Um Simulador para o Processador de Arquitectura Configurável ConCISe

Orlando Moreira, Bernardo Kastrup, Nuno Lau, António Ferrari

Resumo **- O ConCISe é o protótipo de um Processador de Arquitectura Configurável desenvolvido pela Philips Research, que pretende tornar este tipo de sistemas viável para produção em larga escala. Este artigo começa por apresentar a arquitectura ConCISe. Seguidamente, descreve o processo de construção de um simulador para este processador , apresenta e discute os resultados de simulação obtidos. A análise destes resultados é feita sob uma perspectiva das relações custo/desempenho.**

Abstract- **ConCise is a Philips research project on Configurable Architecture Processors, which is expected to make this kind of systems cost-effective for large scale production in the near future. This article starts by presenting the ConCISe architecture. Then it describes the development of a simulator for a ConCISe processor and discusses some of the simulation results. The analysis of these results is done from a cost/performance perspective.**

I. INTRODUÇÃO

A. Processadores de Arquitectura Configurável

O termo *Processador de Arquitectura Configurável* (PAC) diz respeito a um tipo de arquitectura de microprocessadores que tem vindo a ser desenvolvido nos últimos anos, devido ao aparecimento no mercado de Dispositivos Lógicos Programáveis (DLPs), que permitem utilizar o mesmo circuito integrado para implementar vários circuitos digitais diferentes. Um PAC é tipicamente constituído por um *core* de processador genérico (frequentemente RISC) e uma certa quantidade de recursos DLP utilizados para implementar em *hardware* segmentos críticos da aplicação. Embora não existam (ainda) quaisquer produtos deste género no mercado, várias propostas têm sido apresentadas na comunidade científica, como sejam:

- o PRISM[1], em que FPGAs são utilizadas como co-processadores ligados a um processador Motorola comercial;
- o PRISC[2], onde DLPs são inseridos no próprio *datapath* de um processador MIPS, de forma a implementar uma unidade semelhante a uma ALU, mas que implementa operações específicas dependentes da aplicação.

Podem referir-se, além destes, outros sistemas, como sejam o DISC[3], o OneChip[4], o Garp[5] ou o Chimaera[6].

Em todas estas arquitecturas as vantagens em termos de desempenho dos PACs quando comparados com processadores tradicionais podem ser comprovadas. Há que ter em conta, contudo, o seguinte :

- em relação a um circuito especificamente desenhado para a aplicação em causa (um ASIC - *Application Specific Integrated Circuit*), os PACs têm pior desempenho, embora fiquem mais baratos, graças à possibilidade de re-utilização;
- em relação a um processador padrão, os PACs têm frequentemente melhor desempenho, mas custos mais elevados, além de um processo de desenvolvimento de aplicações mais complexo.

A área de aplicação dos PACs parece pois ficar num ponto intermédio, entre sistemas de mais fraco desempenho, mas enorme versatilidade - os processadores genéricos – e dispositivos de alto desempenho, mas orientados para uma única aplicação – os ditos ASICs.

Entretanto, a compilação é um calcanhar de Aquiles deste tipo de sistema. Um bom compilador deveria permitir ao programador de aplicações gerar código em linguagem de alto nível sem se preocupar com a tradução deste em código máquina da arquitectura em causa, mas isso, num PAC, implica que o compilador saiba escolher quais os sectores do código mais adequados para implementar em *hardware* (aqueles em que essa implementação resultará em melhoria significativa do desempenho).

A questão não é de todo trivial e, até à data, não são conhecidas soluções plenamente satisfatórias. O problema avoluma-se se se pensar atribuir à arquitectura a capacidade de configuração dinâmica (ou seja, capacidade de proceder à alteração da programação dos recursos configuráveis durante a execução da aplicação).

B. Trabalho anterior sobre o ConCISe

O ConCISe, é um PAC desenvolvido nos laboratórios da Philips Research em Eindhoven, nos Países-Baixos, que aproveita ideias já existentes (vejam-se sobretudo as similaridades entre esta arquitectura e a arquitectura PRISC), mas que introduz novidades, com o intuito de

facilitar o processo de compilação e melhorar o aproveitamento de recursos configuráveis, evitando ao mesmo tempo a necessidade de configuração dinâmica.

