nl1][ky][kx-1]);
           u3[nl2][ky][kx]=
              u3[nl1][ky][kx]+a31*du1[ky]+a32*du2[ky]+a33*du3[ky] + sig*
                 (u3[nl1][ky][kx+1]-2*u3[nl1][ky][kx]+u3[nl1][ky][kx-1]);
        }
    }
}
loop4 = GetTickCount() - inicio;
```

```
    for (i = 0; i < N; i++)
        printf("x[%d] = %d\tw[%d] = %d\tdu1[%d] = %d\n", i, x[i], i, w[i], i, du1[i]);

    garbage();
    return 0;
}
```

### Loops 9-12

```
#include "livermore.h"

int loop1, loop2, loop3, loop4, time_load;

void startup() {
    int i, j;

    loop = N;
    n = N-1;
    dm22 = 1;
    dm23 = 2;
    dm24 = 3;
    dm25 = 4;
    dm26 = 5;
    dm27 = 6;
    dm28 = 7;
    c0 = 8;
    x = (int*)malloc(sizeof(int)*N);
    y = (int*)malloc(sizeof(int)*N);
    for (i = 0; i < N; i++)
    {
        x[i] = N-i;
        y[i] = i;
        px[i] = (int*)malloc(sizeof(int)*25);
        cx[i] = (int*)malloc(sizeof(int)*25);
        for (j = 0; j < 25; j++)
        {
            px[i][j] = j;
            cx[i][j] = j;
        }
    }
}

void garbage() {
    int i;
```

```
    free(x);
    free(y);
    for (i = 0; i < N; i++)
    {
        free(px[i]);
        free(cx[i]);
    }

}

int main() {

    int k, l, i;
    int inicio;

    /*
     *   Prologue
     */

    inicio = GetTickCount();
    startup();
    time_load = GetTickCount() - inicio;

    /*
     ********************************************************************
     *   Kernel 9 -- integrate predictors
     ********************************************************************
     */
    inicio = GetTickCount();
    for ( l=1 ; l<=loop ; l++ ) {
        for ( i=0 ; i<n ; i++ ) {
            px[i][0] = dm28*px[i][12] + dm27*px[i][11] + dm26*px[i][10] +
                       dm25*px[i][ 9] + dm24*px[i][ 8] + dm23*px[i][ 7] +
                       dm22*px[i][ 6] + c0*( px[i][ 4] + px[i][ 5]) + px[i][ 2];
        }
    }
    loop1 = GetTickCount() - inicio;

    /*
     ********************************************************************
     *   Kernel 10 -- difference predictors
     ********************************************************************
     */

    inicio = GetTickCount();
```

```
for ( l=1 ; l<=loop ; l++ ) {
    for ( i=0 ; i<n ; i++ ) {
        ar        =      cx[i][ 4];
        br        = ar - px[i][ 4];
        px[i][ 4] = ar;
        cr        = br - px[i][ 5];
        px[i][ 5] = br;
        ar        = cr - px[i][ 6];
        px[i][ 6] = cr;
        br        = ar - px[i][ 7];
        px[i][ 7] = ar;
        cr        = br - px[i][ 8];
        px[i][ 8] = br;
        ar        = cr - px[i][ 9];
        px[i][ 9] = cr;
        br        = ar - px[i][10];
        px[i][10] = ar;
        cr        = br - px[i][11];
        px[i][11] = br;
        px[i][13] = cr - px[i][12];
        px[i][12] = cr;
    }
}
loop2 = GetTickCount() - inicio;

/*
 ******************************************************************
 *   Kernel 11 -- first sum
 ******************************************************************
 */

inicio = GetTickCount();
for ( l=1 ; l<=loop ; l++ ) {
    x[0] = y[0];
    for ( k=1 ; k<n ; k++ ) {
        x[k] = x[k-1] + y[k];
    }
}
loop3 = GetTickCount() - inicio;

/*
 ******************************************************************
 *   Kernel 12 -- first difference
 ******************************************************************
 */
```

