Alocação Global de Registradores de Endereçamento para Referências a Vetores em DSPs

Guilherme de Lima Ottoni

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Alocação Global de Registradores de Endereçamento para Referências a Vetores em DSPs

Guilherme de Lima Ottoni[1]

Dezembro de 2002

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo (Orientador)
- Prof. Dr. Tomasz Kowaltowski
  Instituto de Computação – UNICAMP
- Prof. Dr. Roberto da Silva Bigonha
  Departamento de Ciência da Computação – UFMG
- Prof. Dr. Arnaldo Vieira Moura (suplente)
  Instituto de Computação – UNICAMP

[1]**Auxílio financeiro do CNPq (processos 130739/2000-6 e 141958/2000-6) e da CAPES.**

Alocação Global de Registradores de Endereçamento para Referências a Vetores em DSPs

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Guilherme de Lima Ottoni e aprovada pela Banca Examinadora.

Campinas, 17 de dezembro de 2002.

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

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

Substitua pela folha com a assinatura da banca

Resumo

O avanço tecnológico dos sistemas computacionais tem proporcionado o crescimento do mercado de sistemas dedicados, cada vez mais comuns no dia-a-dia das pessoas, como por exemplo em telefones celulares, *palmtops* e sistemas de controle automotivo. Devido às suas características, estas novas aplicações requerem sistemas que aliem baixo custo, alto desempenho e baixo consumo de potência. Uma das maneiras de atender a estes requisitos é utilizando processadores especializados. Contudo, a especialização na arquitetura dos processadores impõe novos desafios para o desenvolvimento de *software* para estes sistemas. Em especial, os compiladores — geralmente responsáveis pela otimização de código — precisam ser adaptados para produzir código eficiente para estes novos processadores.

Na área de processamento de sinais digitais, como em telefonia celular, processadores especializados, denominados DSPs[2], são amplamente utilizados. Estes processadores tipicamente possuem poucos registradores de propósito geral e modos de endereçamento bastante limitados. Além disso, muitas das suas aplicações envolvem o processamento de grandes seqüências de dados, as quais são geralmente armazenadas em vetores. Como resultado, o estudo de técnicas de otimização de referências a vetores tornou-se um problema central em compilação para DSPs. Este problema, denominado *Global Array Reference Allocation* (GARA), é o objeto central desta dissertação. O sub-problema central de GARA consiste em se determinar, para um dado conjunto de referências a vetores que serão alocadas a um mesmo registrador de endereçamento, o menor custo das instruções que são necessárias para manter este registrador com o endereço adequado em cada ponto do programa. Nesta dissertação, este sub-problema é modelado como um problema em grafos, e provado ser NP-difícil. Além disso, é proposto um algoritmo eficiente, baseado em programação dinâmica, para resolver este sub-problema de forma exata sob certas restrições. Com base neste algoritmo, duas técnicas são propostas para resolver o problema de GARA. Resultados experimentais, obtidos pela implementação destas técnicas no compilador GCC, comparam-nas com outros resultados da literatura. Os resultados demonstram a eficácia das técnicas propostas nesta dissertação.

[2]Do original em inglês: *Digital Signal Processors*.

Abstract

The technological advances in computing systems have stimulated the growth of the embedded systems market, which is continuously becoming more ordinary in people's lives, for example in mobile phones, palmtops and automotive control systems. Because of their characteristics, these new applications demand the combination of low cost, high performance and low power consumption. One way to meet these constraints is through the design of specialized processors. However, processor specialization imposes new challenges to the development of software for these systems. In particular, compilers — generally responsible for code optimization — need to be adapted in order to produce efficient code for these new processors.

In the digital signal processing arena, such as in cellular telephones, specialized processors, known as DSPs (Digital Signal Processors), are largely used. DSPs typically have few general purpose registers and very restricted addressing modes. In addition, many DSP applications include large data streams processing, which are usually stored in arrays. As a result, studing array reference optimization techniques became an important task in compiling for DSPs. This work studies this problem, known as *Global Array Reference Allocation* (GARA). The central GARA subproblem consists of determining, for a given set of array references to be allocated to the same address register, the minimum cost of the instructions required to keep this register with the correct address at all program points. In this work, this subproblem is modeled as a graph theoretical problem and proved to be NP-hard. In addition, an efficient algorithm, based on dynamic programming, is proposed to optimally solve this subproblem under some restrictions. Based on this algorithm, two techniques to solve GARA are proposed. Experimental results, from the implementation of these techniques in the GCC compiler, compare them with previous work in the literature. The results show the effectiveness of the techniques proposed in this work.

"As dificuldades foram criadas para serem vencidas,
e nunca devemos desistir até que tenhamos
absoluta certeza de que somos incapazes."

Sebastião Ottoni

Agradecimentos

Em primeiro lugar, à minha esposa, Desirée, por todo o carinho e apoio, e também pela compreensão, especialmente durante os quase dois anos em que a distância se fez necessária para que eu pudesse buscar meus ideais.

Aos meus pais, Elias e Maria Lúcia, e irmãos, Gustavo, Tatiana e Frederico, por todo o afeto e apoio em todos os momentos da minha vida. À minha avó, Maria Christina, por sempre me incentivar na constante busca pelo conhecimento.

Ao meu orientador, professor Guido, pela oportunidade de trabalhar sob sua orientação, pela amizade, e sobretudo pela compreensão.

Aos professores, não apenas do IC-Unicamp, mas também do Instituto de Educação Juvenal Miller, do Colégio Técnico Industrial de Rio Grande, e da Fundação Universidade Federal do Rio Grande, que colaboraram de alguma maneira para a minha formação.

A todos os amigos que tive o prazer de conhecer no IC-Unicamp, em especial aos colegas ingressantes da pós-graduação em 2000 e aos meus contemporâneos do Laboratório de Sistemas de Computação, Rodolfo, Márcio, Marcus, Sandro, Felipe e Borin, e à agregada do LSC, Juliana. Os momentos de descontração são certamente indispensáveis, apesar de às vezes atrasarem um pouco o trabalho. :-)

Aos amigos e colegas de longa data do CTI e da FURG, por toda a amizade e incentivo.

Aos meus colegas de equipe nas Maratonas de Programação da ACM/SBC, Barata, Elbio, Piuí, Vinícius e Ulisses, e também aos amigos de outras universidades que conheci nestes eventos, em especial ao Reuber, ao Pedro e ao Pil. E ainda, aos organizadores desta competição que, mesmo às vezes de forma um pouco precária, incentivaram a mim e a muitos outros alunos a gostar ainda mais de Computação, proporcionando uma forma de os alunos mostrarem seus conhecimentos e habilidades. Tenho consciência de que muitas oportunidades me foram oferecidas, em parte, pelas minhas participações nestas competições.

Ao CNPq e à CAPES, pelo auxílio financeiro para a realização deste trabalho.

A todos vocês, registro aqui a minha mais sincera gratidão.

A todos os jovens — não apenas de nosso País, mas de todo o mundo — que por razões sócio-econômicas não tiveram as mesmas oportunidades de estudo que eu tive. Certamente muitos destes são mais capazes do que eu, e me entristece e envergonha muito saber que estas pessoas não tiveram a chance de mostrar isso.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Nos últimos anos, a Computação tem progressivamente se transferido de sistemas computacionais tradicionais (computadores de grande porte ou pessoais) para sistemas computacionais dedicados, incluindo telefones celulares, *palmtops* e equipamentos de controle automotivo. Esta nova gama de aplicações, que tem tornado a Computação cada vez mais onipresente, traz também novos desafios à área. Novos sistemas de *software* e *hardware* precisam ser projetados de acordo com as necessidades e restrições de cada aplicação ou classe de aplicações.

Muitos destes novos nichos de mercado englobam aplicações que impõem várias restrições críticas de desempenho e custo. Por outro lado, na maioria das vezes, estas aplicações executam em condições de energia limitada (geralmente com bateria), o que impõe fortes restrições quanto ao consumo de energia. Do ponto de vista de sua arquitetura, estes sistemas requerem a adoção de processadores especializados, capazes de atender a estas restrições. Unidades funcionais típicas de processadores de propósito geral pouco utilizadas em determinado domínio de aplicações freqüentemente não são incluídas, e dão lugar a unidades específicas para acelerar operações comuns neste domínio.

Um outro fato importante que vem afetando a maneira como sistemas dedicados são projetados é a crescente demanda por novas aplicações. Isto vem resultando em uma grande procura por ferramentas automatizadas que auxiliem no projeto e no desenvolvimento tanto do *hardware* quanto do *software* destes sistemas. Em particular, compiladores — amplamente utilizados no desenvolvimento de *software* para processadores tradicionais — tornam-se ainda mais importantes para estes novos sistemas, por conta de alguns fatores. Em primeiro lugar, devido à já mencionada especialização da arquitetura de processadores dedicados, a qual torna a programação direta em linguagem de montagem uma tarefa ainda mais árdua do que em processadores de propósito geral. Em segundo lugar, uma vez que os compiladores permitem a programação em linguagens de alto nível, eles aceleram o processo de desenvolvimento destes sistemas e aumentam a

portabilidade do *software*. Isto é particularmente importante devido aos ciclos de projeto de sistemas dedicados modernos serem extremamente curtos (6 meses).

Processadores dedicados trazem novos desafios à área de compiladores, mais especificamente à otimização e à geração de código. Técnicas tradicionais de otimização e geração de código [3], desenvolvidas para processadores de propósito geral, não são suficientes para a obtenção de código eficiente para processadores dedicados. Paulin *et al* [42] e Leupers [31] mencionam que os códigos gerados por compiladores para estes processadores são entre 400% e 1000% piores, tanto em tamanho quanto em tempo de execução, do que códigos correspondentes escritos diretamente em linguagem de montagem.

Alguns dos principais problemas relacionados à otimização de código para processadores dedicados provêm dos poucos modos de endereçamento tipicamente disponíveis nestes processadores. DSPs[1], que são processadores amplamente utilizados em sistemas dedicados de alto desempenho, geralmente não possuem modos de endereçamento elaborados, como o modo indexado. Isto torna o problema de endereçamento em DSPs ainda mais relevante do que em processadores de propósito geral, para os quais há pesquisas mostrando que instruções de endereçamento podem representar mais de 50% de todos os bits de um programa, e uma de cada seis instruções de uma aplicação de propósito geral típica [29].

Várias aplicações típicas de DSPs processam grandes seqüências de dados, quase sempre armazenadas na forma de vetores. Além disto, significativa parcela do tempo de execução destas aplicações é geralmente despendida em laços interiores, processando estes vetores. Este fato, aliado à escassez de modos de endereçamento em processadores dedicados, motivou o estudo de técnicas específicas de otimização para acesso a vetores dentro de laços. Araújo *et al* [6] foram os primeiros a estudar e formular este problema, que depois também foi abordado em [25, 33]. Contudo, todos estes trabalhos tratam o problema apenas para laços constituídos de um único bloco básico. Cintra e Araújo [19] foram os primeiros a propor uma solução para este problema quando laços contêm desvios condicionais.

O presente trabalho estende o estudo da alocação de referências a vetores dentro de laços contendo mais de um bloco básico, para processadores dedicados típicos. Uma introdução sobre unidades de cálculo de endereço nestes processadores, bem como uma revisão dos trabalhos relacionados, são apresentadas no capítulo 2. A maior parte das contribuições teóricas deste trabalho são apresentadas no capítulo 3, centrado em torno do artigo [41], publicado na *10th International Conference on Compiler Construction* (CC'01), realizada em Gênova, Itália, em abril de 2001. Um outro artigo, propondo uma solução ótima exata, apesar de exponencial, para o problema em questão, bem como apresentando resultados experimentais comparando este método com os trabalhos anteriores é o centro do capítulo 4. Este artigo foi recentemente submetido a uma edição especial

[1]Do original em inglês: *Digital Signal Processors*.

do *Micro-Electronics Journal*, a convite do editor desta edição, e é uma versão estendida do artigo [40] publicado no *IEEE Workshop on Embedded System Codesign 2002* (ESCODES'02), realizado em San Jose, EUA, em setembro de 2002. Uma formulação visando utilizar o método proposto em [41] para alocação global de registradores de propósito geral é proposta como trabalho futuro no capítulo 5. As conclusões deste trabalho são apresentadas no capítulo 6.

Capítulo 2

Trabalhos Relacionados

Devido às restrições de desempenho, custo e consumo de potência, processadores dedicados no geral, e DSPs em particular, possuem arquiteturas bastante especializadas. Uma especialização característica destes processadores é quanto aos modos de endereçamento à memória. Tipicamente, DSPs possuem unidades de endereçamento especializadas, sendo principalmente utilizado o modo de endereçamento indireto, no qual a instrução especifica um registrador de endereçamento que contém o endereço a ser lido ou escrito na memória. A ampla utilização deste modo de endereçamento, em detrimento de modos de endereçamento mais elaborados como o modo indexado, deve-se a alguns fatores. Primeiramente, o endereçamento indireto permite a codificação de instruções mais curtas, necessitando apenas de um campo na instrução para especificar um registrador de endereçamento, com número de bits igual ao logaritmo na base 2 do número de registradores de endereçamento do processador. No modo de endereçamento indexado, por outro lado, é necessário um campo adicional na codificação da instrução com tamanho suficiente para armazenar um deslocamento a ser somado ao valor contido no registrador base. Visto que a quantidade de memória em sistemas dedicados é, em geral, bastante limitada, a codificação mais compacta de instruções é muito mais utilizada. Um outro fator a favor do modo de endereçamento indireto é a execução mais rápida das instruções, uma vez que um estágio de *pipeline* que seria utilizado para o cálculo do endereço pode ser economizado, dado que o endereço efetivo é armazenado no registrador de endereçamento.

Todavia, a ausência de modos de endereçamento mais elaborados dificulta a geração de código eficiente por parte do compilador. Em primeiro lugar, a leitura/escrita eficiente de variáveis automáticas na pilha deixa de ser uma tarefa trivial. Em segundo lugar, também é dificultada a referência (leitura ou escrita) a elementos de vetores, especialmente dentro de laços. Estes dois problemas são conhecidos na literatura, respectivamente, como *Offset Assignment* e Alocação de Referências a Vetores (*Array Reference Allocation*). O presente trabalho estende pesquisas anteriores em Alocação de Referências a Vetores. Neste

capítulo, serão introduzidos os trabalhos anteriores que abordam estes dois problemas. Porém, antes será apresentada uma unidade típica de geração de endereços em DSPs.

2.1 Geração de Endereços em DSPs

As unidades funcionais especializadas na geração de endereços em DSPs são denominadas *Address Generation Units* (AGUs). Devido aos motivos já explicados, as AGUs geralmente possuem apenas modo de endereçamento indireto à memória. Visando diminuir o custo da alteração do conteúdo dos registradores de endereçamento (*Address Registers*, ARs), as AGUs dos DSPs geralmente implementam algum mecanismo para modificar o registrador de endereçamento durante a execução de uma instrução sem custo adicional. Este mecanismo, na grande maioria das AGUs, permite uma operação de soma ou subtração de um valor ao registrador de endereçamento utilizado pela instrução que está executando. Este tipo de computação é chamada de auto-incremento quando a operação feita é uma soma, e de auto-decremento quando é uma subtração. Este tipo de operação aritmética é codificada como parte de outras instruções que utilizam o *datapath*, eliminando a necessidade de instruções aritméticas explícitas quando for possível utilizá-las.

O valor que é somado/subtraído do AR pela AGU é denominado *offset*. Tanto o número de registradores de endereçamento quanto o tamanho do *offset* podem variar de processador para processador. Um valor de *offset* para auto-incremento e auto-decremento muito utilizado é 1, visto que há um grande número de acessos seqüenciais nos programas escritos para DSPs. Mas há processadores que, como o ADSP-210x, possuem 4 registradores de endereçamento e *offset* de até 4.

Na figura 2.1, pode-se observar uma AGU com um banco de registradores de endereçamento. Neste tipo de AGU, é possível fazer operações de auto-incremento e auto-decremento (em paralelo à computação), operações de subtração ou soma de um imediato a um AR, e operações de carga de um AR. As operações com imediato e de carga, por necessitarem de mais bits para serem codificadas, não podem ser codificadas juntamente com outras operações em uma única instrução. Assim, elas deixam ociosos recursos do *datapath* enquanto executam, e portanto consomem ciclos adicionais de CPU.

É possível encontrar também, em algumas AGUs, um banco de registradores chamados de *modify registers* (MRs). Estes registradores são responsáveis por conter *offsets* de valores diferentes. Operações do tipo soma ou subtração de um AR com um MR, chamadas de operações *auto-modify*, também não consomem recursos do *datapath*, consumindo somente recursos da AGU.

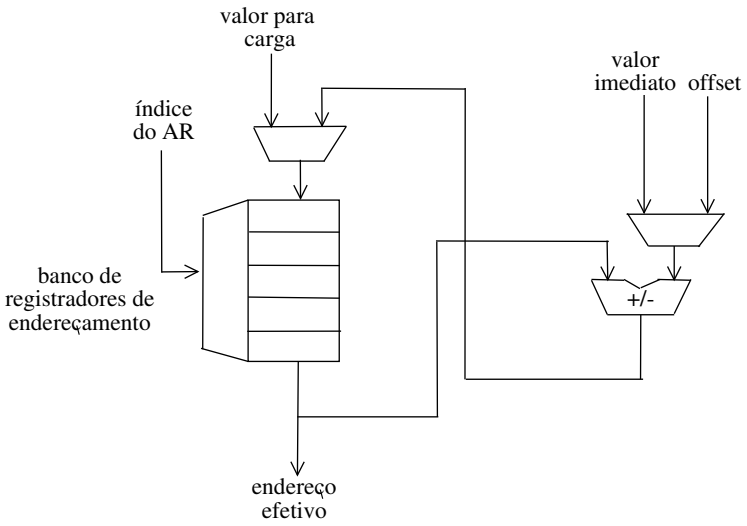

Figura 2.1: Modelo de uma AGU típica.

