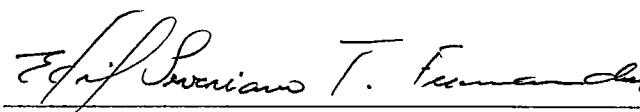
EXPLORAÇÃO DO PARALELISMO DE BAIXO NÍVEL EM MÁQUINAS DO TIPO VLIW

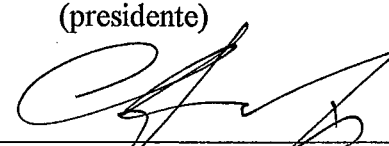
Delfim Xavier Martins

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO

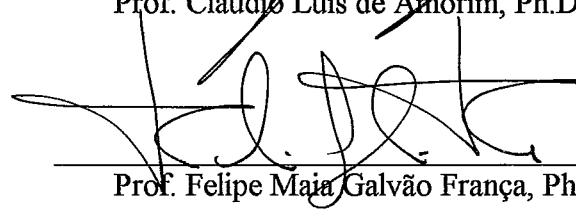
Aprovada por



Prof. Edil Severiano Tavares Fernandes, Ph.D.
(presidente)



Prof. Cláudio Luis de Amorim, Ph.D.



Prof. Felipe Maia Galvão França, Ph.D

Prof. Siang Wun Song, Ph.D

RIO DE JANEIRO, RJ - BRASIL
ABRIL DE 1995

Sumário

Como conseqüência dos recentes avanços tecnológicos tornou-se possível desenvolver arquiteturas de computadores incluindo diversas unidades funcionais que podem ser programadas para operar em paralelo. Estas máquinas são conhecidas como Super Escalares e representam um membro importante dos processadores paralelo: arquiteturas incorporando paralelismo de baixo nível. O maior problema relacionado com esta classe de arquitetura está na geração de código que explore seu paralelismo potencial.

Neste trabalho, descrevemos alguns experimentos com a geração de código paralelo orientado para máquinas Super Escalares. Estes experimentos empregam técnicas tradicionais de extração de concorrência que foram adaptadas às máquinas alvo com concorrência de baixo nível.

Abstract

As a consequence of recent technological achievements, it is now possible to develop Comput Architectures including several independent functional units that can be programmed to operate in parallel. These machines are the so-called, Super-Scalar machines, and they represent an important member of the parallel processors: architectures incorporating low level parallelism. The main problem regarding this class of architectures is how the machine code should be generated in order to exploit their potencial parallelism.

In this work we describe some experiments leading to the generation of parallel code oriented for Super-Scalar machines. The experiments have been carried out with tradicional concurrency extraction techniques that have been modified in order to satisfy the requirements of some target machines with low level concurrency.

Índice

Capítulo I Introdução

Diversas técnicas para melhorar o desempenho de computadores, baseadas tanto em *software* como em *hardware*, têm sido propostas e implementadas ao longo dos anos. Uma das alternativas mais promissoras, utilizadas por muitas destas técnicas consiste na exploração de paralelismo [AIKE88], [FISH81], [LAND80], [LINN88], [NICO85], [NICO90], [TOKO81].

A extração de paralelismo pode se dar em diferente níveis de granularidade e em diferentes fases do desenvolvimento e execução de programas. Por exemplo as máquinas microprogramadas podem ser projetadas segundo duas organizações básicas: a organização vertical, que oferece como vantagem a economia de recursos de *hardware* e a organização horizontal tem como principal atrativo a execução simultânea de várias micro-operações.

O processo de empacotamento das micro-operações de um microprograma em microinstruções de forma que o tempo de execução seja minimizado e que a sequência resultante de microinstruções seja semanticamente equivalente à sequência original denomina-se compactação.

A exploração deste paralelismo potencial não é uma tarefa simples. As primeiras contribuições nesta direção, tinham como objetivo otimizar blocos básicos. A compactação de blocos básicos é conhecida como Compactação Local. Embora apresente bons resultados, a compactação local não é suficiente, uma vez que os blocos básicos são normalmente muito pequenos.

Existem várias possibilidades para a movimentação de operações entre blocos. Os métodos de compactação que consideram o fluxo de controle geral do programa denominam-se de Compactação Global.

Com os avanços tecnológicos tornou-se viável desenvolver processadores, chamados de Super Escalares que incorporam diversas unidades funcionais e vias de dados independentes, que são capazes de executar diversas instruções em paralelo.

Com o objetivo de manter ocupadas as múltiplas unidades funcionais, explorando assim mais eficientemente o paralelismo de baixo nível provido pelas máquinas Super Escalares partimos para a utilização de uma técnica de compactação global, o *percolation scheduling* ao inves de uma técnica de compactação local.

Nossos experimentos foram conduzidos numa máquina Super Escalar do tipo VLIW cujo número de unidades funcionais, largura das instruções de máquina, tempo de latência e outros dados arquiteturais são complemetamente parametrizáveis. Sendo a arquitetura utilizada baseia-se no processador i860, isto é , utilizamos o repertório de instruções, tipos de unidades funcionais e tempo de latência das operações do processador i860 como base para o nossos teste. A utilização do i860 como base deve-se ao fato deste processador ter sido utilizado na máquina de alto desempenho desenvolvido pelo grupo da COPPE.

Este trabalho está organizado em cinco capítulos. O capítulo 2 descreve as técnicas de extração de paralelismo. A descrição da arquitetura utilizada e do processo de compactação encontram-se no capítulo 3. O capítulo 4 apresenta os experimentos práticos executados juntamente com os resultados obtidos. O estágio atual do trabalho, as linhas de pesquisa para o futuro e as conclusões são apresentadas no capítulo 5.

Compactação Local

Os primeiros trabalhos propostos para a extração de paralelismo tratavam o problema a nível de microcódigo e consideravam apenas otimização dos trechos de microprogramas delimitados pelos blocos básicos.

Caminho Crítico

Este algoritmo trata o grafo de dependências de dados como um grafo PERT, no qual para cada aresta (v,w) associa-se um peso que corresponde a duracão da micro-operacão. O algortimo encontra o caminho crítico do grafo PERT e os tempos mais cedo (ET) e mais tarde (LT) nos quais cada micro-operacão pode ser executada.

Na primeira fase as micro-operacões críticas, isto é, aquelas com ET = LT, são alocadas em quadros de forma que as micro-operacões com ET = LT = **I** sejam posicionadas no quadro **i**. Esta alocação inicial não considera possíveis conflitos entre micro-operacões em um mesmo quadro. Na fase seguinte, os quadros são particionados em tantos quanto forem necessários para a resolução destes conflitos.

Na fase final, procura-se alocar as micro-operacões restantes (não críticas) em quadros cujo número seja maior que ET e menor que LT e que possam comportar a micro-operação sem gerar conflitos de recursos. Caso a alocação não seja possível, um quadro adicional é criado e inserido observando a posição correta entre os quadros que garanta os tempos mais cedo e mais tarde para execução da micro-operação.

Branch and Bound

Basicamente, este algoritmo prevê a construção de uma árvore que representa todas as possíveis ordenações das microinstruções de um microprograma. A exploração completa desta árvore conduziria à ordenação ótima porém apresentaria complexidade exponencial. Outra alternativa é a utilização de uma heurística para "podar" a árvore. O sucesso do algoritmo fica então dependente da heurística utilizada. Vários experimentos foram feitos na intenção de avaliar diversas heurísticas para o algoritmo.

List Scheduling

O algoritmo de *List Scheduling* pode ser considerado um caso particular do algoritmo Branch and Bound, no qual a heurística utilizada consiste em seguir sempre aquele que parece ser o melhor caminho a partir de um ponto, transformando a árvore em apenas um ramo (uma lista). A avaliação do "melhor caminho" depende de uma métrica a ser escolhida, e que afeta o resultado da compactação. Um exemplo de medida sugerida por Wood[WOOD78] associa um peso a cada micro-operação que corresponde ao número de descendentes que esta possui no grafo de dependências.

Os algoritmos descritos acima produziram bons resultados, mas ainda restaria muito a ser explorado se fosse possível fazer uma compactação além dos limites dos blocos básicos. Os microprogramas normalmente possuem muitas micro-operações de desvio, resultando em blocos de tamanho bastantes reduzido. Se os blocos são compactados separadamente, muitas microinstruções deixam de utilizar todos os recursos disponíveis, deixando espaços vazios para micro-operações que o processo de otimização não foi capaz de aproveitar. O paralelismo obtido através da Compactação Local não é muito satisfatório, embora seja fundamental para qualquer outro método de compactação.

Como decorrência das limitações impostas pela Compactação Local, vários estudos foram feitos na tentativa de se obter uma Compactação Global de microprogramas. Na compactação Global não existem fronteiras para a movimentação das micro-operações desde que a equivalência semântica do microprograma seja preservada.

Alguns trabalhos com esse objetivo podem ser citados como pioneiros. Dentre eles destacamos a técnica chamada MORIF, proposta por Tokoro, Tamura e Takizuka em [TOKO81]; a técnica de *Trace Scheduling* proposta por Fisher em [FISH81] e a técnica de *Percolation Scheduling* proposta por Nicolau [NICO85], [AIKE88] e utilizada no desenvolvimento deste trabalho. Estas técnicas são descritas a seguir.

Trace Scheduling

A técnica *trace scheduling*, desenvolvida em 1981 por Fisher[], destinava-se originalmente à compactação global de micro-código. Motivado pelos trabalhos de outros pesquisadores Fisher concluiu que a compactação de blocos básicos sem levar em consideração as potencialidades dos blocos vizinhos não era suficiente.

O algoritmo do *Trace Scheduling* faz o escalonamento das instruções ao longo do fluxo de controle do programa como um todo, i.e, a técnica explora o paralelismo em trechos (traces) de execução do programa. O *Trace scheduling* opera em trilhas ao invés de blocos básicos. Uma trilha é uma sequência de instruções sem ciclos, que pode ser executada contiguamente para algum(ns) dado(s) de entrada.

O algoritmo inclui quatro fases:

1. Escolha dos trechos.
2. Pré-processamento do trecho.
3. Compactação do trecho.
4. Pós-processamento do trecho.

1. Escolha dos Trechos

Usando métodos heurísticos, o algoritmo seleciona o trecho do programa que será executado mais freqüentemente. As instruções de desvio condicional do trecho

selecionado são tratadas da mesma forma como as instruções dos outros tipos. A probabilidade de execução de cada trecho do programa desempenha um importante papel na qualidade do código gerado.

Nem sempre o método heurístico pode determinar automaticamente a frequência de execução de cada *trace*. Por esse motivo, em alguma situações, o programador pode interferir no processo de compactação. Alternativamente, o programa pode ser instrumentado, e em seguida executado seqüencialmente usando um conjunto de dados representativo como entrada. Os resultados produzidos durante a execução seqüencial monitorada serão utilizados como parâmetros pelo algoritmo *trace scheduling*.

Ao compactar longos trechos de programa com elevada probabilidade de execução, uma taxa maior de concorrência pode ser detectada e explorada pelo algortimo.

2. Pré-processamento do Trecho

Ao invés de compactar prontamente o trecho selecionado, o algoritmo realiza um pré-processamento no *trace*. Esse tratamento impede que movimentações errôneas sejam feitas durante o escalonamento. Tendo em vista que comandos serão movimentados ao longo dos traces de programa sem respeitar os limites definidos pelos blocos básicos, torna-se necessário incluir novas arestas no grafo de dependência de dados. Essas arestas conectarão blocos contendo uma instrução de desvio condicional aos blocos com instruções que neutralizam o efeito da execução antecipada e indevida de comandos pertencentes ao outro ramo do desvio (com maior probabilidade de execução). Uma descrição mais detalhada desses comandos, denominados código de reparo encontra-se em [FISH83] e [FISH84].

3. Escalonamento das Instruções do Trecho

Após a fase de pré-processamento, o escalonador inicia o processo de compactação do *trace* corrente. O algoritmo *list scheduling* [LAND80] é usado para executar esta tarefa. Embora tenha sido projetado originalmente para compactar comandos no interior de blocos básicos, o algoritmo *list scheduling* foi modificado para aceitar como entrada um conjunto de blocos básicos, i.e., um *trace* do programa de aplicação. Por esse motivo, durante a compactação, as fronteiras delimitadas pelos blocos básicos não impõem restrições às movimentações de comandos realizadas por esta versão do algoritmo *list scheduling*.

4. Pós-processamento do Trecho

Uma vez concluído o escalonamento do *trace*, é necessário corrigir o efeito provocado pela movimentação de alguns comandos. A fase de pós-processamento cria alguns blocos básicos para essa finalidade. Esses blocos atuarão como interface entre os pontos de entrada ou de saída (blocos R e S respectivamente) do trecho compactado e o restante do programa. Em outras palavras, usando esses blocos podemos conectar o *trace* (que foi isolado e compactado) ao restante do programa.

Os comandos no interior dos blocos R e S garantirão a equivalência semântica

do *trace* após a compactação, permitindo assim que ele seja incorporado ao restante do programa.

O algoritmo *trace scheduling* explora o paralelismo de laços (loops), " desenrolando-os" em diversas iterações. Nesse caso, os teste realizados na variável de controle do comando iterativo são transformados em desvios condicionais. O conjunto de blocos básicos provenientes do " desenrolamento" de laços, forma um *trace* de programa, e por essa razão é compactado como um outro trecho qualquer.

A especificação e implementação do algoritmo *trace scheduling* criou novas oportunidades para detecção e exploração do paralelismos de baixo nível presentes em programas de aplicação. As significativas taxas de aceleração obtidas durante a interpretação (via simulador) do código compactado pelo algoritmo, motivaram a criação de um processador para explorar o potencial oferecido pela técnica *trace scheduling*. Esse processador Super Escalar, denominado *Trace*, é do tipo VLIW e foi fabricado e comercializado pela Multiflow.

Embora possa ser empregado em aplicações de uso geral, o despacho de instruções proposto por Fisher [FISH83] é mais eficiente para aplicações científicas: empregando o algoritmo *trace scheduling*, o desempenho de um processador Super Escalar pode apresentar taxas de aceleração variando de 10 a 30 vezes de acordo com o programa de aplicação.

Percolation Scheduling

Percolation scheduling (PS) é uma outra técnica de compactação global de código cujo objetivo é a exploração do paralelismo fora das fronteiras dos blocos básicos. A experiência obtida durante a implementação do *software* de suporte para a arquitetura ELI-512 de Universidade de Yale [FISH84], viabilizou o desenvolvimento dessa nova técnica de detecção e extração de paralelismo de baixo nível [NICO85].

Conforme apontado por Aiken e Nicolau [AIKE88], as seguintes características da técnica *trace scheduling* motivaram a criação do meio ambiente de geração de código paralelo:

- A técnica *trace scheduling* compacta somente trechos que são isolados do restante do programa como se eles fossem blocos básicos.