```
    inicio = GetTickCount();
    for ( l=1 ; l<=loop ; l++ ) {
        for ( k=0 ; k<n ; k++ ) {
            x[k] = y[k+1] - y[k];
        }
    }
    loop4 = GetTickCount() - inicio;

    for (i = 0; i < N; i++)
        printf("x[%d] = %d\t",i,  x[i]);

    for (i = 0; i < N; i++)
        for (l = 0; l < 25; l++)
            printf("px[%d][%d] = %d\t", i, l, px[i][l]);

    garbage();
    return 0;
}
```

### Loops 13-16

```
#include <malloc.h>
#include "livermore.h"

int loop1, loop2, loop3, loop4, time_load;

void startup() {
    int i, j;

    loop = N;
    n = N-32;
    vx = (int*)malloc(sizeof(int)*4);
    xx   = (int*)malloc(sizeof(int)*N);
    ix   = (int*)malloc(sizeof(int)*N);
    xi   = (int*)malloc(sizeof(int)*N);
    y    = (int*)malloc(sizeof(int)*N);
    z    = (int*)malloc(sizeof(int)*N);
    e    = (int*)malloc(sizeof(int)*N);
    f    = (int*)malloc(sizeof(int)*N);
    grd  = (int*)malloc(sizeof(int)*N);
    zone = (int*)malloc(sizeof(int)*N);
    plan = (int*)malloc(sizeof(int)*N);
    d    = (int*)malloc(sizeof(int)*N);

    ex   = (int*)malloc(sizeof(int)*N);
```

```
ex1  = (int*)malloc(sizeof(int)*N);
dex  = (int*)malloc(sizeof(int)*N);
dex1 = (int*)malloc(sizeof(int)*N);
ir   = (int*)malloc(sizeof(int)*N);
rx   = (int*)malloc(sizeof(int)*N);
rh   = (int*)malloc(sizeof(int)*2*N);

for (i = 0; i < N; i++)
{
    y[i] = z[i]  =  0;
    e[i] = f[i] = 0;
    grd[i] = zone[i] = plan[i] = d[i] = i;
    ex[i] = dex[i] = dex1[i] = i;
    ir[i] = rx[i] = i;
}

for (i = 0; i < 25; i++)
{
    vy[i] = (int*)malloc(sizeof(int)*N);
    for (j = 0; j < N; j++)
        vy[i][j] = j;
}

for (i = 0; i < 7; i++)
{
    vs[i] =  (int*)malloc(sizeof(int)*N);
    vh[i] =  (int*)malloc(sizeof(int)*N);
    vf[i] =  (int*)malloc(sizeof(int)*N);
    vg[i] =  (int*)malloc(sizeof(int)*N);
    for (j = 0; j < N; j++)
    {
        vs[i][j] = j;
        vy[i][j] = j;
        vh[i][j] = j;
        vf[i][j] = j;
        vg[i][j] = j;
    }
}

for (i = 0; i < N; i++)
{
    p[i][0] = i;
    p[i][1] = i;
    p[i][2] = i;
    p[i][3] = i;
    for (j = 0; j < N; j++)
```

```
        {
            b[i][j] = j;
            c[i][j] = j;
            h[i][j] = j;
        }
    }

}

void garbage() {
    int i;

    free(d);
    free(plan);
    free(zone);
    free(y);
    free(z);
    free(vx);
    free(xx);
    free(ix);
    free(xi);
    free(ex);
    free(ex1);
    free(dex);
    free(dex1);
    free(ir);
    free(rx);
    free(rh);
    for (i = 0; i < N; i++)
      free(p[i]);
    for (i = 0; i < 7; i++)
    {
        free(vs[i]);
        free(vh[i]);
        free(vf[i]);
        free(vg[i]);
    }
}

int main() {

    int k , l , ip, j4, j5;
    int j, ii, lb, k3, k2, m;
    int i1 , j1 , i2 , j2;
    int ng, nz, ar, br, tmp;
    int inicio;
```

```
/*
 *   Prologue
 */

inicio = GetTickCount();
startup();
time_load = GetTickCount() - inicio;