2.2 *Offset Assignment*

Em processadores dedicados, como DSPs, nos quais há poucos registradores de propósito geral e o principal modo de endereçamento disponível é o indireto com auto-incremento e auto-decremento, o problema de se determinar a disposição das variáveis automáticas escalares na pilha é bastante importante, pois tem uma forte relação com um aumento/redução do código de acesso a estas variáveis. Isto porque, por meio de auto-incremento e auto-decremento, acessos consecutivos a variáveis próximas na pilha podem ser feitos sem a necessidade de instruções explícitas de cálculo de endereço. Este problema é conhecido como *Offset Assignment*, e foi formulado e resolvido originalmente por Bartley [8], que mostrou que diferentes disposições das variáveis automáticas na pilha podem resultar em códigos (para acessá-las) com grandes diferenças de desempenho. Bartley modelou este problema como um problema em grafos, definindo o *grafo de acessos*. Neste grafo, a cada variável automática escalar é associado um vértice, e entre cada par de vértices u e v há uma aresta (u, v), não orientada, cujo peso é dado pelo número de acessos consecutivos às variáveis correspondentes a estes dois vértices.

Posteriormente, Liao *et al* [35] provaram que este problema, para o caso particular de um único registrador de endereçamento – *Simple Offset Assignment* (SOA) – é NP-difícil. Eles também propuseram uma solução alternativa, baseada em uma adaptação do algoritmo de Kruskal para o problema de árvore geradora de custo mínimo, e propuseram uma solução para o problema de *Offset Assignment* no caso de múltiplos registradores de endereçamento, denominado *General Offset Assignment* (GOA). A solução apresentada

para GOA, com k registradores de endereçamento, consiste em particionar as variáveis escalares automáticas em até k conjuntos, e aplicar SOA em cada um deles. Posteriormente, vários outros trabalhos abordaram o problema de SOA, destacando-se entre eles [48, 34, 45, 7].

2.3 Alocação de Referências a Vetores

O problema de alocação global de referências a vetores dentro de laços para arquiteturas de propósito geral foi estudado em [14, 10]. Nestes dois trabalhos, os elementos dos vetores são alocados a registradores de propósito geral. À medida que as iterações do laço são executadas, os elementos dos vetores são movidos entre registradores, como em um *pipeline*. Com isto, apenas algumas poucas instruções de load e move são necessárias a cada iteração. Contudo, a grande maioria dos DSPs possui apenas uns poucos registradores especializados, o que torna esta técnica impraticável para estas arquiteturas.

2.3.1 Alocação Local de Referências a Vetores

O problema de alocação de referências a vetores dentro de laços em DSPs foi originalmente proposto e resolvido por Araújo *et al* [6]. Esta formulação englobava apenas laços formados por um único bloco básico, ou seja, sem instruções de desvio no seu corpo. Por este motivo, esta formulação é o que se denomina Alocação Local de Referências a Vetores (*Local Array Reference Allocation* – LARA). Araújo *et al* [6] modelaram este problema como um problema em grafos, definindo o *Grafo de Indexação* (IG[1]). No IG, há um vértice correspondente a cada referência a vetor dentro de um laço, e existe uma aresta $(u \rightarrow v)$ se auto-incremento/decremento puder ser utilizado para redirecionar o registrador de endereçamento da referência correspondente a u para a correspondente a v. Além disto, uma aresta $(u \rightarrow v)$ só é inserida se a referência correspondente a u aparece antes da correspondente a v no código, uma vez que a aresta de retorno do laço é, a princípio, desconsiderada. Desta forma, um caminho no IG corresponde à alocação do mesmo registrador de endereçamento a uma seqüência de referências a vetor. O problema então se torna obter uma cobertura por caminhos disjuntos em vértices do IG que minimize o número de caminhos. Este problema pode ser resolvido eficientemente usando-se um algoritmo para emparelhamento máximo em grafos bipartidos [11]. Cada caminho resultante da cobertura é atribuído a um registrador de endereçamento. Uma heurística de junção de caminhos é utilizada quando a cardinalidade da cobertura supera o número de registradores de endereçamento disponíveis no processador-alvo.

[1]Em inglês: *Indexing Graph*

Gebotys [25] estendeu esta abordagem, através de uma modelagem do problema em fluxo em redes, com o intuito de minimizar o número de instruções explícitas de atualização dos registradores de endereçamento, dado o número de registradores de endereçamento disponíveis no processador.

Posteriormente, Leupers *et al* [33] propuseram uma solução exata para LARA, baseada na técnica de *branch-and-bound*, a qual utiliza a solução proposta em [6] como limitante inferior na sua solução.

2.3.2 Alocação Global de Referências a Vetores

Alocação Global de Referências a Vetores (*Global Array Reference Allocation* – GARA) é a generalização de LARA para laços de programa contendo múltiplos blocos básicos. Assim como ocorre com a maioria dos problemas em otimização de código, a generalização de soluções locais para soluções globais no caso do problema de alocação de referências a vetores é bastante difícil, uma vez que passam a existir diversos *traces* de execução que o fluxo de controle pode seguir.

A primeira solução para GARA foi proposta por Cintra e Araújo [19]. Esta solução é baseada em heurísticas, dada a dificuldade inerente ao problema, uma vez que GARA é uma generalização de LARA, o qual é NP-difícil [5]. A presente dissertação estende este trabalho em vários pontos.

Cintra e Araújo [19] usam o conceito de *live range* para designar um conjunto de referências a vetor que são alocadas a um mesmo registrador de endereçamento. Inicialmente, cada referência a vetor do laço sendo otimizado é colocada separadamente em um *live range*. A seguir, é utilizado um algoritmo heurístico, denominado *Live Range Growth* (LRG), o qual faz a junção sucessiva de pares de *live ranges* até que o número total delas atinja o número de registradores de endereçamento disponíveis. Para decidir qual par de *live ranges* será unido a cada iteração de LRG, todas as combinações de pares dos *live ranges* correntes são avaliadas, e a que resulta em um menor custo é escolhida. O custo de um *live range* é medido pelo número de instruções explícitas de ajuste do registrador de endereçamento que são necessárias. Assim, o problema central desta técnica é a determinação, para um dado *live range*, do número destas instruções que são necessárias para manter o registrador de endereçamento apontando para a referência correta em todos os pontos do laço e em todos os possíveis *traces* de execução.

Para resolver este problema de determinar o custo de um *live range*, Cintra e Araújo [19] utilizam uma solução heurística, apesar de não provarem que este problema é NP-difícil. Na sua solução, eles transformam o laço com as referências do *live range* para uma forma de representação que eles definem como *Single Reference Form* (SRF). Nesta representação, baseada na *Static Single Assignment (SSA) Form* [22], funções ϕ são inseridas no início

dos blocos básicos que pertencem à fronteira de dominância iterada dos blocos básicos que contêm alguma referência do *live range*. As funções ϕ representam os pontos onde uma decisão deve ser tomada a respeito de qual elemento do vetor será apontado pelo registrador de endereçamento. Assim, *resolver* uma função ϕ significa escolher o elemento do vetor que será apontado pelo registrador de endereçamento no correspondente ponto do programa. Para fazer esta escolha, primeiramente é feita a *análise de referências* [20], que é uma análise de fluxo de dados parecida com *reaching definitions* [3]. A análise de referências produz, para cada função ϕ_i, duas listas de referências: (1) a das que alcançam (isto é, podem preceder) ϕ_i, e (2) a das que podem ser alcançadas (ou seja, podem suceder) ϕ_i. É importante notar que estas listas podem conter, inclusive, outras funções ϕ. Cintra e Araújo [19] sempre resolvem as funções ϕ em uma ordem arbitrária, a saber, de baixo para cima (*Tail-Head*) no código. Seguindo esta ordenação, a solução de cada função ϕ é escolhida dentre as referências nas suas listas produzidas pela análise de referências. Outras funções ϕ que já tenham sido resolvidas anteriormente são substituídas pelas suas soluções correspondentes; as demais funções ϕ são ignoradas. Para cada possível solução, é calculado o custo de redirecionar o registrador de endereçamento de/para as referências que precedem/sucedem a função ϕ que está sendo resolvida. Modos de endereçamento com auto-incremento e auto-decremento são utilizados sempre que possível para ajustar o registrador de endereçamento sem a necessidade de uma instrução explícita. A solução que resultar no menor custo é escolhida.

Cintra [20] também fez experimentos considerando uma alternativa para a construção dos *live ranges* iniciais. Ao invés de colocar cada referência isoladamente em um *live range*, foi colocado em cada *live range* um caminho da cobertura por caminhos disjuntos do grafo de indexação, da mesma forma que havia sido feito para LARA em [6]. Todavia, os resultados apresentados em [20] demonstram que não houve ganhos com esta construção inicial de *live ranges*.

Capítulo 3

Computação do Custo de *Live Ranges*

Este capítulo é basicamente constituído do artigo [41] entitulado *"Optimal Live Range Merge for Address Register Allocation in Embedded Programs"*, apresentado na *10th International Conference on Compiler Construction* (CC'01), realizada em Gênova, Itália, em abril de 2001. Este artigo, de cunho mais teórico, estende o trabalho de Cintra e Araújo [19] da seguinte forma. Em primeiro lugar, o problema de se determinar o custo mínimo de instruções de ajuste do registrador de endereçamento para um *live range* é modelado como um problema em grafos e provado ser NP-difícil em geral. É introduzido o conceito de Φ-*Dependence Graph* (DG_ϕ), o qual modela as dependências entre as funções ϕ, e é proposto um algoritmo eficiente (polinomial), baseado em programação dinâmica, para encontrar uma solução ótima para as funções ϕ quando o DG_ϕ é acíclico. Por fim, é proposta uma nova técnica para alocação global de referências a vetores, também baseada em *live range growth*, combinando-se o algoritmo ótimo para DG_ϕ's acíclicos com a heurística *Tail-Head* proposta em [19].

Optimal Live Range Merge for Address Register Allocation in Embedded Programs

Guilherme Ottoni, Sandro Rigo, Guido Araujo,
Subramanian Rajagopalan and Sharad Malik

Abstract

The increasing demand for wireless devices running mobile applications has renewed the interest on the research of high performance low power processors that can be programmed using very compact code. One way to achieve this goal is to design specialized processors with short instruction formats and shallow pipelines. Given that it enables such architectural features, indirect addressing is the most used addressing mode in embedded programs. This paper analyzes the problem of allocating address registers to array references in loops using auto-increment addressing mode. It leverages on previous work, which is based on a heuristic that merges address register live ranges. We prove, for the first time, that the merge operation is NP-hard in general, and show the existence of an optimal linear-time algorithm, based on dynamic programming, for a special case of the problem.

1 Introduction

Complex embedded programs running on compact wireless devices has become a typical computer system nowadays. Although this trend is the result of a mature computing technology, it brings a new set of challenges. It has become increasingly hard to meet the stringent design requirements of such systems: low power consumption, high performance and small code size. Due to this design constraints, many embedded programs are written in assembly, and run on specialized embedded processors like *Digital Signal Processors* (DSPs) (e.g. ADSP 21000 [4]) or commercial CISC machines (e.g. Motorola 68000 [38]).

The increase in the size of embedded applications has put a lot of pressure towards the development of optimizing compilers for these architectures. One way to achieve this goal is to design specialized processors with short instruction formats and shallow pipelines. Since it allows such architectural improvements, indirect addressing is the most common addressing mode found in embedded programs. Again, due to cost constraints, the number of address registers available in the processor is fairly small, specially in DSPs (typically ≤ 8). This paper analyzes the problem of allocating address registers to array references in embedded program loops.

The register allocation problem has been extensively studied in the compiler literature [16, 13, 18, 28] for general-purpose processors, but not enough work has been done for the case of highly constrained embedded processors. *Local Reference Allocation* (LRA) is the problem of allocating address registers to array references in a basic block such that the number of address registers and instructions required to update them are minimized. LRA has been studied before in [6, 25, 33]. These are efficient graph-based solutions, when references are restricted to basic block boundaries. In particular, Leupers et al [33] is a very efficient solution to LRA. Unfortunately, not much work has been developed towards finding solutions to the global form of this problem, when array references are spread across different basic blocks, a problem we call *Global Reference Allocation* (GRA). The basic concepts required to understand GRA are described in Sect. 3. In [19] we proposed a solution to GRA, based on a technique that we call *Live Range Growth* (LRG). In Sect. 3 of this paper we give a short description of the LRG algorithm. In the current work, we extend our study of GRA in two ways. First, we perform the complexity analysis of the merge operation required by LRG, and prove it to be NP-hard (Sect. 4). Moreover, we prove the existence of an optimal linear-time algorithm for a special case of merge (Sect. 5). In Sect. 6 we evaluate the performance of this algorithm, and show that it can save update instructions. Section 7 summarizes the main results.

2 Problem Definition

Global Reference Allocation (GRA) is the problem of allocating address registers (`ar's`) to array references such that the number of simultaneously live address registers is kept below the maximum number of such registers in the processor, and the number of new instructions required to do that is minimized. In many compilers, this is done by assigning address registers to similar references in the loop. This results in efficient code, when traditional optimizations like induction variable elimination and loop invariant removal [3, 39] are in place. Nevertheless, assigning the same address register to different instances of the same reference does not always result in the best code. For example, consider the code fragment of Fig. 1(a). If one register is assigned to each reference, $a[i+1]$ and $a[i+2]$, two address registers are required for the loop. Now, assume that only one address register is available in the processor. If we rewrite the code from Fig. 1(a) using a single pointer `p`, as shown in Fig. 1(b), the resulting loop uses only one address register that is allocated to `p`. The cost of this approach is the cost of a pointer *update instruction* (`p += 1`) introduced on one of the loop control paths, which is considerably smaller than the cost of spilling/reloading one register. The C code fragment of Fig. 1(b) is a source level model of the *intermediate representation* code resulting after we apply the optimization described in the next sections.

```
(1) for (i = 0; i < N-2; i++) {
(2)   if (i % 2) {
(3)     avg += a[i+1] << 2;
(4)     a[i+2] = avg * 3;
(5)   }
(6)   if (avg < error)
(7)     avg -= a[i+1] - error/2;
(8)   else
(9)     avg -= a[i+2] - error;
(10) }
(11)
(12)
(13)
```

(a)

```
p = &a[1];
for (i = 0; i < N-2; i++){
  if (i % 2) {
    avg += *p++ << 2;
    *p-- = avg * 3;
  }
  if (avg < error)
    avg -= *p++ - error/2;
  else {
    p += 1;
    avg -= *p - error;
  }
}
```

(b)

Figure 1: (a) Code fragment; (b) Modified code that enables the allocation of one register to all references.

3 Basic Concepts

This section contains a short overview of some basic concepts that are required to formulate the GRA problem and to study its solutions. For a more detailed description, the reader should report to [19].

First of all assume, for the rest of this paper, that the array data type is a memory word (a typical characteristic of embedded programs). Moreover, assume that each array reference is atomic (i.e. encapsulated into a specific data type in the compiler intermediate representation), and array subscript expressions are not considered for *Common Subexpression Elimination* (CSE).

A central issue in GRA is to bound allocation to the number of address registers in the target processor. As a consequence, sometimes it is desirable that two references share the same register. This is done by inserting an instruction between the references or by using the auto-increment (decrement) mode. The possibility of using auto-increment (decrement) can be measured by the *indexing distance*, which is basically the distance between two index expressions of two array references.

The concept of live range used in this paper is a straightforward extension of idea of *variable liveness* adopted in the compiler literature [3]. The set of array references of a program can be divided into sets of control-flow paths, each of them defining a live range. For example, in Fig. 2(a) references are divided into live ranges R and S. In our notation for live range, little squares represent array references, and squares with the same color are

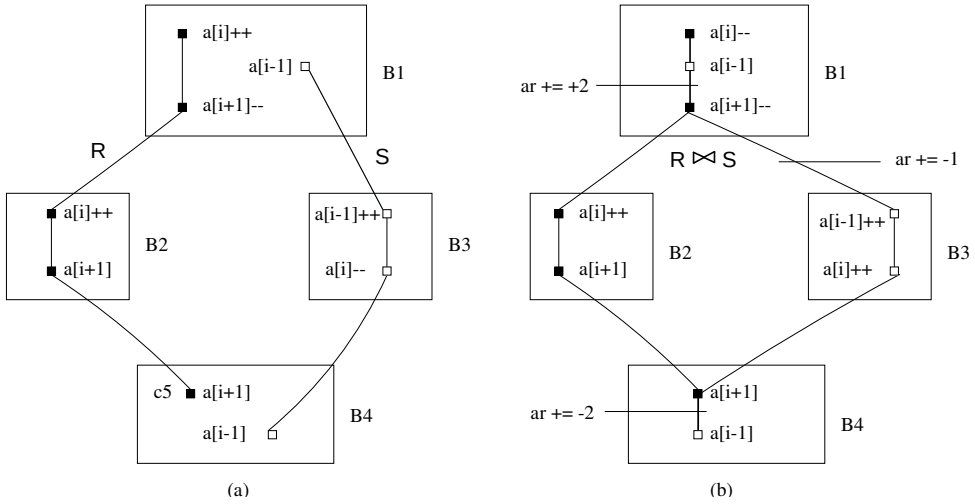

Figure 2: (a) Two live ranges R and S; (b) A single live range is formed after merging $R \bowtie S$.

in the same range. Moreover, an edge between two references of a live range indicates that the references are glued together through auto-increment mode or an update instruction. In Fig. 2, symbol "`++`" ("`--`") following a reference assigns a post-increment (decrement) mode to that reference. Notice that a live range can include references across distinct basic blocks.

