Explorando o Espaço de Projeto com o Objetivo de Redução do Consumo de Energia usando *Reduced Bit-width Instruction Set Architecture* (rISA)

Jonatan Fracasso da Silva
Universidade de Caxias do Sul
jfsilva4@ucs.br

Flávio Rech Wagner
Universidade Federal do Rio Grande do Sul
flavio@inf.ufrgs.br

Sandro Neves Soares
Universidade de Caxias do Sul
snsoares@ucs.br

Resumo

Este trabalho apresenta um framework para a exploração do espaço de projeto usando Reduced Bit-width Instruction Set Architecture (rISA). rISA é um recurso arquitetural empregado para a redução de código e, também, para a redução do consumo de energia em processadores embarcados. O framework rISA herda, da sua infraestrutura de modelagem, recursos que o distinguem de outras ferramentas, relacionados à facilidade de acesso e de uso. Além disso, ele implementa um algoritmo de redução de código que tem, como diferencial, um tratamento mais eficiente dos desvios reduzidos. O uso deste framework permitiu a descoberta de um novo formato rISA, não considerado por outros trabalhos correlatos, que é mais simples e que apresenta melhores resultados do que outros conjuntos mais complexos em termos de redução no consumo de energia. A simplicidade deste novo formato advém do emprego de menos instruções no algoritmo de redução e os resultados obtidos, em experimentos com programas do MiBench, foram superiores a propostas anteriores em até 14%.

1. Introdução

Telefones celulares, *MP3 players*, *handhelds* e outros equipamentos têm o seu funcionamento sustentado, em muitas situações, por baterias, que fornecem algumas horas de funcionamento aos seus proprietários. A redução do consumo de energia em equipamentos como estes é um aspecto importante e assunto de pesquisa, atualmente, na área de sistemas embarcados. Como exemplo, pode-se citar uma experiência recente no Japão que mostrou que os consumidores não irão adotar um novo produto, ainda que com mais recursos, se ele fornece menos tempo de conversa ou de bateria do que outros com os quais eles já estão acostumados, o que foi comprovado na baixa venda inicial dos primeiros celulares 3G em relação aos da geração 2G [1].

rISA, ou *Reduced Bit-width Instruction Set Architecture* [2], é um recurso arquitetural empregado para reduzir o tamanho do código de programas. Processadores com este recurso executam instruções de dois conjuntos distintos: o *normal* e o *reduzido*. O conjunto de instruções reduzidas engloba as instruções mais freqüentes, no código do programa, codificadas em um número menor de bits. São exemplos de processadores rISA: o ARM7TDMI, o MIPS32/16 bit TinyRISC, o STMicro's ST100 e o ARC Tangent.

Os processadores rISA expandem dinamicamente as instruções reduzidas em suas correspondentes instruções normais. Geralmente, cada instrução reduzida conta com uma equivalente no conjunto de instruções *normal*. Isto faz com que o processo de tradução seja simples e direto, exigindo a inclusão apenas de uma simples unidade de tradução no hardware do processador. Nenhum outro módulo de hardware é necessário [2].

Em adição à redução de tamanho do código do programa, um programa com instruções reduzidas requer menos acessos à memória de instruções quando é executado. Conseqüentemente, há um decréscimo no consumo de energia neste subsistema. Se todo o código do programa pudesse ser expresso em termos de instruções reduzidas, então aproximadamente 50% de redução de tamanho seria obtida, e conseqüentemente haveria uma proporcional redução no consumo de energia na memória de instruções [3].

Este trabalho apresenta um framework, denominado framework rISA, que pode ser empregado pelo projetista para experimentar diferentes conjuntos de instruções reduzidas, a fim de obter o mais adequado para uma dada aplicação, quando o objetivo é a redução do consumo de energia. O framework é de código aberto e está disponível na internet. Ele é

baseado em um modelo do processador MIPS e dispõe de recursos visuais e interativos para que o projetista configure os experimentos e analise os resultados obtidos. Além disso, ele implementa um algoritmo de redução de código que impõe menos restrições à existência de desvios entre blocos reduzidos. O uso deste framework possibilitou uma descoberta importante, relacionada a um novo formato de instruções reduzidas, não considerado por outros trabalhos correlatos [2], que é mais simples e que apresenta resultados melhores em até 14% do que outros conjuntos mais complexos, no que diz respeito à redução no consumo de energia.

O texto que segue está organizado da seguinte forma: a Seção 2 detalha mais o recurso rISA. A Seção 3 discute trabalhos que tratam desta técnica. A Seção 4 descreve o método utilizado para a redução do código de programas. A Seção 5 descreve a implementação do framework e a Seção 6, a sua aplicação para a exploração do espaço de projeto com rISA. A Seção 7 descreve resultados obtidos com o uso de um novo formato de instruções reduzidas, e a Seção 8 apresenta as conclusões e discute trabalhos futuros.