/*
 ********************************************************************
 *   Kernel 13 -- 2-D PIC (Particle In Cell)
 ********************************************************************
 */

inicio = GetTickCount();
for ( l=1 ; l<=loop ; l++ ) {
    for ( ip=0 ; ip<n ; ip++ ) {
        i1 = p[ip][0];
        j1 = p[ip][1];
        i1 &= N-1;
        j1 &= N-1;
        p[ip][2] += b[j1][i1];
        p[ip][3] += c[j1][i1];
        p[ip][0] += p[ip][2];
        p[ip][1] += p[ip][3];
        i2 = p[ip][0];
        j2 = p[ip][1];
        i2 = ( i2 & N-1 ) - 1 ;
        j2 = ( j2 & N-1 ) - 1 ;
        p[ip][0] += y[i2+32];
        p[ip][1] += z[j2+32];
        i2 += e[i2+32];
        j2 += f[j2+32];
        h[j2][i2] += 1;
    }
}
loop1 = GetTickCount() - inicio;

/*
 ********************************************************************
 *   Kernel 14 -- 1-D PIC (Particle In Cell)
 ********************************************************************
 */

inicio = GetTickCount();
```

```
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=0 ; k<n ; k++ ) {
        vx[k] = 0;
        xx[k] = 0;
        ix[k] = grd[k];
        xi[k] = ix[k];
        ex1[k] = ex[ ix[k] - 1 ];
        dex1[k] = dex[ ix[k] - 1 ];
    }
    for ( k=0 ; k<n ; k++ ) {
        vx[k] = vx[k] + ex1[k] + ( xx[k] - xi[k] )*dex1[k];
        xx[k] = xx[k] + vx[k]  + flx;
        ir[k] = xx[k];
        rx[k] = xx[k] - ir[k];
        ir[k] = ( ir[k] & 2048-1 ) + 1;
        xx[k] = rx[k] + ir[k];
    }
    for ( k=0 ; k<n ; k++ ) {
        rh[ ir[k]-1 ] += 1 - rx[k];
        rh[ ir[k]   ] += rx[k];
    }
}
loop2 = GetTickCount() - inicio;

/*
 ******************************************************************
 *   Kernel 15 -- Casual Fortran.  Development version
 ******************************************************************
 */

inicio = GetTickCount();
for ( l=1 ; l<=loop ; l++ ) {
    ng = 7;
    nz = n;
    ar = 1;
    br = 2;
    for ( j=1 ; j<ng ; j++ ) {
        for ( k=1 ; k<nz ; k++ ) {
            if ( (j+1) >= ng ) {
                vy[j][k] = 0;
                continue;
            }
            if ( vh[j+1][k] > vh[j][k] ) {
                t = ar;
            }
            else {
```

```
                t = br;
            }
            if ( vf[j][k] < vf[j][k-1] ) {
                if ( vh[j][k-1] > vh[j+1][k-1] )
                    r = vh[j][k-1];
                else
                    r = vh[j+1][k-1];
                s = vf[j][k-1];
            }
            else {
                if ( vh[j][k] > vh[j+1][k] )
                    r = vh[j][k];
                else
                    r = vh[j+1][k];
                s = vf[j][k];
            }
            vy[j][k] = sqrt( vg[j][k]*vg[j][k] + r*r )* t/s;
            if ( (k+1) >= nz ) {
                vs[j][k] = 0;
                continue;
            }
            if ( vf[j][k] < vf[j-1][k] ) {
                if ( vg[j-1][k] > vg[j-1][k+1] )
                    r = vg[j-1][k];
                else
                    r = vg[j-1][k+1];
                s = vf[j-1][k];
                t = br;
            }
            else {
                if ( vg[j][k] > vg[j][k+1] )
                    r = vg[j][k];
                else
                    r = vg[j][k+1];
                s = vf[j][k];
                t = ar;
            }
            vs[j][k] = sqrt( vh[j][k]*vh[j][k] + r*r )* t / s;
        }
    }
}
loop3 = GetTickCount() - inicio;

/*
 ****************************************************************
 *   Kernel 16 -- Monte Carlo search loop
```