As mentioned before, not much research work has been done towards finding solutions to the allocation of address registers for a whole procedure. A solution to this problem has been proposed recently in [19], that is based on repeatedly merging pairs of live ranges R and S, such that all references in the new range (called $R \bowtie S$) are allocated to the same register. The algorithm described in [19] starts by partitioning all references across a set of initial live ranges. The initial set of ranges($\mathcal{R}$) is formed by the individual references in the loop[1]. Ranges are merged pairwise until its number is smaller or equal to the number of address registers available in the processor. This technique is called *Live Range Growth* (LRG). Merge operations for similar problems have also been studied in [35, 23]. The cost of merging two ranges R and S ($cost_{\bowtie}(R, S)$) is the total number of cycles of the update instructions required by the merge. For example, after ranges R and S in Fig. 2(a) are merged, a single range $R \bowtie S$ results (Fig. 2(b)) which requires three update instructions. At each step of the algorithm the pair of ranges of minimum cost is merged.

[1]Each array access is assigned to a unique range.

3.1 The Merge Operator ($R \bowtie S$)

In order to enable references to share the same address register, the LRG algorithm needs to enforce, at compile time, that each reference b should be reached by only one reference a, since this allows the computation of the indexing distance $d(a, b)$ at compile time. As a consequence, the compiler can decide, at compile time, between using auto-increment (decrement) mode for a, or inserting an update instruction on the path from a to b. This requirement can be satisfied if the references in the CFG are in *Single Reference Form* (SRF) [19], a variation of *Static Single Assignment (SSA) Form* [22].

After a program is in SRF, any loop basic block for which array references converge will have a ϕ-equation, including the header and the tail of the loop. After reference analysis, which is performed by solving *reaching definitions* [3, 39] with the data items being the array references, any ϕ-equation has associated to it sets UD and DU, with elements $a_i \in UD$ and $b_j \in DU$. The value w that results from the ϕ-equation depends on how distant the elements in UD and DU are. We want to select a reference w that reduces the number of update instructions required to merge all a_i to all b_j. This can be done by substituting exhaustively w for all the elements in $UD \cup DU$, and measuring the cost due to the update instructions.

The problem of merging two live ranges can now be formulated as follows. Assume that, during some intermediate step in the LRG algorithm, two ranges R and S are considered for merge. Consider, for the following analysis, only those references which are in $R \cup S$. The set of ϕ-equations associated to these references forms a system of equations. The unknown variables in this system are virtual references w_k. These equations may have circular dependencies, caused basically by the following reasons: (a) it originates from a loop; (b) each ϕ-function depends on its sets UD_ϕ and DU_ϕ. Therefore, any optimal solution for variables w_k cannot always be determined exactly. Consider, for example, the CFG of Fig. 4(a) corresponding to the loop in Fig. 1(a), after ϕ-equations are inserted and reference analysis is performed. Three ϕ-equations result in blocks B_1, B_3 and B_6, namely:

$$s1_\phi : w_1 = \phi(w_3, a[i+1], w_2) \tag{3.1}$$

$$s2_\phi : w_2 = \phi(a[i+2], w_1, a[i+1], a[i+2]) \tag{3.2}$$

$$s3_\phi : w_3 = \phi(a[i+1], a[i+2], w_1) \tag{3.3}$$

The work in [19] assumed that the optimal merge of any two ranges is a NP-complete problem. For those instances of the problem in which the exact solution is too expensive, a heuristic should be applied. Any candidate heuristic must choose an evaluation order for the ϕ-equations such that, at each assignment $w_k = \phi(.)$, any argument of ϕ that is a virtual reference is simply ignored. For example, during the estimate of value w_3, in

Equation 3.3, the argument w_1 is ignored and the resulting assignment becomes $w_3 = \phi(a[i+1], a[i+2])$, which results in $w_3 = a[i+1]$ (or $a[i+2]$, for the same cost). The final cost of merge is dependent on the order that the heuristic uses to evaluate the set of ϕ-equations. In the heuristic proposed in [19] (called *Tail-Head*), the first equation in the evaluation order is the one at the tail of the loop. From this point on, the heuristic proceeds backward from the tail of the loop up to the header evaluating each equation as it is encountered, and computing the new value of w_k, which is then used in the evaluation of future equations.

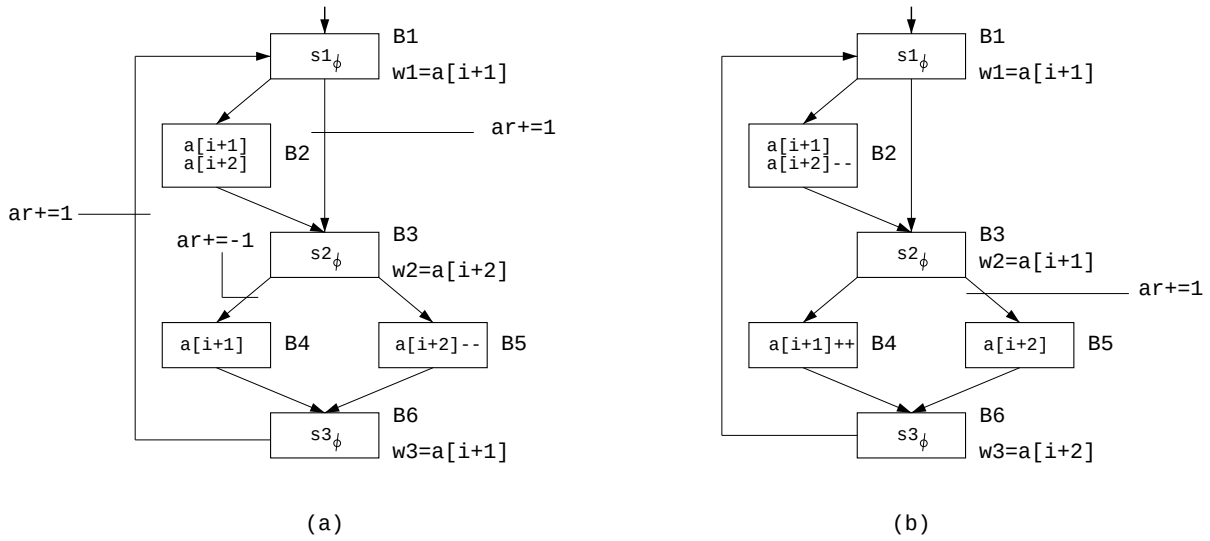

Figure 3: (a) Mode and update-instruction insertion for Tail-Head heuristic; (b) Optimal mode and update-instruction insertion using dynamic programming.

Example 1 Consider the application of this heuristic to the Equations 3.1-3.3, from the code fragment in Fig. 1(a). The result is shown in Fig. 3(a). First, the ϕ-equation $s3_\phi$ at the tail of the loop (block B_6) evaluates to $w_3 = a[i+1]$. This reference is equivalent to $a[i]$ in the next loop iteration. Next, $s2_\phi$ and $s1_\phi$ are evaluated, resulting in $w_2 = a[i+2]$ and $w_1 = a[i+1]$, respectively. At this point, all virtual references (i.e. the values of w_k) have been computed. In the last step, update instructions and auto-increment(decrement) modes are inserted into the code such that the references associated to the live ranges being merged satisfy the computed ϕ-functions. This results in the addition of auto-decrement mode to $a[i+2]$ in B_5, and the insertion of three update instructions: (a) $ar+ = 1$ on the edge (B_1, B_3); (b) $ar+ = 1$ on the edge (B_6, B_1); and (c) $ar+ = -1$ on the edge (B_3, B_4).

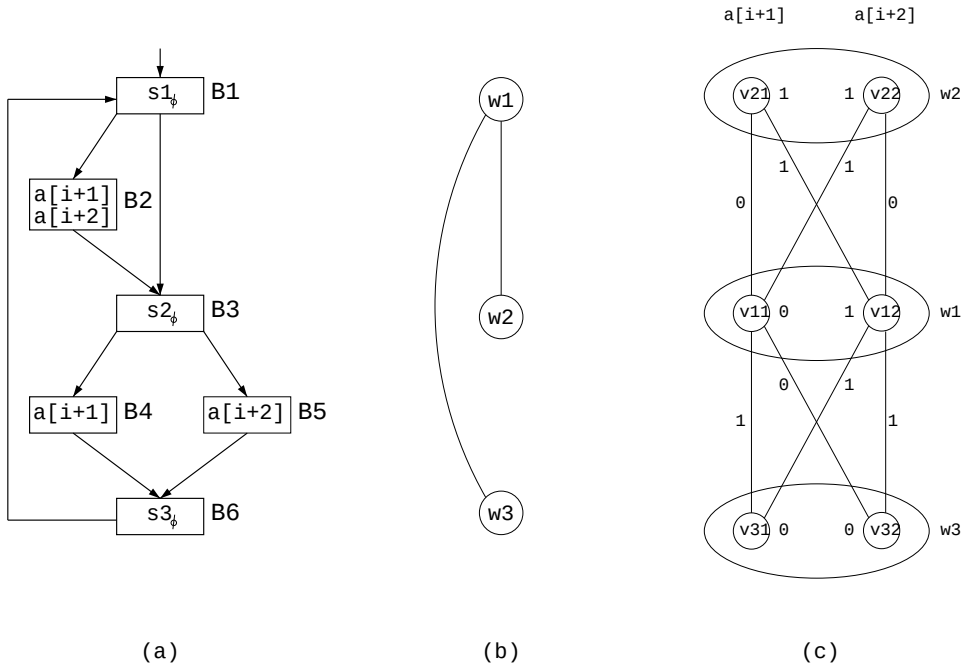

Figure 4: (a) The CFG in SRF; (b) The DG_ϕ representation; (c)The SG_ϕ for the DG_ϕ.

4 Complexity Analysis of $R \bowtie S$

Given that the merge operation is central to the LRG algorithm, in the following sections we perform the analysis of this operation. We complete the work started in [19] by proving that the merge operation (i.e., finding the minimal cost for it) is an NP-hard problem in general, through a reduction from the minimal vertex-covering problem. Moreover, we show that in a special case the merge operation admits an optimal solution, based on a dynamic programming algorithm. For the other cases, we show that a simple heuristic leads to effective solutions.

4.1 The Φ-Dependence Graph

The problem of solving the set of equations above can be formulated as a graph theoretical one as follows. Consider the code fragment of Fig. 4(a). We build a dependence graph where vertices are associated to the ϕ-equations and edges define dependencies between equations. This graph is called *ϕ-Dependence Graph* (DG_ϕ) and is constructed as follows:

Definition 1 *The DG_ϕ is an undirected graph for which there exists a vertex associated to each ϕ-equation, and there exist an edge (w_i, w_j), between two vertices w_i and w_j, if and only if w_i is a ϕ-function of w_j and vice-versa.*

Figure 4(b) shows the DG_ϕ corresponding to the equations in Fig. 4(a).

4.2 The Φ-Solution Graph

Having constructed the DG_ϕ for a set of ϕ-equations, we build a graph which reflects all possible solutions for these equations. But before doing so, we need to define some concepts. Let m be the number of distinct references that can result from the ϕ-equations. The range of references can be obtained by a simple code inspection, searching for the minimum and maximum values for each reference in the loop[2], which are stored into vector refs$[i], 1 \leq i \leq m$. Using these concepts we can now construct a graph that encodes all the possible solutions for the ϕ-equations.

Definition 2 *The Φ-Solution Graph (SG_ϕ) is an undirected graph constructed from the DG_ϕ as follows: (1) For each vertex u in DG_ϕ insert m vertices into SG_ϕ, one for each possible solution of the equation associated to u; (2) If there is an edge in DG_ϕ between two vertices u and v, and being sets $\{u_1, u_2, \ldots, u_m\}$ and $\{v_1, v_2, \ldots, v_m\}$ the vertices in the SG_ϕ associated with u and v respectively, insert edges (u_i, v_j) into SG_ϕ, for all $1 \leq i, j \leq m$.*

Figure 4(c) shows the SG_ϕ obtained from the DG_ϕ in Fig. 4(b). Notice that SG_ϕ is a multipartite graph, i.e. $\mathrm{SG}_\phi = \{P_1, P_2, \ldots, P_n\}$, with $P_i = \{v_{ij}, \mid 1 \leq j \leq m\}$. SG_ϕ is a weighted graph, with costs both on its vertices and edges. We assign a local cost $lcost(v_{ij})$ to each vertex v_{ij}, where $1 \leq i \leq n$ and $1 \leq j \leq m$, and define it as follows:

$$lcost(v_{ij}) = \sum_{j=1}^{m} cost(\mathrm{refs}[j], a) \ , \tag{3.4}$$

for each real reference a in $args[w_i]$, where $args[w_i]$ is the set of all arguments of the ϕ-function associated with w_i. The local cost is computed using the same cost function defined in [19] (which considers the possibility of using the auto-increment(decrement) modes), by assuming that $w_i = \mathrm{refs}[j]$ and considering only the arguments of ϕ_i which are real references (i.e. not w). In other words, *lcost* gives the contribution to the total merge cost when the solution for the ϕ-equation associated to w_i is $w_i = \mathrm{refs[j]}$. We also assign to each edge $e = (v_{ij}, v_{kl})$ in SG_ϕ, a global cost given by

$$gcost(v_{ij}, v_{kl}) = cost(\mathrm{refs}[j], \mathrm{refs}[l]) \ . \tag{3.5}$$

Notice that both arguments of *gcost* are virtual references, and reflect the cost of assigning solutions refs$[j]$ and refs$[l]$ respectively to virtual references w_i and w_k. Figure 4(c) illustrates the local and global costs associated with each SG_ϕ vertex and edge.

[2]It may be necessary to consider the loop increment.

Now we are able to state the solution of the ϕ-equation system in terms of the SG_ϕ. We want to determine the set $V = \{v_i | v_i \in P_i, 1 \leq i \leq n\}$ such that the total merge cost $cost_\bowtie[V] = cost_\bowtie(R, S)$ is minimum, where

$$cost_\bowtie[V] = \sum lcost(v_i) + \sum gcost(e) \ , \tag{3.6}$$

for all $1 \leq i \leq n$ and $e \in \mathrm{SG}_\phi[V]$, where $\mathrm{SG}_\phi[V]$ denotes the sub-graph of SG_ϕ induced by the vertices of V.

4.3 $R \bowtie S$ is NP-hard

In this subsection, we prove that the problem stated above is NP-hard. To do this, we restate it as a decision problem, and call it the *Minimum Cost Induced Sub-graph Problem (MCISP).*

Definition 3 (MCISP) *Let G be an instance of a SG_ϕ and k a positive integer. Find a subset V of the vertices of G (as described in Sect. 4.2) such that $cost_\bowtie[V] \leq k$.*

We show that MCISP is NP-hard through a reduction from the *Vertices Covering Problem* (VCP) [24], defined as follows.

Definition 4 (VCP) *Given a graph G and an integer k, decide whether G has a vertex-cover C with at most k vertices, such that each edge of G has at least one of its incident vertices in C.*

Theorem 1 *MCISP is NP-complete.*

Proof: First, it must be noticed that MCISP $\in$ NP. This is an easy task, as we can verify in polynomial-time if a given solution V has a total cost less than or equal to k. We will not give more details on this.

Now, for MCISP to be NP-complete, it is missing to show that it is NP-hard. To do this, we use a reduction from a known NP-complete problem to MCISP. We choose the vertex-cover problem (VCP) for this purpose.

Consider now the reduction of VCP, with instance graph G and integer k, to MCISP, with G' and k'. First we show how to create G' from G. For example, consider the graph G in Fig. 5(a). For each vertex $v \in G$, we insert two vertices v_0 and v_1 in G'. Then, for each edge $e = (u, v) \in G$, we insert four edges in G': (u_0, v_0), (u_0, v_1), (u_1, v_0) and (u_1, v_1).

Next, we add the costs to G', both to its vertices and edges, to reflect the local and global costs defined above. For each v_0 we make $lcost(v_0) = 0$, and for each vertex v_1 we set $lcost(v_1) = 1$.

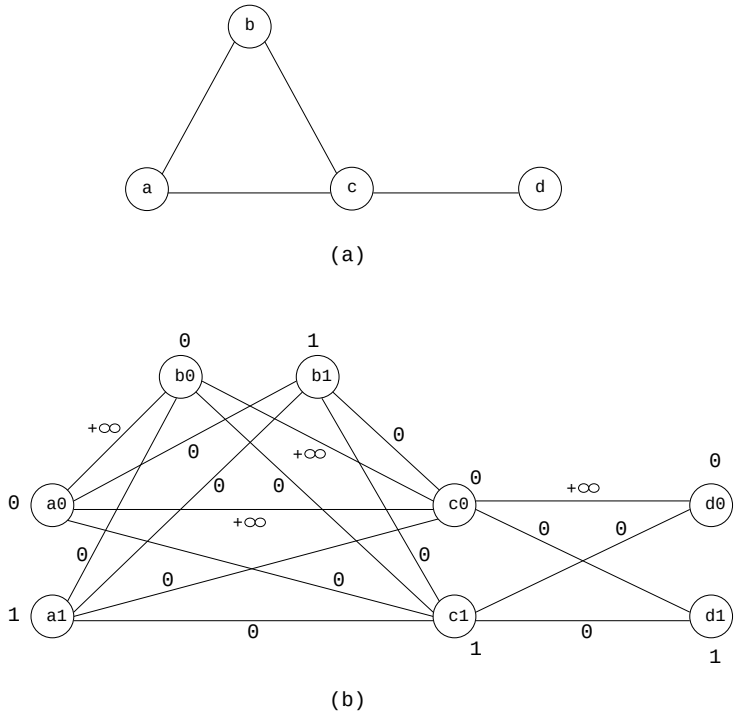

Figure 5: (a) VCP on original instance graph G; (b) MCISP on reduced instance graph G'.