- Código de reparo é incluído nos pontos de entrada e de saída do *trace* compactado para que a equivalência semântica do programa seja preservada.

A inflexibilidade da técnica, que compacta somente um *trace* por vez, impede alcançar a taxa de paralelismo que seria detectada se dois ou mais traces fossem comapctados em conjunto.

- A interação "usuário X processo" de compactação é muito reduzida na técnica *trace scheduling*. O estágio corrente das técnicas de analise de fluxo sugerem que uma participação maior do usuário seria muito útil durante a geração do código

paralelo. A técnica *trace scheduling* é efetiva somente para uma classe específica de aplicações (programas científicos).

O objetivo da técnica *percolation scheduling* consiste em maximizar o nível de paralelismo do programa compactado através da movimentação de operações entre os nós, aumentando desse modo, o número de operações que serão executadas simultaneamente. A técnica utiliza um pequeno conjunto de primitivas define as movimentações de operações permitidas entre nós adjacentes em um grafo paralelo do programa. Cada nó desse grafo paralelo contem uma ou uma ou mais operações que podem ser executadas em paralelo. As arestas do grafo determinam as vias de execução do programa.

O processo se inicia com um grafo do fluxo de controle do programa, onde cada nó representa uma ou mais operações. Nicolau ressalta que o código fonte original não precisa ser necessariamente sequencial, podendo já ter sido previamente otimizado. Utilizando as primitivas de movimentação de operações, o grafo original é transformado em outro, semanticamente equivalente, porém mais " paralelo", i.e, com um número menor de nós.

Estas primitivas simples baseiam-se no fluxo de controle e no grafo de dependências do programa. Uma movimentação só se concretiza após sua validade tenha sido verificada. Todas as movimentações do código são realizadas a partir destas operações básicas, de forma que a equivalência semântica entre os grafos inicial e final fique assegurada.

Quatro primitivas de transformação foram definidas. Elas operam em nós adjacentes do grafo. A aplicação destas transformações repetidas vezes permite que as operações se movam (*percolate*) para várias partes do programa em direção ao nó inicial. Em seguida, faremos uma breve descrição de cada uma das transformações.

As quatro primitivas de transformação do algoritmo são:

Delete:

Nós vazios podem ocorrer como resultado de outras transformações ou como parte do grafo original. Uma vez que não afetam a execução e a semântica do programa de nenhuma forma, tais nós podem ser removidos, desde que seus predecessores passem a apontar para seus sucessores.

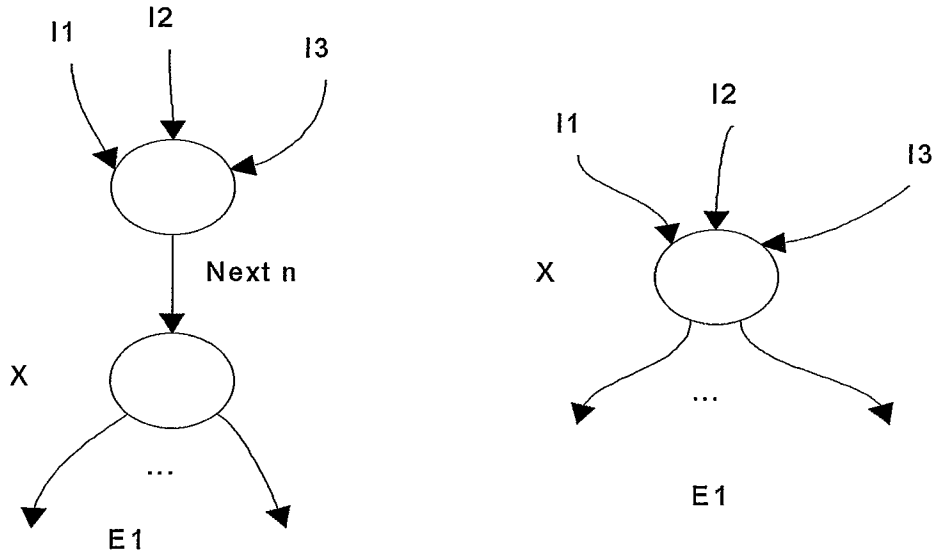


Figura 1 Primitiva Delete.

Move-Op:

Esta transformação move uma operação do nó **n** para o nó **m**, através da aresta (m,n), desde que não exista nenhuma dependência entre o nó **m** e a componente sendo movida. Ao realizar a movimentação, deve-se tomar cuidado para não afetar caminhos que passem por **n** e não passem por **m**.

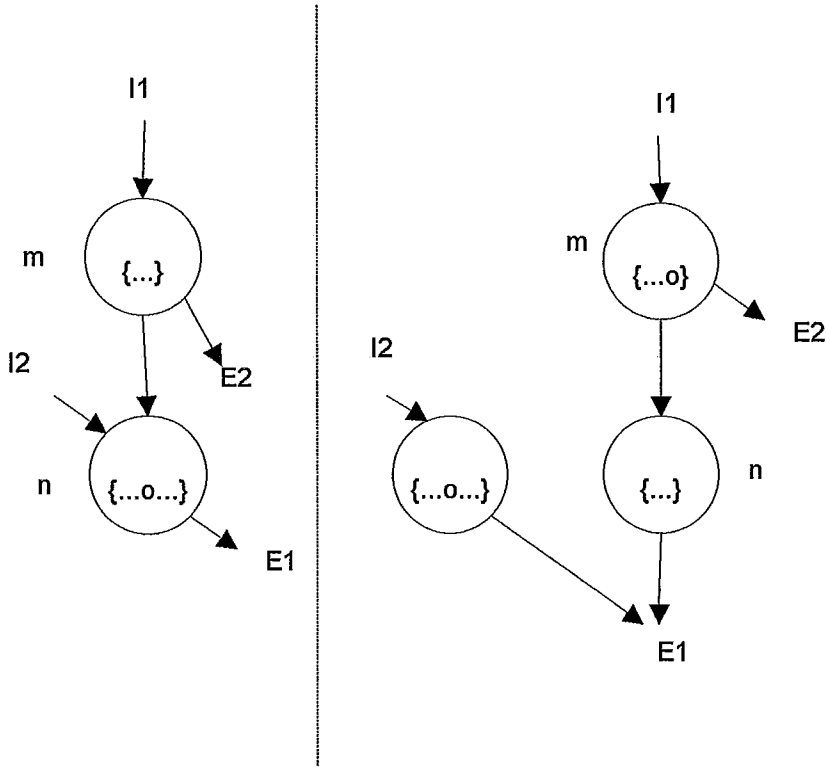


Figura 2 Primitiva Move-Op

Move-Cj:

Esta primitiva move uma operação de desvio condicional do nó **n** para o nó **m** através da aresta **(m,n)**, desde que não exista nenhuma dependência entre o nó **m** e a componente sendo movida. Ao realizar a movimentação, deve-se tomar cuidado para não afetar caminhos que passem por **n** mas não passem por **m**.

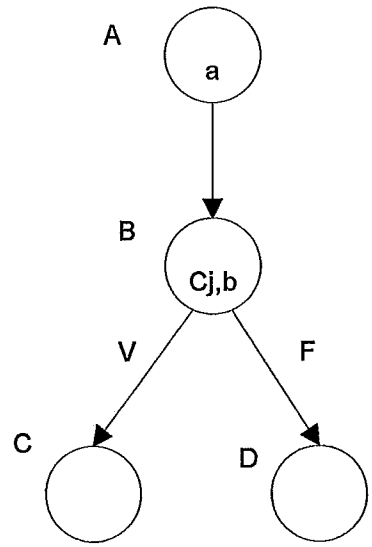


Figura 3 Primitiva Move-Cj

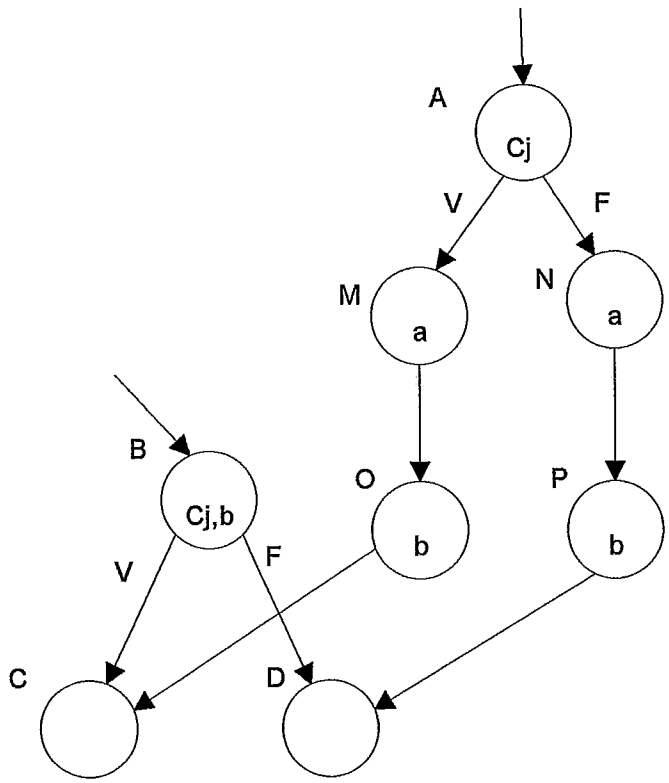


Figura 4 Primitiva Move-Cj

Unificação:

A unificação consiste na movimentação da cópia de operações idênticas de um conjunto de nós S = { n0, n1, n2, ... } para um predecessor **m**. Esta movimentação só pode ser feita se não existir nenhuma dependência entre o nó **m** e a componente que está sendo movida e se existirem caminhos **(m,ni)** para todos os nós pertencentes a S. ao realizar a movimentação deve-se tomar cuidado para não afetar caminhos que passem pelos **ni's** e não passem por **m**.

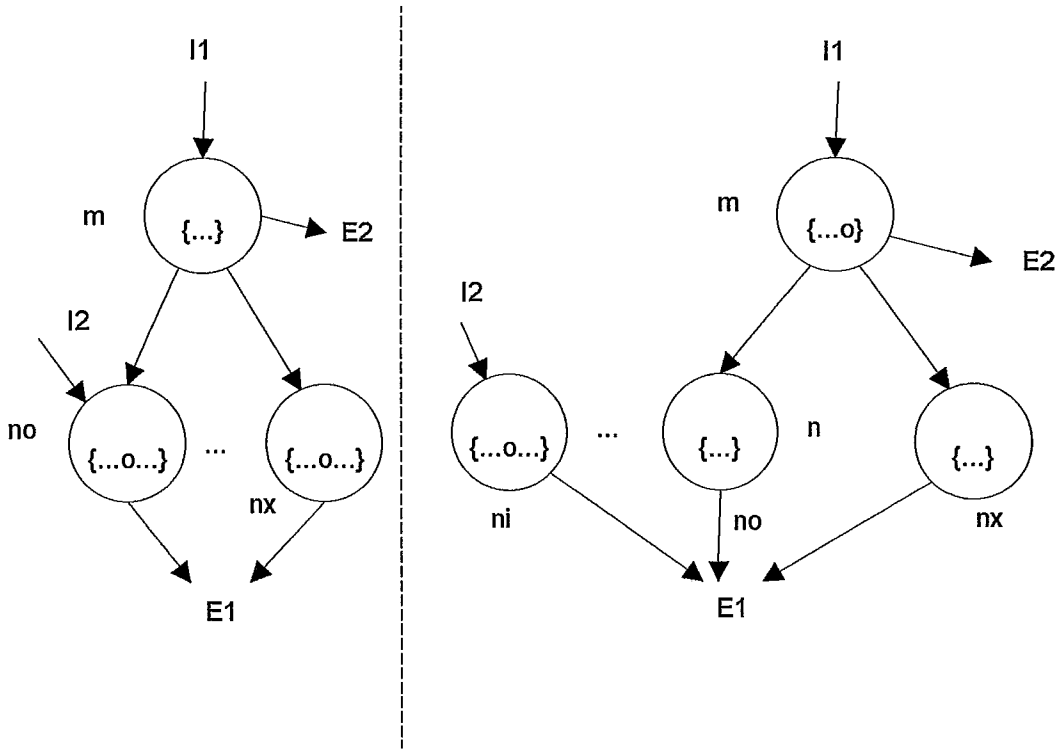


Figura 5 Primitiva Unificação

Capítulo III Paralelização de Codigo para máquinas VLIW

Compactação de μ-programas

A compactação de μ-código é um tópico de pesquisa que tem sido bastante estudado. A necessidade de automatizar a produção de μ-código efeciente para máquinas incorporando paralelismo de baixo nível, é responsável pelo desenvolvimento de inúmeros trabalhos nessa área.

O desenvolvimento das técnicas de compactação resultou na criação das chamadas máquinas Super Escalares, que são caracterizadas pela presença de paralelismo no nível de instruções em linguagem de máquina. As arquiteturas Super Escalares são capazes de executar simultaneamente diversas instruções distintas provenientes de um mesmo programa. Por essa razão, o potencial de processamento dessas máquinas pode ser comparado com o de um supercomputador [JOUP89].

Como ocorre com as suas ancestrais (i.e, as μ-máquinas com μ-instruções horizontais), a grande barreira responsável pela redução no nível de paralelismo que pode ser extraído de uma arquitetura Super Escalar, consiste na dificuldade de programá-la. O desenvolvimento manual de programas de aplicação somente é viável naqueles casos quando um reduzido número de instruções é suficiente para especificar a tarefa desejada. O desenvolvimento de ferramentas para a produção automática de código paralelo encontra-se ainda num estado embrionário, e por essa razão, esquemas interativos, com o programador participando do processo de paralelização de código para essas arquiteturas, têm sido investigados.

Dependência de Dados

Dois tipos de dependências limitam o número de primitivas que podem ser ativadas em paralelo: dependência de controle e dependência de dados. Dependência de controle (ou procedural) resulta do fluxo de controle do programa em execução. Por exemplo, quando da execução do comando *if* <*cond*> *then* $list_1$ *else* $list_2$, não podemos determinar precisamente qual das duas listas será selecionada, sem que saibamos a *priori* qual o resultado da avaliação da condição <*cond*>.