```
 ********************************************************************
 */

inicio = GetTickCount();
ii = n / 3;
lb = ii + ii;
k3 = k2 = 0;
for ( l=1 ; l<=loop ; l++ ) {
    i1 = m = 1;
    label410:
    j2 = ( n + n )*( m - 1 ) + 1;
    printf("j2 = %d\t", j2);
    for ( k=1 ; k<=n ; k++ ) {
        k2++;
        j4 = j2 + k + k;
        j5 = zone[j4-1];
        printf("j5 = %d\t", j5);
        if ( j5 < n ) {
            if ( j5+lb < n ) {
                tmp = plan[j5-1] - t;
            } else {
                if ( j5+ii < n ) {
                    tmp = plan[j5-1] - s;
                } else {
                    tmp = plan[j5-1] - r;
                }
            }
        } else if( j5 == n ) {
            break;
        } else {
            k3++;
            tmp=(d[j5-1]-(d[j5-2]*(t-d[j5-3])*(t-d[j5-3])+(s-d[j5-4])*
                        (s-d[j5-4])+(r-d[j5-5])*(r-d[j5-5])));
        }
        printf("temp = %d\t", tmp);
        if ( tmp < 0 ) {
            if ( zone[j4-2] < 0 )
                continue;
            else if ( !zone[j4-2] )
                break;
        } else if ( tmp ) {
            if ( zone[j4-2] > 0 )
                continue;
            else if ( !zone[j4-2] )
                break;
        } else break;
```

```
            m++;
            if ( m > zone[0] )
                m = 1;
            if ( i1-m )
                goto label410;
            else
                break;
        }
    }
    loop4 = GetTickCount() - inicio;

    garbage();

    printf("tmp = %d\t", tmp);

    for (k = 0; k < N; k++)
        printf("ix[%d] = %d\t", k, ix[k]);
    puts("");
    for (k = 0; k < N; k++)
        printf("ex1[%d] = %d\t", k, ex1[k]);


    for (l = 0; l < 25; l++)
        for (k = 0; k < N; k++)
            printf("vy[%d][%d] = %d\t", l, k, vy[l][k]);
    puts("");
    for (l = 0; l < 7; l++)
        for (k = 0; k < N; k++)
            printf("vs[%d][%d] = %d\t", l, k, vs[l][k]);

    return 0;
}
```

### Loops 17-20

```
#include "livermore.h"

int loop1, loop2, loop3, loop4, time_load;

void startup() {
    int i,j;

    loop = 100;
    n = N-1;
    vstp = (int*)malloc(sizeof(int)*N);
```

```
    vxne = (int*)malloc(sizeof(int)*N);
    vxnd = (int*)malloc(sizeof(int)*N);
    vlin = (int*)malloc(sizeof(int)*N);
    vsp = (int*)malloc(sizeof(int)*N);
    vlr = (int*)malloc(sizeof(int)*N);
    ve3 = (int*)malloc(sizeof(int)*N);
    b5 = (int*)malloc(sizeof(int)*N);
    sa = (int*)malloc(sizeof(int)*N);
    sb = (int*)malloc(sizeof(int)*N);
    xx = (int*)malloc(sizeof(int)*N);
    vx = (int*)malloc(sizeof(int)*N);
    y = (int*)malloc(sizeof(int)*N);
    g = (int*)malloc(sizeof(int)*N);
    z = (int*)malloc(sizeof(int)*N);
    v = (int*)malloc(sizeof(int)*N);
    w = (int*)malloc(sizeof(int)*N);
    x = (int*)malloc(sizeof(int)*N);
    u = (int*)malloc(sizeof(int)*N);

    for (i = 0; i < N; i++)
    {
        vsp[i] = vstp[i] = vxne[i] = vxnd[i] = b5[i] = i+1;
        sa[i] = sb[i] = y[i] = g[i] = z[i] = xx[i] = i+1;
        v[i] = w[i] = x[i] = vx[i] = u[i] = i+1;
        for (j = 0; j < 8; j++)
        {
            za[j][i] = zp[j][i] = zq[j][i] = zr[j][i] = j+1;
            zm[j][i] = zb[j][i] = zu[j][i] = zv[j][i] = j+1;
            zz[j][i] = j+1;
        }
    }
}

void garbage() {

    free(vstp);
    free(vxne);
    free(vxnd);
    free(vlin);
    free(vlr);
    free(vsp);
    free(ve3);
    free(b5);
    free(sa);
    free(sb);
    free(xx);
```