Utilizando um protótipo da cadeia de compilação [7] e um *profiler*, foram feitas contagens do número de instruções do ConCISe para algoritmos de criptografia. Desta forma, obtiveram-se estimativas da melhoria de desempenho que esta arquitectura ofereceria em relação a um processador RISC comum. Estes resultados foram publicados em [8]. Contudo, não foi analisada a influência nos tempos de execução de efeitos como os diversos *pipelines stalls* causados, por exemplo, pelos acessos à memória externa das *caches* de instruções e dados. Além disso, faltava verificar a correcção funcional da arquitectura ConCISe.

C. Trabalho realizado

Este artigo descreve o trabalho de simulação e estudo da arquitectura ConCISe, durante um estágio de 6 meses do primeiro autor deste artigo no laboratório de investigação Nat.Lab, da Philips Research, em Eindhoven, nos Países-Baixos. O trabalho efectuado constou do seguinte :

- construir um simulador para o ConCISe, utilizando um simulador já existente do processador MIPS[1];
- alterar um *assembler* MIPS já existente para reconhecer a instrução *rfui*, e produzir código executável pelo simulador;
- utilizar o simulador para verificar a correcção funcional da arquitectura ConCISe;
- utilizar algoritmos de criptografia para fazer *benchmarking* da arquitectura;
- analisar os resultados das simulações.

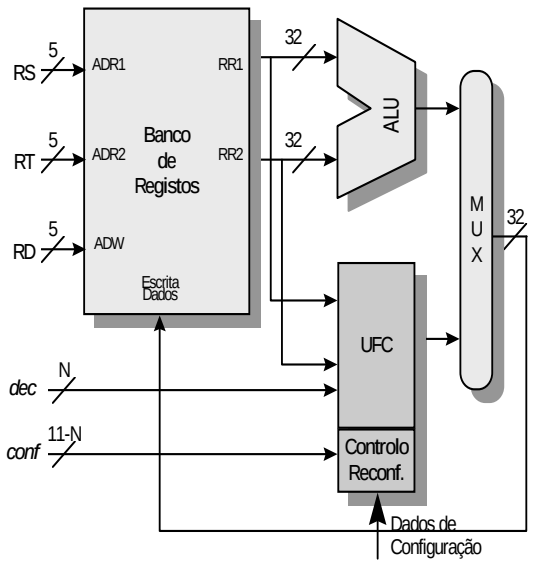

Fig. 1 A arquitectura ConCISe (esquema simplificado).

[1] Como veremos na seccção II, a arquitectura ConCISe baseia-se num *core* MIPS alterado.

D. Organização do texto

A secção I é esta Introdução. Na secção II descrevemos a Arquitectura ConCISe. Em III relatamos a construção do simulador. A secção IV apresenta os resultados das simulações feitas e analisa-os. A secção V contém as conclusões.

II. A ARQUITECTURA CONCISE

A. O Reportório de Instruções

No ConCISe os recursos configuráveis constituem uma Unidade Funcional Configurável (UFC), integrada no *datapath* de um processador RISC[9] de forma similar a outras unidades funcionais (eg a ALU), e em paralelo com estas (ver Fig. 1). O objectivo desta UFC é melhorar o desempenho do processador em processos de computação que requeiram manipulação de *bits*, tarefa para a qual são pouco adequadas as funções disponibilizadas pela ALU (funções aritméticas/lógicas sobre palavras de comprimento fixo).

Para tirar maior proveito dos recursos configuráveis e evitar proceder à reconfiguração destes durante a execução (o que degrada significativamente o desempenho do sistema), utilizou-se a estratégia de codificar múltiplas instruções em cada configuração da UFC.

Ao conjunto de instruções do processador hospedeiro é adicionada uma nova instrução, chamada *rfui*, que acede à UFC. O formato de codificação dessa instrução é:

6	5	5	5	N	11-N
opcode	RS	RT	RD	Dec	conf

Trata-se de uma instrução tipo-R (registo-a-registo) padrão do conjunto de instruções RISC, em que o campo *func* (11 bits) é dividido em dois outros: *conf* e *dec*. *Conf* define a configuração da UFC que corresponde a uma determinada instância da instrução *rfui*, ao passo que *dec* contém os sinais de descodificação responsáveis pela selecção de uma instrução específica dentro da configuração seleccionada da UFC.