A dependência de dados resulta do fluxo de dados durante a execução de um programa: o conteúdo das variáveis é alterado pelas transferências impostas pelos comandos de atribuição. Em outras palavras, considerando que o valor de uma expressão depende do conteúdo das variáveis e dos operandos que a formam, e que esse valor pode ser usado como fonte de um comando de atribuição, é de se esperar então que esse comando seja dependente do outro dado que é produzido pela avaliação da expressão. Por exemplo, considere o seguinte trecho de programa:

```
...     ....
i.      A := B + C;
j.      E := A + 7;
...     ....
```

Dois comandos de atribuição estão especificados no fragmento de programa acima. O i-ésimo comando depende de dois comandos precedentes que alteram (pela última vez) o conteúdo das variáveis B e C. Assim, podemos dizer que o comando i é dependente desses dois comandos. Analogamente, podemos dizer que o j-ésimo comando do trecho depende do dado produzido pelo i-ésimo comando (a variável A, usada como fonte pela instrução j, é alterada pelo comando i).

A partir desse exemplo, podemos verificar que as dependências de dados (e as procedurais também) estabelecem uma relação de precedência entre os comandos de um programa, i.e. o ínicio de um comando pode estar condicionado ao término de um outro.

Dependências atuam como barreiras, forçam o seqüenciamento dos comandos de um programa, sendo portanto uma das responsáveis pela queda no desempenho de processadores com paralelismo de granularidade fina. Diversas técnicas de otimização para eliminar dependências têm sido desenvolvidas, incluindo aquelas que introduzem código para reparar o efeito provocado pela execução antecipada e indevida de alguns comandos.

Existem três tipos de dependência de dados:

1. Dependência verdadeira
2. Anti-dependência
3. Dependência de saída

Com o objetivo de faciltar a detecção e tratamento das dependências (ou interações) dos comando de um programa, costuma-se empregar uma estrutura de dados denominada de grafo de dependência de dados de um programa *(Data Dependence Graph*, ou simplesmente DDG). Os comandos do programa são representados por um nó do grafo, e as arestas (que são orientadas) correspondem às relações de precedência dos comandos. Por exemplo, o DDG do trecho do programa apresentado anteriormente pode ser assim representado:

Figura 6 Exemplo de DDG

Grafo de Dependência

Para construir o DDG de um programa precisamos das seguintes definições: conjuntos de operandos fonte e destino das instruções (In(i) e Out(i) respectivamente; ordem de execução dos comandos (Θ); e relações de dependência (δ,δ,δ^{o} e δ^{*}).

Uma instrução de atribuição pode ser representada por:

I: D := F_1 OP F_2

onde I é o rotulo do comando, OP o seu código de operação , F_1 e F_2 são os operandos fontes da operação e D a variável destino. Segundo essa definição, estamos assumindo que cada instrução da máquina hospedeira incorpora, além do código de operação, três endereços que podem ser registradores ou posições da memoria principal. Estamos supondo ainda, que somente comandos do tipo atribuição fazem parte do programa. Embora não seja realista, essa suposição não restringe o alcance do tratamento. Finalmente, caso tenhamos uma expressão simples do lado direito do comando, então o código de operação e o operando F_2 serão omitidos.

Conjuntos de Operandos Fonte e Destino

Denominaremos In(i) o conjunto de operandos fonte do comando I, e Out(i) o seu conjunto de operandos destino. Esses operandos podem ser registradores, palavras da memória, ou identificadores de variáveis armazenadas em outros tipos de elementos. Constantes ficam excluídas dos conjuntos de operandos da instrução I, é obvio que para alguns comandos, os conjuntos podem ser vazios.

Ordem de Execução dos Comandos

Usamos a relação θ (lê-se precede) para denotar a ordem de execução das instruções de um programa. Dessa forma, a relação i θ j (i precede j) indica que o comando i deve ser executado antes do comando j. Basicamente, a relação θ estabelece a ordem de execução dos comandos de um programa.

Relações de Dependência

Mesmo sendo independentes, duas ou mais instruções podem competir por recursos do *hardware* requeridos durante sua execução. No caso delas serem dependentes, a ordem de execução precisa ser mantida. Caso contrário, poderíamos ter resultados incorretos sendo produzidos. Por essas razões, é fundamental determinar se duas ou mais instruções interagem. Essa determinação torna-se extremamente relevante se desejarmos modificar a ordem de precedência original, permitindo que a execução antecipada e concorrente das instruções de um mesmo programa.

Levando em conta que a interação de duas instruções i e j, depende das variáveis por elas compartilhadas, então podemos dizer que:

- a instrução j apresenta uma ***dependência de dados verdadeira*** em relação à intrução i, denotado por $i \delta j$, quando

  $Out(i) \cap In(j) \neq 0$;

- o comando j é ***anti-depedente*** do comando i, denotado por $i \delta j$, quando

  $In(i) \cap Out(j) \neq 0$;

- a instrução j apresenta uma ***dependência de dado de saída*** em relação ao comando i, denotado por $i \delta^0 j$, quando

  $Out(i) \cap Out(j) \neq 0$;

Toda vez que a interseção dos conjuntos não for vazia, estamos diante de uma dependência, e por essa razão, quando da geração do código de máquina, a seqüencialidade dos comandos i e j deve ser assegurada (i.e., $i \theta j$).

Usaremos a notação de $i \delta^* j$ para referenciar qualquer um dos três tipos de dependência de dados entre os comandos i e j. Se $i \delta^* j$ e $j \delta^* k$, então $\exists$ uma relação de precedência entre i e k. Essa relação nem sempre é direta, i.e., nem sempre existe uma aresta no DDG conectando os nós i e k. Esse tipo de dependência, denominada de dependência indireta, é denotada por $I \Delta k$. Assim, na relação $a_1 \delta^* \cdots \delta^* a_n$ podemos representar uma dependência entre dois comandos da seqüencia, simplesmente por

$$a_j \Delta a_k, \forall\ 1 \leq j \leq k \leq n.$$

Finalmente, dizemos que duas instruções são independentes se elas não interagirem. Nesse caso, elas podem ser executadas simultaneamente.

Existem algoritmos implementados em *software* e em *hardware* capazes de eliminar, freqüentemente, as dependências 2 e 3 (chamadas de *dependências falsas* pois envolvem mais de uma atribuição ao mesmo operando destino). Tal não ocorre com a primeira dependência, e é por essa razão que ela é denomidada *dependência verdadeira.*

Combinando as condições de dependência e de ordem de execução de comandos, temos:

$i \delta j$ então $i \theta j$ e $Out(i) \cap In(j) \neq 0$.
$i \delta j$ então $i \theta j$ e $In(i) \cap Out(j) \neq 0$.
$i \delta^0 j$ então $i \theta j$ e $In(i) \cap Out(j) \neq 0$.

As operações primitivas descritas na técnica de *Percolation Scheduling*, e que serão utilizadas neste trabalho, nada mais são do que mudanças simples realizadas no fluxo de controle e na distribuição das instruções de um programa nas adjacências de um nó (chamaremos nó cada vértice do Grafo de Fluxo de Controle).

Antes de movimentar uma operação é necessário verificar a validade da movimentação. Para tal necessitamos do fluxo de controle do programa que está sendo examinado.

Conforme mencionado anteriormente, as operações básicas atuam sobre o fluxo de controle do programa, podendo alterá-lo. Desta forma procuramos uma representação adequada para esta estrutura de dados. Para facilidade de implementação, o grafo foi construído de forma a permitir que o percorressemos em qualquer direção e a partir de qualquer ponto.

A cada nó (vertice) do grafo associamos uma lista de instruções " alocadas" àquele nó. Inicialmente, cada nó contém uma única instrução (programa sequencial). A medidade que o processo de compactação progride, um grupo de instruções (programa já parcialmente paralelizado) pode estar incluído na lista de instruções alocadas aos nós. A cada momento, o conteúdo dos nós do fluxo de controle pode ser interpretado como sendo o conjunto de instruções que seriam executadas simultaneamente sempre que o nó fosse alcançado ao longo do processamento.

Ao final do processo de otimização, o grafo de fluxo de controle juntamente com as instruções alocadas a cada nó correspondem ao programa compactado pelo sistema.

É importante ressaltar que nem todos os campos estarão preenchidos: isso depende do programa de entrada, da interdependência de suas instruções, da disponibilidade de recursos do *hardware*, e do algoritmo responsável pela escolha da seqüencia em que as instruções do programa fonte devem ser tratadas.

Capítulo IV Descrição da máquina

Processador Intel i860

Em 1989 a Intel lançou o microprocessador 80860 (ou simplesmente i860). Incorporando cerca de 1 milhão de transistores num único circuito integrado, esse processador foi projetado para atender os requisitos de alto desempenho exigidos por *workstations*, aplicações numéricas, acelaradores, supercomputadores paralelos e etc.

Incluindo num mesmo chip blocos funcionais para operações com inteiros e com reais, e memórias *cache* de instrução e de dados, o i860 é capaz de executar até 80 milhões de instruções em ponto flutuante por segundo (80 MFLOPS) (precisão simples) ou 60 MFLOPS com precisão dupla.

Os seguinte blocos funcionais constituem o circuito integrado:

- Processador inteiro de 32 bits (RISC *core*);
- Unidade de gerenciamento de memória (MMU);
- Unidade de controle de ponto flutuante;
- Unidade de soma de ponto flutuante;
- Multiplicador de ponto flutuante;
- Unidade gráfica 3D;
- Memória *cache* de instruções com 4 *Kbytes* ;
- Memória *cache* de dados com 8 *Kbytes* ;
- Unidade de controle do barramento.

Os barramento internos e externos do processador são constituídos de 64 bits, exceto o que conecta a cache de dados com a unidade de controle de ponto flutuante (128 bits) e os 4 barramento da *Core Unit* (o processador inteiro manipula operandos de 32 bits). O padrão IEEE 754-1985 foi adotado na implementação das operações com reais (32 bits nas operçòes de precisão simples e 64 bits para precisão dupla).

O i860 possui dois modos de execução: simples e dual. No modo simples, uma instrução para a *Core* ou para unidade de controle de ponto flutuante é despachada em cada ciclo de máquina.

No modo de execução dual, em cada ciclo de máquina, a unidade *Core* faz a busca de duas instruções de 32 bits: uma das instruções é transferida para o processador inteiro e a outra para a unidade de controle de ponto flutuante. Tendo em vista que o i860 é uma arquitetura Super Escalar, essas duas instruções são executadas simultaneamente. Graças ao modo dual é que um alto desempenho pode ser atingido pelo processador: a unidade *Core*, através de instruções de *load e store*, assegura o necessário fluxo de operandos para que as unidades de ponto flutuante permaneçam em contínua operação, além de executar instruções de controle de loops.

O processador inclui dois bancos de registradores: um banco constituido de 32 registradores de 32 bits usado pelo processador de inteiros; e um banco com 32 registradores de 32 bits usados pela unidade de controle de ponto flutuante.

Registradores de ponto flutuante podem ser endereçados como 32 ítens individuais, ou como 16 registradores de 64 bits, ou então como 8 ítens de 128 bits.

Além de fazer a busca, a unidade *Core* executa os seguintes tipos de instruções:

a - load/store registradores
b - transferências entre registradores
c - adição e subtração (com e sem sinal)
d - operações de deslocamento
e - operações lógicas
f - desvios
g - instruções de controle

As instruções do tipo a incluem operações de load/store envolvendo os registradores de ponto flutuante também. As instruções de controle (tipo g) incluem o descarte do conteúdo da memória cache de dados, load/store registrador de controle e *lock/unlock.*

A maioria das instruções da *Core* é executada durante um ciclo de máquina. Na implementação dessa unidade, um *pipeline* com quatro estágios foi empregado: busca, decodificação, execução e armazenamento do resultado. Uma instrução de leitura (load) envolvendo a memória principal é executada num único ciclo de máquina. Esse detalhe de implementação é viabilizado pela existência de um *scoreboard* que garante a continuidade do processamento.

Se uma instrução referenciar um registrador cujo conteúdo ainda não foi atualizado (i.e., a transferência do dado a partir da memória principal ainda não terminou), o processador (usando a técnica do *scoreboarding*) interrompe a instrução corrente até que o dado esteja disponível. Caso o dado esteja armazenado na memória cache, a instrução de load requer um ciclo.

A unidade de gerenciamento de memória, implementa o conceito de memória virtual via paginação, com um espaço virtual constituído por 4 gigabytes. A memória é endereçada por bytes, e o formato *litle endian* (i.e o byte menos significativo é aquele com o endereco de memória mais baixo) é normalmente usado no endereçamento de dados constituídos por diversos bytes. O processador inclui um bit num registrador de controle que permite indicar o formato *big endian* (i.e., byte mais significativo no endereço mais baixo). Dados podem ser armazenados em qualquer um dos dois formatos. O código objeto sempre é armazenado no formato *little endian.*

A MMU implementa proteção a nível de páginas, e o mecanismo de paginação pode ser habilitado e desabilitado pelo programador. Se habilitado, o endereço virtual é traduzido - através da tabela de páginas - para um endereço físico da memória principal. No i860, as páginas são organizadas em dois níveis hierarquicos: o diretório de tabelas e tabela de páginas.

Usando a técnica LRU, a MMU mantem informações relacionadas com as 64 últimas páginas refernciadas. Essas informações ficam armazenadas no interior do processador (num buffer associativo) e permitem eliminar o ovrhead imposto pelas consultas aos dois níveis hierarquicos de tabelas, pois o endereço físico do bloco armazenado a página pode estar armazenado no *Translation Lookaside Buffer* (TLB).

Somente se o endereço do bloco não estiver armazenado no buffer associativo, é que o diretório e a tabela de páginas (residentes na memória principal) serão consultados.

Além da Unidade de controle, o sub-sistema de Ponto Flutuante (FPU) do I860 inclui 2 dispositivos separados (*adder e multiplier*) capazes de operar em paralelo. A unidade de controle de ponto flutuante despacha instruções para esses dois dispositivos funcionais, manipula as excessões e ataliza os bits do registrador de status da unidade (*floating point status register*). Os barramentos de 64 bits são usados para transferir os operandos e os resultados das operações executadas pelos dois dispositivos (e pela unidade gráfica).

Empregando um pipeline de três estágios na implementação de suas unidades (i.e., do somador e do multiplicador), a FPU pode produzir até dois resultados por ciclo de máquina. Por exemplo, a 40 Mhz, cada unidade pode produzir um resultado em precisão simples por ciclo (80 MFLOPS). A FPU opera em dois modos: escalar e pileline. Se duas instruções adjacentes apresentam dependência de dados, o modo escalar é usado. No modo pipeline (vetorial), os três estágios da arquitetura devem ser levados em conta pelo programador (ou pelo compilador). O modo vetorial pode ser especificado no modo dual de execução (i.e., instruções despachadas simultaneamente para a *Core* e para FPU). é importante observar que compete ao programador a tarefa de especificar o modo de execução desejado (simples ou dual) e como a FPU deverá operar (instrções escalares ou vetorias). Essa especificação é feita através da escolha apropriada dos códigos de operação.