```
    free(vx);
    free(y);
    free(g);
    free(z);
    free(v);
    free(w);
    free(x);
    free(u);
}

int main() {

    int k , l , i, kb5i;
    int j, ink, kn, jn, stb5;
    int scale, xnm, xnc, xnei, e6, t, s, di, dn, dk, e3;
    int inicio;

    /*
     *   Prologue
     */

    inicio = GetTickCount();
    startup();
    time_load = GetTickCount() - inicio;

    /*
     ********************************************************************
     *   Kernel 17 -- implicit, conditional computation
     ********************************************************************
     */

    inicio = GetTickCount();
    for ( l=1 ; l<=loop ; l++ ) {
        i = n-1;
        j = 0;
        ink = -1;
        scale = 5/ 3;
        xnm = 1 / 3;
        e6 = 1 / 3;
        goto l61;
l60:    e6 = xnm*vsp[i] + vstp[i];
        vxne[i] = e6;
        xnm = e6;
        ve3[i] = e6;
        i += ink;
        if ( i==j ) goto l62;
```

```
l61:    e3 = xnm*vlr[i] + vlin[i];
        xnei = vxne[i];
        vxnd[i] = e6;
        xnc = scale*e3;
        if ( xnm > xnc ) goto l60;
        if ( xnei > xnc ) goto l60;
        ve3[i] = e3;
        e6 = e3 + e3 - xnm;
        vxne[i] = e3 + e3 - xnei;
        xnm = e6;
        i += ink;
        if ( i != j ) goto l61;
l62:;
    }
    loop1 = GetTickCount() - inicio;


    /*
     ********************************************************************
     *   Kernel 18 - 2-D explicit hydrodynamics fragment
     ********************************************************************
     */

    inicio = GetTickCount();
    for ( l=1 ; l<=loop ; l++ ) {
        t = 1;
        s = 2;
        kn = 6;
        jn = n;
        for ( k=1 ; k<kn ; k++ ) {
          for ( j=1 ; j<jn ; j++ ) {
              za[k][j] = ( zp[k+1][j-1] +zq[k+1][j-1] -zp[k][j-1] -zq[k][j-1] )*
                         ( zr[k][j] +zr[k][j-1] ) / ( zm[k][j-1] +zm[k+1][j-1]);
              zb[k][j] = ( zp[k][j-1] +zq[k][j-1] -zp[k][j] -zq[k][j] ) *
                         ( zr[k][j] +zr[k-1][j] ) / ( zm[k][j] +zm[k][j-1]);
          }
        }
        for ( k=1 ; k<kn ; k++ ) {
            for ( j=1 ; j<jn ; j++ ) {
                zu[k][j] += s*( za[k][j]   *( zz[k][j] - zz[k][j+1] ) -
                                za[k][j-1] *( zz[k][j] - zz[k][j-1] ) -
                                zb[k][j]   *( zz[k][j] - zz[k-1][j] ) +
                                zb[k+1][j] *( zz[k][j] - zz[k+1][j] ) );
                zv[k][j] += s*( za[k][j]   *( zr[k][j] - zr[k][j+1] ) -
                                za[k][j-1] *( zr[k][j] - zr[k][j-1] ) -
                                zb[k][j]   *( zr[k][j] - zr[k-1][j] ) +
```

```
                            zb[k+1][j] *( zr[k][j] - zr[k+1][j] ) );
        }
    }
    for ( k=1 ; k<kn ; k++ ) {
        for ( j=1 ; j<jn ; j++ ) {
            zr[k][j] = zr[k][j] + t*zu[k][j];
            zz[k][j] = zz[k][j] + t*zv[k][j];
        }
    }
}
loop2 = GetTickCount() - inicio;

/*
 ******************************************************************
 *   Kernel 19 -- general linear recurrence equations
 ******************************************************************
 */

inicio = GetTickCount();
kb5i = 0;
stb5 = 1;
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=0 ; k<n ; k++ ) {
        b5[k+kb5i] = sa[k] + stb5*sb[k];
        stb5 = b5[k+kb5i] - stb5;
    }
    for ( i=1 ; i<=n ; i++ ) {
        k = n - i ;
        b5[k+kb5i] = sa[k] + stb5*sb[k];
        stb5 = b5[k+kb5i] - stb5;
    }
}
loop3 = GetTickCount() - inicio;