2. O Recurso Arquitetural rISA

A Figura 1 relaciona o código normal de um trecho do programa CRC32 do MiBench [6] (Figura 1a), executando no processador MIPS, com o respectivo código reduzido obtido (Figura 1b). Este último constitui um bloco de instruções reduzidas ou bloco reduzido. Um programa compilado com a técnica rISA conterá diversos destes blocos, além de trechos compostos apenas de instruções normais. As instruções de troca de modo de execução e *nops* reduzidos (*change mode* e *rISA_nop*) serão explicadas na Seção 4. Cada linha preenchida nestas figuras corresponde a uma posição da memória de instruções. A redução obtida neste caso é de, aproximadamente, 22% (7 posições ocupadas no lugar de 9 – a instrução *change mode* ocupa 1 posição).

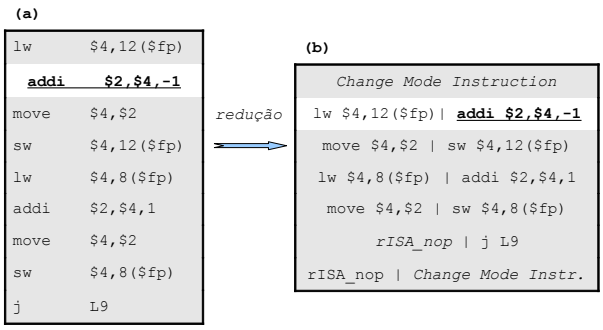

Figura 1. Código Normal (a) e Reduzido (b)

A Figura 2 mostra a conversão da instrução *addi,* presente no código da Figura 1, de 32 bits originalmente (Figura 2a), na sua correspondente de 16 bits (Figura 2b), usando o formato *rISA_4444*. Este formato é assim chamado porque emprega 4 seqüências de 4 bits para especificar, respectivamente, o *opcode*, o registrador fonte (rs), o registrador destino (rd) e o imediato (imm).

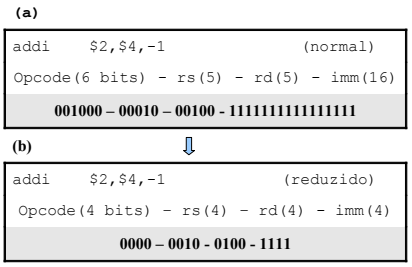

Figura 2. Instrução Addi: Normal (a) e Reduzida (b)

Como resultado do reduzido número de bits nas instruções reduzidas para a especificação do *opcode*, apenas um subconjunto das diferentes instruções de um programa (distintas pelo seu *opcode,* neste caso) pode ser efetivamente reduzido. No formato *rISA_4444*, por exemplo, são apenas 16, ou, mais especificamente, todas as ocorrências destas 16 instruções com *opcodes* diferentes entre si. Assim, a escolha correta das diferentes instruções a serem reduzidas, para um dado programa, passa a ser o principal problema a ser solucionado, independentemente se o objetivo é a redução de código (onde a escolha recai sobre as instruções mais freqüentes) ou a redução do consumo de energia (onde a escolha recai sobre as instruções mais executadas). É natural que as instruções mais freqüentes, ou as mais executadas, não sejam as mesmas para diferentes aplicações.

A simplicidade do novo formato, proposto neste trabalho, advém do fato de que ele seleciona menos instruções distintas (com diferentes *opcodes*) do código original para a redução (apenas 8 *opcodes* distintos). Conseqüentemente, há uma simplificação na lógica da unidade de tradução, que precisa expandir menos instruções diferentes em suas correspondentes normais. A quantidade maior de bits que sobra da especificação do *opcode* das instruções reduzidas pode ser então usada para a especificação de valores imediatos, o que diminui o número de instruções que não podem ser reduzidas devido à impossibilidade de representar estes valores com os bits disponíveis. Estas instruções que são selecionadas para a redução, mas que não podem ser efetivamente reduzidas, são ditas *descartadas* da redução. Como vai se ver mais adiante neste texto, o número de instruções que deixam de ser descartadas supera o número daquelas que deixam de ser

selecionadas para a redução, levando-se em consideração as várias ocorrências de cada *opcode*. Como resultado de mais instruções reduzidas no programa, dentre aquelas que são mais executadas, a redução do consumo de energia na memória de instruções aumenta.