On the edges, the following costs are assigned: $gcost(u_0, v_0) = +\infty$, and $gcost(u_0, v_1) = gcost(u_1, v_0) = gcost(u_1, v_1) = 0$. Fig. 5(b) illustrates the graph G' obtained from G in Fig. 5(a). It is clear that this reduction can be done in polynomial-time. Now we are going to prove that, having obtained G' from G as described above, and setting $k = k'$, VCP on G and k is satisfied if and only if MCISP on G' and k' is satisfied. Let's first see why the forward argument holds. Let C be a solution to VCP. So, for each edge in G, at least one of its incident vertices is in C. We argue that $V = V_1 \cup V_0$, where $V_1 = \{v_1 | v \in C\}$ and $V_0 = \{v_0 | v \notin C\}$ is a solution to MCISP. Let $q \leq k$ be the cost of C (i.e., the number of vertices). Consider now the cost of the solution V to MCISP. The first summation on Equation 3.6 is exactly the number of vertices in V_1, which is also the number of vertices in C (i.e. q). Now consider the second summation in Equation 3.6. As C is a solution to VCP, there are no two vertices $u_0, v_0 \in V$ such that the edge $(u_0, v_0) \in G'[V]$. Hence, every edge in $G'[V]$ has its *gcost* equal to 0, and $cost_{\bowtie}[V]$ is exactly q.

We still have to prove the backward argument, i.e., that given a solution V to MCISP with cost $q \leq k'$, there is a solution C to VCP with the same cost q. Being k' a finite integer number, $G'[V]$ has no edge between any two vertices u_0 and v_0. Therefore, choosing $C = \{v | v_1 \in V\}$, any edge in G will have at least one of its incident vertices in C, and so C is a vertex-cover for G.

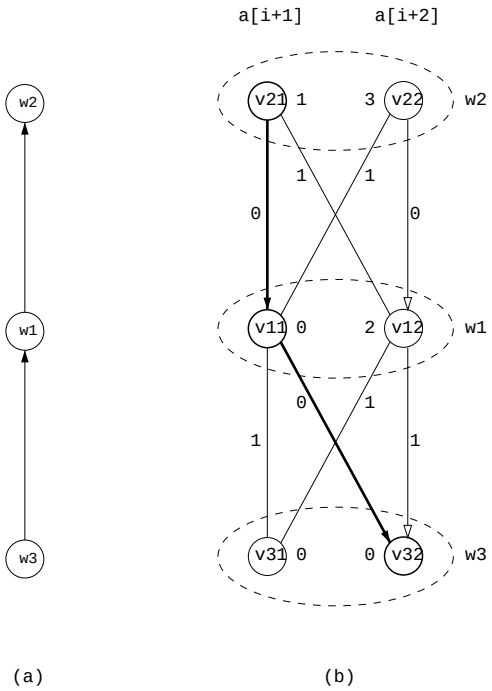

Figure 6: (a) The DDG_ϕ for the DG_ϕ of Fig. 4; (b) The DSGϕ for the DDG_ϕ in (a).

Now it is just missing to show that the cost of solution C to VCP is also q. This is easy to see, as the cost q is exactly the number of vertices in V with subscript 1, which is also the number of vertices in C. And so, the correctness of the polynomial-time reduction is proved.

Having proved the decision version of the problem to be NP-complete, and being the problem an optimization one, we conclude that the problem of finding the minimal cost for the merge operation is NP-hard.

□

5 The Case when DG_ϕ is a Tree

As shown in the previous section, the merge operation is NP-hard. In this section we present an optimal exact algorithm for merging two live ranges, under the assumption that the corresponding DG_ϕ is a tree. The algorithm is based on the dynamic programming technique, and uses a special order to traverse the nodes of the DG_ϕ. Ordering the traversal of the vertices actually means imposing a direction on each edge of the dependence graph. The arc $(u \rightarrow v)$ obtained by choosing a direction for the edge (u, v) implies that w_u is an argument of w_v, unlike in the previous form of the DG_ϕ, when edge (u, v)

indicated that w_v was also an argument of w_u. In this directed form of the DG_ϕ (called DDG_ϕ) the other arguments of the ϕ-functions are not changed.

Our algorithm is based on a particular traversal of the DDG_ϕ that we call *Leaves Removal Order* (LRO). This order is obtained by successively removing any leaf of the tree. If l is a leaf, it has a unique adjacent node, say v, such that the direction of (l, v) is set to $l \rightarrow v$. When l is removed, w_v is dropped out of the list of arguments of the ϕ-function for w_l.

This ordering algorithm can be implemented with time-complexity $O(n)$[3], for example by taking into account the degree of each vertex.

Figure 6(a) shows the DDG_ϕ obtained from the DG_ϕ of Fig. 4(a), using the algorithm just described above. We call the attention to the fact that, although we described this ordering algorithm separately, in practice it can be implemented on the same pass as the next step.

The next step in our solution is a dynamic programming algorithm which computes the minimum merge cost for any solution of our problem. We call this algorithm the *LRO Algorithm*, and show its pseudo-code in Alg. 1. We need a vector $accost[1, \ldots, m]$ for each vertex of DDG_ϕ, to hold the cumulative cost at this vertex, for each one of the m possible solutions of the corresponding ϕ-equation, from refs[1] to refs[m], respectively. In other words, for each vertex v of the DG_ϕ we assign an element $accost[i]$ to each one of its corresponding v_i vertices in SG_ϕ. The *accost* vectors are initialized with the respective *lcost* vectors. Then, following the LRO, for each leaf node l being removed, such that $l \rightarrow v$ is its last arc, we add the cost c_{v_i} to $accost_v[i]$, where c_{v_i} is the minimum value between $accost_l[j] + gcost(v_i, l_j)$, for all $1 \leq j \leq m$.

The minimum total merge cost is the minimum value in the DG_ϕ last vertex's *accost* vector. For simplicity, we omitted from Alg. 1 the code needed to recover the solution (i.e. the set $S = \{v_i | v_i \in P_i, 1 \leq i \leq n\}$), though it should be clear that it can be implemented without any extra computational complexity cost. The time-complexity of this algorithm is $O(n.m^2)$. In practice, since m is generally bounded by a small constant[4], the algorithm complexity reduces to $O(n)$.

Lemma 1 *Given a LRO sequence $S = (v_1, v_2, ..., v_n)$ of DG_ϕ's vertices, the LRO Algorithm properly computes the minimal value for $accost[v_{ij}]$, for every $1 \leq i \leq n$ and $1 \leq j \leq m$.*

Proof: First, it must be noticed that, as long as DG_ϕ is a tree, there always exists an LRO. Moreover, the ϕ-functions are inserted such that the cost of every arc in the CFG

[3]Remember that n is the number of vertices of DG_ϕ

[4]Remember that m is the number of references (inside the loop) in the range defined by the one with the minimum and the one with the maximum index value.

Algorithm 1 LRO algorithm

```
(1) procedure LRO(DG_φ, SG_φ)
(2) for each vertex v_ij ∈ SG_φ, where 1 ≤ i ≤ n and 1 ≤ j ≤ m, do
(3)     accost_i[j] ← lcost(v_ij)
(4) for all but the last vertex v_p ∈ DG_φ, according to LRO, do
(5)     let (v_p,v_q) be the last edge incident to v_p
(6)     for j ← 1 to m do
(7)         min ← +∞
(8)         for i ← 1 to m do
(9)             if accost_p[i] + gcost(v_pi, v_qj) < min then
(10)                min ← accost_p[i] + gcost(v_pi, v_qj)
(11)        accost_q[j] ← accost_q[j] + min
```

affected by the ϕ-functions' values is considered exactly once. Let $S_k = (v_1, ..., v_k)$. We prove this lemma by induction in k.

Basis: $k = 1$. In this case, $accost[v_{1j}]$, $1 \leq j \leq m$, is just equal to $lcost(v_{1j})$, which is minimal, as all the $lcost$'s are chosen to be locally optimal.

Induction Hypothesis: We assume that, for every $i < k$, the LRO Algorithm computes the minimal values for $accost[v_{ij}]$.

Inductive Step: We define the set $A = \{v_i | i < k \text{ and } (v_i, v_k) \in \mathrm{DG}_\phi\}$. We now need to show how to compute the minimal value for each $accost[v_{kj}]$, $1 \leq j \leq m$. The $lcost(v_{kj})$ is always in $accost[v_{kj}]$. So we have to minimize

$$\sum_{v_i \in A} min\{accost[v_{iy}] + gcost(v_{kj}, v_{iy}) | 1 \leq y \leq m\}$$

for each $1 \leq j \leq m$. $gcost(v_{kj}, v_{iy})$ is a precalculated constant. By the induction hypothesis, $accost[v_{iy}]$ is minimal, for each $v_i \in A$ and $1 \leq y \leq m$. So, it turns out that $accost[v_{kj}]$ is correctly computed, for each $1 \leq j \leq m$, as we minimize a value among the minimal values for all possible choices. And so the proof is complete, and the LRO Algorithm computes the minimal values for $accost[v_{kj}]$, for all valid values for k and j.

□

Theorem 2 *The LRO Algorithm is optimal when the DG_ϕ is a tree, in the sense that it results in the smallest possible cost for $R \bowtie S$.*

Proof: The Lemma 1 states that the LRO Algorithm computes the minimum cost for $accost[v_{ij}]$, for all vertices $v_{ij} \in \mathrm{SG}_\phi$, if the corresponding DG_ϕ is a tree. So, for

the algorithm to compute the smallest cost for $R \bowtie S$, we just have take the minimum between $\{accost[v_{nj}] | v_{nj} \in \mathrm{SG}_\phi$, and the v_n is the last vertex in the LRO for $\mathrm{DG}_\phi\}$.

□

Example 2 Figure 6(b) shows the result of running this algorithm on the SG_ϕ from Fig. 4(c), according to the LRO from Fig. 6(a). In this figure, we show the global costs on the edges, and the final cumulative cost on each vertex. At each vertex v, we use arrows to point to the other vertices u (a single vertex in this example), such that u is used to reach the minimum value at v. The vertices and edges that compose the solution are highlighted in Fig. 6(b).

Figure 3(b) shows the code fragment of Fig. 1(a) with the modifications associated to the solution illustrated in Fig. 6(b). Notice that only one update instruction is necessary, which is in accordance with cost $accost[v_{21}]$ in Fig. 6(b). Figure 3(a) shows the solution found using the Tail-Head heuristic described in [19], for the instance of the problem shown in Fig. 1(a). This solution has a higher cost (3 update instructions).

6 Experimental Results

Our experimental framework is based on programs from the MediaBench benchmark [30], which are typically found in many embedded applications. We have implemented the LRG heuristic [19], and the LRO algorithm described in Sect. 5 into the IMPACT compiler [17]. Both algorithms run at `lcode` level, just before all standard intermediate representation optimizations [3]. At each merge in the LRG algorithm, the DG_ϕ graph is tested for the presence of cycles. If DG_ϕ is cyclic the Tail-Head (TH) heuristic is applied. On the other hand, if DG_ϕ is acyclic, the LRO algorithm is used to compute the ϕ-functions.

For each program, we measured the number of update instructions required by its inner loops [5], such that one address register is used to access all array references of each array (Table 6). This is a worst case scenario which reveals the improvement resulting from the LRO algorithm. This table also shows an estimate of the number of times when DG_ϕ is a tree during the live range growth process. These preliminary results indicate that this may be a frequent situation in embedded applications. We acknowledge that measuring the final cycle count would be more realistic, but unfortunately at this point we do not have a complete retarget of the IMPACT architecture to a real-world DSP processor. On the other hand, DSP compilers have few address registers available and some of them are used to perform other tasks (e.g. stack access), such that only very few are left for address register allocation. In this case, the above scenario approaches

[5] Only those loops which make access to arrays have been considered.

Program Name	Loop	Minimum No. ARs	% Trees	# Update Instr.	
				TH	LRO+TH
jpeg	1	2	100	4	3
	2	1	64	4	3
	3	2	46	4	4
epic	1	1	78	0	0
	2	1	100	3	3
	3	2	77	2	1
pgp	1	2	100	0	0
	2	1	100	1	1
mpeg	1	1	75	3	3
mesa	1	1	75	3	3
	2	1	100	2	2
pegwit	1	1	67	2	2
	2	2	100	1	0
gsm	1	2	100	0	0
	2	1	89	1	1

Table 1: Comparison between the Tail-Head (TH) and LRO+TH approaches.

what occurs in many real situations. Notice from Table 6, that the improvement due to the LRO algorithm is typically one less update instruction in the loop body, what considerably improves loop performance, particularly for tight inner loops found in many critical signal processing applications.

Even more relevant, these numbers also reveal that the TH heuristic that we proposed in [19] results in near optimal allocation.

7 Conclusions and Future Work

This paper addresses the problem of allocating address registers to array references in loops of embedded programs. It is based on a previous work that assigns address registers to live ranges by merging ranges together. The contributions of this work are two. First, it proves that the optimal merge of address register live ranges is an NP-hard problem. Second, it proves the existence of an optimal linear-time algorithm to merge live ranges when the dependency graph formed by the set of ϕ-equations on the references is a tree.

A full evaluation of this technique on a real-world DSP running a wide set of benchmarks is under way. We are also working to add support to new DSP features. One example is the support to modify registers. This kind of register is present in many DSPs

and we intend to use them to increment address registers by values greater than one, incurring in no additional cost. By doing so, we can avoid issuing some update instructions and, as a consequence, decrease the update instruction overhead.

8 Acknowledgments

This work was partially supported by CNPq (300156/97-9), ProTem CNPq/NSF Collaborative Research Project (68.0059/99-7) and by fellowship grants CNPq (141710/2000-4) and (141958/2000-6). We also thank the reviewers for their comments.

Capítulo 4

Solução Exata para GARA e Comparação com Métodos Anteriores

Este capítulo é constituído de um outro artigo, recentemente submetido a uma edição especial do *Micro-Electronics Journal*, a convite do editor desta edição. Trata-se de uma versão estendida do artigo [40], entitulado "*Efficient Array Reference Allocation for Loops in Embedded Processors*", apresentado no *IEEE Workshop on Embedded System Codesign 2002* (ESCODES'02), realizado em San Jose, EUA, em setembro de 2002. Neste artigo, é proposta uma solução ótima exata, apesar de exponencial, para o problema de alocação global de referências a vetores, sendo apresentados resultados experimentais comparando este método com os trabalhos anteriores. Esta solução exata utiliza, sempre que possível, o algoritmo proposto em [41]. Os resultados experimentais mostram que a proporção de Grafos de Dependências ϕ que são acíclicos é bastante elevada, tanto nesta solução exata quanto na heurística apresentada no capítulo 3. Além disto, é descrita também a *Extended Single Reference Form* (ESRF), uma extensão da *Single Reference Form* (SRF) [19] necessária para garantir a otimalidade da solução exata para DG_ϕ's acíclicos. A ESRF não havia sido publicada em [41] por restrições técnicas quanto ao espaço.

Address Register Allocation for Arrays in Loops of Embedded Programs

Guilherme Ottoni

Guido Araújo

Abstract

Efficient address register allocation has been shown to be a central problem in code generation for processors with restricted addressing modes. This paper extends previous work on *Global Array Reference Allocation* (GARA), the problem of allocating address registers to array references in loops. It describes two heuristics to the problem, based on the SSA Form, presenting experimental data to support them. In addition, it proposes an approach to solve GARA optimally which, albeit computationally exponential, is useful to measure the efficiency of other methods. Experimental results, using the MediaBench benchmark, reveal that the proposed heuristics can solve the majority of the benchmark loops near optimality in polynomial-time. A substantial execution time speedup is reported for the benchmark programs, after compiled with the original and the optimized versions of GCC.

1 Introduction

The increase in the size and complexity of embedded system applications has induced designers to adopt architectures that offer low power consumption, enhanced performance and reduced cost. Processors that run embedded programs range from commercial CISC machines (e.g. Motorola 68000) to specialized *Digital Signal Processors* (DSPs) (e.g. DSP16xx [37]), and encompass a considerable share of the processors produced every year.

Address computation takes a large fraction of the execution time for most programs. Addressing can account for over 50% of all program bits and 1 out of every 6 instructions for a typical general-purpose program [29]. In order to speedup address computation, most embedded processors offer specialized *addressing modes*. A typical example is the auto-increment (decrement) mode, which enables the encoding of very short instructions. All commercial DSPs and most CISC processors *Instruction Set Architectures* (ISAs) have auto-increment (decrement) modes. In fact, in order to reduce the instruction size, many embedded processors do not allow the typical base-register plus offset addressing mode

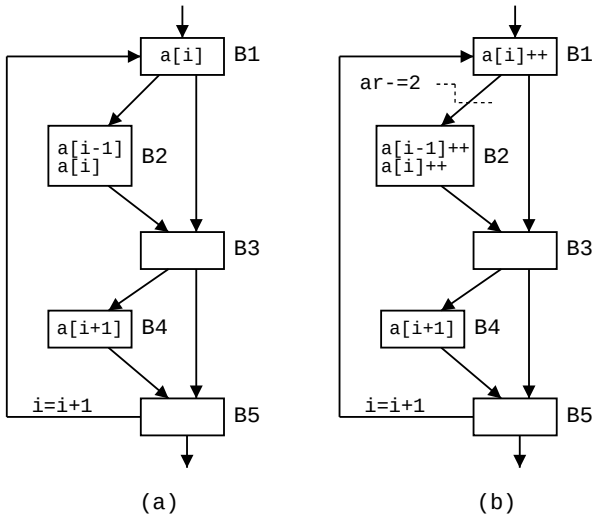

Figure 1: (a) CFG fragment; (b) Inserting auto-increment mode and update instructions.

frequently found in general-purpose architectures. Even worse, very few registers are available in these processors (typically 4-16), and addressing is usually performed only through specialized address register.

This paper extends previous work [19, 41] on *Global Array Reference Allocation* (GARA), which is the problem of allocating address registers to array references in loops running on embedded processors. As an example, consider the *Control-Flow Graph* (CFG) from Figure 1(a), where only the array references are shown. Solving GARA on this code, with a single address register available, produces the code in Figure 1(b). In Figure 1(b), all the array references are performed through the single address register (`ar`), and symbol `++` (`--`) following a reference implies that post-increment (post-decrement) addressing mode is used.

This paper describes two heuristics to GARA, presenting experimental data to support them. In addition, it proposes an approach to solve GARA optimally which, albeit computationally exponential, is useful to measure the efficiency of the other methods. The experimental results are very encouraging. Using the exact solution as the baseline, the experiments reveal that program loops in the MediaBench benchmark [30] can be solved near optimality in polynomial-time through the heuristics. An average speedup of 14.3% is reported for the benchmark programs after compiled with the original and optimized versions of GCC.