/*
 ******************************************************************
 *   Kernel 20 -- Discrete ordinates transport, conditional recurrence on xx
 ******************************************************************
 */

inicio = GetTickCount();
dk = 1;
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=0 ; k<n ; k++ ) {
        di = y[k] - g[k] / ( xx[k] + dk );
        dn = 2;
```

```
        if ( di ) {
            dn = z[k]/di ;
            if ( t < dn ) dn = t;
            if ( s > dn ) dn = s;
        }
        x[k] = ( ( w[k] + v[k]*dn )* xx[k] + u[k] ) / ( vx[k] + v[k]*dn );
        xx[k+1] = ( x[k] - xx[k] )* dn + xx[k];
    }
}
loop4 = GetTickCount() - inicio;

garbage();

for (l = 0; l < N; l++)
    printf("vxne[%d] = %d\t", l, vxne[l]);
puts("");
for (l = 0; l < N; l++)
    printf("ve3[%d] = %d\t", l, ve3[l]);
puts("");
for (l = 0; l < N; l++)
    printf("vxnd[%d] = %d\t", l, vxnd[l]);


for (l = 0; l < 8; l++)
    printf("b5[%d] = %d\t", l, b5[l]);
puts("");


for (l = 0; l < N; l++)
    printf("x[%d] = %d\t", l, x[l]);
puts("");
for (l = 0; l < N; l++)
    printf("xx[%d] = %d\t", l, xx[l]);
puts("");
*/

for (l = 0; l < 8; l++)
    for (k = 0; k < N; k++)
        printf("za[%d][%d] = %d\t", l, k, za[l][k]);
puts("");
for (l = 0; l < 8; l++)
    for (k = 0; k < N; k++)
        printf("zb[%d][%d] = %d\t", l, k, za[l][k]);
puts("");
for (l = 0; l < 8; l++)
    for (k = 0; k < N; k++)
```

```
            printf("zu[%d][%d] = %d\t", l, k, zu[l][k]);
    puts("");
    for (l = 0; l < 8; l++)
        for (k = 0; k < N; k++)
            printf("zv[%d][%d] = %d\t", l, k, zv[l][k]);
    puts("");
    for (l = 0; l < 8; l++)
        for (k = 0; k < N; k++)
            printf("zr[%d][%d] = %d\t", l, k, zr[l][k]);
    puts("");
    for (l = 0; l < 8; l++)
        for (k = 0; k < N; k++)
            printf("zz[%d][%d] = %d\t", l, k, zz[l][k]);
    puts("");

    return 0;
}
```

### Loops 21-24

```
#include "livermore.h"

int loop1, loop2, loop3, loop4, time_load;

void startup() {
    int i, j;

    n = N-1;
    loop = 100;

    w = (int*)malloc(sizeof(int)*N);
    u = (int*)malloc(sizeof(int)*N);
    v = (int*)malloc(sizeof(int)*N);
    x = (int*)malloc(sizeof(int)*N);
    y = (int*)malloc(sizeof(int)*N);
    for (i = 0; i < N; i++)
        x[i] = y[i] = v[i] = u[i] = w[i] = i+1;

    for (j = 0; j < 25; j++)
    {
        vy[j] = (int*)malloc(sizeof(int)*N);
        for (i = 0; i < N; i++)
            vy[j][i] = j+1;
    }
    for (j = 0; j < N; j++)
```

```
    {
        px[j] = (int*)malloc(sizeof(int)*N);
        cx[j] = (int*)malloc(sizeof(int)*N);
        for (i = 0; i < N; i++)
            px[j][i] = cx[j][i] = j+1;
    }
    for (j = 0; j < 7; j++)
    {
        za[j] = (int*)malloc(sizeof(int)*N);
        zp[j] = (int*)malloc(sizeof(int)*N);
        zq[j] = (int*)malloc(sizeof(int)*N);
        zr[j] = (int*)malloc(sizeof(int)*N);
        zm[j] = (int*)malloc(sizeof(int)*N);
        zb[j] = (int*)malloc(sizeof(int)*N);
        zu[j] = (int*)malloc(sizeof(int)*N);
        zv[j] = (int*)malloc(sizeof(int)*N);
        zz[j] = (int*)malloc(sizeof(int)*N);
        for (i = 0; i < N; i++)
        {
            za[j][i] = zp[j][i] = zq[j][i] = zr[j][i] = zm[j][i] = j+1;
            zb[j][i] = zu[j][i] = zv[j][i] = zz[j][i] = j+1;
        }
    }