3. Trabalhos Relacionados

Em [2] é apresentado o uso de rISA para a redução do tamanho do código de programas e, conseqüentemente, para a redução do tamanho da memória ROM exigida no sistema, o que diminui o custo do produto final. Isso assume grande importância para equipamentos com grande volume de produção e de vendas, como telefones celulares, PDAs, roteadores, e outros que empregam sistemas embarcados. Processadores contemporâneos que usam rISA, e suas implementações, são também descritos em [2]. Foram empregados nos experimentos diversos formatos de instruções reduzidas. Estes formatos variam na quantidade de bits que empregam para a especificação dos diferentes campos da instrução (*opcode*, registradores e imediatos), assim como no emprego ou não de instruções especiais, que não fazem parte do código original do programa, e que têm como objetivo complementar informações que não podem ser representadas com o reduzido número de bits disponível no formato em questão. Estas instruções reduzidas especiais, como *rISA_extend* por exemplo, usada para completar valores imediatos, requerem um processo de tradução mais complexo e geram um impacto negativo no desempenho da aplicação, como será explicado na próxima seção. O número de bits usados para especificar o *opcode*, nestes formatos abordados, é 4 e 7, o que permite a redução de 16 e 128 instruções distintas (diferentes *opcodes*) do código original, respectivamente. O trabalho relata que o formato *rISA_4444* (16 *opcodes* distintos e acesso a 16 registradores por operando) apresenta o melhor resultado em termos de redução de código, levando-se em consideração o compromisso entre a redução obtida e a complexidade da unidade de tradução.

O trabalho descrito em [3] afirma que a principal vantagem de rISA é a alta taxa de compressão de código e a economia de energia obtidas, com um mínimo de alterações no hardware do processador. Neste trabalho, é mostrado que um algoritmo que busque apenas a redução de código não atinge os melhores resultados em termos de consumo de energia, uma vez que não leva em consideração o aspecto dinâmico de execução da aplicação. Por isso, um algoritmo que leva em consideração este aspecto dinâmico é apresentado e aplicado sobre um modelo de processador MIPS com e sem memória cache. É relatada uma economia média de 26% no acesso à memória de instruções sobre um modelo base sem rISA, numa organização sem memória cache. No algoritmo de redução descrito, e detalhado em [5], é dito que as instruções selecionadas para a redução precisam pertencer a blocos básicos, isto é, precisam estar em porções de código que não contêm desvios, exceto no seu início ou no seu final, nem contêm instruções que são destinos de desvio.

Não há menção nestes trabalhos, e em outros que tratam de rISA [4,5], ao uso de um formato com um número de *opcodes* inferior a 16, nem a preocupação de quantificar as possíveis causas de descarte de instruções dentre aquelas passíveis de redução. Com mais instruções descartadas, há uma fragmentação dos blocos reduzidos, o que impacta negativamente tanto a redução de código quanto a redução do consumo de energia. Igualmente não há informações sobre como acessar as ferramentas utilizadas para executar os experimentos.

4. O Método

O trabalho descrito em [2] mostra que um formato rISA simples, *rISA_4444* (16 *opcodes* distintos e acesso a 16 registradores por operando, ou imediatos de 4 bits), apresenta resultados semelhantes, em termos de redução de código, a formatos mais complexos e, além disso, apresenta os seguintes benefícios: (1) mantém a simplicidade da unidade de tradução; e (2) compromete menos o desempenho. Este compromisso norteou a nossa decisão de usar, primeiramente, este formato em um modelo do processador MIPS com pipeline. Ele será chamado, no texto que segue, de *rISA_16ops*.

A simplicidade do *rISA_16ops* advém do fato de que este formato não usa instruções especiais, como *rISA_extend*, por exemplo, para completar valores imediatos. Estas instruções especiais exigem uma lógica adicional na unidade de tradução, uma vez que elas não contam com instruções correspondentes no conjunto *normal*. Além disso, como não fazem parte do código original do programa (tomando, como base, o programa com instruções normais apenas), ao serem executadas, elas impactam o desempenho, pois gastam ciclos de relógio adicionais, e, ainda, elas aumentam o tamanho dos blocos de instruções reduzidas. Neste formato *rISA_16ops*, se o valor não pode ser representado com a quantidade de bits disponível, a instrução correspondente é descartada da redução.

A Figura 3 apresenta o nosso algoritmo para a conversão do código original em código reduzido usando rISA. Se o modelo de processador foi configurado pelo projetista para usar o recurso arquitetural rISA (*mips.usingRISA*), o método *mapRegisters* encarrega-se de alterar o código do programa de forma que ele atinja o seu objetivo usando apenas metade dos registradores disponíveis, isto é, 16 de um total de 32. O método *markCandidates* é chamado para marcar as instruções do programa que podem ser reduzidas, uma vez que os seus *opcodes* pertençam ao conjunto das diferentes instruções a serem reduzidas, conjunto este previamente especificado pelo projetista. As instruções marcadas, ou selecionadas, não se restringem àquelas pertencentes a blocos básicos, como em [3].