A UFC (baseada numa CPLD) tem, além das duas entradas de 32 *bits* para os operandos, uma entrada adicional de N *bits*. Por esta entrada recebe os sinais *dec*. Todas as configurações da UFC devem incluir um descodificador de instruções para interpretar os sinais *dec*. Um registo tem de ser acrescentado para transportar *conf* e *dec* entre as secções da *pipeline*. Uma unidade de controlo de reconfiguração pode ser usada para interpretar o sinal *conf*. Para isso, as configurações necessárias a uma determinada aplicação devem estar numeradas (de *0* a $2^{11-N}-1$), guardando a unidade de controlo o número da

configuração presentemente carregada na UFC. Sempre que uma instrução *rfui* é processada, o valor de *conf* é verificado, e caso não corresponda à configuração carregada, a *pipeline* é parada enquanto a configuração pedida é carregada.

Vamos, a partir daqui, considerar que *N=4*, tendo assim $2^4=16$ instruções codificadas por configuração e $2^{11-4}=128$ configurações possíveis. Além disso, utilizaremos o MIPS R3000 [9] como processador hospedeiro.

B. A UFC baseada num CPLD

Uma arquitectura de DLP do tipo CPLD foi escolhida para implementar a UFC do ConCISe. Isto deve-se ao facto de este tipo de arquitecturas terem um modelo de temporização previsível e serem especialmente eficientes para circuitos com um pequeno número de níveis lógicos.

O CPLD utilizado é uma versão simplificada dos dispositivos da família XPLAII (tecnologia recentemente vendida pela Philips à Xilinx), com memória de configuração implementada à custa de SRAM.

O dispositivo dispõe de 2 conjuntos de 32 *bits* de entrada, correspondentes aos valores dos operandos, mais 4 sinais relativos ao sinal de descodificação de função, *dec*. Estes sinais são encaminhados por um *cross-point switch (LZIA)*, e através de dois barramentos de N *bits* (valor ainda em estudo) para dois blocos lógicos com 16 saídas cada um, totalizando os 32 *bits* de saída, correspondentes ao resultado da operação (Fig. 2). Cada bloco lógico (Fig. 3) é constituído por uma combinação de matrizes PAL e PLA, que permitem, sob a forma de soma de produtos em módulo-2, contruir funções lógicas. Cada pino de saída tem 4 termos-produto (TPs) dedicados. Além disso, até 32 TPs da PLA podem também ser utilizados para cada termo-soma (TS), estando assim disponíveis um máximo de 36 TPs para cada TS. Isto permite a implementação de funções bastante complexas, com apenas uma passagem através do dispositivo.

Assumindo que qualquer função lógica que

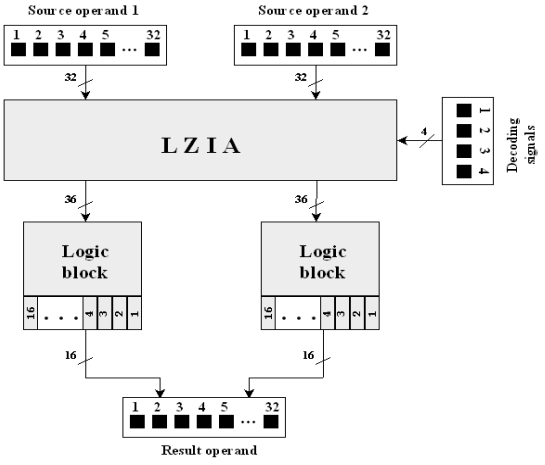

Fig. 2 A arquitectura da UFC baseada numa CPLD.