A unidade de soma executa instruções de adição, subtração, comparação e conversão, todas operando com valores em precisão simple ou dupla. No modo escalar, o somador requer três ciclos de máquina para produzir um resulatdo. No modo pipeline, um resultado é produzido a cada ciclo de máquina.

A unidade gráfica 3D do i860 opera em paralelo com a *Core* Unit e usa o banco de registradores de ponto flutuante. Esta unidade é capaz de executar comandos manipulando simultaneamente 2,4 ou 8 pixels, dependendo da largura do pixel (24/32, 16 ou 8 bits respectivamente). As primitivas da unidade gráfica suportam algoritmos para traçados em três dimensões, tais como, procedimentos para eliminar superfícies ocultas, para avaliar perpectiva, para produzir sombras etc.

As unidades de memória *cache* (instruções e dados) são separadas e são formadas por 4 K e 8 K bytes. A *cache* de instruções transfere até 64 bits por ciclo de máquina enquanto que e de dados pode transferir até 128 bits por ciclo. A política *write back* é implementada na *cache* de dados. Através do *software*, é possível inibir a atuação da memória *cache* de dados.

A unidade de controle do barramento é capaz de transferir 64 bits a cada dois ciclos de máquina. O barramento pode ser conectado ao sistema de memória principal ou a um segundo nível de memória *cache*. Com o objetivo de aumentar a taxa de transferência de dados, as operções envolvendo a memória principal podem ser executadas no modo *pipeline*. Conseqüentemente , a cada dois ciclos, o barramento externo pode fornecer um operando da memória principal para o par de instruções do tipo *add/multiply* em dupla precisão, ou para dois pares de instruções adjacentes do tipo *add/multiply* em precisão simples. Em ambos os casos, estamos supondo que o

segundo operando das instruções estejam na memória *cache* de dados.

Modelo da Máquina VLIW

As técnicas utilizadas neste trabalho podem ser aplicadas à geração de código paralelo para uma grande variedade de máquinas, tanto no nível convencional de programação quanto ao nível de microprogramação. Porém, para a implementação do algoritmo, tornou-se necessária a definição de um modelo de arquitetura alvo, para o qual o ambiente desenvolvido se direciona.

Em nosso modelo cada instrução é constituida de varias operações, isto é, a instrução possui diversos campos que especificam grupos de atividades que poderão ou não ser executadas a cada ciclo de máquina.

O modelo de arquitetura utilizado é identico a arquitetura do processador i860, i.e utlizamos o mesmo conjunto de instruções, barramentos e unidades funcionais. Porém, a latência das unidades funcionais em nossa arquitetura pode ser modificada atraves de parâmtros. O número de instrucoes executadas por ciclo de máquina também pode ser configurado antes da execução do algoritmo de compactação, sendo possivel indicar dentro do número de instruções que serão executadas a cada ciclo quantas unidades funcionais de cada tipo existirão.

Capítulo V Processo de Compactação

Descrição do Processo de compactação

Dispondo das primitivas do *percoltion scheduling* podemos definir o processo de compactação quer foi utilizado neste trabalho.

O processo de compactação utiliza como entrada um programa em linguagem *assembly* do i860 [MASM90]. A partir deste código, é contruido o grafo de fluxo do programa em questão. Para cada nó do grafo temos as seguintes informações:

1. Lista de instruções alocadas neste nó;
2. Um *scoreboard* das unidades que estão sendo utilizadas;
3. A próxima instrução após a execução deste nó;
4. Uma lista de todas as instruções que tem este nó como *target*;

Além do grafo de fluxo de controle também criamos uma lista de todos os blocos básicos existentes no programa. Utilizando as operações Move-Op e Deleção, iniciamos o processo de compactação sobre os blocos básicos.

O algoritmo de compactação seleciona as instruções que irão sofrer o processo de migração até que não haja possibilidade de migrar nenhuma instrução. Damos então por encerrado o processo de compactação. Vale ressaltar que a política de escolha da próxima instrução a ser migrada consiste apenas em tomar um bloco básico a partir da sua última instrução e tentá-la migrar até que não seja mais possível.

Descrição do Algoritmo

A partir do grafo de fluxo do programa original e das informações sobre o tempo de latência de cada unidade funcional, o programa é reajustado de forma que respeite as características da arquitetura VLIW que esta sendo utilizada, i.e. como cada ciclo uma janela da VLIW é executada, se uma instrução tem um tempo de latência maior que um ciclo então teremos que inserir *nops* de forma a garantir que a unidade funcional utilizada pela instrução em questão não seja ocupada por outra instrução subsequente.

A partir deste novo programa gerado (com *nops*) podemos aplicar as primitivas de movimentação (Move-Op e Deleção) de forma a compactar o programa. Neste trabalho as primitivas do *percolation scheduling* foram utilizadas para executar a compactação de blocos básicos.

As primitivas acima mencionadas foram combinadas de modo a gerar uma "função" chamada *Migrate*. O objetivo desta função é tentar migrar uma instrução o mais para o topo do bloco básico possível. Esta migração migração leva em conta as dependências de dados existentes entre a instrução sendo movidada e instrução alvo, e a disponibilidade de unidades funcionais. O processo de migração das instruções que compoem o bloco básico termina quando não for mais possível migrar nenhuma instrução.

Capítulo VI Experimentos

Descrição dos programas compactados

Como forma de avaliar a eficácia o processo de extração de paralelismo que implementamos, tomamos como base um exemplo prático que é a *Biblioteca Vetorial (VECLIB)*. As rotinas que compoem esta biblioteca proveem um conjunto de operações básicas que podem ser chamadas de um programa em C e em Fortran. De programas em C, as chamadas destas rotinas substituem os loops utilizados nos calculos correspondentes.

As rotinas são definidads para dupla e simples precisão e a biblioteca vetorial inclue a várias rotinas do BLAS (Basic Linear Algebra Subprograms) e utiliza as mesmas convenções de nome de chamada do BLAS.

As rotinas utilizadas foram descritos em uma linguagem de alto-nível (Linguagem C) e submetidos ao compilador High C 860 produzido pela Metaware. Todas as opcoes de otimização foram desabilitadas para que o código gerado pelo compilador não influenciasse no processo de otimização implementado.

A seguir apresentamos uma breve descrição (uma descrição mais completa pode ser encontrada no apêndice) de todas as rotinas utilizadas como exemplo:

dcopy	Copy vector
dfill	Fill vector
dneg	Change sign
dswap	Swap vectors
dsadd	Scalar plus vector
dscal	Scalar times a vector to itself
dsdiv	Scalar Scalar divided by vector
dsmul	Scalar times vector
dssub	Scalar vector subtraction
dvadd	Vector addition
dvdiv	Element-wise vector division
dvmul	Vector element-wise multiplication
dvneg	Negate Vector
dvrecp	Vector reciprocal
dvsub	Vector subtraction

Tabela 1 Primitivas Matemáticas

daxpy	(Scalar x vector) + vector

dsvmvt	(Scalar - vector) x vector
dsvpvt	(Scalar + vector) x vector
dsvtsp	(Scalar x vector) + scalar
dsvtvm	(Scalar x vector) - scalar
dsvtvp	(Scalar x vector) + vector
dsvvmt	Scalar x (vector - vector)
dsvvpt	Scalar - (vector + vector)
dsvvtm	Scalar - (vector x vector)
dsvvtp	Scalar + (vector x vector)
dvvmvt	(Vector - vector) x vector
dvvpvt	(Vector + vector) x vector
dvvtvm	(Vector x vector) - vector
dvvtvp	(Vector x vector) + vector
dvvvtm	Vector - (vector x vector)

Tabela 2 Operações Triad

Avaliação dos resultados

A seguir apresentamos os resultados obtidos após o processo de compactação. Nas tabelas apresentamos os seguintes dados:

- Taxa de ocupação das unidades funcionais.
- Número de instruções executadas a cada ciclo de máquina.
- Tamanho do bloco básico original i.e. antes da compactação.
- Tamanho do bloco básico compactado.
- Total de operações executadas pelo bloco básico.

para todas as rotinas somente apresentamos os dados referentes ao bloco básico que compoem o loop principal da rotina em questão.

Rotina	LS	ALU	GRP	BR	FPADD	FPMUL	Instr/Ciclo	B.B Original	B.B Compactado	Total Operacoes
dcopy	1.00	0.63	0.00	0.13	0.00	0.00	1.38	10	8	11
dfill	1.33	0.67	0.00	0.17	0.00	0.00	1.67	9	6	10
dneg	0.50	0.50	0.00	0.13	0.38	0.00	1.13	8	8	9
dswap	2.00	0.63	0.00	0.13	0.00	0.00	1.88	16	8	15
dsadd	0.75	0.63	0.00	0.13	0.38	0.00	1.38	10	8	11
dscal	0.75	0.50	0.00	0.13	0.00	0.38	1.25	9	8	10
dsdiv	0.33	0.39	0.06	0.06	0.50	1.33	1.33	23	18	24
dsmul	0.75	0.63	0.00	0.13	0.00	0.38	1.38	10	8	11
dssub	0.29	0.71	0.14	0.07	0.21	0.43	1.36	18	14	19
dvadd	0.60	0.60	0.00	0.10	0.30	0.00	1.20	11	10	12
dvdiv	0.30	0.40	0.05	0.05	0.45	1.20	1.25	24	20	25
dvmul	0.60	0.60	0.00	0.10	0.00	0.30	1.20	11	10	12
dvneg	0.50	0.63	0.00	0.13	0.38	0.00	1.25	9	8	10
dvrecp	0.24	0.41	0.06	0.06	0.53	1.24	1.29	21	17	22
dvsub	0.60	0.60	0.00	0.10	0.30	0.00	1.20	11	10	12

Tabela 3 Primitivas Matemáticas Compactadas

Rotina	LS	ALU	GRP	BR	FPADD	FPMUL	Instr/Ciclo	B.B Original	B.B Compactado	Total Operacoes
daxpy	0.80	0.50	0.00	0.10	0.30	0.30	1.30	12	10	13
dsvmvt	0.80	0.60	0.00	0.10	0.30	0.30	1.40	13	10	14
dsvpvt	0.80	0.60	0.00	0.10	0.30	0.30	1.40	13	10	14
dsvtsp	0.80	0.50	0.00	0.10	0.60	0.00	1.30	12	10	13
dsvtvm	0.80	0.60	0.00	0.10	0.30	0.30	1.40	13	10	14
dsvtvp	0.80	0.60	0.00	0.10	0.30	0.30	1.40	13	10	14
dsvvmt	0.67	0.50	0.00	0.08	0.25	0.25	1.17	13	12	14
dsvvpt	0.67	0.50	0.00	0.08	0.50	0.00	1.17	13	12	14
dsvvtm	0.67	0.50	0.00	0.08	0.25	0.25	1.17	13	12	14
dsvvtp	0.67	0.50	0.00	0.08	0.25	0.25	1.17	13	12	14
dvvmvt	0.67	0.58	0.00	0.08	0.25	0.25	1.25	14	12	15
dvvpvt	0.67	0.58	0.00	0.08	0.25	0.25	1.25	14	12	15
dvvtvm	0.67	0.58	0.00	0.08	0.25	0.25	1.25	14	12	15
dvvtvp	0.67	0.58	0.00	0.08	0.25	0.25	1.25	14	12	15
dvvvtm	0.67	0.58	0.00	0.08	0.25	0.25	1.25	14	12	15

Tabela 4 Primitivas Triad Compactadas

O resultados apresentados acima foram obtidos com utilizando a configuração que melhor se adaptava a rotina em questão. As seguintes configurações foram utilizadas:

Configuração	LS	ALU	GRP	BR	Fpadd	Fpmul
1	2	2	1	1	1	1
2	4	2	1	1	1	1
3	1	2	1	1	1	1
4	6	2	1	1	1	1
5	2	3	1	1	2	2
6	2	2	1	1	2	1

Tabela 5 Configurações da VLIW

a seguir apresentamos a configuração que foi utilizada:

Rotina	Confiuração
dcopy	1
dfill	2
dneg	3
dswap	4
dsadd	1
dscal	1
dsdiv	5
dsmul	1
dssub	1
dvadd	3
dvdiv	5
dvmul	3
dvneg	3
dvrecp	5
dvsub	3

Tabela 6 Configuração Primitivas Matemáticas

Rotina	Configuração
daxpy	1
dsvmvt	1
dsvpvt	1
dsvtsp	6
dsvtvm	1
dsvtvp	1
dsvvmt	3
dsvvpt	6
dsvvtm	3
dsvvtp	3
dvvmvt	3
dvvpvt	3
dvvtvm	3
dvvtvp	3
dvvvtm	3

Tabela 7 Configuração Primitivas Triad

Capitulo VII Conclusão

Neste trabalho, apresentamos algumas técnicas propostas para o problema da extração de paralelismo. Vimos que, as mesmas técnicas usadas na compactação de microprogramas podem ser aplicadas ao nível convencional de programação e na geração de código para máquinas Super Escalares.

Concentramos nossos esforços na técnica *Percolation Scheduling* descrita por Nicolau em [NICO85,AIKE88]. Desenvolvemos uma ferramenta que, aplicando as primitivas do *Percolation Scheduling*, gera código paralelo para uma máquina Super Escalar, a partir de um código inicial puramente sequencial. Nos experimentos que realizamos obtivemos taxas de compactação dos blocos básicos que variaram de 10% a 50 %.

Devido a característica da biblioteca vetorial ser basicamente um conjunto de loops criados simularem operações vetoriais através da utilização dos recursos disponíveis na arquitetura do processaodr i860, tornou-se inviável a utilização da compactação global. Isto se deve porque todas as rotinas são basicamente compostas de apenas um loop principal, o qual executa toda a função descrita pela rotina. Inicialmente insistimos em utilizar compactação global porém chegamos a conclusão de que não haveria nenhum ganho na sua utilização.

A utilização da VECLIB teve o objetivo de avaliarmos que características seriam importantes o i860 ter. Como podemos ver pelos dados obtidos, a maioria das rotinas teve o seu desempenho melhorado quando utilizamos uma arquitetura com 2 undades de load/store e 2 ALU's.

Ainda existe muito trabalho a ser feito para a construção de uma ferramenta automatizada para exploração do paralelismo. A compactação manual, é certamente um trabalho muito ingrato, e talvez fosse interessante, como o próprio Nicolau prevê, oferecer uma interface amigável que permita ao usuário interferir diretamente no processo de otimização.

Bibliografia

[AIKE88] A. Aiken and A. Nicolau, "A development Enviroment for Horizontal Microcode", IEEE Transactions on Computer, Vol 14, No. 5, May 1988.