O método *IsPossibleToReduceCandidates* descarta instruções marcadas que não podem ser reduzidas devido ao limitado número de bits para especificar os seus operandos, situação chamada de *overflow* de operandos. *DiscardSmallBlocks* é usada para montar blocos de instruções marcadas adjacentes, e também para descartar aqueles blocos que não são de interesse para a redução, devido ao seu pequeno tamanho, já levando-se em consideração o *overhead* causado pela posterior inserção das instruções de troca de modo de execução. Cada bloco reduzido (ou bloco de instruções reduzidas) do programa possui, no seu início, uma instrução de troca de modo de execução, de *normal* para *reduzido* (*mx* [2]) e, no seu final, a correspondente instrução de troca de modo de *reduzido* para *normal* (*rISA_mx* [2]). A primeira é uma instrução normal e a segunda, uma instrução reduzida. Elas indicam ao processador se a unidade de tradução deve ou não ser ativada.

```
if (mips.usingRISA ( )) {
   mips.rISA.mapRegisters ( );
   mips.rISA.markCandidates ( );
   mips.rISA.isPossibleToReduceCandidates();
   mips.rISA.discardSmallBlocks ();
   while(mips.rISA.treatBranchesAndJumps())
        mips.rISA.discardSmallBlocks ();
   mips.rISA.countFinalBlocks ();
   mips.rISA.translateToRISAstep1 ( );
   mips.rISA.translateToRISAstep2 ( );
   mips.rISA.generateFinalCode ( output);
}
```

Figura 3. Algoritmo para a Conversão de Código

TreatBranchesAndJumps é chamado para descartar as instruções marcadas que estão envolvidas, seja como origem ou destino, em *branches* ou *jumps* de um bloco reduzido para um bloco normal (ou bloco de instruções normais) e vice-versa. A exceção fica por conta de um *branch* ou *jump* ocorrendo de um bloco de instruções normais para o início de um bloco de instruções reduzidas. Este método é chamado, repetidamente, em conjunto com o método *DiscardSmallBlocks* até que não haja mais *branches* ou *jumps* a serem descartados. Ele, em conjunto com o método *markCandidates*, extrapola as regiões de redução para além da fronteira dos blocos básicos, isto é, instruções marcadas adjacentes podem se estender para além de um bloco básico.

CountFinalBlocks é um método para a captura de dados estatísticos. O método *translateToRISAstep1*, efetivamente, cria os blocos reduzidos, inserindo as instruções de troca de modo de execução, *mx* e *rISA_mx*, além de inserir *nops* reduzidos para completar o tamanho de palavras no final dos blocos, a fim de manter o alinhamento de instruções nas palavras de memória. Tais *nops* são, igualmente, inseridos para que *branches* e *jumps* reduzidos fiquem posicionados na segunda metade de uma palavra de memória (ver a instrução *j L9* na Figura 1b), o que simplifica a lógica da unidade de tradução, uma vez que ela não precisa tratar a instrução reduzida que segue o salto. *translateToRISAstep2* recalcula os *offsets* (valores a serem adicionados ao *Program Counter* para o cálculo do destino de saltos) de *branches* e *jumps*, gera as seqüências de 16 bits das instruções reduzidas, assim como as encapsula em palavras de 32 bits. *GenerateFinalCode* é usado para gerar o código final no formato empregado pelo modelo do processador MIPS.

Apesar de este algoritmo ser muito parecido com o descrito em [3] e [4], ele apresenta, como diferencial, o fato de que um bloco reduzido pode conter instruções que extrapolam os limites de um único bloco básico. Isso faz com que mais desvios entre blocos reduzidos possam ser executados, não apenas aqueles desvios para o início de um outro bloco, o que, por sua vez, tem o mesmo efeito de combinar diferentes blocos reduzidos. Por outro lado, o algoritmo descrito em [3] e [4] possui um módulo adicional para determinar a viabilidade de reduzir, ou não, um bloco previamente selecionado, que leva em consideração o impacto desta redução sobre a compressão de código e também sobre o desempenho.

A Figura 4 mostra o número de instruções, originalmente marcadas para a redução pelo método *markCandidates*, que foram descartadas no programa

qsort do MiBench, devido a uma das seguintes causas: *overflow* de operandos, tratamento de *branches* e *jumps*, ou tamanho pequeno do bloco. O conjunto de instruções reduzidas utilizado, no formato *rISA_16ops*, inclui as 16 instruções distintas mais executadas no *qsort*. Os dados nesta figura, que mostram um grande número de instruções descartadas devido ao reduzido número de bits – *overflow* de operandos, motivaram-nos a repetir os experimentos com um número ainda menor de instruções no conjunto: 8 no lugar de 16. Com apenas 8 instruções, sobra 1 bit da especificação do *opcode* para ser utilizado na especificação de valores imediatos. A Figura 5 mostra as mesmas informações da Figura 4, só que agora usando um formato rISA que prevê 8 instruções distintas. Ele foi denominado de *rISA_8ops*. O conjunto de instruções reduzidas utilizado inclui as 8 instruções distintas mais executadas no *qsort*.