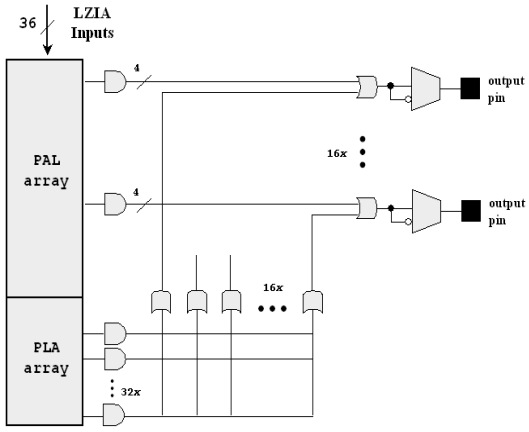

Fig. 3 A estrutura de um bloco lógico das CPLDs XPLA2

implementemos necessita de apenas uma passagem pelo dispositivo, o atraso total de propagação é sempre menor ou igual ao caminho que passa pela matriz PLA. Esta predictabilidade garante-nos que a UFC pode executar as suas funções dentro de um passo da *pipeline* do processador. Com um tempo de propagação total através da PLA estimado em 5,5ns (num processo de 0,35µm), o ConCISe pode atingir velocidades de relógio até ≈180 MHz (desprezando-se o tempo de propagação através dos multiplexadores relacionados com a *pipeline*).

As estimativas feitas indicam que a UFC ocupa cerca de 4mm^2 de silicio, no mesmo processo de 0,35µm, se incluirmos a unidade de controlo de reconfiguração. Contudo, como mostraremos mais adiante, na maior parte das aplicações do ConCISe uma só configuração é necessária, codificando todas as funções que permitem aceleração significativa do processamento. Nesses casos, a unidade de controlo de reconfiguração não é necessária. Se a eliminarmos, o valor estimado da área da UFC decresce para 2.5mm^2, uma superfície comparável à de uma *cache* de 8KB, se utilizarmos a mesma tecnologia de fabrico.

C. O Protótipo da Cadeia de Compilação

O problema da compilação consiste na detecção e selecção de blocos de instruções aritméticas/lógicas que possam ser sintetizadas num circuito lógico passível de ser implementado pela UFC numa só passagem pela matriz e que tenha como operandos dois registos de entrada e um de saída. A blocos de instruções com estas características chamamos **blocos candidatos**.

A cadeia de compilação (Fig. 4) desenvolvida procede da seguinte maneira:

a) a partir do código em C, um **compilador padrão** (*gcc)* produz código *assembly* para o processador hospedeiro utilizado (MIPS R3000);

b) o **módulo de detecção de blocos candidatos** analiza o código *assembly* produzido, e gera uma lista de blocos de código passíveis de síntese em *hardware*;
c) manualmente e tendo disponível informação sobre o tempo de execução do código, obtida através de um **profiler**, são seleccionados os blocos a sintetizar (este passo deverá ser entretanto automatizado);
d) os blocos seleccionados são convertidos pelo **módulo tradutor** em descrições em PHDL (uma linguagem de descrição de *hardware* utilizada pela Philips, similar a ABEL);
e) as descrições em PHDL são convertidas num ficheiro de configuração da UFC utilizando as **ferramentas de projecto comerciais das XPLAII** (*XPLA Designer*);
f) o código *assembly* é alterado, substituindo-se os blocos sintetizados pelas instâncias apropriadas da instrução *rfui*;
g) o código *assembly* é convertido por um ***assembler* MIPS alterado** (*gas - GNU cross-assembler)* para reconhecer a instrução *rfui* em código executável, que juntamente com o ficheiro de configuração da UFC permite correr a aplicação num simulador do ConCISe.

III. SIMULADOR DO CONCISE

A. Um simulador do MIPS

Tomou-se a decisão de utilizar um simulador já existente do MIPS e acrescentar-lhe a funcionalidade relacionada com a UFC. Esta abordagem tem vantagens em comparação com a escrita integral de um novo simulador. Permite reduzir o tempo de desenvolvimento do simulador, visto que grande parte do código é aproveitado, e, além disso, a capacidade do simulador de reproduzir o funcionamento de um processador MIPS já foi testada, pelo que apenas as novas funcionalidades precisam de ser verificadas.