[FISH81] J. A. Fisher, "Trace Scheduling: A techinque for Global Microcode Compaction", IEEE Transactions on Computers, Vol C-30, No. 7, 1981, pp-478-490

[FISH83] J. A. Fisher, "Very Long Instruction Word Architectures and the ELI-512", Proceedings of the Annual International Symposium on Computer Architectures, IEEE Computer Society and the Association for Computing Machinery, June 1983, pp. 140-150.

[HENN83] John Hennessy and Thomas Gross, "Postpass Code Optimization of Pipelined Constrians", ACM Transaction on Programming Languages and Systems, Vol 5, No. 3, July 1983, pp. 442-448.

[INTE89] i860TM 64-Bit Micropocessor Programmer's Reference Manual.

[JOUP89] N. Jouppi and D. Wall, "Available Instruction Level Parallelism for Super-Scalar and Super-Pipelined Machines", Proceedings of Third ASPLOS (April 1989), in ACM SIGPLAN Notices, Vol 14, May 1989.

[LAND80] D. Landskov, S. Davidson, B. Shriver and P.W. Mallet, "Local microcode compaction", Computing Surveys, Vol 12, Sept. 1980, pp 261-294

[LINN88] J. L. Linn, "Horizontal Microcode Compaction", In Microprogramming and Firmware Engineering Methods, Edited by Stanley Habib, Van Nostrand Reinhold, New York, 1988, pp. 381-431.

[NICO85] A. Nicolau, "Percolation Scheduling: A Parallel Compilation Techinique", Technical Report TR-85-678 , May 1985

[NICO90] A. Nicolau and R. Potasman, "Realistic Scheduling: Compaction for Pipelined Architectures", Proceedings of the 23rd Annual Workshop on Microprogramming and Microarchitectures MICRO-23, November 1990, pp. 69-79

[TOKO81] M. Tokoro, E. Tamura and T. Takisuka, "Optimization of Microprograms", IEEE Transactions on Computers, Vol C-30, No. 7, pp. 491-504, 1981

[WOOD78] G. Wood, "On the packing of micro-operations into micro-instructio words", Proc 11th Annual Workshop on Microprogramming, SIGMICRO, Dec. 1978, pp. 51-55.

Apêndices

Apresentamos a seguir o código original e o código otimizado dos programas utilizados como exemplo.

Primitivas Matemáticas

```
        .file   "XCOPY.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dcopy |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|void dcopy(n,x,y)
// 2|int n;
// 3|double *x,*y;
// 4|{
_dcopy::
// 5|  int i;
// 6|
// 7|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        bnc     .L.L001.3
.L.L001.2:
// 8|    y[i] = x[i];
        shl     3,r23,r22
.L.LL001.4:
        addu    r22,r17,r18
        ld.l    0(r18),r20
        ld.l    4(r18),r21
        addu    r22,r19,r18
        st.l    r20,0(r18)
        addu    1,r23,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        st.l    r21,4(r18)
        bc.t    .L.LL001.4
        shl     3,r23,r22
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XCOPY.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dcopy |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|void dcopy(n,x,y)
// 2|int n;
// 3|double *x,*y;
// 4|{
_dcopy::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0
        bnc     .L.L001.3
.L.L001.2:
// 8|     y[i] = x[i];
        shl     3,r23,r22
.L.LL001.4:
        addu    r22,r17,r18     addu    1,r23,r23
        ld.l    0(r18),r20      ld.l    4(r18),r21
        addu    r22,r19,r18
        st.l    r20,0(r18)      st.l    r21,4(r18)
        nop
        subs    r23,r16,r0
        bc.t    .L.LL001.4
        shl     3,r23,r22
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XFILL.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dfill |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|void dfill(n,alpha,x)
// 2|int n;
// 3|double *alpha,*x;
// 4|{
_dfill::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        bnc     .L.L001.3
.L.L001.2:
// 8|     x[i] = *alpha;
        shl     3,r22,r19
.L.LL001.4:
        addu    r19,r18,r19
        ld.l    0(r17),r20
        ld.l    4(r17),r21
        st.l    r20,0(r19)
        addu    1,r22,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        st.l    r21,4(r19)
        bc.t    .L.LL001.4
        shl     3,r22,r19
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XFILL.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dfill |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|void dfill(n,alpha,x)
// 2|int n;
// 3|double *alpha,*x;
// 4|{
_dfill::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0
        bnc     .L.L001.3
.L.L001.2:
// 8|     x[i] = *alpha;
        shl     3,r22,r19
.L.LL001.4:
        addu    r19,r18,r19     addu    1,r22,r22       ld.l    4(r17),r21      ld.l
        0(r17),r20
        st.l    r21,4(r19)      st.l    r20,0(r19)
        nop
        subs    r22,r16,r0
        bc.t    .L.LL001.4
        shl     3,r22,r19
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file  "XNEG.C"
        .text
        .align 8
_L00TEXT:
        .text; .align 4
//--------------| dneg |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|void dneg(n,x)
// 2|int n;
// 3|double *x;
// 4|{
_dneg::
// 5|  int i;
// 6|
// 7|  for(i=0;i<n;i++)
        mov    r0,r19
        subs   r19,r16,r0     // cc:r19 < r16  !cc:r19 >= r16
        bnc    .L.L001.3
.L.L001.2:
// 8|    x[i] = -x[i];
        shl    3,r19,r18
.L.LL001.4:
        addu   r18,r17,r18
        fld.d  0(r18),f16
        fsub.dd f0,f16,f16
        addu   1,r19,r19
        subs   r19,r16,r0     // cc:r19 < r16  !cc:r19 >= r16
        fst.d  f16,0(r18)
        bc.t   .L.LL001.4
        shl    3,r19,r18
.L.L001.3:
// 9|}
        bri    r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XNEG.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dneg |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|void dneg(n,x)
// 2|int n;
// 3|double *x;
// 4|{
_dneg::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r19
        subs    r19,r16,r0
        bnc     .L.L001.3
.L.L001.2:
// 8|     x[i] = -x[i];
        shl     3,r19,r18
.L.LL001.4:
        addu    r18,r17,r18         addu    1,r19,r19
        fld.d   0(r18),f16
        fsub.dd f0,f16,f16
        fst.d   f16,0(r18)
        nop
        subs    r19,r16,r0
        bc.t    .L.LL001.4
        shl     3,r19,r18
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XSWAP.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dswap |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|void dswap(n,x,y)
// 2|int n;
// 3|double *x,*y;
// 4|{
_dswap::
// 5|   int i;
// 6|   double t;
// 7|
// 8|   for(i=0;i<n;i++) {
        mov     r0,r25
        subs    r25,r16,r0      // cc:r25 < r16  !cc:r25 >= r16
        bnc     .L.L001.3
.L.L001.2:
// 9|     t = y[i];
        shl     3,r25,r20
.L.LL001.4:
        addu    r20,r19,r18
        ld.l    0(r18),r23
        ld.l    4(r18),r24
//10|     y[i] = x[i];
        addu    r20,r17,r20
        ld.l    0(r20),r21
        ld.l    4(r20),r22
        st.l    r21,0(r18)
        st.l    r22,4(r18)
//11|     x[i] = t;
        st.l    r23,0(r20)
        addu    1,r25,r25
        subs    r25,r16,r0      // cc:r25 < r16  !cc:r25 >= r16
        st.l    r24,4(r20)
        bc.t    .L.LL001.4
        shl     3,r25,r20
.L.L001.3:
//12|   }
//13|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//14|
```

```
        .file   "XSWAP.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dswap |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|void dswap(n,x,y)
// 2|int n;
// 3|double *x,*y;
// 4|{
_dswap::
// 5|   int i;
// 6|   double t;
// 7|
// 8|   for(i=0;i<n;i++) {
        mov     r0,r25
        subs    r25,r16,r0
        bnc     .L.L001.3
.L.L001.2:
// 9|     t = y[i];
        shl     3,r25,r20
.L.LL001.4:
        addu    r20,r19,r18     addu    1,r25,r25
        ld.l    0(r18),r23      addu    r20,r17,r20     ld.l    4(r18),r24
        ld.l    0(r20),r21      st.l    r24,4(r20)      st.l    r23,0(r20)      ld.l
        4(r20),r22
        st.l    r21,0(r18)      st.l    r22,4(r18)
        nop
//10|     y[i] = x[i];
//11|     x[i] = t;
        subs    r25,r16,r0
        bc.t    .L.LL001.4
        shl     3,r25,r20
.L.L001.3:
//12|   }
//13|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//14|
```

```
        .file   "XSADD.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsadd |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|void dsadd(n,alpha,x,y)
// 2|int n;
// 3|double *alpha,*x,*y;
// 4|{
_dsadd::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        bnc     .L.L001.3
.L.L001.2:
// 8|     y[i] = *alpha + x[i];
        shl     3,r22,r21
.L.LL001.4:
        addu    r21,r18,r19
        fld.d   0(r19),f18
        fld.d   0(r17),f16
        addu    r21,r20,r19
        fadd.dd f16,f18,f16
        addu    1,r22,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        fst.d   f16,0(r19)
        bc.t    .L.LL001.4
        shl     3,r22,r21
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XSADD.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsadd |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|void dsadd(n,alpha,x,y)
// 2|int n;
// 3|double *alpha,*x,*y;
// 4|{
_dsadd::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0
        bnc     .L.L001.3
.L.L001.2:
// 8|     y[i] = *alpha + x[i];
        shl     3,r22,r21
.L.LL001.4:
        addu    r21,r18,r19     addu    1,r22,r22       fld.d   0(r17),f16
        fld.d   0(r19),f18
        addu    r21,r20,r19     fadd.dd f16,f18,f16
        fst.d   f16,0(r19)
        nop
        subs    r22,r16,r0
        bc.t    .L.LL001.4
        shl     3,r22,r21
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XSCAL.C"
        .text
        .align  8
_L00TEXT:
        .text; .align  4
//--------------| dscal |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|void dscal(n,alpha,x)
// 2|int n;
// 3|double *alpha,*x;
// 4|{
_dscal::
// 5|  int i;
// 6|
// 7|  for(i=0;i<n;i++)
        mov     r0,r20
        subs    r20,r16,r0      // cc:r20 < r16  !cc:r20 >= r16
        bnc     .L.L001.3
.L.L001.2:
// 8|    x[i] = *alpha * x[i];
        shl     3,r20,r19
.L.LL001.4:
        addu    r19,r18,r19
        fld.d   0(r19),f16
        fld.d   0(r17),f18
        fmul.dd f18,f16,f16
        addu    1,r20,r20
        subs    r20,r16,r0      // cc:r20 < r16  !cc:r20 >= r16
        fst.d   f16,0(r19)
        bc.t    .L.LL001.4
        shl     3,r20,r19
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XSCAL.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dscal |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|void dscal(n,alpha,x)
// 2|int n;
// 3|double *alpha,*x;
// 4|{
_dscal::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r20
        subs    r20,r16,r0
        bnc     .L.L001.3
.L.L001.2:
// 8|     x[i] = *alpha * x[i];
        shl     3,r20,r19
.L.LL001.4:
        addu    r19,r18,r19     addu    1,r20,r20       fld.d   0(r17),f18
        fld.d   0(r19),f16
        fmul.dd f18,f16,f16
        fst.d   f16,0(r19)
        nop
        subs    r20,r16,r0
        bc.t    .L.LL001.4
        shl     3,r20,r19
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XSDIV.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsdiv |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|
// 2|void dsdiv(n,alpha,x,y)
// 3|int n;
// 4|double *alpha,*x,*y;
// 5|{
_dsdiv::
// 6|  int i;
// 7|
// 8|  for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        bnc     .L.L001.3
.L.L001.2:
// 9|    y[i] = *alpha / x[i];
        shl     3,r22,r21
.L.LL001.4:
        addu    r21,r18,r19
        fld.d   0(r19),f24
        fld.d   0(r17),f22
        orh     16384,r0,r31    // 16384 = 0x4000
        ixfr    r31,f17
        fmov.ss f0,f16
        frcp.dd f24,f18
        fmul.dd f24,f18,f20
        fsub.dd f16,f20,f20
        fmul.dd f18,f20,f18
        fmul.dd f24,f18,f20
        fsub.dd f16,f20,f20
        fmul.dd f18,f20,f18
        fmul.dd f24,f18,f20
        fsub.dd f16,f20,f20
        fmul.dd f20,f18,f20
        addu    r21,r20,r19
        fmul.dd f22,f20,f16
        addu    1,r22,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        fst.d   f16,0(r19)
        bc.t    .L.LL001.4
        shl     3,r22,r21
.L.L001.3:
//10|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//11|
```

```
        .file   "XSDIV.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsdiv |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|
// 2|void dsdiv(n,alpha,x,y)
// 3|int n;
// 4|double *alpha,*x,*y;
// 5|{
_dsdiv::
// 6|  int i;
// 7|
// 8|  for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0
        bnc     .L.L001.3
.L.L001.2:
// 9|    y[i] = *alpha / x[i];
        shl     3,r22,r21
.L.LL001.4:
        addu    r21,r18,r19     addu    1,r22,r22       fmov.ss f0,f16  orh     16384,r0,r31
        fld.d   0(r17),f22
        fld.d   0(r19),f24      ixfr    r31,f17
        addu    r21,r20,r19     frcp.dd f24,f18
        fmul.dd f24,f18,f20
        fsub.dd f16,f20,f20
        fmul.dd f18,f20,f18
        fmul.dd f24,f18,f20
        fsub.dd f16,f20,f20
        fmul.dd f18,f20,f18
        fmul.dd f24,f18,f20
        fsub.dd f16,f20,f20
        fmul.dd f20,f18,f20
        fmul.dd f22,f20,f16
        fst.d   f16,0(r19)
        nop
        subs    r22,r16,r0
        bc.t    .L.LL001.4
        shl     3,r22,r21
.L.L001.3:
//10|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//11|
```

```
        .file   "XSMUL.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsmul |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|
// 2|void dsmul(n,alpha,x,y)
// 3|int n;
// 4|double *alpha,*x,*y;
// 5|{
_dsmul::
// 6|   int i;
// 7|
// 8|   for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0      //  cc:r22 < r16  !cc:r22 >= r16
        bnc     .L.L001.3
.L.L001.2:
// 9|     y[i] = *alpha * x[i];
        shl     3,r22,r21
.L.LL001.4:
        addu    r21,r18,r19
        fld.d   0(r19),f18
        fld.d   0(r17),f16
        addu    r21,r20,r19
        fmul.dd f16,f18,f16
        addu    1,r22,r22
        subs    r22,r16,r0      //  cc:r22 < r16  !cc:r22 >= r16
        fst.d   f16,0(r19)
        bc.t    .L.LL001.4
        shl     3,r22,r21       // conditionally executed
.L.L001.3:
//10|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//11|
```

```
        .file   "XSMUL.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsmul |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|
// 2|void dsmul(n,alpha,x,y)
// 3|int n;
// 4|double *alpha,*x,*y;
// 5|{
_dsmul::
// 6|  int i;
// 7|
// 8|  for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0
        bnc     .L.L001.3
.L.L001.2:
// 9|    y[i] = *alpha * x[i];
        shl     3,r22,r21
.L.LL001.4:
        addu    r21,r18,r19     addu    1,r22,r22       fld.d   0(r17),f16
        fld.d   0(r19),f18
        addu    r21,r20,r19     fmul.dd f16,f18,f16
        fst.d   f16,0(r19)
        nop
        subs    r22,r16,r0
        bc.t    .L.LL001.4
        shl     3,r22,r21
.L.L001.3:
//10|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//11|
```

```
        .file  "xssub.c"
        .text
        .align  8
_L00TEXT:
        .text; .align  4
//-------------| dssub |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|void dssub(n,alpha,x,y)
// 2|int n;
// 3|double alpha,*x,*y;
// 4|{
_dssub::
        fmov.dd f18,f22
// 5|  int i;
// 6|
// 7|  for(i=0;i<n;i++)
        mov    r0,r22
        subs   r22,r16,r0     // cc:r22 < r16  !cc:r22 >= r16
        bnc    .L.L001.3
.L.L001.2:
// 8|    y[i*incy] = alpha - x[i*incx];
        ixfr   r22,f18
.L.LL001.4:
        ixfr   r19,f20
        fmlow.dd       f18,f20,f18
        fxfr   f18,r17
        shl    3,r17,r17
        addu   r17,r18,r17
        fld.d  0(r17),f16
        ixfr   r22,f18
        fsub.dd f22,f16,f16
        ixfr   r21,f20
        fmlow.dd       f18,f20,f18
        fxfr   f18,r17
        shl    3,r17,r17
        addu   r17,r20,r17
        addu   1,r22,r22
        subs   r22,r16,r0     // cc:r22 < r16  !cc:r22 >= r16
        fst.d  f16,0(r17)
        bc.t   .L.LL001.4
        ixfr   r22,f18
.L.L001.3:
// 9|}
        bri    r1
        nop
```

```
        .file   "xssub.c"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dssub |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|void dssub(n,alpha,x,y)
// 2|int n;
// 3|double alpha,*x,*y;
// 4|{
_dssub::
        fmov.dd f18,f22   mov      r0,r22
        nop
        nop
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        subs      r22,r16,r0
        bnc       .L.L001.3
.L.L001.2:
// 8|     y[i*incy] = alpha - x[i*incx];
        ixfr      r22,f18
.L.LL001.4:
        ixfr      r19,f20
        fmlow.dd          f18,f20,f18
        ixfr      r21,f20  fxfr      f18,r17
        shl       3,r17,r17         ixfr      r22,f18
        addu      r17,r18,r17       fmlow.dd          f18,f20,f18       addu      1,r22,r22
        fld.d     0(r17),f16
        fsub.dd   f22,f16,f16       fxfr      f18,r17
        shl       3,r17,r17
        addu      r17,r20,r17
        fst.d     f16,0(r17)
        nop
        subs      r22,r16,r0
        bc.t      .L.LL001.4
        ixfr      r22,f18
.L.L001.3:
// 9|}
        bri       r1
        nop
```

```
        .file   "XVADD.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dvadd |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|void dvadd(n,x,y,z)
// 2|int n;
// 3|double *x,*y,*z;
// 4|{
_dvadd::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        bnc     .L.LL001.3
.L.L001.2:
// 8|     z[i] = x[i] + y[i];
        shl     3,r22,r20
.L.LL001.4:
        addu    r20,r17,r18
        fld.d   0(r18),f18
        addu    r20,r19,r18
        fld.d   0(r18),f16
        addu    r20,r21,r18
        fadd.dd f16,f18,f16
        addu    1,r22,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        fst.d   f16,0(r18)
        bc.t    .L.LL001.4
        shl     3,r22,r20
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XVADD.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dvadd |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|void dvadd(n,x,y,z)
// 2|int n;
// 3|double *x,*y,*z;
// 4|{
_dvadd::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0
        bnc     .L.L001.3
.L.L001.2:
// 8|     z[i] = x[i] + y[i];
        shl     3,r22,r20
.L.LL001.4:
        addu    r20,r17,r18     addu    1,r22,r22
        fld.d   0(r18),f18
        addu    r20,r19,r18
        fld.d   0(r18),f16
        fadd.dd f16,f18,f16     addu    r20,r21,r18
        fst.d   f16,0(r18)
        nop
        subs    r22,r16,r0
        bc.t    .L.LL001.4
        shl     3,r22,r20
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XVDIV.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dvdiv |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|void dvdiv(n,x,y,z)
// 2|int n;
// 3|double *x,*y,*z;
// 4|{
_dvdiv::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        bnc     .L.L001.3
.L.L001.2:
// 8|     z[i] = x[i] / y[i];
        shl     3,r22,r20
.L.LL001.4:
        addu    r20,r17,r18
        fld.d   0(r18),f24
        addu    r20,r19,r18
        fld.d   0(r18),f16
        orh     16384,r0,r31    // 16384 = 0x4000
        ixfr    r31,f21
        fmov.ss f0,f20
        frcp.dd f16,f18
        fmul.dd f16,f18,f22
        fsub.dd f20,f22,f22
        fmul.dd f18,f22,f18
        fmul.dd f16,f18,f22
        fsub.dd f20,f22,f22
        fmul.dd f18,f22,f18
        fmul.dd f16,f18,f22
        fsub.dd f20,f22,f22
        fmul.dd f22,f18,f22
        addu    r20,r21,r18
        fmul.dd f24,f22,f16
        addu    1,r22,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        fst.d   f16,0(r18)
        bc.t    .L.LL001.4
        shl     3,r22,r20
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XVDIV.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dvdiv |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|void dvdiv(n,x,y,z)
// 2|int n;
// 3|double *x,*y,*z;
// 4|{
_dvdiv::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0
        bnc     .L.L001.3
.L.L001.2:
// 8|     z[i] = x[i] / y[i];
        shl     3,r22,r20
.L.LL001.4:
        addu    r20,r17,r18     addu    1,r22,r22       fmov.ss f0,f20  orh     16384,r0,r31
        fld.d   0(r18),f24      ixfr    r31,f21
        addu    r20,r19,r18
        fld.d   0(r18),f16
        addu    r20,r21,r18     frcp.dd f16,f18
        fmul.dd f16,f18,f22
        fsub.dd f20,f22,f22
        fmul.dd f18,f22,f18
        fmul.dd f16,f18,f22
        fsub.dd f20,f22,f22
        fmul.dd f18,f22,f18
        fmul.dd f16,f18,f22
        fsub.dd f20,f22,f22
        fmul.dd f22,f18,f22
        fmul.dd f24,f22,f16
        fst.d   f16,0(r18)
        nop
        subs    r22,r16,r0
        bc.t    .L.LL001.4
        shl     3,r22,r20
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XVMUL.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dvmul |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|void dvmul(n,x,y,z)
// 2|int n;
// 3|double *x,*y,*z;
// 4|{
_dvmul::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        bnc     .L.L001.3
.L.L001.2:
// 8|     z[i] = x[i] * y[i];
        shl     3,r22,r20
.L.LL001.4:
        addu    r20,r17,r18
        fld.d   0(r18),f18
        addu    r20,r19,r18
        fld.d   0(r18),f16
        addu    r20,r21,r18
        fmul.dd f16,f18,f16
        addu    1,r22,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        fst.d   f16,0(r18)
        bc.t    .L.LL001.4
        shl     3,r22,r20
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XVMUL.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dvmul |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|void dvmul(n,x,y,z)
// 2|int n;
// 3|double *x,*y,*z;
// 4|{
_dvmul::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0
        bnc     .L.L001.3
.L.L001.2:
// 8|     z[i] = x[i] * y[i];
        shl     3,r22,r20
.L.LL001.4:
        addu    r20,r17,r18     addu    1,r22,r22
        fld.d   0(r18),f18
        addu    r20,r19,r18
        fld.d   0(r18),f16
        fmul.dd f16,f18,f16     addu    r20,r21,r18
        fst.d   f16,0(r18)
        nop
        subs    r22,r16,r0
        bc.t    .L.LL001.4
        shl     3,r22,r20
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XVNEG.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dvneg |-----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|void dvneg(n,x,y)
// 2|int n;
// 3|double *x,*y;
// 4|{
_dvneg::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r21
        subs    r21,r16,r0      // cc:r21 < r16  !cc:r21 >= r16
        bnc     .L.L001.3
.L.L001.2:
// 8|     y[i] = - x[i];
        shl     3,r21,r20
.L.LL001.4:
        addu    r20,r17,r18
        fld.d   0(r18),f16
        addu    r20,r19,r18
        fsub.dd f0,f16,f16
        addu    1,r21,r21
        subs    r21,r16,r0      // cc:r21 < r16  !cc:r21 >= r16
        fst.d   f16,0(r18)
        bc.t    .L.LL001.4
        shl     3,r21,r20
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XVNEG.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//-------------| dvneg |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|void dvneg(n,x,y)
// 2|int n;
// 3|double *x,*y;
// 4|{
_dvneg::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r21
        subs    r21,r16,r0
        bnc     .L.L001.3
.L.L001.2:
// 8|     y[i] = - x[i];
        shl     3,r21,r20
.L.LL001.4:
        addu    r20,r17,r18         addu    1,r21,r21
        fld.d   0(r18),f16
        fsub.dd f0,f16,f16          addu    r20,r19,r18
        fst.d   f16,0(r18)
        nop
        subs    r21,r16,r0
        bc.t    .L.LL001.4
        shl     3,r21,r20
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XVRECP.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//-------------| dvrecp |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|void dvrecp(n,x,y)
// 2|int n;
// 3|double *x,*y;
// 4|{
_dvrecp::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r21
        subs    r21,r16,r0      // cc:r21 < r16  !cc:r21 >= r16
        bnc     .L.L001.3
.L.L001.2:
// 8|     y[i] = ((double) 1.0) / x[i];
        shl     3,r21,r20
.L.LL001.4:
        addu    r20,r17,r18
        fld.d   0(r18),f22
        orh     16384,r0,r31    // 16384 = 0x4000
        ixfr    r31,f21
        fmov.ss f0,f20
        frcp.dd f22,f18
        fmul.dd f22,f18,f16
        fsub.dd f20,f16,f16
        fmul.dd f18,f16,f18
        fmul.dd f22,f18,f16
        fsub.dd f20,f16,f16
        fmul.dd f18,f16,f18
        fmul.dd f22,f18,f16
        fsub.dd f20,f16,f16
        addu    r20,r19,r18
        fmul.dd f16,f18,f16
        addu    1,r21,r21
        subs    r21,r16,r0      // cc:r21 < r16  !cc:r21 >= r16
        fst.d   f16,0(r18)
        bc.t    .L.LL001.4
        shl     3,r21,r20
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XVRECP.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dvrecp |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|void dvrecp(n,x,y)
// 2|int n;
// 3|double *x,*y;
// 4|{
_dvrecp::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r21
        subs    r21,r16,r0
        bnc     .L.L001.3
.L.L001.2:
// 8|     y[i] = ((double) 1.0) / x[i];
        shl     3,r21,r20
.L.LL001.4:
        addu    r20,r17,r18     addu    1,r21,r21       fmov.ss f0,f20  orh     16384,r0,r31
        fld.d   0(r18),f22      ixfr    r31,f21
        addu    r20,r19,r18     frcp.dd f22,f18
        fmul.dd f22,f18,f16
        fsub.dd f20,f16,f16
        fmul.dd f18,f16,f18
        fmul.dd f22,f18,f16
        fsub.dd f20,f16,f16
        fmul.dd f18,f16,f18
        fmul.dd f22,f18,f16
        fsub.dd f20,f16,f16
        fmul.dd f16,f18,f16
        fst.d   f16,0(r18)
        nop
        subs    r21,r16,r0
        bc.t    .L.LL001.4
        shl     3,r21,r20
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XVSUB.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dvsub |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|void dvsub(n,x,y,z)
// 2|int n;
// 3|double *x,*y,*z;
// 4|{
_dvsub::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        bnc     .L.L001.3
.L.L001.2:
// 8|     z[i] = x[i] - y[i];
        shl     3,r22,r20
.L.LL001.4:
        addu    r20,r17,r18
        fld.d   0(r18),f18
        addu    r20,r19,r18
        fld.d   0(r18),f16
        addu    r20,r21,r18
        fsub.dd f18,f16,f16
        addu    1,r22,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        fst.d   f16,0(r18)
        bc.t    .L.LL001.4
        shl     3,r22,r20
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