Figura 4. Nro.de Instruções Descartadas e Causas – *rISA_16ops*

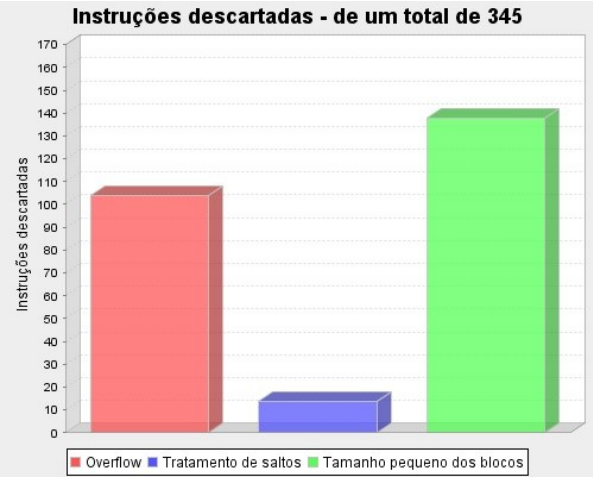

Figura 5. Nro.de Instruções Descartadas e Causas – *rISA_8ops*

Os números menores mostrados na Figura 5, em comparação aos da Figura 4, levaram-nos a empregar em nossos experimentos com rISA o novo formato *rISA_8ops,* uma vez que, com mais instruções reduzidas no programa, dentre aquelas que são mais executadas, maior será a redução do consumo de energia na memória de instruções. Nestes casos abordados, a redução do consumo de energia foi de 14% e 26%, respectivamente. É interessante notar que o total de instruções originalmente marcadas para a redução (antes do descarte) não difere muito nas duas situações.

5. O Framework

T&D-Bench [7, 8] é um framework para a exploração do espaço de projeto de processadores. Ele estende o conceito de *Architecture Description Languages* [9], amplamente empregadas no projeto de processadores, fornecendo, de maneira complementar a uma linguagem de descrição de fácil uso, chamada *T&D-Simplified Description Language*, ou TDSDL, um conjunto de métodos especializados, denominados *macros*. As macros são usadas para o acesso e a manipulação dos aspectos de arquitetura, organização e temporização do processador. Os seus argumentos são constituídos de informações sobre o processador modelado, fornecidas pelo projetista na TDSDL.

O T&D-Bench busca fornecer um processo de projeto simplificado e rápido, em conjunto com a flexibilidade para a modelagem de mecanismos mais complexos ou específicos de um dado processador, como o recurso arquitetural rISA por exemplo. Tais mecanismos são difíceis de serem modelados usando-se unicamente as construções de uma linguagem declarativa. Os modelos de processadores do T&D-Bench podem incorporar, automaticamente, recursos visuais e interativos para que o projetista acompanhe e conduza as rodadas de simulação.

No caso dos experimentos com rISA, adotou-se o simulador do processador MIPS do T&D-Bench, um dos modelos disponíveis no framework. A unidade de tradução foi descrita em conjunto com a unidade de decodificação, codificada em um método da classe *processor* (chamada de classe *mips* neste modelo). Ela é a classe principal nos modelos de processadores do T&D-Bench e mantém todas as informações, provenientes da TDSDL, em estruturas de dados que podem ser manipuladas usando-se as macros. Foi, igualmente, adicionado um componente rISA ao modelo do MIPS (como uma instância pertencente à classe *mips* – ver Figura 2) para executar o método descrito na seção anterior. Quatro programas do

MiBench, CRC32, bitcount, qsort e stringsearch, estão disponíveis para serem empregados nos experimentos com rISA. O código C destes programas, e de outros que venham a ser criados, é compilado para Assembly usando-se o *gnu-gcc cross-compiler*. As simulações podem ser executadas em modo *batch*, mas os recursos visuais e interativos disponíveis são de extrema importância, principalmente para a análise de resultados, conforme será apresentado mais adiante neste texto. A Figura 6 mostra o nosso framework para a exploração do espaço de projeto com rISA que, na verdade, é construído a partir da especialização de um modelo de processador disponível no T&D-Bench.