This paper is divided as follows. Section 2 lists the previous work on GARA. Section 3 describes, for the first time, the *Extended Single Reference Form* (ESRF), which is required to guarantee the optimality of the dynamic programming algorithm proposed in [41]. Two heuristics for GARA are summarized in Section 4, and Section 5 proposes a

method to compute its exact solution. Finally, Section 6 reports the experimental results when our implementation in GCC compiles MediaBench programs.

2 Previous Work

Register allocation is a well studied problem in compilers. Many of the first problems in code generation involved finding good algorithms for register allocation [47, 2, 46]. Global register allocation is an important problem in code generation which has been extensively studied [16, 13, 18, 28]. Other researchers have considered the interaction of register allocation and scheduling in code generation for RISC machines [27, 12], and inter-procedural register allocation [15]. The allocation of local variables to the stack-frame, using auto-increment (decrement) mode, has been studied in [32, 34, 8, 36, 45, 23].

Local Array Reference Allocation (LARA) is the problem of allocating address registers to array references in a basic block such that the number of address registers and instructions required to update them are minimized. LARA has been studied in [33, 6, 25], which are efficient graph-based solutions, when references are restricted to basic block boundaries. Global register allocation for array references, on general-purpose architectures, has been studied before by Bodik and Gupta [10] and Callahan et al [14]. In [10] and [14] array references are allocated to general-purpose registers. As the loop iteration progresses, references are moved among registers in a *pipelined* fashion. Unfortunately, many embedded processors are highly constrained architectures containing very few specialized registers, what makes the application of these techniques impossible.

In [19], a technique based on live range growth and a variation of *Static Single Assignment* (SSA) Form [22] was proposed. It consists on consecutively merging pairs of live ranges until the number of ranges equals the number of address registers in the target processor architecture. A heuristic was used to decide which pair of live ranges should be merged. The problem of finding the minimal number of update instructions when merging pairs of live ranges has been proved to be NP-complete in general [41]. The difficulty of the problem lies on choosing the best (minimum cardinality) set of update instructions among a combinatorial number of possible sets. The large number of sets results from the need to keep correct, on every possible execution path, the value of the address register for the array references on the merged live range. In [41], Ottoni et al. proves the existence of an optimal dynamic programming algorithm to find the minimal set of update instructions, for a special case of live range topology. Preliminary experimental results in [41] speculated that this particular topology would be very common in practice, although not enough benchmark data was presented to support that. The experimental results from Section 6 confirm this hypothesis to be true for the MediaBench programs.

3 The Extended Single Reference Form (ESRF)

Any approach that aims at solving the GARA optimization problem should be able to perform two central tasks. First, it has to choose the points inside the code where update instructions would be needed to adjust the address registers. Second, it has to allocate an address register to each array reference such that the cost of the required update instructions is minimized. In order to achieve an optimal GARA solution, both tasks have to be performed optimally. In this section, we show how to solve the first part of the problem optimally, i.e. deciding the points where update instructions could be required. Notice that the need of an update instruction, at some selected point, will depend on how efficient is the algorithm that assigns address registers to the array references, as discussed in Section 4.

In [19] it was realized that the problem of deciding where update instructions are needed resembles the problem of choosing places to insert ϕ-functions in the SSA-Form [22]. The points where to insert ϕ-functions are those on the *Iterated Dominance Frontier* [22] of the basic blocks that contain array references. The program representation resulting after ϕ-functions are inserted was called *Single Reference Form* (SRF). For example, Figure 2(a) shows the CFG of a loop body in SRF. However, SRF is not enough to guarantee the optimality of the approach presented in [41].

The problem with SRF is that it identifies the points where more than one array reference reach, but not those points that reach multiple array references (which is the case of the point at the exit of B1 in Figure 2(a)). As a result, in SRF it is possible to have update instructions that are not associated with any ϕ-function, although their values depend on the choice of addressing modes. In order to fix this problem, we propose what we call the *Extended Single Reference Form* (ESRF). In this form, in addition to the ϕ-functions inserted at the *beginning* of the basic blocks that form the Iterated Dominance Frontier (ϕ_b), we also insert ϕ-functions at the exit of the basic blocks that are on the *Iterated Post-dominance Frontier* [39] (ϕ_e). But it is still possible that the insertion of ϕ-functions into one of the dominance frontiers will require the insertion of ϕ-functions into the other one. In order to deal with this, another iteration level is used to compute ESRF such that, at the end, each array reference has only one reference in each of its DU/UD-chains.

We call this approach *Combined Dominance Frontier* (CDF) and describe it in Algorithm 1. To illustrate how Algorithm 1 works, consider Figure 2(b), ignoring the ϕ-functions shown. Table 1 shows the value of the sets S, IDF and IPF as they are computed in Algorithm 1. For example, in step 3, S is set to $\{1, 4, 5, 6\}$, and so the iterated post-dominance frontier in step 4 is calculated as if there were array references in all of these basic blocks. At the end of the algorithm, IDF=$\{1, 4, 5, 6\}$ and IPF=$\{1, 2, 3, 6\}$. The

Algorithm 1 Combined Dominance Frontier

```
(1) function CDF(Ref_BBs :  Set_of_BBs)
(2)     var IDF, IPF : Set_of_BBs;
(3)     S ← Ref_BBs;
(4)     do
(5)         S' ← S;
(6)         IDF ← Iterated_Dominance_Frontier(S);
(7)         S ← S ∪ IDF;
(8)         IPF ← Iterated_Postdominance_Frontier(S);
(9)         S ← S ∪ IPF;
(10)    while S ≠ S';
(11)    return (IDF, IPF);
```

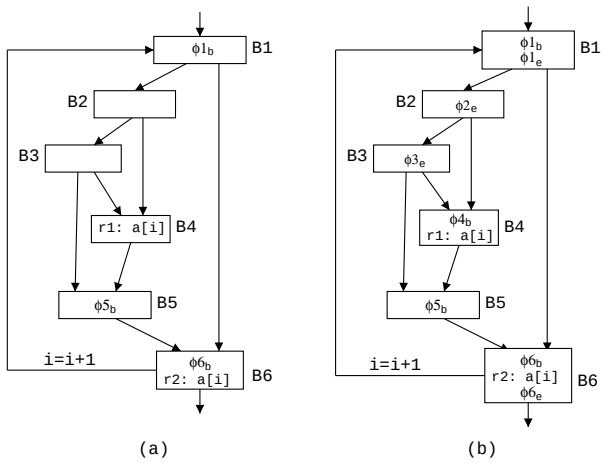

Figure 2: (a) CFG in SRF; (b) CFG in ESRF.

resulting code in ESRF is illustrated in Figure 2(b), with the corresponding ϕ-functions inserted.

ESRF holds the important property that any update instruction depends on a ϕ-function[1]. Thus, update instructions are inserted between a ϕ-function and an array reference or another ϕ-function, but never between two array references. As the variables of our optimization problem are the values to be chosen for the ϕ-functions, this is a fundamental property that guarantees the optimality of the algorithm described in [41].

[1]Update instructions may not depend on a ϕ-function, for example for consecutive array references inside a single basic block. But these cases can be locally solved in an optimal fashion.

Step	CDF Line	S	IDF	IPF
1	3	4,6		
2	6		1,5,6	
3	7	1,4,5,6		
4	8			1,2,3,6
5	9	1,2,3,4,5,6		
6	6		1,4,5,6	
7	7	1,2,3,4,5,6		
8	8			1,2,3,6
9	9	1,2,3,4,5,6		

Table 1: CDF computation for the code from Figure 2(b).

4 GARA Heuristics

In this section we summarize the methods proposed in [19] and [41]. Both techniques are based on merging live ranges. Initially, each array reference from the loop being optimized is assigned to a separate live range, and then a sequence of live range merge operations is performed, until the number of live ranges reaches the number of address registers available on the target processor. In both approaches, the choice for the pair of live ranges to be merged, at each step, is performed by computing the cost of all live ranges obtained by pairwisely merging the current live ranges, and then choosing the least costly one. The two approaches differ on the way the cost for a live range is computed: [19] uses a heuristic called *Tail-Head* (TH), while [41] uses an optimal dynamic programming algorithm called *Leaves Removal Order* (LRO) whenever the topology of the live range allows, resorting to TH otherwise. We call this combined approach LRO-TH.

In order to illustrate both methods, we use a loop from the `pegwit` program in MediaBench. Figure 3 shows the CFG representation of this loop and its corresponding array references. The loop has nine references (`r1` to `r9`), associated to two arrays (`a` and `b`), and the loop step is 2. The edges are labeled with the corresponding estimated execution frequencies. When an update instruction is needed, we use the estimated execution frequency on the edge or basic block where it will be inserted as its corresponding cost. This way, our goal becomes to minimize the total estimated execution frequency for the required update instructions, instead of simply minimizing the number of update instructions as in [19, 41].

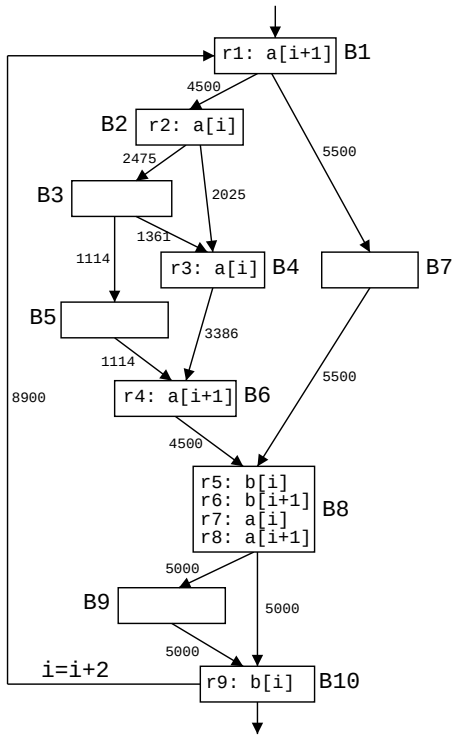

Figure 3: The control-flow graph for a loop example from `pegwit`.

4.1 The Tail-Head Heuristic

In this section we shortly describe how the Tail-Head (TH) heuristic is used to estimate the cost of a merge operation during GARA. For further details the reader should report to [19].

When GARA starts, the live range growth approach takes place, merging at each step the pair of current live ranges that leads to the best total cost. Hence, at each merge operation the cost of the new live range must be determined. When the TH heuristic is used to compute the cost, the following operations take place. Initially, the loop is transformed to SRF, in order to determine the points where ϕ-functions are required. Then, the ϕ-functions are solved, starting at the loop tail toward the loop head. For each ϕ-function, the solution is chosen among the values of all references in its UD/DU-chains, ignoring the ϕ-functions which have not been solved yet. The cost of the merged live range is given by the summation of the expected execution frequency of the update instructions required to set the address registers correctly. Zero cost auto-modify addressing modes are used whenever possible.

As an example, Figure 4 shows the live range formed by merging references `r3`, `r7` and `r8`. The UD/DU-chains for the ϕ-functions associate to these references are also shown.

First, ϕ_{8b} is solved, resulting in `a[i]` (the only value in its UD/DU-chains). Then, ϕ_{6b} results in `a[i]`, for the same reason. Finally, ϕ_{1b} is solved, and in this case two solutions are possible: `a[i]`, because of `r3` and because it is the solution for ϕ_{6b} and ϕ_{8b}, and `a[i-1]`, because of the reference `r8` from the previous loop iteration (note that `a[i+1]` from the previous iteration is `a[i-1]` in the current one, as the loop step is 2). `a[i]` is chosen because it leads to cost zero, with the insertion of a post-increment addressing mode at reference `r8`.

Example: GARA Using the TH Heuristic

We now illustrate the GARA solution using the TH heuristic. Consider the code from Figure 3. Table 2 shows the initial set of live ranges. For each live range, this table presents the basic blocks where ϕ-functions are required when transforming it to SRF (column 2), and the live range cost (column 3) calculated using the Tail-Head heuristic.

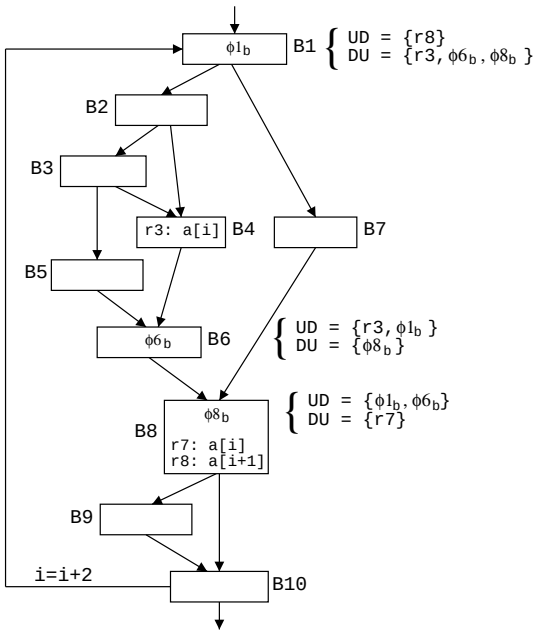

Figure 4: The live range formed by references `r3`, `r7` and `r8` in SRF.

As GARA progresses, the live range growth approach takes place, starting with the ranges in Table 2. Table 3 shows the sequence of merge operations that are performed. In column 2, the resulting set of live ranges after each merge is illustrated, with the just merged range highlighted. The third column contains the basic blocks where ϕ-functions are required for the just formed range, and column 4 shows the cost of the merge operation, computed using the Tail-Head heuristic. The last column lists the total cost of all current live ranges at this step of the execution.

The merging of live ranges is performed until two ranges remain (step 7 in Table 3).

Ref.	LR ϕ-funcs.	LR cost
1	1b	10000
2	1b,8b	8900
3	1b,6b,8b	13386
4	1b,8b	8900
5	1b	8900
6	1b	8900
7	1b	8900
8	1b	8900
9	1b	8900
Total cost		85686

Table 2: Initial live ranges; costs computed using the Tail-Head heuristic.

Step	LRs	LR ϕ-funcs.	LR cost	Total cost
0	[1][2][3][4][5][6][7][8][9]	–	–	85686
1	[1][2][3][4]**[5,6]**[7][8][9]	1b	0	67886
2	[1][2][3][4][5,6]**[7,8]**[9]	1b	0	50086
3	[1][2]**[3,7,8]**[4][5,6][9]	1b,6b,8b	0	36700
4	**[1,2]**[3,7,8][4][5,6][9]	1b,8b	8900	26700
5	[1,2]**[3,4,7,8]**[5,6][9]	1b,6b,8b	1114	18914
6	[1,2][3,4,7,8]**[5,6,9]**	1b	8900	18914
7	**[1,2,3,4,7,8]**[5,6,9]	1b,6b,8b	12286	21186

Table 3: Live Range Growth using the Tail-Head heuristic.

These ranges cannot be merged, as they refer to different arrays (`a` and `b`), and thus cannot share the same address register. Assuming that 3 address registers are available (which is the case of the target processor we used), we have two possibilities: either using 2 address registers (one for each final live range), or using 3 address registers (one for each live range after step 6). We choose the last alternative, as it leads to a smaller total cost (18914). The final allocation and the update instructions inserted are shown in Table 4.

Address Register	References	Update Instructions		
		Instr.	Edge	Cost
ar_0	r_1:a[i+1]-- r_2:a[i]	ar_0+=3	B10→B1	8900
ar_1	r_3:a[i]++ r_4:a[i+1]-- r_7:a[i]++ r_8:a[i+1]++	ar_1+=1	B3→B5	1114
ar_2	r_5:b[i]++ r_6:b[i+1]-- r_9:b[i]	ar_2+=2	B10→B1	8900

Table 4: The final allocation using TH.

4.2 The Leaves Removal Order Algorithm

The LRO approach for computing a live range cost, in opposition to the Tail-Head heuristic, guarantees that the optimal solution is found for the values of the ϕ-functions, although it does not apply to every code in ESRF. Fortunately, the experimental results in Section 6 show that the cases to which this method applies are indeed very common in practice.

In [41] we introduced the concept of *ϕ-Dependence Graph* (DG_ϕ). This is an undirected graph in which there is one vertex for each ϕ-function, and an edge between two vertices if and only if the solution to one of the ϕ-functions depends on the solution to the other. The DG_ϕ can be constructed using an algorithm similar to reaching definitions and DU/UD-chains [3] on the ϕ-functions. In order to illustrate the concepts of ESRF and DG_ϕ, Figure 5(a) presents, for the same example from Figure 3, the live range formed by references `r1`, `r2` and `r4` in ESRF. The corresponding DG_ϕ is shown in Figure 5(b). For example, there is an edge between ϕ_{1e} and ϕ_{8b} because $\phi_{8b} \in DU_{1e}$ (and so $\phi_{1e} \in UD_{8b}$).