}

void garbage() {
    int j;

    free(w);
    free(u);
    free(v);
    free(x);
    free(y);

    for (j = 0; j < 25; j++)
        free(vy[j]);
    for (j = 0; j < N; j++)
    {
        free(px[j]);
        free(cx[j]);
    }
    for (j = 0; j < 7; j++)
    {
        free(za[j]);
        free(zp[j]);
```

```
        free(zq[j]);
        free(zr[j]);
        free(zm[j]);
        free(zb[j]);
        free(zu[j]);
        free(zv[j]);
        free(zz[j]);
    }
}

int main() {

    int k , l , i, j;
    int expmax, qa, m;
    int inicio;

    /*
     *   Prologue
     */

    inicio = GetTickCount();
    startup();
    time_load = GetTickCount() - inicio;

    /*
     ********************************************************************
     *   Kernel 21 -- matrix*matrix product
     ********************************************************************
     */

    inicio = GetTickCount();
    for ( l=1 ; l<=loop ; l++ ) {
        for ( k=0 ; k<25 ; k++ ) {
            for ( i=0 ; i<25 ; i++ ) {
                for ( j=0 ; j<n ; j++ ) {
                    px[j][i] += vy[k][i] * cx[j][k];
                }
            }
        }
    }
    loop1 = GetTickCount() - inicio;

    /*
     ********************************************************************
     *   Kernel 22 -- Planckian distribution
     ********************************************************************
```

```
 */

inicio = GetTickCount();
expmax = 20;
u[n-1] = 1 * expmax * v[n-1];
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=0 ; k<n ; k++ ) {
        y[k] = u[k] / v[k];
        w[k] = x[k] / ( exp( y[k] ) -1);
    }
}
loop2 = GetTickCount() - inicio;

/*
 ********************************************************************
 *   Kernel 23 -- 2-D implicit hydrodynamics fragment
 ********************************************************************
 */
// esta dando erro no Xingo por causa do ponto flutuante abaixo

inicio = GetTickCount();
for ( l=1 ; l<=loop ; l++ ) {
    for ( j=1 ; j<6 ; j++ ) {
        for ( k=1 ; k<n ; k++ ) {
            qa = za[j+1][k]*zr[j][k] + za[j-1][k]*zb[j][k] +
                 za[j][k+1]*zu[j][k] + za[j][k-1]*zv[j][k] + zz[j][k];
            //za[j][k] += 0.175 *( qa - za[j][k] );
            za[j][k] += 1 *( qa - za[j][k] );
        }
    }
}
loop3 = GetTickCount() - inicio;

/*
 ********************************************************************
 *   Kernel 24 -- find location of first minimum in array
 ********************************************************************
 */

inicio = GetTickCount();
x[n/2] = -1;
for ( l=1 ; l<=loop ; l++ ) {
    m = 0;
    for ( k=1 ; k<n ; k++ ) {
        if ( x[k] < x[m] ) m = k;
    }
```

```
    }
    loop4 = GetTickCount() - inicio;

    garbage();

    for ( j=0 ; j<N ; j++ )
        for ( l=0 ; l<N ; l++ )
            printf("px[%d][%d] = %d\t", j, l, px[j][l]);
    puts("");

    for ( l=0 ; l<N ; l++ )
        printf("y[%d] = %d\t", l, y[l]);
    puts("");
    for ( l=0 ; l<N ; l++ )
        printf("w[%d] = %d\t", l, w[l]);
    puts("");

    for ( j=0 ; j<7 ; j++ )
        for ( l=0 ; l<N ; l++ )
            printf("za[%d][%d] = %d\t", j, l, za[j][l]);

    printf("m = %d\n", m);

    return 0;
}
```