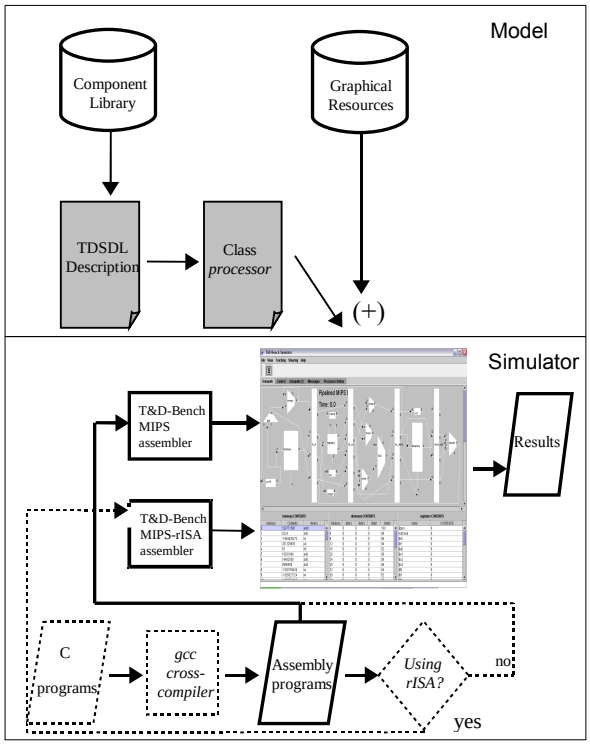

Figura 6. Framework rISA

6. Explorando o Espaço de Projeto

A Figura 7 mostra a GUI (*Graphical User Interface*) do framework rISA. As janelas que aparecem nesta figura são acessadas através do item *Simular com rISA* do menu *Simular*. Antes de configurar o experimento através das opções disponíveis nestas janelas, o projetista deve escolher o programa a ser executado. Alguns programas, em Assembly, já estão disponíveis no framework, como os programas CRC32, bitcount, qsort e stringsearch do MiBench. Novos programas em C podem ser desenvolvidos e compilados para Assembly usando-se o *gnu-gcc*, configurado para gerar código MIPS.

As opções de configuração de um experimento incluem: não usar o recurso arquitetural, a seleção de conjuntos de instruções reduzidas predefinidos e, inclusive a personalização de um conjunto, tanto em formato (uso de 8 ou 16 *opcodes*), quanto em conteúdo, isto é, a definição de quais instruções distintas serão incluídas no conjunto. Ainda na configuração, existe a possibilidade de optar pela geração de um gráfico no final da simulação. Tal gráfico pode mostrar uma série de informações a respeito do experimento, isto é, da execução do programa reduzido com os conjuntos selecionados pelo projetista (e até mesmo sem redução). Estas informações incluem: números relacionados a causas do descarte de instruções (*overflow* de operandos, tratamento de *branches* e *jumps* e tamanho pequeno do bloco), assim como quantidade e tamanho dos blocos reduzidos, quantidade de instruções buscadas no sistema de memória, ciclos de execução e tempo de relógio da simulação. Elas são fundamentais para a análise detalhada dos experimentos por parte do projetista. A Figura 8 mostra a apresentação dos resultados, em termos de número de instruções buscadas e de ciclos de relógio, para a execução do programa *qsort* sem o uso da técnica rISA e com outros cinco conjuntos de instruções reduzidas predefinidos.

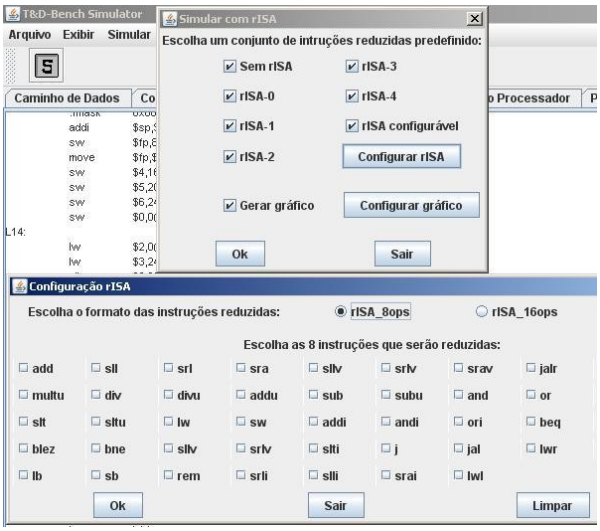

Figura 7. GUI do Framework rISA

7. Resultados

O experimento, cujos resultados são descritos a seguir, foi feito usando-se os programas CRC32, bitcount, qsort e stringsearch do MiBench. O uso destes programas, ao invés de novos programas criados especificamente para usá-los no experimento, foi nossa escolha porque eles fazem parte de um conjunto de

benchmarks representativo comercialmente e bastante empregado no projeto de sistemas embarcados. Representativo comercialmente significa que há diferentes categorias de aplicações no MiBench, cada uma dirigida a uma área específica do mercado de sistemas embarcados.

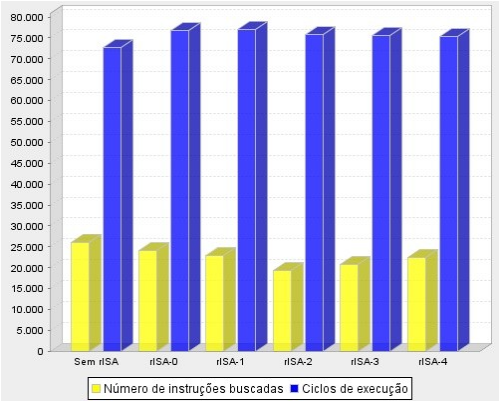

Figura 8. Visualização dos Resultados do Experimento

Primeiramente, usando o simulador do MIPS monociclo, disponível no T&D-Bench, descobriu-se, para cada um dos 4 programas, quais as suas 16 instruções distintas (diferente *opcodes*) mais executadas. Com elas, foram formados quatro diferentes conjuntos de instruções reduzidas. Cada conjunto foi aplicado no seu respectivo programa, usando o formato *rISA_16ops*, e o código reduzido foi simulado. Depois, mais 4 conjuntos reduzidos foram definidos, cada um contendo apenas as 8 instruções distintas mais executadas em cada um dos 4 programas. Foram executadas, então, mais 4 simulações, nas quais cada conjunto de 8 *opcodes* foi aplicado no seu respectivo programa, usando o formato *rISA_8ops*, e o código reduzido resultante foi simulado.