```
        .file   "XVSUB.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dvsub |-----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|void dvsub(n,x,y,z)
// 2|int n;
// 3|double *x,*y,*z;
// 4|{
_dvsub::
// 5|   int i;
// 6|
// 7|   for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0
        bnc     .L.L001.3
.L.L001.2:
// 8|     z[i] = x[i] - y[i];
        shl     3,r22,r20
.L.LL001.4:
        addu    r20,r17,r18         addu    1,r22,r22
        fld.d   0(r18),f18
        addu    r20,r19,r18
        fld.d   0(r18),f16
        fsub.dd f18,f16,f16         addu    r20,r21,r18
        fst.d   f16,0(r18)
        nop
        subs    r22,r16,r0
        bc.t    .L.LL001.4
        shl     3,r22,r20
.L.L001.3:
// 9|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//10|
```

Operácoes Triad

```
        .file   "DAXPY.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| daxpy |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|daxpy(n,alpha,x,y)
// 2|int n;
// 3|double *alpha;
// 4|double *x;
// 6|double *y;
// 8|{
_daxpy::
// 9|  int i;
//10|
//11|  for (i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        bnc     .L.L001.3
.L.L001.2:
//12|    y[i] = (*alpha * x[i]) + y[i];
        shl     3,r22,r21
.L.LL001.4:
        addu    r21,r18,r19
        fld.d   0(r19),f18
        fld.d   0(r17),f16
        addu    r21,r20,r19
        fmul.dd f16,f18,f16
        fld.d   0(r19),f18
        fadd.dd f18,f16,f16
        addu    1,r22,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        fst.d   f16,0(r19)
        bc.t    .L.LL001.4
        shl     3,r22,r21
.L.L001.3:
//13|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//14|
//15|
```