Whenever DG_ϕ is a *tree*, a dynamic programming algorithm can be used to solve the ϕ-functions optimally [41]. To achieve that, we must first find all possible solutions for any ϕ-function in ESRF. These values are exactly the array references that appear in the loop, and the references in the next loop iteration. In practice, the number of possible solutions for the ϕ-functions is usually restricted to a few values. The dynamic programming algorithm executes in a bottom-up fashion, following a *Leaves Removal Order* (LRO) of the DG_ϕ. For each tree leaf l, all the possible pairs of solutions for l and its single adjacent vertex v are tested, and the best costs are accumulated in v. Before, the costs related to other vertices adjacent to l have already been accumulated in l. This

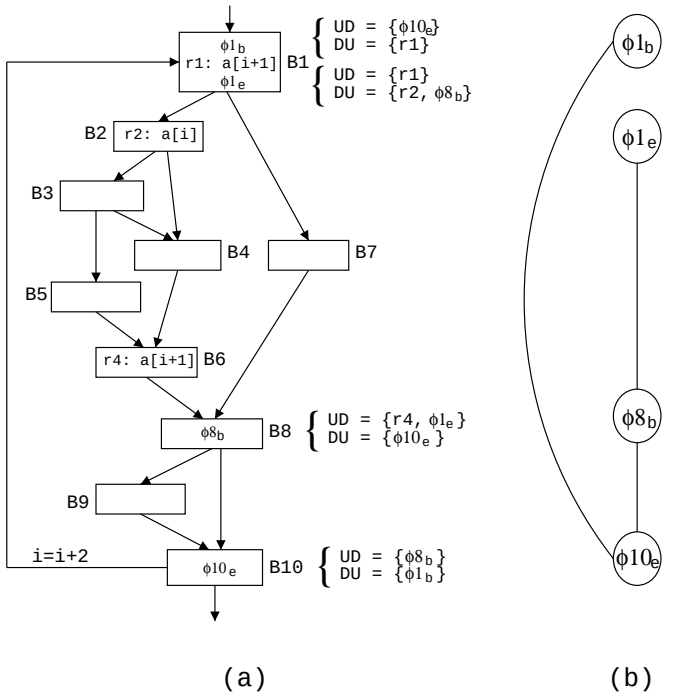

Figure 5: (a) The live range formed by references `r1`, `r2` and `r4` in ESRF. (b) The corresponding DG_ϕ.

algorithm has a running time linear on the number of ϕ-function. More details on this algorithm are described in [41].

Example: GARA Using the LRO-TH Approach

We now illustrate the GARA solution using the LRO-TH approach. Consider again the code from Figure 3. Table 5 shows the initial live ranges for the live range growth approach. Column 3 shows whether the corresponding DG_ϕ is a tree or not. Whenever DG_ϕ is a tree, the LRO algorithm is used to compute the optimal cost of the range resulting after merge. Otherwise, the Tail-Head heuristic cost estimate is determined.

Table 6 shows the sequence of live range mergings, similarly as in Table 3. Here again, two options for allocation are possible: using either 2 or 3 address registers. The best choice is again to allocate 3 address registers to the live ranges in step 6 of Table 6. This gives a total cost of 17800, which is slightly better than the cost achieved with the Tail-Head-only approach (18914). The final allocation is presented in Table 7.

Ref.	LR ϕ-functions	DG_ϕ acyclic	LR cost
1	1b,10e	yes	8900
2	1b,1e,8b,10e	no	8900
3	1b,1e,2e,3e,4b,6b,8b,10e	no	13386
4	1b,1e,8b,10e	no	8900
5	1b,10e	yes	8900
6	1b,10e	yes	8900
7	1b,10e	yes	8900
8	1b,10e	yes	8900
9	1b,10e	yes	8900
Total cost			85686

Table 5: Initial live ranges; costs computed using the LRO algorithm whenever possible, and the Tail-Head heuristic otherwise.

Step	LRs	LR ϕ-functions	DG_ϕ acyclic	LR cost	Total cost
0	[1][2][3][4][5][6][7][8][9]	–	–	–	84586
1	[1][2][3][4]**[5,6]**[7][8][9]	1b,10e	yes	0	66786
2	[1][2][3][4][5,6]**[7,8]**[9]	1b,10e	yes	0	48986
3	[1][2]**[3,7,8]**[4][5,6][9]	1b,1e,2e,3e,4b,6b,8b,10e	no	0	35600
4	**[1,2]**[3,7,8][4][5,6][9]	1b,1e,8b,10e	yes	8900	26700
5	**[1,2,4]**[3,7,8][5,6][9]	1b,1e,8b,10e	yes	8900	17800
6	[1,2,4][3,7,8]**[5,6,9]**	1b,10e	yes	8900	17800
7	**[1,2,3,4,7,8]**[5,6,9]	1b,1e,2e,3e,4b,6b,8b,10e	no	12286	21186

Table 6: Live Range Growth using the LRO algorithm whenever possible, and the TH heuristic otherwise.

5 The GARA Exact Solution

In this section, we present a method (EXACT) for computing the exact, minimum cost GARA solution for a given loop. This approach relies on the LRO algorithm whenever possible. As the experimental results from Section 6 show, LRO is applicable in the great majority of the cases, and this is what makes the EXACT approach feasible[2].

[2]By *feasible* here we mean that it requires a computational time that we can deal with for the purposes of this research, although it may not be practical to be performed inside a compiler.

Address Register	References	Update Instructions		
		Instr.	Edge	Cost
ar_0	r_1:a[i+1]-- r_2:a[i]++ r_4:a[i+1]--	ar_0+=3	B10→B1	8900
ar_1	r_3:a[i] r_7:a[i]++ r_8:a[i+1]++	–	–	–
ar_2	r_5:b[i]++ r_6:b[i+1]-- r_9:b[i]	ar_2+=2	B10→B1	8900

Table 7: The final allocation using LRO-TH.

Let R be the number of address registers in the target processor which are available for allocation, and A be the number of array references in a loop. First of all, we should identify all the A array references inside the loop, and partition them such that two references are put into the same partition if and only if their *indexing distance* [50] can be statically determined. We call K the number of partitions inside the loop, and $P_1, ..., P_K$ the partitions themselves.

It is clear that only references in the same partition are eligible for sharing an address register, although references in the same partition can be allocated to different address registers. Therefore, one of the decisions that must be made is how to divide the R address registers among the K partitions. The second decision EXACT has to make is how to sub-partition each partition P_j into live ranges. Finally, for each live range, we should choose the best solution for the ϕ-functions in a way to minimize the update instruction cost. Algorithm 2 describes a top-level pseudo-code for our approach.

Procedure `EXACT` (Algorithm 2) is the entry point for the pseudo-code. Its first step is to identify and partition the array references inside the loop. Then, for each partition P_j, it calls the procedure `Compute_Minimum_Costs`, which fills in the j^{th} column of the matrix C (C_{ij} is the minimum possible cost if i address registers are assigned to partition P_j). In order to fill in this column, `Compute_Minimum_Costs` exhaustively generates all the possibilities of sub-partitioning the array references in P_j in a number of live ranges that varies from 1 to R. For each live range, the corresponding minimum update instruction cost is computed using the LRO algorithm if possible, or a brute force, exponential algorithm otherwise, which simply tests all the combinations of solutions to the ϕ-functions. In addition, an estimate of the cost if no address register is assigned to this partition

Algorithm 2 GARA Exact Solution

```
(1) procedure EXACT (L :  loop)
(2)     identify the array references in L, partitioning them
(3)       in P_1, ..., P_K;
(4)     for each P_j, 1 ≤ j ≤ K do
(5)       Compute_Minimum_Costs(P_j);
(6)     Optimal_AR_Distribution({P_1, ..., P_K}, C);
(7)
(8) procedure Compute_Minimum_Costs(P_j)
(9)     fill in C_0j with the cost estimated if no address
(10)      register is allocated to P_j;
(11)    C_ij ← +∞, 1 ≤ i ≤ R;
(12)    for each combination of partitioning the references
(13)        in P_j in live ranges {LR_1, ..., LR_i} , 1 ≤ i ≤ R do
(14)      total_cost ← 0;
(15)      for each LR_k , 1 ≤ k ≤ i , do
(16)        build DG_φ for LR_k;
(17)        if DG_φ is a tree then
(18)          cost ← LRO_cost(LR_k);
(19)        else
(20)          cost ← Brute_Force_cost(LR_k);
(21)        total_cost ← total_cost + cost;
(22)      if total_cost < C_ij then
(23)        C_ij ← total_cost;
(24)
(25) procedure Optimal_AR_Distribution({P_1, ..., P_K},C)
(26)    min_cost ← +∞;
(27)    for each combination of values r_1, r_2, ..., r_K |
(28)        Σ_{i=1}^{K} r_i ≤ R and r_i ≥ 0 do
(29)      cost ← 0;
(30)      for k ← 1 to K do
(31)        cost ← cost + C_{r_k,k};
(32)      if cost < min_cost then
(33)        min_cost ← cost;
(34)
(35) procedure Brute_Force_cost(LR_k)
(36) /* Backtracking to generate all the combinations of
(37)    solutions for the φ-functions in the ESRF of LR_k.
(38)    Return the minimum cost among the costs for all
(39)    of these combinations. */
```

# ARs	Partition			
	`A (Array a)`		`B (Array b)`	
	Cost	LRs	Cost	LRs
0	140000	–	80000	–
1	10014	[1,2,3,4,7,8]	8900	[5,6,9]
2	8900	[1,2,4][3,7,8]	8900	[5,6][9]
3	14400	[1,3][2,4][7,8]	26700	[5][6][9]

Table 8: The C matrix holding the best solution for each entry in the $\#\ ARs \times Partition$ space. Here each partition corresponds to one of the arrays from Figure 3.

is made. This estimate is dependent on the target processor, and considers any other addressing mode available, or the cost of spilling an address register. Table 8 shows the C matrix computed for the loop from Figure 3.

Having the C table computed, the `EXACT` procedure calls the procedure `Optimal_AR_Distribution`, which is responsible for choosing the best way to divide the R address registers among the K partitions. This is another brute force algorithm, which explores all the possibilities for making this distribution. This procedure uses the precomputed values previously stored in the C table, in order to avoid recomputing the minimum costs at each time. For our example, using the C matrix from Table 8, the best possible solution is to attribute 2 ARs to partition `A` and 1 to partition `B`, resulting in a total cost of 17800. Note that this is the same solution found by the LRO-TH approach in Section 4.2.

6 Experimental Results

In order to test the methods described in this paper, we have implemented all approaches inside GCC version 3.0.2 [1]. The GARA optimization takes place right after the traditional loop optimizations, making use of the loop induction variable information available at this point. Moreover, we used the GCC infrastructure for profiling-driven optimizations to improve the update instruction cost estimation (as described in Section 4).

The target processor for the experiments was the Lucent DSP16xx [37], which has a total of 4 address registers (one of which is used as stack-pointer), and post-increment (decrement) addressing modes. The results presented here are based on static profiling information. We measured the expected execution cycles for inner-most loops from MediaBench [30] applications. Only loops with any array reference have been considered. Three set of experiments have been performed. In the first set (Table 9), the speedup be-

tween all approaches and the original GCC was measured. The second set of experiments (Table 10) aimed at comparing the compilation time between the heuristics and the exact solution. The last set of experiments (Table 11) computed the percentage of DG_ϕ graphs in the loops which are trees.

Table 9 shows a comparison between the following approaches: (a) live range growth using the Tail-Head heuristic (TH); (b) live range growth using the combination of the Leaves Removal Order algorithm and the Tail-Head heuristic (LRO-TH); and (c) the exact solution (EXACT). The speedup was measured with respect to the original GCC implementation [1]. Table 9 shows that the speedup achieved by LRO-TH approaches the speedup of the time-consuming EXACT method (average difference of 0.09%). In addition, the TH approach also leads to a speedup close to the exact solution (average difference of 0.54%), although not as good as LRO-TH does.

Program	# of loops	Speedup (%)		
		TH	LRO-TH	EXACT
`adpcm`	2	0.80	0.80	1.01
`epic`	6	10.24	11.24	11.50
`g721`	1	0.00	0.00	0.00
`ghostscript`	37	13.46	13.95	13.96
`jpeg`	32	13.86	14.42	14.44
`mpeg2`	7	13.13	13.13	13.93
`pegwit`	5	25.22	25.22	25.22
`pgp`	3	23.96	23.96	23.96
Average	–	13.85	14.30	14.39

Table 9: Comparison in terms of speedup between the original GCC, the Tail-Head (TH) approach, the Leaves Removal Order and Tail-Head (LRO-TH) combined approach, and the EXACT solution.

Table 10 compares the execution time performance of the GCC implementations of TH, LRO-TH and EXACT. It shows that the TH and LRO-TH heuristics do not increase the compilation time noticeably (in fact, they reduced the average compilation time slightly). On the other hand, the EXACT method demands a great amount of time for loops with many array references, as in some MediaBench programs (e.g. `pegwit`). This is due to the intrinsic exponential time-complexity of EXACT. Notice that for some programs (e.g. `mpeg2`), GARA improved the compilation time, what should be due to the simplifications it performs in the code, accelerating other optimizations.

Table 11 presents data regarding the topology of the DG_ϕ's, during the application of both the LRO-TH and the EXACT techniques. The results show that the great majority

Program	Compilation Time (s)			
	Baseline GCC	TH	LRO-TH	EXACT
adpcm	0.800	0.800	0.790	0.800
epic	4.430	4.530	4.530	4.880
g721	0.790	0.750	0.760	0.760
ghostscript	41.400	40.980	41.830	273.470
jpeg	23.410	22.240	23.380	259.310
mpeg2	12.020	10.840	10.280	2.440
pegwit	6.160	4.000	4.540	38933.750
pgp	4.690	4.330	4.320	4.620
Average	11.713	11.059	11.304	4935.004

Table 10: Comparison in terms of compilation time between the original GCC, the Tail-Head (TH) approach, the Leaves Removal Order and Tail-Head (LRO-TH) combined approach, and the EXACT solution.

of the DG_ϕ's are trees, meaning that the linear-time optimal LRO algorithm for computing the cost of live ranges is frequently executed in LRO-TH and EXACT. In LRO-TH, the execution of LRO reduces the cost of the update instructions by diminishing the number of times that the Tail-Head heuristic is evoked. In EXACT, the optimal cost for the live ranges can almost always be computed in linear time by LRO, thus enabling EXACT to run in feasible time.

Finally, it is worth noting that, even though most of the DG_ϕ's happen to be trees, the results obtained by TH approaches that of LRO-TH, meaning that even the simple Tail-Head heuristic leads to a good solution to the GARA problem.

7 Conclusions

In this paper we extended previous work on GARA. We presented the Extended Single Reference Form, which is needed for the optimality of the LRO algorithm [41]. We proposed an exact, optimal algorithm for GARA, which uses the LRO algorithm. The detailed experimental results show that the LRO-TH approach generally achieves solutions close to optimal. The average speedup of LRO-TH for the loops of the MediaBench benchmark was 14.3%, when comparing to the GCC's original address register allocation technique. In addition, we showed that the great majority of DG_ϕ are trees, making it possible to the exact technique to run fast for most of the loops, despite its exponential time-complexity.

Program	LRO-TH			EXACT		
	trees	total	% trees	trees	total	% trees
`adpcm`	2	4	50.00	2	4	50.00
`epic`	33	45	73.33	59	113	52.21
`g721`	1	1	100.00	1	1	100.00
`ghostscript`	2259	2400	94.12	602599	604353	99.71
`jpeg`	2190	2329	94.03	583676	585426	99.70
`mpeg2`	21	23	91.30	24	26	92.31
`pegwit`	165	199	82.91	1583239	1584209	99.94
`pgp`	7	7	100.00	10	10	100.00
Total	4678	5008	93.41	2769610	2774142	99.84

Table 11: Proportion of DG_ϕ's that are trees when applying the LRO-TH and the EXACT approaches.

8 Acknowledgments

We would like to thank Gang-Ryung Uh from Agere Systems Inc. for his support on the DSP16xx processor, and Michael Collison from Mindspeed Inc. for creating and maintaining the DSP16xx GCC port.

Capítulo 5

Trabalhos Futuros

Em otimização de código, a generalização de técnicas locais (dentro de um único bloco básico) para métodos globais (para vários blocos básicos) é quase sempre um problema difícil. Visando tornar mais simples estas generalizações, a *Static Single Assignment (SSA) Form* [22] foi proposta. Nesta forma de representação, cada uso de uma variável é alcançado apenas por uma única definição. Esta característica é trivialmente verdadeira quando cada bloco básico é considerado isoladamente, e portanto torna várias otimizações globais mais semelhantes aos seus correspondentes métodos locais. A *Single Reference Form (SRF)*, proposta por Cintra e Araújo [19], é uma adaptação de SSA para referências a vetores, possuindo a propriedade de que cada referência pode somente ser precedida por uma única outra referência, o que é essencial para se saber para que endereço o registrador de endereçamento está apontando. No presente trabalho (capítulo 4), foi proposta uma extensão de SRF, denominada *Extended Single Reference Form (ESRF)*, a qual adiciona a propriedade de que uma referência pode somente ser sucedida por uma única outra referência. Em SRF e ESRF, é importante a forma com que as funções ϕ são resolvidas, pois isto se reflete no número de instruções de ajuste do registrador de endereçamento que serão necessárias. Em SSA, quando o código precisa ser traduzido de volta para uma representação intermediária sem funções ϕ, instruções de cópia são inseridas, as quais várias vezes não são necessárias. Contudo, ao contrário de SRF e ESRF, no caso de SSA outras otimizações depois podem ser aplicadas para a remoção de cópias desnecessárias.

Para a área de compiladores, as principais contribuições deste trabalho são, possivelmente, o conceito de *Grafo de Dependências* ϕ (ϕ *Dependence Graph* – DG_ϕ) e o algoritmo polinomial para resolver as funções ϕ de forma ótima quando DG_ϕ for acíclico. Pensando a respeito da possível aplicação destes resultados a outros problemas de otimização de código, foi estudada uma formulação para problema de alocação global de registradores de propósito geral, que será detalhada na seção 5.1. A utilização desta formulação para uma solução baseada em *live range growth* poderá constituir uma alternativa efetiva ao

método de alocação global de registradores por coloração de grafos [16, 13], que tem sido o método dominantemente adotado para este problema nas últimas duas décadas.

5.1 Alocação Global de Registradores de Propósito Geral

O problema de Alocação Global de Registradores (GRA[1]) é um dos problemas fundamentais de otimização de código, sendo estudado há algumas décadas. O primeiro alocador global de registradores baseado em coloração de grafos foi desenvolvido na IBM por Chaitin *et al* [16]. Este trabalho foi posteriormente estendido por várias outras contribuições, entre elas [13, 26]. Apesar de ser o método padrão adotado para a solução deste problema, vários pesquisadores ainda tentam encontrar métodos mais eficientes (quanto a algum critério) para este problema. Isto porque, em alguns casos o alocador global de registradores por coloração gera alguns trechos de código de baixa qualidade [21]. Proebsting e Fischer [44] apresentam uma técnica baseada em probabilidades para guiar a alocação de registradores. Bergner *et al* [9] descrevem uma técnica para fazer *spilling* parcial de *webs*. Outros trabalhos, como [49, 43], propõem métodos mais rápidos para alocação global de registradores, visando a aplicação em compiladores *just in time*.