Os gráficos nas Figuras 9 a 12, gerados pelo próprio framework rISA, apresentam os valores da redução de consumo de energia, para cada programa, com os dois conjuntos de instruções reduzidas, o de 16 *opcodes* e o de 8 *opcodes*, em comparação à execução do código sem redução. Esta redução de consumo de energia no subsistema de memória é expressa pelo reduzido número de *fetches*, isto é, pelo menor número de acessos a este subsistema, mesmo parâmetro de comparação empregado em [3]. Não se está levando em consideração o consumo de energia da unidade de tradução, ainda que ela seja mais simples no formato *rISA_8ops*, porque o foco da comparação, neste momento, são os dois formatos rISA, que seriam implementados em processadores com o recurso rISA. Os valores nas figuras estão normalizados.

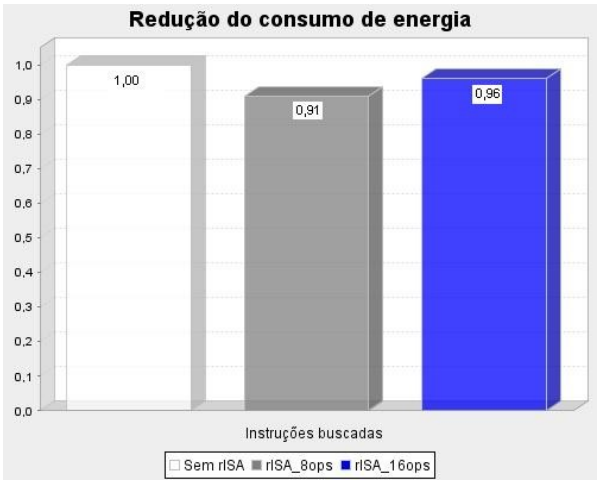

Figura 9. Resultados **bitcount** – 8 *opcodes* X 16 *opcodes*

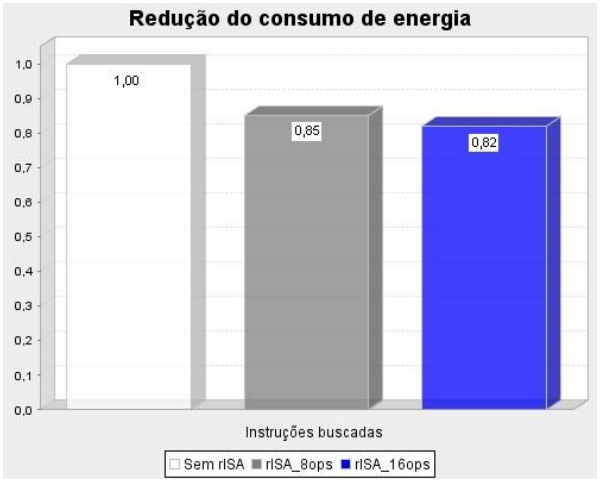

Figura 10. Resultados **CRC32** – 8 *opcodes* X 16 *opcodes*

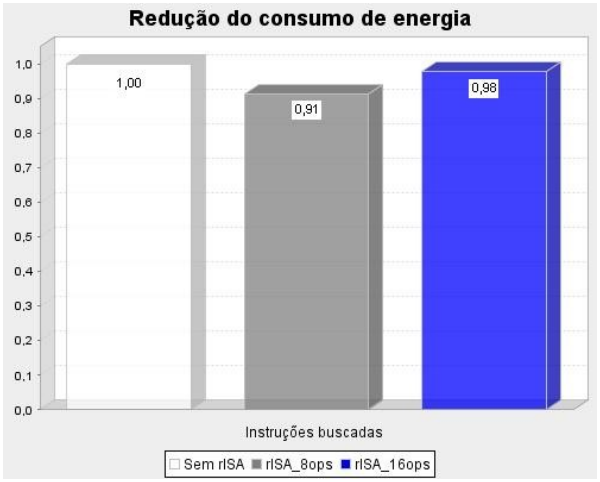

Figura 11. Resultados **stringsearch** – 8 *opcodes* X 16 *opcodes*

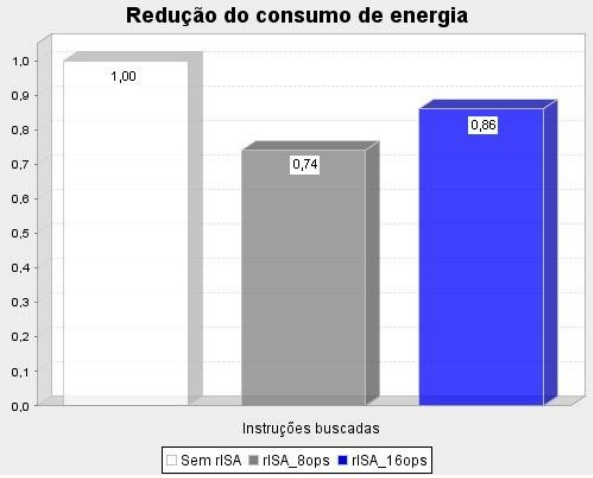

Figura 12. Resultados **qsort** – 8 *opcodes* X 16 *opcodes*

Em 3 das 4 aplicações, houve uma maior economia de energia quando foi usado um formato rISA com 8 *opcodes*. Isso é devido ao menor número de instruções descartadas da redução neste formato e, também, à natureza das aplicações, cujos laços não repetem muitas instruções distintas. A exceção fica por conta do laço principal do programa CRC32, onde um grande número de operações lógicas são executadas.

8. Conclusão e Trabalhos Futuros

Este trabalho mostrou que, devido ao menor número de instruções descartadas da redução em um novo formato rISA com 8 *opcodes*, em comparação a um formato com 16 *opcodes*, o primeiro apresenta, para 3 das 4 aplicações (programas do MiBench) usadas no experimento, melhores resultados em termos de redução do consumo de energia. Em uma das aplicações, o ganho chegou a 14%. Além disso, o formato *rISA_8ops* simplifica a unidade de tradução, que precisa expandir apenas 8 instruções distintas em suas correspondentes instruções normais.

Tais resultados foram obtidos usando-se um framework cujos recursos o distinguem de outras ferramentas, relacionados à facilidade de acesso e de uso. Por exemplo, o projetista conta com recursos visuais e interativos que facilitam o trabalho de configuração das rodadas de simulação e, principalmente, a análise dos resultados obtidos (todos os gráficos apresentados neste texto foram gerados pelo framework rISA). Além disso, ele implementa um algoritmo de redução de código que tem, como diferencial, um tratamento mais eficiente dos desvios reduzidos.