```
        .file   "DAXPY.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| daxpy |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|daxpy(n,alpha,x,y)
// 2|int n;
// 3|double *alpha;
// 4|double *x;
// 6|double *y;
// 8|{
_daxpy::
// 9|  int i;
//10|
//11|  for (i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0
        bnc     .L.L001.3
.L.L001.2:
//12|    y[i] = (*alpha * x[i]) + y[i];
        shl     3,r22,r21
.L.LL001.4:
        addu    r21,r18,r19     addu    1,r22,r22       fld.d   0(r17),f16
        fld.d   0(r19),f18
        addu    r21,r20,r19     fmul.dd f16,f18,f16
        fld.d   0(r19),f18
        fadd.dd f18,f16,f16
        fst.d   f16,0(r19)
        nop
        subs    r22,r16,r0
        bc.t    .L.LL001.4
        shl     3,r22,r21
.L.L001.3:
//13|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//14|
//15|
```

```
        .file   "DSVMVT.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsvmvt |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|
// 2|dsvmvt(n,alpha,x,y,z)
// 3|int n;
// 4|double *alpha;
// 5|double *x;
// 6|double *y;
// 7|double *z;
// 8|{
// 9|
//10|
//11|
_dsvmvt::
//12|  int i;
//13|
//14|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        bnc     .L.L001.3
.L.L001.2:
//15|    z[i] = (*alpha - x[i]) * y[i];
        shl     3,r23,r21
.L.LL001.4:
        addu    r21,r18,r19
        fld.d   0(r19),f18
        fld.d   0(r17),f16
        addu    r21,r20,r19
        fsub.dd f16,f18,f16
        fld.d   0(r19),f18
        addu    r21,r22,r19
        fmul.dd f18,f16,f16
        addu    1,r23,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        fst.d   f16,0(r19)
        bc.t    .L.LL001.4
        shl     3,r23,r21
.L.L001.3:
//16|
//17|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//18|
```

```
        .file   "DSVMVT.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsvmvt |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|
// 2|dsvmvt(n,alpha,x,y,z)
// 3|int n;
// 4|double *alpha;
// 5|double *x;
// 6|double *y;
// 7|double *z;
// 8|{
// 9|
//10|
//11|
_dsvmvt::
//12|  int i;
//13|
//14|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0
        bnc     .L.L001.3
.L.L001.2:
//15|    z[i] = (*alpha - x[i]) * y[i];
        shl     3,r23,r21
.L.LL001.4:
        addu    r21,r18,r19     addu    1,r23,r23       fld.d   0(r17),f16
        fld.d   0(r19),f18
        addu    r21,r20,r19     fsub.dd f16,f18,f16
        fld.d   0(r19),f18
        fmul.dd f18,f16,f16     addu    r21,r22,r19
        fst.d   f16,0(r19)
        nop
        subs    r23,r16,r0
        bc.t    .L.LL001.4
        shl     3,r23,r21
.L.L001.3:
//16|
//17|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//18|
```

```
        .file   "DSVPVT.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsvpvt |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|dsvpvt(n,alpha,x,y,z)
// 2|int n;
// 3|double *alpha;
// 4|double *x;
// 5|double *y;
// 6|double *z;
// 7|{
_dsvpvt::
// 8|  int i;
// 9|
//10|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        bnc     .L.L001.3
.L.L001.2:
//11|    z[i] = (*alpha + x[i]) * y[i];
        shl     3,r23,r22
.L.LL001.4:
        addu    r22,r18,r21
        fld.d   0(r21),f18
        fld.d   0(r17),f16
        addu    r22,r19,r21
        fadd.dd f16,f18,f16
        fld.d   0(r21),f18
        addu    r22,r20,r21
        fmul.dd f18,f16,f16
        addu    1,r23,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        fst.d   f16,0(r21)
        bc.t    .L.LL001.4
        shl     3,r23,r22
.L.L001.3:
//12|}
        bri     r1
        nop             //      delayed
        .oVhc2.3a =: 0
```

```
        .file   "DSVPVT.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsvpvt |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|dsvpvt(n,alpha,x,y,z)
// 2|int n;
// 3|double *alpha;
// 4|double *x;
// 5|double *y;
// 6|double *z;
// 7|{
_dsvpvt::
// 8|  int i;
// 9|
//10|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0
        bnc     .L.L001.3
.L.L001.2:
//11|    z[i] = (*alpha + x[i]) * y[i];
        shl     3,r23,r22
.L.LL001.4:
        addu    r22,r18,r21     addu    1,r23,r23       fld.d   0(r17),f16
        fld.d   0(r21),f18
        addu    r22,r19,r21     fadd.dd f16,f18,f16
        fld.d   0(r21),f18
        fmul.dd f18,f16,f16     addu    r22,r20,r21
        fst.d   f16,0(r21)
        nop
        subs    r23,r16,r0
        bc.t    .L.LL001.4
        shl     3,r23,r22
.L.L001.3:
//12|}
        bri     r1
        nop
        .oVhc2.3a =: 0
```

```
        .file   "DSVTSP.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsvtsp |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|
// 2|dsvtsp(n,alpha,beta,x,y)
// 3|int n;
// 4|double *alpha;
// 5|double *beta;
// 6|double *x;
// 8|double *y;
//10|{
_dsvtsp::
//11|  int i;
//12|
//13|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        bnc     .L.L001.3
.L.L001.2:
//14|    y[i] = (*alpha + x[i]) + *beta;
        shl     3,r23,r22
.L.LL001.4:
        addu    r22,r19,r20
        fld.d   0(r20),f18
        fld.d   0(r17),f16
        fadd.dd f16,f18,f18
        fld.d   0(r18),f16
        addu    r22,r21,r20
        fadd.dd f16,f18,f16
        addu    1,r23,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        fst.d   f16,0(r20)
        bc.t    .L.LL001.4
        shl     3,r23,r22
.L.L001.3:
//15|
//16|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//17|
```

```
        .file   "DSVTSP.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsvtsp |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|
// 2|dsvtsp(n,alpha,beta,x,y)
// 3|int n;
// 4|double *alpha;
// 5|double *beta;
// 6|double *x;
// 8|double *y;
//10|{
_dsvtsp::
//11|  int i;
//12|
//13|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0
        bnc     .L.L001.3
.L.L001.2:
//14|    y[i] = (*alpha + x[i]) + *beta;
        shl     3,r23,r22
.L.LL001.4:
        addu    r22,r19,r20     addu    1,r23,r23       fld.d   0(r17),f16
        fld.d   0(r20),f18
        addu    r22,r21,r20     fadd.dd f16,f18,f18
        fld.d   0(r18),f16
        fadd.dd f16,f18,f16
        fst.d   f16,0(r20)
        nop
        subs    r23,r16,r0
        bc.t    .L.LL001.4
        shl     3,r23,r22
.L.L001.3:
//15|
//16|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//17|
```

```
        .file   "DSVTVM.C"
        .text
        .align  8
_L00TEXT:
        .text; .align 4
//--------------| dsvtvm |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|
// 2|dsvtvm(n,alpha,x,y,z)
// 3|int n;
// 4|double *alpha;
// 5|double *x;
// 7|double *y;
// 9|double *z;
//11|{
_dsvtvm::
//12|  int i;
//13|
//14|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        bnc     .L.L001.3
.L.L001.2:
//15|    z[i] = *alpha * x[i] - y[i];
        shl     3,r23,r21
.L.LL001.4:
        addu    r21,r18,r19
        fld.d   0(r19),f18
        fld.d   0(r17),f16
        addu    r21,r20,r19
        fmul.dd f16,f18,f16
        fld.d   0(r19),f18
        addu    r21,r22,r19
        fsub.dd f16,f18,f16
        addu    1,r23,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        fst.d   f16,0(r19)
        bc.t    .L.LL001.4
        shl     3,r23,r21
.L.L001.3:
//16|
//17|}
        bri     r1
        nop             //      delayed
        .oVhc2.3a =: 0
//18|
```

```
        .file   "DSVTVM.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsvtvm |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|
// 2|dsvtvm(n,alpha,x,y,z)
// 3|int n;
// 4|double *alpha;
// 5|double *x;
// 7|double *y;
// 9|double *z;
//11|{
_dsvtvm::
//12|  int i;
//13|
//14|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0
        bnc     .L.L001.3
.L.L001.2:
//15|    z[i] = *alpha * x[i] - y[i];
        shl     3,r23,r21
.L.LL001.4:
        addu    r21,r18,r19     addu    1,r23,r23       fld.d   0(r17),f16
        fld.d   0(r19),f18
        addu    r21,r20,r19     fmul.dd f16,f18,f16
        fld.d   0(r19),f18
        fsub.dd f16,f18,f16     addu    r21,r22,r19
        fst.d   f16,0(r19)
        nop
        subs    r23,r16,r0
        bc.t    .L.LL001.4
        shl     3,r23,r21
.L.L001.3:
//16|
//17|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//18|
```

```
        .file   "DSVTVP.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsvtvp |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|dsvtvp(n,alpha,x,y,z)
// 2|int n;
// 3|double *alpha;
// 4|double *x;
// 6|double *y;
// 8|double *z;
//10|{
_dsvtvp::
//11|  int i;
//12|
//13|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        bnc     .L.L001.3
.L.L001.2:
//14|    z[i] = *alpha * x[i] + y[i];
        shl     3,r23,r21
.L.LL001.4:
        addu    r21,r18,r19
        fld.d   0(r19),f18
        fld.d   0(r17),f16
        addu    r21,r20,r19
        fmul.dd f16,f18,f16
        fld.d   0(r19),f18
        addu    r21,r22,r19
        fadd.dd f18,f16,f16
        addu    1,r23,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        fst.d   f16,0(r19)
        bc.t    .L.LL001.4
        shl     3,r23,r21
.L.L001.3:
//15|
//16|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//17|
```

```
        .file   "DSVTVP.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsvtvp |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|dsvtvp(n,alpha,x,y,z)
// 2|int n;
// 3|double *alpha;
// 4|double *x;
// 6|double *y;
// 8|double *z;
//10|{
_dsvtvp::
//11|  int i;
//12|
//13|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0
        bnc     .L.L001.3
.L.L001.2:
//14|    z[i] = *alpha * x[i] + y[i];
        shl     3,r23,r21
.L.LL001.4:
        addu    r21,r18,r19     addu    1,r23,r23       fld.d   0(r17),f16
        fld.d   0(r19),f18
        addu    r21,r20,r19     fmul.dd f16,f18,f16
        fld.d   0(r19),f18
        fadd.dd f18,f16,f16     addu    r21,r22,r19
        fst.d   f16,0(r19)
        nop
        subs    r23,r16,r0
        bc.t    .L.LL001.4
        shl     3,r23,r21
.L.L001.3:
//15|
//16|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//17|
```

```
        .file   "DSVVMT.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsvvmt |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|dsvvmt(n,alpha,x,y,z)
// 2|int n;
// 3|double *alpha;
// 4|double *x;
// 5|double *y;
// 6|double *z;
// 7|{
_dsvvmt::
// 8|  int i;
// 9|
//10|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        bnc     .L.L001.3
.L.L001.2:
//11|    z[i] = *alpha * (x[i] - y[i]);
        shl     3,r23,r22
.L.LL001.4:
        addu    r22,r18,r21
        fld.d   0(r21),f18
        addu    r22,r19,r21
        fld.d   0(r21),f16
        fsub.dd f18,f16,f16
        fld.d   0(r17),f18
        addu    r22,r20,r21
        fmul.dd f18,f16,f16
        addu    1,r23,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        fst.d   f16,0(r21)
        bc.t    .L.LL001.4
        shl     3,r23,r22
.L.L001.3:
//12|}
        bri     r1
        nop             //      delayed
        .oVhc2.3a =: 0
```

```
        .file   "DSVVMT.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsvvmt |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|dsvvmt(n,alpha,x,y,z)
// 2|int n;
// 3|double *alpha;
// 4|double *x;
// 5|double *y;
// 6|double *z;
// 7|{
_dsvvmt::
// 8|  int i;
// 9|
//10|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0
        bnc     .L.L001.3
.L.L001.2:
//11|    z[i] = *alpha * (x[i] - y[i]);
        shl     3,r23,r22
.L.LL001.4:
        addu    r22,r18,r21         addu    1,r23,r23
        fld.d   0(r21),f18
        addu    r22,r19,r21
        fld.d   0(r21),f16
        addu    r22,r20,r21         fsub.dd f18,f16,f16
        fld.d   0(r17),f18
        fmul.dd f18,f16,f16
        fst.d   f16,0(r21)
        nop
        subs    r23,r16,r0
        bc.t    .L.LL001.4
        shl     3,r23,r22
.L.L001.3:
//12|}
        bri     r1
        nop
        .oVhc2.3a =: 0
```

```
        .file   "DSVVPT.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsvvpt |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|dsvvpt(n,alpha,x,y,z)
// 2|int n;
// 3|double *alpha;
// 4|double *x;
// 5|double *y;
// 6|double *z;
// 7|{
_dsvvpt::
// 8|  int i;
// 9|
//10|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        bnc     .L.L001.3
.L.L001.2:
//11|    z[i] = *alpha - (x[i] + y[i]);
        shl     3,r23,r22
.L.LL001.4:
        addu    r22,r18,r21
        fld.d   0(r21),f18
        addu    r22,r19,r21
        fld.d   0(r21),f16
        fadd.dd f16,f18,f16
        fld.d   0(r17),f18
        addu    r22,r20,r21
        fsub.dd f18,f16,f16
        addu    1,r23,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        fst.d   f16,0(r21)
        bc.t    .L.LL001.4
        shl     3,r23,r22
.L.L001.3:
//12|}
        bri     r1
        nop
        .oVhc2.3a =: 0
```

```
        .file   "DSVVPT.C"
        .text
        .align  8
_L00TEXT:
        .text; .align 4
//--------------| dsvvpt |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|dsvvpt(n,alpha,x,y,z)
// 2|int n;
// 3|double *alpha;
// 4|double *x;
// 5|double *y;
// 6|double *z;
// 7|{
_dsvvpt::
// 8|  int i;
// 9|
//10|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0
        bnc     .L.L001.3
.L.L001.2:
//11|    z[i] = *alpha - (x[i] + y[i]);
        shl     3,r23,r22
.L.LL001.4:
        addu    r22,r18,r21         addu    1,r23,r23
        fld.d   0(r21),f18
        addu    r22,r19,r21
        fld.d   0(r21),f16
        addu    r22,r20,r21         fadd.dd f16,f18,f16
        fld.d   0(r17),f18
        fsub.dd f18,f16,f16
        fst.d   f16,0(r21)
        nop
        subs    r23,r16,r0
        bc.t    .L.LL001.4
        shl     3,r23,r22
.L.L001.3:
//12|}
        bri     r1
        nop
        .oVhc2.3a =: 0
```