Nesta seção, é proposta uma formulação para resolver o problema de alocação global de registradores usando a técnica de *live range growth*. Ao longo desta seção, supõe-se uma arquitetura *load/store*, tipo RISC. Ou seja, considera-se que todas as computações são feitas em registradores, e as únicas instruções que operam com a memória são exatamente load e store. Esta suposição visa tornar a formulação aqui apresentada menos complexa. Todavia, acredita-se que uma generalização para outras arquiteturas possa ser obtida a partir do material aqui apresentado.

Assim como para o problema de GARA, o problema central para se obter uma solução baseada em *live range growth* para alocação global de registradores é a determinação do menor custo para um dado *live range*. Contudo, uma outra noção de *live range* precisa ser introduzida para o problema de GRA. Em GRA, considera-se um *live range* como sendo um conjunto de variáveis que serão alocadas a um mesmo registrador. Na verdade, para eliminar o problema da reutilização de um mesmo nome de variáveis, considera-se que as variáveis foram previamente renomeadas de forma que cada *web* [39] seja associada a um nome de variável (ou registrador virtual) diferente. Analogamente à GARA, todas as variáveis de um mesmo *live range* compartilham o mesmo registrador, de forma que todos os usos e definições destas variáveis serão feitos através deste registrador.

Para o problema de GRA, também é importante que cada referência (isto é, uso ou

[1]Em inglês: *Global Register Allocation*

definição) de uma variável seja precedido por uma única outra referência a uma variável do seu *live range*. Esta propriedade é importante para que se saiba, em cada ponto do programa, qual variável estará armazenada no registrador, independentemente do caminho executado. Contudo, o fato de uma referência ser sucedida por uma única outra perde importância neste problema. Isto porque, durante a execução de uma instrução, não há alternativas para fazer com que o registrador passe a conter outra variável do *live range* que não a especificada na instrução. Ou seja, sendo x uma variável do *live range* L, depois de uma instrução que define x, o registrador associado a este *live range* certamente terá que conter esta variável. O mesmo acontece no caso de uma instrução que usa x — desde que esta instrução não defina uma outra variável y do mesmo *live range* L, pois neste caso o registrador conterá y, e não x, após a execução desta instrução.

Assim, da mesma forma que é feito com o problema de GARA, o primeiro passo a ser feito é a inserção das funções ϕ. No caso de GRA, estas funções devem ser inseridas na entrada de cada bloco básico que pertença à fronteira de dominância iterada dos blocos básicos que usam ou definem alguma variável do *live range*. Isto se faz necessário visto que tanto um uso quanto uma definição de uma variável forçam que a mesma esteja no registrador.

5.1.1 Análise de Fluxo de Dados

Na modelagem aqui proposta para alocação global de registradores, é crucial a informação, para um determinado *live range*, de qual variável está alocada ao seu registrador em cada ponto do programa. Além disto, é importante também a informação de se a variável correntemente no registrador possui um valor igual ou diferente do seu valor armazenado na memória. Isto porque, em caso de ser necessário utilizar o registrador para outra variável do *live range* em um determinado ponto do programa, uma instrução de store pode ser economizada se os valores no registrador e em memória forem idênticos. No caso de estes valores serem iguais, diz-se que o valor está *consistente*; caso contrário, o valor é dito *inconsistente*.

Para se calcular a variável que estará alocada ao registrador de um *live range* em cada ponto do programa, bem como o seu estado de consistência, é feita uma análise de fluxo de dados, denominada *Register Consistency Reachability (RCR)*. Assim, os itens desta análise serão formados por pares (v, e), onde v é uma variável do *live range* e e é um estado de consistência. Denota-se V o conjunto das variáveis do *live range*, `C` o estado de consistência e `I` o de inconsistência.

Uma atenção especial é necessária às funções ϕ. O resultado de uma função ϕ_i precisa também ser formado por um par, constituído por uma variável de V e um estado de consistência (`C` ou `I`). Contudo, para se escolher as soluções para as funções ϕ, é necessário

ter os resultados da análise RCR, para que se possa saber quais estados de registrador alcançam uma determinada função ϕ e assim avaliar qual a melhor solução para ela. Define-se então que a solução de uma função ϕ_i é o par (ω_i, σ_i). Também são definidos os conjuntos $V_\phi = \{\omega_i\}$ e $E_\phi = \{\sigma_i\}$. Os ω_i's e os σ_i's são as variáveis do problema de otimização, que visa diminuir o número de instruções de load e store. Ou seja, assim como em GARA, deseja-se escolher as soluções das funções ϕ de forma a minimizar o custo do *live range*.

Mais formalmente, os itens da análise de fluxo de dados RCR são pares (v, e), nos quais:

- $v \in V \cup V_\phi \cup \{\epsilon\}$, onde ϵ simboliza que nenhuma variável está no registrador.
- $e \in \{\mathtt{C}, \mathtt{I}, \mathtt{X}\} \cup E_\phi$, sendo que X indica que não se sabe se o valor contido no registrador está consistente ou não.

A utilidade dos símbolos ϵ e X ficará evidente a seguir.

Definidos os itens da análise de fluxo de dados, o próximo passo é a definição de como estes itens são computados ao longo do programa. De forma a tornar mais simples a apresentação, supõe-se que um conjunto de itens é calculado para cada ponto entre duas referências do programa. Ou seja, dada uma instrução (em código de três endereços) do tipo x := y + z, sendo x e y variáveis do *live range* em questão, considera-se um ponto do programa antes da leitura de y, outro entre a leitura de y e escrita em x, e outro ponto ainda após a escrita de x. Como z não pertence ao mesmo *live range*, o seu uso pode ser ignorado. É importante notar também que um *live range* contendo y e z não pode ser considerado, uma vez que pelo conceito de *live range* ambas as variáveis compartilhariam o mesmo registrador, e nesta instrução ambas as variáveis são usadas simultaneamente. Assim, estas restrições devem ser levadas em consideração quando da escolha dos *live ranges* a serem unidos no algoritmo de *live range growth*. Por outro lado, não há nenhum problema em x e y pertencerem ao mesmo *live range*; no pior caso, y teria um valor inconsistente antes da instrução x := y + z, necessitando um store, mas este store pode ser inserido antes desta instrução.

Para cada referência r, o conjunto de elementos de RCR que alcança a sua entrada é dado por:

- $in[r] = \bigcup_{p \ \mathtt{pred}\ r} out[p]$

Ou seja, os estados do registrador que alcançam o ponto antes de uma referência r são todos os que alcançam a saída de alguma referência p que pode preceder r no fluxo de controle do programa. Além disso, define-se que $in[r_0] = \{(\epsilon, \mathtt{X})\}$, sendo r_0 a primeira referência do programa.

Neste instante, é importante salientar que, em todos os pontos que precedem uma referência, o conjunto de itens de RCR só pode conter um único elemento. Esta propriedade é resultado da inserção das funções ϕ, e é análoga ao fato de que, em SSA, cada uso é alcançado por uma única definição.

A seguir, tem-se as expressões para $out[r]$, que relacionam estes conjuntos aos respectivos conjuntos $in[r]$, para cada referência. Existem três casos, dependendo do tipo de referência (uso ou definição), ou de se tratar de uma função ϕ:

- Caso 1: r: $x := ... \mid x \in LR$

  $out[r] = \{(x, \texttt{I})\}$

- Caso 2: r: $... := x \mid x \in LR$

  $$out[r] = \begin{cases} \{(x,s)\} & \text{, se } in[r] = \{(x,s)\}, \forall s \\ \{(x, \texttt{C})\} & \text{, se } in[r] = \{(y,s)\},\ y \in (V \cup \{\epsilon\}) - \{x\},\ \forall s \\ \{(x, \texttt{X})\} & \text{, se } in[r] = \{(\omega_i, \sigma_i)\},\ \omega_i \in V_\phi \end{cases}$$

- Caso 3: r: ϕ_i

  $out[r] = \{(\omega_i, \sigma_i)\}$

No caso 1, x será atribuído um novo valor, e portanto o registrador conterá, após esta referência, a variável x com um valor inconsistente com o valor desta variável em memória.

O caso 2 se subdivide em três sub-casos. O primeiro, quando $in[r]$ é constituído de um estado no qual x está no registrador. Neste caso, um uso de x fará com que o registrador continue com x, no mesmo estado de consistência que havia antes. O segundo sub-caso é quando $in[r]$ possui um estado com uma outra variável y no registrador, ou com o registrador vazio (sem variável alguma). Nesta situação, independentemente da consistência de y (o que pode requerer um store ou não), será necessário fazer um load de x antes da referência r, o que fará com que o registrador contenha esta variável, com um valor consistente com o seu valor em memória. O terceiro sub-caso trata a situação de a referência r ser alcançada pela solução de uma função ϕ. Nesta situação, pode-se apenas garantir que o registador conterá o valor de x, porém o estado de consistência deste é indeterminado, dependendo da solução que for escolhida para a função ϕ. Por exemplo, se a solução da função ϕ for (x, I), este também será o valor em $out[r]$, mas se a solução for ter uma outra variável no registrador, um load será necessário e resultará em $out[r] = \{(x, \texttt{C})\}$.

Por último, o caso 3 trata das funções ϕ, que podem ser consideradas como um tipo especial de referência. Uma função ϕ deixará o registrador em um estado dado pela solução que for escolhida para ela.

Estas expressões para os conjuntos *in* e *out* podem ser utilizadas para se encontrar, iterativamente, estes conjuntos para todos os pontos do programa. Uma formulação

utilizando o conceito de blocos básicos pode ser utilizada para tornar mais eficiente o cálculo destes conjuntos. Contudo, optou-se por esta formulação em nível de referências visando tornar mais simples a apresentação.

5.1.2 Emissão de Instruções Não Dependentes das Funções ϕ

Conforme já explicado, após a inserção das funções ϕ, é válida a propriedade de que toda referência a alguma variável do *live range* é alcançada por um único estado de registrador (ou seja, $|in[r]| = 1$, para toda referência r). Valendo-se desta propriedade, para cada referência que não depende da solução de funções ϕ, o conjunto mínimo de instruções requeridas para obter o conteúdo do registrador necessário podem ser calculadas. Os seguintes casos identificam as instruções que não dependem da solução de funções ϕ, bem como as instruções necessárias em cada caso (denota-se $live_{in}[r]$ o conjunto de variáveis vivas no ponto antes da referência r):

1. Caso: r: $x := ... \mid x \in LR$, $in[r] = \{(v, e)\}$
   (a) Caso $(v \in V - \{x\})$ e $(e = \texttt{I})$ e $(v \in live_{in}[r])$
   Instruções necessárias, antes de r: store v
   Neste caso, o registrador contém uma variável v diferente de x, a qual está inconsistente e viva neste ponto do programa. Assim, é necessário armazenar o valor de v. Como a referência a x trata-se de uma definição, não é necessário carregar o valor de x da memória.
   (b) Caso contrário, nenhuma instrução é necessária, ou sua necessidade depende da solução de alguma função ϕ.
2. Caso: r: $... := x \mid x \in LR$, $in[r] = \{(v, e)\}$
   (a) Caso $v = \epsilon$
   Instruções necessárias, antes de r: load x
   Neste caso, o registrador não contém variável alguma. E como a referência a x trata-se de um uso, é necessário carregar o valor de x da memória.
   (b) Caso $(v \in V - \{x\})$ e $((e \in \{\texttt{C},\texttt{X}\})$ ou $((e = \texttt{I})$ e $(v \notin live_{in}[r]))))$
   Instruções necessárias, antes de r: load x
   Neste caso, o registrador contém uma variável v diferente de x, a qual está consistente, possui estado de consistência indefinido (depende de alguma função ϕ), ou ainda a qual está inconsistente mas não está viva neste ponto. Assim, não é necessário armazenar o valor de v (caso $e = \texttt{X}$, um store poderá ser necessário dependendo da solução das funções ϕ). Porém, como a referência a x trata-se de um uso, é necessário carregar o valor de x da memória.

(c) Caso $(v \in V - \{x\})$ e $(e = \texttt{I})$ e $(v \in live_{in}[r])$
Instruções necessárias, antes de r: store v ; load x
Neste caso, o registrador contém uma variável v diferente de x, a qual está inconsistente e viva neste ponto do programa. Assim, é necessário armazenar o valor de v. Além disso, como a referência a x trata-se de um uso, é necessário carregar o valor de x da memória.

(d) Caso contrário, nenhuma instrução é necessária, ou sua necessidade depende da solução de alguma função ϕ.

5.1.3 Solução das Funções ϕ

Inseridas as instruções de load e store que não dependem de funções ϕ, conforme descrito anteriormente na seção 5.1.2, resta a inserção das instruções dependentes das soluções das funções ϕ. Mais do que isso, resta ainda resolver as funções ϕ, ou seja, decidir em que estado o registrador estará em cada um dos pontos do programa onde uma função ϕ foi inserida. Assim como no problema de GARA, os valores escolhidos para estas funções são muito importantes, uma vez que eles influenciam no número de instruções de load e store necessárias.

Dada uma função $\phi_i = (\omega_i, \sigma_i)$, para se avaliar o custo de uma solução $(\omega_i, \sigma_i) = (w, f)$, deve-se analisar todas as referências do programa que:

1. são alcançadas por ϕ_i, ou seja, possuem (ω_i, σ_i) no seu conjunto in; ou
2. são alcançadas pelo registrador com uma variável em estado indefinido (X) por causa da solução de ϕ_i; ou
3. alcançam ϕ_i

Para resolver os casos 1 e 2, basta propagar-se a solução (w, f) da função ϕ_i como na resolução da análise de fluxo de dados RCR, e utilizar a análise de casos apresentada na seção 5.1.2, uma vez que agora os estados de registrador na entrada destas referências estarão bem determinados.

As referências do caso 3, que são as que podem preceder a função ϕ_i no fluxo de execução, são exatamente as contidas em $in[\phi_i]$. Assim, para avaliar o custo de ϕ_i relacionado a cada uma destas referências r, deve-se fazer a seguinte análise de casos, comparando-se a solução de ϕ_i com cada $(v, e) \in in[\phi_i]$:

- Caso 1: $(w = \epsilon)$ e $(v \in V)$ e $(e = \texttt{I})$ e $(v \in live_{in}[\phi_i])$
Instruções necessárias, após r: store v
Neste caso, em que a solução de ϕ_i é não ter variável alguma utilizando o registrador,

uma variável v que esteja viva e inconsistente no registrador deve ser armazenada em memória.

- Caso 2: $(w \neq \epsilon)$ e $(v = w)$ e $(f = \texttt{C})$ e $(e = \texttt{I})$
  Instruções necessárias, após r: store v
  Neste caso, o registrador contém a variável requerida, porém em estado inconsistente. Como a solução é que esta variável esteja consistente, uma instrução de store é necessária.

- Caso 3: $(w \neq \epsilon)$ e $((v = \epsilon)$ ou $((v \in V - \{w\})$ e $((e = \texttt{C})$ ou $(v \notin live_{in}[\phi_i])))$
  Instruções necessárias, após r: load w
  Neste caso, o registrador não contém variável alguma, ou contém uma variável v diferente da solução w, mas v está consistente ou não está viva neste ponto. Assim, não é necessário armazenar v na memória, sendo preciso apenas carregar w. Observa-se que, mesmo que $f = \texttt{I}$, o resultado desta instrução resultará em v consistente com a memória, sem custo adicional para isto.

- Caso 4: $(w \neq \epsilon)$ e $(v \in V - \{w\})$ e $(e = \texttt{I})$ e $(v \in live_{in}[\phi_i])$
  Instruções necessárias, após r: store v ; load w
  Neste caso, o registrador contém uma variável v diferente da solução w, e v está inconsistente e viva neste ponto. Assim, é necessário armazenar v na memória, e carregar w. Observa-se que, mesmo que $f = \texttt{I}$, o resultado destas instruções resultará em v consistente com a memória, sem custo adicional para isto.

- Caso contrário: nenhuma instrução é requerida.

A título de observação, nota-se que, como possível solução para uma função ϕ_i, não faz sentido considerar uma variável w que não esteja viva neste ponto do programa, bem como variáveis que não tenham referências alcançando ou sendo alcançadas por ϕ_i. Além disso, observa-se que é necessário considerar a possível solução na qual o registrador está vazio (não contendo variável alguma), pois às vezes isto pode levar ao menor custo.

5.1.4 Dependência entre Funções ϕ

Para a resolução de uma função ϕ, nas relações apresentadas na seção 5.1.3, foi ignorado que o valor de uma função ϕ_i pode depender de outra função ϕ_j. Contudo, esta dependência pode existir, e deve ser levada em consideração de alguma forma, assim como no problema de GARA.

Primeiramente, uma estratégia gulosa pode ser utilizada, assim como a heurística *Tail-Head* proposta em [19]. A idéia básica desta técnica é ordenar as funções ϕ segundo algum

critério, e então resolvê-las seguindo esta ordenação. Para cada função ϕ, pode-se tentar todas as suas possíveis soluções, e escolher a que resultar no menor custo. Para o cálculo destes custos, para uma dada função ϕ_i, pode-se utilizar as soluções para as outras funções ϕ já resolvidas das quais ϕ_i depende. Outras funções ϕ das quais ϕ_i depende, mas ainda não resolvidas, são ignoradas durante o cálculo do custo de ϕ_i.