Na continuação do trabalho, pretende-se explorar a idéia de um conjunto de instruções reduzidas escolhido, dinâmica e especificamente, para uma dada aplicação, ou para diferentes trechos de uma mesma aplicação. Esta idéia parece ser viável uma vez que a simplicidade do novo formato *rISA_8ops* abre espaço para uma lógica de reconfiguração a ser inserida na unidade de tradução.

Referências

[1] A. Goren. Multimedia needs multiprocessor SoCs. http://www.eetimes.com/story/OEG20030702S0054. Acesso em: 28 ago. 2007.

[2] A. Shrivastava, P. Biswas, A. Halambi, N. Dutt and A. Nicolau. A Design Space Exploration Framework for Reduced Bit-width Instruction Set Architecture (rISA) Design. Proceedings of the International Symposium on System Synthesis – ISSS 2002.

[3] A. Shrivastava, P. Biswas, A. Halambi, N. Dutt and A. Nicolau. Energy Efficient Code Generation using rISA. Proceedings of the Asia and South Pacific Design Automation Conference – ASPDAC 2004.

[4] A. Shrivastava, P. Biswas, A. Halambi, N. Dutt and A. Nicolau. An Efficient Compiler Technique for Code Size Reduction using Reduced Bit-width ISAs. Proceedings of the International Conference on Design Automation and Test in Europe – DATE 2002.

[5] A. Shrivastava, P. Biswas, A. Halambi, N. Dutt and A. Nicolau. "Compilation Framework for Code Size Reduction using Reduced Bit-width ISAs". ACM TODAES: ACM Transactions on Design Automation of Electronic Systems, 2005.

[6] M.R.Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown, MiBench: A Free, Commercially Representative Embedded Benchmark Suite, 4th Workshop on Workload Characterization, Dec. 2001.

[7] S. Soares and F.R. Wagner. From Classroom to Research: Providing Different Services for Computer Architecture Education. Workshop on Computer Architecture Education, 2007, San Diego, Califórnia.

[8] S. Soares and F.R. Wagner. Design Space Exploration using T&D-Bench. Anais do XVI Symposium on Computer Architecture and High Performance Computing, 2004, Foz do Iguaçu, PR. p. 40-47.

[9] P. Biswas, S. Pasricha, P. Mishra, A. Shrivastava, N. Dutt and A. Nicolau. EXPRESSION. Users Manual. Version 1.0. 2003. Department of Information and Computer Science, University of Califórnia, Irvine. Available at: <http://www.ics.uci.edu/~express/>.