```
        .file   "DSVVTM.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsvvtm |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|dsvvtm(n,alpha,x,y,z)
// 2|int n;
// 3|double *alpha;
// 4|double *x;
// 6|double *y;
// 8|double *z;
//10|{
_dsvvtm::
//11|  int i;
//12|
//13|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        bnc     .L.L001.3
.L.L001.2:
//14|    z[i] = *alpha - x[i] * y[i];
        shl     3,r23,r21
.L.LL001.4:
        addu    r21,r18,r19
        fld.d   0(r19),f18
        addu    r21,r20,r19
        fld.d   0(r19),f16
        fmul.dd f16,f18,f16
        fld.d   0(r17),f18
        addu    r21,r22,r19
        fsub.dd f18,f16,f16
        addu    1,r23,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        fst.d   f16,0(r19)
        bc.t    .L.LL001.4
        shl     3,r23,r21
.L.L001.3:
//15|
//16|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//17|
```

```
        .file   "DSVVTM.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsvvtm |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|dsvvtm(n,alpha,x,y,z)
// 2|int n;
// 3|double *alpha;
// 4|double *x;
// 6|double *y;
// 8|double *z;
//10|{
_dsvvtm::
//11|  int i;
//12|
//13|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0
        bnc     .L.L001.3
.L.L001.2:
//14|    z[i] = *alpha - x[i] * y[i];
        shl     3,r23,r21
.L.LL001.4:
        addu    r21,r18,r19         addu    1,r23,r23
        fld.d   0(r19),f18
        addu    r21,r20,r19
        fld.d   0(r19),f16
        addu    r21,r22,r19         fmul.dd f16,f18,f16
        fld.d   0(r17),f18
        fsub.dd f18,f16,f16
        fst.d   f16,0(r19)
        nop
        subs    r23,r16,r0
        bc.t    .L.LL001.4
        shl     3,r23,r21
.L.L001.3:
//15|
//16|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//17|
```

```
        .file   "DSVVTP.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsvvtp |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|dsvvtp(n,alpha,x,y,z)
// 2|int n;
// 3|double *alpha;
// 4|double *x;
// 6|double *y;
// 8|double *z;
//10|{
_dsvvtp::
//11|  int i;
//12|
//13|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        bnc     .L.L001.3
.L.L001.2:
//14|    z[i] = *alpha + x[i] * y[i];
        shl     3,r23,r21
.L.LL001.4:
        addu    r21,r18,r19
        fld.d   0(r19),f18
        addu    r21,r20,r19
        fld.d   0(r19),f16
        fmul.dd f16,f18,f16
        fld.d   0(r17),f18
        addu    r21,r22,r19
        fadd.dd f18,f16,f16
        addu    1,r23,r23
        subs    r23,r16,r0      // cc:r23 < r16  !cc:r23 >= r16
        fst.d   f16,0(r19)
        bc.t    .L.LL001.4
        shl     3,r23,r21
.L.L001.3:
//15|
//16|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//17|
```

```
        .file   "DSVVTP.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dsvvtp |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|dsvvtp(n,alpha,x,y,z)
// 2|int n;
// 3|double *alpha;
// 4|double *x;
// 6|double *y;
// 8|double *z;
//10|{
_dsvvtp::
//11|  int i;
//12|
//13|  for(i=0;i<n;i++)
        mov     r0,r23
        subs    r23,r16,r0
        bnc     .L.L001.3
.L.L001.2:
//14|    z[i] = *alpha + x[i] * y[i];
        shl     3,r23,r21
.L.LL001.4:
        addu    r21,r18,r19         addu    1,r23,r23
        fld.d   0(r19),f18
        addu    r21,r20,r19
        fld.d   0(r19),f16
        addu    r21,r22,r19         fmul.dd f16,f18,f16
        fld.d   0(r17),f18
        fadd.dd f18,f16,f16
        fst.d   f16,0(r19)
        nop
        subs    r23,r16,r0
        bc.t    .L.LL001.4
        shl     3,r23,r21
.L.L001.3:
//15|
//16|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//17|
```

```
        .file   "DVVMVT.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dvvmvt |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|dvvmvt(n,w,x,y,z)
// 2|int n;
// 3|double *w;
// 5|double *x;
// 7|double *y;
// 9|double *z;
//11|{
_dvvmvt::
//12|  int i;
//13|
//14|  for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        bnc     .L.L001.3
.L.L001.2:
//15|    z[i] = (w[i] - x[i]) * y[i];
        shl     3,r22,r20
.L.LL001.4:
        addu    r20,r17,r18
        fld.d   0(r18),f18
        addu    r20,r19,r18
        fld.d   0(r18),f16
        addu    r20,r21,r18
        fsub.dd f18,f16,f16
        fld.d   0(r18),f18
        addu    r20,r23,r18
        fmul.dd f18,f16,f16
        addu    1,r22,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        fst.d   f16,0(r18)
        bc.t    .L.LL001.4
        shl     3,r22,r20
.L.L001.3:
//16|
//17|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//18|
```

```
        .file   "DVVMVT.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dvvmvt |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|dvvmvt(n,w,x,y,z)
// 2|int n;
// 3|double *w;
// 5|double *x;
// 7|double *y;
// 9|double *z;
//11|{
_dvvmvt::
//12|  int i;
//13|
//14|  for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0
        bnc     .L.L001.3
.L.L001.2:
//15|    z[i] = (w[i] - x[i]) * y[i];
        shl     3,r22,r20
.L.LL001.4:
        addu    r20,r17,r18         addu    1,r22,r22
        fld.d   0(r18),f18
        addu    r20,r19,r18
        fld.d   0(r18),f16
        fsub.dd f18,f16,f16         addu    r20,r21,r18
        fld.d   0(r18),f18
        fmul.dd f18,f16,f16         addu    r20,r23,r18
        fst.d   f16,0(r18)
        nop
        subs    r22,r16,r0
        bc.t    .L.LL001.4
        shl     3,r22,r20
.L.L001.3:
//16|
//17|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//18|
```

```
        .file  "DVVPVT.C"
        .text
        .align  8
_L00TEXT:
        .text; .align  4
//--------------| dvvpvt |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|
// 2|dvvpvt(n,w,x,y,z)
// 3|int n;
// 4|double *w;
// 6|double *x;
// 8|double *y;
//10|double *z;
//12|{
_dvvpvt::
//13|  int i;
//14|
//15|  for(i=0;i<n;i++)
        mov    r0,r22
        subs   r22,r16,r0     // cc:r22 < r16  !cc:r22 >= r16
        bnc    .L.L001.3
.L.L001.2:
//16|    z[i] = (w[i] + x[i]) * y[i];
        shl    3,r22,r20
.L.LL001.4:
        addu   r20,r17,r18
        fld.d  0(r18),f18
        addu   r20,r19,r18
        fld.d  0(r18),f16
        addu   r20,r21,r18
        fadd.dd f16,f18,f16
        fld.d  0(r18),f18
        addu   r20,r23,r18
        fmul.dd f18,f16,f16
        addu   1,r22,r22
        subs   r22,r16,r0     // cc:r22 < r16  !cc:r22 >= r16
        fst.d  f16,0(r18)
        bc.t   .L.LL001.4
        shl    3,r22,r20
.L.L001.3:
//17|
//18|}
        bri    r1
        nop
        .oVhc2.3a =: 0
//19|
```

```
        .file   "DVVPVT.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//-------------| dvvpvt |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|
// 2|dvvpvt(n,w,x,y,z)
// 3|int n;
// 4|double *w;
// 6|double *x;
// 8|double *y;
//10|double *z;
//12|{
_dvvpvt::
//13|  int i;
//14|
//15|  for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0
        bnc     .L.L001.3
.L.L001.2:
//16|    z[i] = (w[i] + x[i]) * y[i];
        shl     3,r22,r20
.L.LL001.4:
        addu    r20,r17,r18         addu    1,r22,r22
        fld.d   0(r18),f18
        addu    r20,r19,r18
        fld.d   0(r18),f16
        fadd.dd f16,f18,f16         addu    r20,r21,r18
        fld.d   0(r18),f18
        fmul.dd f18,f16,f16         addu    r20,r23,r18
        fst.d   f16,0(r18)
        nop
        subs    r22,r16,r0
        bc.t    .L.LL001.4
        shl     3,r22,r20
.L.L001.3:
//17|
//18|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//19|
```

```
        .file   "DVVTVM.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dvvtvm |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|
// 2|dvvtvm(n,w,x,y,z)
// 3|int n;
// 4|double *w;
// 6|double *x;
// 8|double *y;
//10|double *z;
//12|{
_dvvtvm::
//13|  int i;
//14|
//15|  for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        bnc     .L.L001.3
.L.L001.2:
//16|    z[i] = w[i] * x[i] - y[i];
        shl     3,r22,r20
.L.LL001.4:
        addu    r20,r17,r18
        fld.d   0(r18),f18
        addu    r20,r19,r18
        fld.d   0(r18),f16
        addu    r20,r21,r18
        fmul.dd f16,f18,f16
        fld.d   0(r18),f18
        addu    r20,r23,r18
        fsub.dd f16,f18,f16
        addu    1,r22,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        fst.d   f16,0(r18)
        bc.t    .L.LL001.4
        shl     3,r22,r20
.L.L001.3:
//17|
//18|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//19|
```

```
        .file   "DVVTVM.C"
        .text
        .align  8
_L00TEXT:
        .text; .align 4
//-------------| dvvtvm |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|
// 2|dvvtvm(n,w,x,y,z)
// 3|int n;
// 4|double *w;
// 6|double *x;
// 8|double *y;
//10|double *z;
//12|{
_dvvtvm::
//13|  int i;
//14|
//15|  for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0
        bnc     .L.L001.3
.L.L001.2:
//16|    z[i] = w[i] * x[i] - y[i];
        shl     3,r22,r20
.L.LL001.4:
        addu    r20,r17,r18     addu    1,r22,r22
        fld.d   0(r18),f18
        addu    r20,r19,r18
        fld.d   0(r18),f16
        fmul.dd f16,f18,f16     addu    r20,r21,r18
        fld.d   0(r18),f18
        fsub.dd f16,f18,f16     addu    r20,r23,r18
        fst.d   f16,0(r18)
        nop
        subs    r22,r16,r0
        bc.t    .L.LL001.4
        shl     3,r22,r20
.L.L001.3:
//17|
//18|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//19|
```

```
        .file  "DVVTVP.C"
        .text
        .align  8
_L00TEXT:
        .text; .align  4
//--------------| dvvtvp |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|dvvtvp(n,w,x,y,z)
// 2|int n;
// 3|double *w;
// 5|double *x;
// 7|double *y;
// 9|double *z;
//11|{
_dvvtvp::
//12|  int i;
//13|
//14|  for(i=0;i<n;i++)
        mov    r0,r22
        subs   r22,r16,r0     // cc:r22 < r16  !cc:r22 >= r16
        bnc    .L.L001.3
.L.L001.2:
//15|    z[i] = w[i] * x[i] + y[i];
        shl    3,r22,r20
.L.LL001.4:
        addu   r20,r17,r18
        fld.d  0(r18),f18
        addu   r20,r19,r18
        fld.d  0(r18),f16
        addu   r20,r21,r18
        fmul.dd f16,f18,f16
        fld.d  0(r18),f18
        addu   r20,r23,r18
        fadd.dd f18,f16,f16
        addu    1,r22,r22
        subs   r22,r16,r0     // cc:r22 < r16  !cc:r22 >= r16
        fst.d  f16,0(r18)
        bc.t   .L.LL001.4
        shl    3,r22,r20
.L.L001.3:
//16|
//17|}
        bri    r1
        nop
        .oVhc2.3a =: 0
//18|
```

```
        .file   "DVVTVP.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dvvtvp |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|dvvtvp(n,w,x,y,z)
// 2|int n;
// 3|double *w;
// 5|double *x;
// 7|double *y;
// 9|double *z;
//11|{
_dvvtvp::
//12|  int i;
//13|
//14|  for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0
        bnc     .L.L001.3
.L.L001.2:
//15|    z[i] = w[i] * x[i] + y[i];
        shl     3,r22,r20
.L.LL001.4:
        addu    r20,r17,r18     addu    1,r22,r22
        fld.d   0(r18),f18
        addu    r20,r19,r18
        fld.d   0(r18),f16
        fmul.dd f16,f18,f16     addu    r20,r21,r18
        fld.d   0(r18),f18
        fadd.dd f18,f16,f16     addu    r20,r23,r18
        fst.d   f16,0(r18)
        nop
        subs    r22,r16,r0
        bc.t    .L.LL001.4
        shl     3,r22,r20
.L.L001.3:
//16|
//17|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//18|
        .file   "DVVVTM.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dvvvtm |----------------------: frame was never referenced.
//Stack allocation: Autos:0
// 1|dvvvtm(n,w,x,y,z)
// 2|int n;
// 3|double *w;
// 5|double *x;
// 7|double *y;
// 9|double *z;
//11|{
_dvvvtm::
//12|  int i;
```

```
//13|
//14| for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0
        bnc     .L.L001.3
.L.L001.2:
//15|   z[i] = w[i] - x[i] * y[i];
        shl     3,r22,r20
.L.LL001.4:
        addu    r20,r19,r18     addu    1,r22,r22
        fld.d   0(r18),f18
        addu    r20,r21,r18
        fld.d   0(r18),f16
        fmul.dd f16,f18,f16     addu    r20,r17,r18
        fld.d   0(r18),f18
        fsub.dd f18,f16,f16     addu    r20,r23,r18
        fst.d   f16,0(r18)
        nop
        subs    r22,r16,r0
        bc.t    .L.LL001.4
        shl     3,r22,r20
.L.L001.3:
//16|
//17|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//18|
//19|
```

```
        .file   "DVVVTM.C"
        .text
        .align  8
_L00TEXT:
        .text;  .align  4
//--------------| dvvvtm |----------------------: frame was never referenced.
//Stack allocation: Autos:0

// 1|dvvvtm(n,w,x,y,z)
// 2|int n;
// 3|double *w;
// 5|double *x;
// 7|double *y;
// 9|double *z;
//11|{
_dvvvtm::
//12|  int i;
//13|
//14|  for(i=0;i<n;i++)
        mov     r0,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        bnc     .L.L001.3
.L.L001.2:
//15|    z[i] = w[i] - x[i] * y[i];
        shl     3,r22,r20
.L.LL001.4:
        addu    r20,r19,r18
        fld.d   0(r18),f18
        addu    r20,r21,r18
        fld.d   0(r18),f16
        addu    r20,r17,r18
        fmul.dd f16,f18,f16
        fld.d   0(r18),f18
        addu    r20,r23,r18
        fsub.dd f18,f16,f16
        addu    1,r22,r22
        subs    r22,r16,r0      // cc:r22 < r16  !cc:r22 >= r16
        fst.d   f16,0(r18)
        bc.t    .L.LL001.4
        shl     3,r22,r20
.L.L001.3:
//16|
//17|}
        bri     r1
        nop
        .oVhc2.3a =: 0
//18|
//19|
```