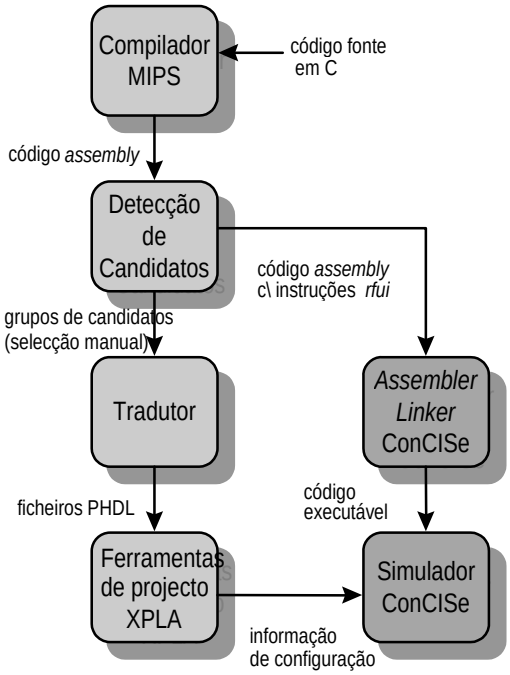

Fig. 4 Cadeia de compilação para o simulador do ConCISe.

Entre os simuladores do MIPS propostos, foi feita a opção de utilizar o TWIPS, desenvolvido na Philips, que, sem ser tão complexo como outros simuladores existentes, apresenta uma simulação correcta da hierarquia de memória, bem como do funcionamento da *pipeline*, o que não acontecia com modelos como, por exemplo, o conhecido SPIM/SAL de James Larus, bastante utilizado para fins pedagógicos, mas que, mesmo na versão *cycle-SPIM*, que procura simular o comportamento da *pipeline* do processador, utiliza várias simplificações na representação da hierarquia de memória, que não se coadunavam com os nossos objectivos.

O TWIPS funciona sob o ambiente TSS (*Tool for System Simulation*) da Philips. O TSS é um ambiente de simulação baseado em ciclos, que permite uma descrição de alto-nível de arquitecturas de *hardware* através de módulos escritos em linguagem C. Estes módulos descrevem a funcionalidade e interface de componentes *hardware*. Ligando entre si instâncias dos vários componentes obtém-se o sistema a ser simulado. A interface com o utilizador durante a simulação é feita através de comandos em TCL (*Tool Command Language*).

O TWIPS é um módulo TSS, capaz de simular um MIPS R3000, incluindo as *caches* de dados e de instruções, o coprocessador 0 e o comportamento da *pipeline*.

B. Simular a UFC

A ideia original para executar a simulação da UFC era utilizar o simulador comercial do *XPLA Designer* e apenas programar uma interface entre este simulador e o TWIPS. Esta abordagem, contudo, mostrou ter graves inconvenientes:

- o simulador comercial é computacionalmente bastante pesado, porque simula os atrasos no dispositivo porta lógica a porta lógica, o que é desnecessário, sabendo-se, *a priori*, o tempo de atraso máximo da UFC. A utilização deste simulador integrado no TWIPS implicaria simulações extremamente lentas;
- o simulador comercial não está orientado para processamento ciclo-a-ciclo, aceitando antes uma sequência temporal de transições dos sinais de entrada como argumento, e processando em seguida toda a série temporal de uma só vez;

- o simulador comercial é constituído por 3 programas independentes, os quais comunicam entre si através de ficheiros; este processo é lento, embora fosse possível tentar fundir os códigos dos 3 executáveis num só programa;
- fazer alterações ao código do simulador comercial é complexo: os comentários no código são escassos e a documentação inexistente.

Tendo isto em conta, foi desenvolvido um novo módulo de simulação funcional da UFC. Este módulo foi escrito em C, devido à necessidade de compatibilidade com o TSS. A estrutura do código foi orientada a objectos (ver diagrama de classes na Fig. 5). Não sendo o C uma linguagem orientada a objectos, foi necessário "traduzir" cada uma destas classes numa estrutura de dados e funções operando sobre essa estrutura. O simulador comercial foi utilizado para comprovar os resultados obtidos com o módulo de simulação funcional.