Observa-se que a complexidade do problema de resolver as funções ϕ de forma ótima, para o problema de alocação global de registradores, é a mesma do problema de GARA, ou seja, o problema é NP-difícil. Assim, um método geral, eficiente e ótimo para resolver estas funções não é conhecido. Contudo, a definição de grafo de dependências ϕ (DG_ϕ), proposta no capítulo 3, pode ser utilizada novamente. Aqui mais uma vez, no caso deste grafo ser acíclico, o algoritmo LRO pode ser utilizado para determinar, de forma eficiente e ótima, as soluções para todas as funções ϕ.

5.1.5 Exemplo

De forma a ilustrar o método de união de *live ranges* proposto nesta seção, será utilizado o exemplo da figura 5.1(a). Esta figura apresenta um grafo de fluxo de controle e as referências a duas variáveis, x e y. Funções ϕ já foram inseridas na entrada dos blocos da fronteira de dominância iterada dos blocos básicos que contêm referência a alguma destas variáveis. Além disso, todos os pontos do programa já estão rotulados com os respectivos conjuntos de RCR, calculados iterativamente pelo uso das relações descritas na seção 5.1.1.

O próximo passo é a inserção das instruções de load e store que não dependem da solução das funções ϕ, ou seja, que inevitavelmente serão necessárias. Usando a análise de casos apresentada na seção 5.1.2, identifica-se que as instruções de load e store ilustradas na figura 5.1(b) são necessárias. Nesta figura, todas as referências às variáveis x e y já foram substituídas pelo registrador r.

Neste ponto, resta resolver as funções ϕ, inserindo as demais instruções necessárias. A figura 5.2(a) apresenta o grafo de dependências ϕ para a *live range*. A única aresta neste grafo é entre os vértices ϕ_1 e ϕ_2. A dependência entre as soluções destas funções existe por causa do estado de consistência em que a variável x pode alcançar ϕ_2 pela aresta $B_1 \rightarrow B_3$, o que depende da solução de ϕ_1. Como o DG_ϕ é acíclico neste caso, pode-se utilizar o algoritmo LRO, apresentado no capítulo 3, para computar o mínimo de instruções para esta *live range*. Nota-se que o vértice ϕ_3 está em um componente conexo separado, o que significa que a solução desta função ϕ não tem dependência com as demais. Assim, pode-se simplesmente testar todas possíveis soluções para esta função, e escolher a melhor. Nota-se que, neste ponto, a variável x está morta, devido à sua definição no bloco básico B_6. Desta forma, restam como candidatos para solução os estados de registrador que têm

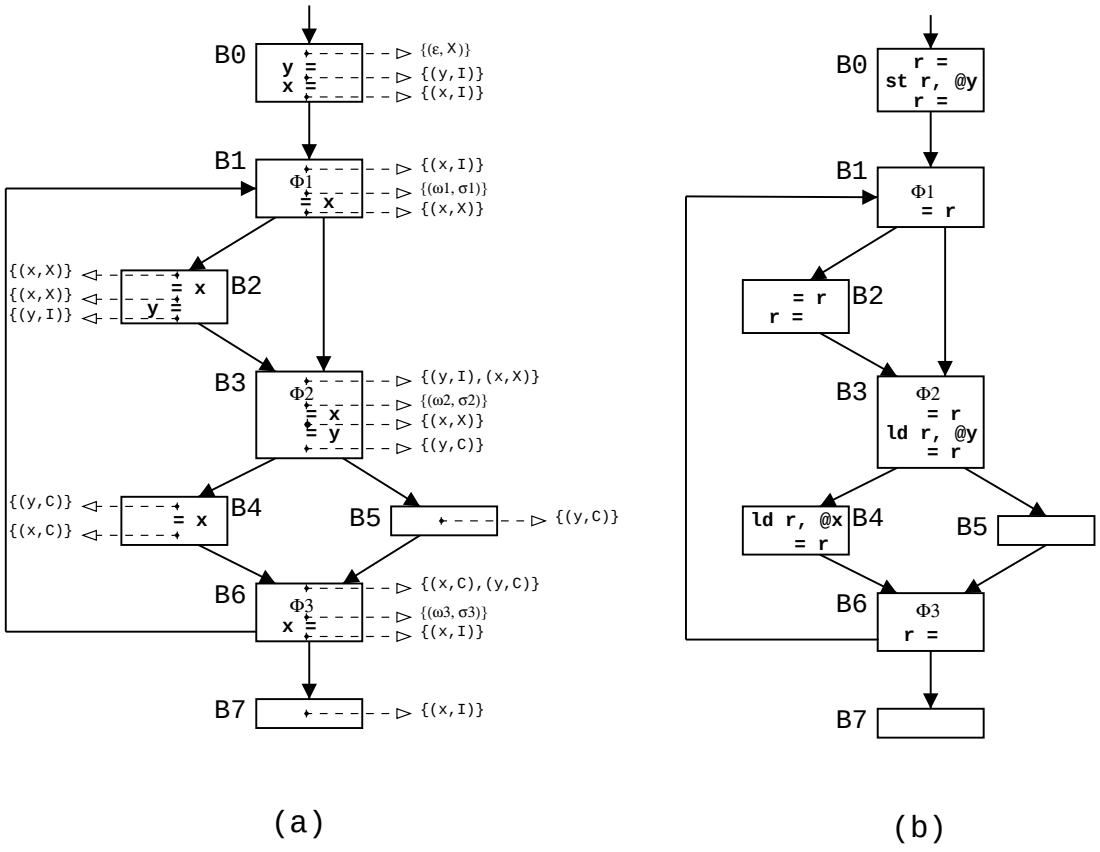

Figura 5.1: Exemplo de alocação global de registradores. (a) *Live range* com as variáveis x e y, anotado com os conjuntos de RCR. (b) Código com as instruções não dependentes das funções ϕ.

como variável y, ou seja (y,C) e (y,I), além do estado com o registrador vazio. Analisando-se estas alternativas, vê-se que, para qualquer estado com a variável y, seria necessário um load desta variável na aresta $B_4 \rightarrow B_6$. Já para o estado com o registrador vazio, nenhuma instrução de load ou store é necessária, e portanto esta é a solução escolhida para a função ϕ_3.

Para resolução da funções ϕ_1 e ϕ_2, utiliza-se o algoritmo LRO. Para o caso particular de um componente conexo que é uma aresta, o que acontece aqui, este algoritmo se resume a testar todos os pares de combinações das possíveis soluções para as duas funções ϕ. As soluções candidatas para a função ϕ_1 são apenas (x,I), (x,C) e $(\epsilon, \mathtt{X})$, uma vez que nenhuma referência à variável y alcança ou é alcançada por ϕ_1. Para ϕ_2, as soluções candidatas são todos os estados de registrador possíveis, ou seja, (x,I), (x,C), (y,I), (y,C) e $(\epsilon, \mathtt{X})$. As soluções escolhidas para estas duas funções são $\phi_1 =$ (x,C) e $\phi_2 = (\epsilon, \mathtt{X})$, com um custo total de três instruções inseridas. O código resultante após a resolução de todas as funções ϕ é mostrado na figura 5.2(b).

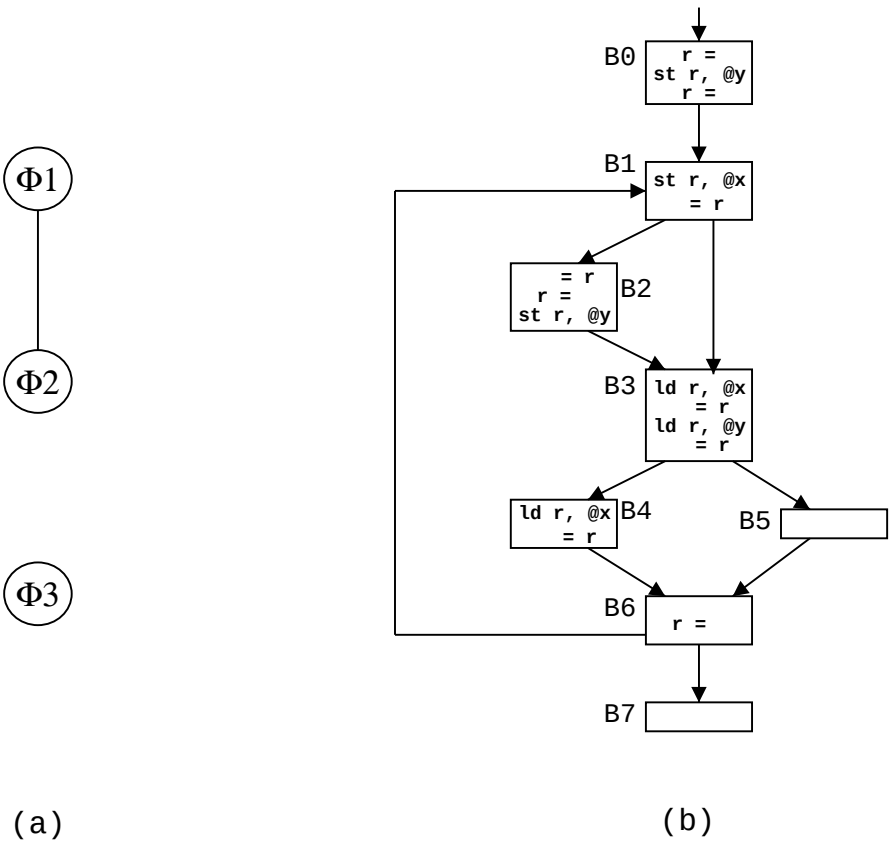

Figura 5.2: (a) Grafo de dependências ϕ. (b) Código final resultante.

Capítulo 6

Conclusões

Neste trabalho, foi estudado o problema de alocação global de registradores de endereçamento para referências a vetores dentro de laços de programas para DSPs. Este problema é denominado *Global Array Reference Allocation* (GARA). A relevância deste problema provém dos poucos registradores de propósito geral existentes e dos modos de endereçamento restritos tipicamente disponíveis nestes processadores, bem como do fato de várias aplicações dos mesmos incluir muito processamento de seqüências de dados geralmente armazenadas em vetores.

As contribuições do presente trabalho para o estudo de GARA podem ser divididas em contribuições teóricas e práticas. Do ponto de vista teórico, foi feito um estudo sobre o sub-problema central de GARA. Este sub-problema consiste em, dado um registrador de endereçamento e o conjunto de referências a vetores de um laço que serão alocadas a ele, encontrar o mínimo de instruções necessárias para mantê-lo apontando para o endereço correto em todos os pontos do laço. No capítulo 3, este sub-problema foi modelado como um problema em grafos, definindo-se então o *Grafo de Dependências* ϕ (DG_ϕ), e provado ser NP-difícil em geral. Além disso, foi proposto um algoritmo eficiente, baseado na técnica de programação dinâmica, para resolver este sub-problema quando o DG_ϕ é acíclico. Baseado nestes resultados, foram propostos dois métodos para resolver GARA. O primeiro é uma extensão do método proposto em [19], sendo uma solução heurística. O segundo, resolve o problema de GARA de forma ótima, apesar de ser computacionalmente ineficiente.

Do ponto de vista prático, a implementação no compilador GCC das técnicas propostas neste trabalho e em [19] possibilitou a comparação destes métodos. Os resultados obtidos mostraram que as heurísticas encontram soluções bastante próximas da solução ótima para GARA. Além disso, os resultados experimentais também mostraram que a proporção dos DG_ϕs que são acíclicos é bastante grande. Na prática, isso possibilita que a heurística proposta no capítulo 3 produza bons resultados, bem como permite ao método exato,

apresentado no capítulo 4, executar em um tempo razoável.

Por fim, acredita-se que a definição do Grafo de Dependências ϕ possa vir a ser útil para outras otimizações globais de código em compiladores. Como sugestão, foi proposta uma modelagem para utilizar as técnicas criadas para GARA para o problema de alocação global de registradores de propósito geral (seção 5.1). Fica como trabalho futuro a investigação prática da adequação desta abordagem.

Bibliografia

[1] The GNU Compiler Collection Project. http://gcc.gnu.org.

[2] A.V. Aho and S.C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[3] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison Wesley, Boston, 1986.

[4] Analog Devices. *ADSP-2100 Family User's Manual*.

[5] G. Araujo. *Code Generation Algorithms for Digital Signal Processors*. PhD thesis, Princeton University, May 1997.

[6] Guido Araujo, Ashok Sudarsanam, and Sharad Malik. Instruction set design and optimizations for address computation in dsp processors. In *Proceedings of the 9th International Symposium on System Synthesis*, pages 31–37. IEEE, November 1996.

[7] Sunil Atri, J. Ramanujam, and Mahmut Kandemir. Improving offset assignment for embedded processors. In *Proceedings of 13th Workshop on Languages and Compilers for Parallel Computing, LCPC'2000, LNCS 2017*, pages 158–172, August 2000.

[8] D. H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Software Practice and Experience*, 22(2):101, February 1992.

[9] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill code minimization via interference region spilling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–295, June 1997.

[10] R. Bodik and R. Gupta. Array data-flow analysis for load-store optimizations in superscalar architectures. *International Journal of Parallel Programming*, 24(6):481–512, 1996.

[11] F.T. Boesch and J.F. Gimpel. Covering the points of a digraph with point-disjoint paths and its application to code optimization. *Journal of the ACM*, 24(2):192–198, April 1977.

[12] D.G. Bradlee, S.J. Eggers, and R.R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, April 1991.

[13] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proc. of the ACM SIGPLAN'89 on Conference on Programming Language Design and Implementation*, pages 275–284, July 1989.

[14] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 53–65, June 1990.

[15] D. Callahan and B Koblenz. Register allocation via hierarchical graph coloring. In *Proc. of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 192–203, June 1991.

[16] G. Chaitin. Register allocation and spilling via graph coloring. In *Proc. of the ACM SIGPLAN'82 Symposium on Compiler Construction*, pages 98–105, June 1982.

[17] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Water, and Wen mei W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. pages 266–275, May 1991.

[18] F.C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, October 1990.

[19] M. Cintra and G. Araujo. Array reference allocation using SSA-Form and live range growth. In *Proceedings of the ACM SIGPLAN LCTES 2000*, pages 26–33, June 2000.

[20] Marcelo Cintra. Address Register Global Allocation using Indexing Graph and an SSA Form Variation. Master's thesis, University of Campinas, April 2000. (in Portuguese).

[21] Keith Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers, 2003 (expected).

[22] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. An efficient method of computing static single assignment form. In *Proc. of the ACM POPL'89*, pages 23–25, 1989.

[23] Erik Eckstein and Andreas Krall. Minimizing cost of local variables access for DSP-processors. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 20–27, May 1999.

[24] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, New York, 1979.

[25] Catherine Gebotys. DSP address optimization using a minimum cost circulation technique. In *Proceedings of the International Conference on Computer-Aided Design*, pages 100–103. IEEE, November 1997.

[26] L. George and A.W. Appel. Iterated register coalescing. *ACM Trans. Programming Language and Systems*, 18(3):300–324, May 1997.

[27] J.R. Goodman and A.W. Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 1988 Conference on Supercomputing*, pages 442–452, July 1988.

[28] R Gupta, M.L. Soffa, and D. Ombres. Efficient register allocation via coloring using clique separators. *ACM Trans. Programming Language and Systems*, 16(3):370–386, May 1994.

[29] C.Y. Hitchcock III. *Addressing Modes for Fast and Optimal Code Generation.* PhD thesis, Carnegie-Mellon University, December 1987.

[30] Cunho Lee, Miodrag Potkonjak, and Willian H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. December 1997.

[31] R. Leupers. Code generation for embedded processors. In *Proceedings of the International Symposium on System Synthesis 2000.* IEEE, September 2000.

[32] R. Leupers and Peter Marwedel. *Retargetable Compiler Technology for Embedded Systems.* Kluwer Academic Publishers, 2001.

[33] Rainer Leupers, Anupam Basu, and Peter Marwedel. Optimized array index computation in DSP programs. In *Proceedings of the ASP-DAC.* IEEE, February 1998.

[34] Rainer Leupers and Fabian David. A uniform optimization technique for offset assignment problems. In *Proceedings of the ACM SIGDA 11th International Symposium on System Synthesis*, pages 3–8, December 1998.

[35] S. Liao, S. Devadas, K. Keutzer, and A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18:235–253, May 1996.

[36] S.Y. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. In *Proc. of 1995 ACM Conference on Programming Language Design and Implementation*, 1995.

[37] Lucent Technologies Inc. *DSP1611/17/18/27/28/29 Digital Signal Processor*, January 1998.

[38] Motorola. *DSP56000/DSP56001 Digital Signal Processor User's Manual*, 1990.

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

[40] Guilherme Ottoni and Guido Araujo. Efficient array reference allocation for loops in embedded processors. In *Proceedings of the IEEE Workshop on Embedded System Codesign, ESCODES'02*, pages 63–68, September 2002.

[41] Guilherme Ottoni, Sandro Rigo, Guido Araujo, Subramanian Rajagopalan, and Sharad Malik. Optimal live range merge for address register allocation in embedded programs. In *Proceedings of the 10th International Conference on Compiler Construction, CC2001, LNCS 2027*, pages 274–288, April 2001.

[42] P. Paulin, C. Liem, M. Cornero, F. Naçabal, and G. Goossens. Embedded software in real-time signal processing systems: Application and architecture trends. In *Proceedings of the IEEE*, volume 85. IEEE, March 1997.

[43] Massimiliano Poletto and Vivek Sarkar. Linear time register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, September 1999.

[44] Todd Proebsting and Charles Fischer. Demand-driven register allocation. *ACM Transactions on Programming Languages and Systems*, 18(6):683–710, November 1996.

[45] Amit Rao and Santosh Pande. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 128–138, May 1999.

[46] R. Sethi. Complete register allocation problems. *SIAM J. Computing*, 4(3):226–248, September 1975.

[47] R. Sethi and J.D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, October 1970.

[48] Ashok Sudarsanam, Stan Liao, and Srinivas Devadas. Analysis and evaluation of address arithmetic capabilities in custom DSP architectures. In *Design Automation Conference*, pages 287–292, 1997.

[49] Omri Traub, Glenn Holloway, and Michael Smith. Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151, June 1998.

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