As classes utilizadas pelo simulador funcional são:

Parser- esta classe representa um leitor de ficheiros de configuração. Lê ficheiros do tipo *ph1*, os quais são ficheiros em PHDL, mostrando apenas as equações lógicas optimizadas para cada pino de saída da configuração da CPLD. Estas equações correspondem à função lógica dos pinos de saída na forma de somas de termos-produto (fig. 6). Os ficheiros *ph1* são produzidos automaticamente pelas ferramentas de projecto XPLAII. O analizador léxico do *Parser*, foi implementado com o auxílio do *lex*[12], e o analizador sintático estruturado de forma similar a uma máquina de estados finita. No final da leitura de um ficheiro, uma estrutura do tipo *Configuration* é disponibilizada.

RFU- esta estrutura representa a UFC. É constituída por uma lista de estruturas *Configuration* que são carregadas através do *Parser*.

Configuration- esta classe é a representação interna do simulador para uma configuração da UFC. É constituída por uma lista de 32 objectos *Sterm*.

Sterm- esta estrutura representa uma soma de termos produto, isto é, uma das equações atribuídas a uma porta OR à saída da UFC. É constituída por uma lista de objectos *Pterm*, e dispõe de um método que calcula o valor à saída, dados os valores actuais de

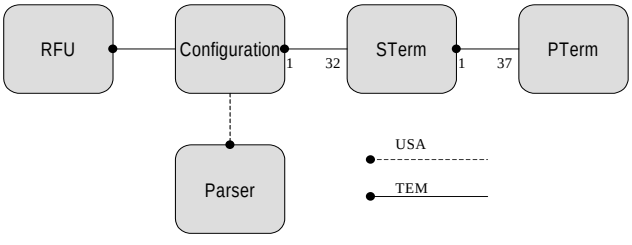

Fig. 5 Diagrama de Classes do simulador da UFC.

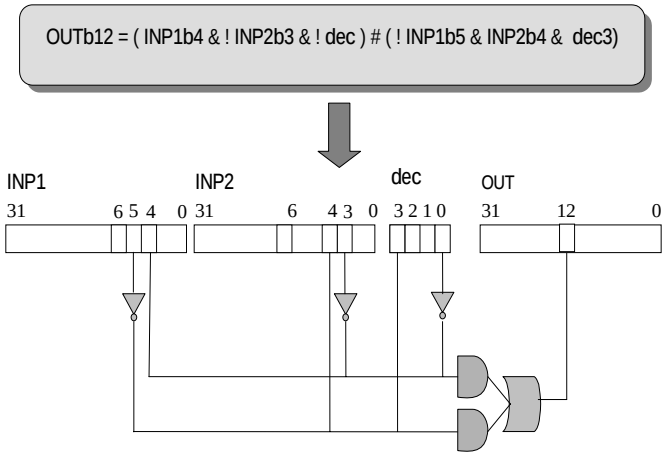

Fig. 6 Exemplo da leitura de uma equação optimizada de um ficheiro *ph1* em esquema de circuito digital. O circuito apresentado é directamente implementável num bloco lógico da UFC.

todas as entradas da UFC (2 entradas de 32 *bits* e os 4 *bits* do sinal *dec*).

Pterm- representa um TP das entradas. Corresponde a uma porta AND das matrizes PLA/PAL do CPLD. Contem uma lista dos pinos de entradas que lhe estão ligados, de forma a poder calcular o valor da saída.

O simulador funcional da UFC construído é capaz de:

- ler ficheiros (produzidos pelas ferramentas de projecto da XPLAII) contendo as configurações desejadas da UFC;
- dados os valores dos dois registos operandos e do sinal *dec*, calcular o resultado da UFC;
- mudar a configuração da UFC.

Este simulador não calcula atrasos do dispositivo. Não necessita, tendo em conta que a arquitectura da UFC impõe o já referido modelo temporal determinístico.

C. O simulador TWIPS/ConCISe

O simulador TWIPS foi alterado de forma a reconhecer a instrução *rfui* e utilizar o módulo de simulação funcional para executar a nova instrução, bem como para carregar mapas de configuração no início do processamento.

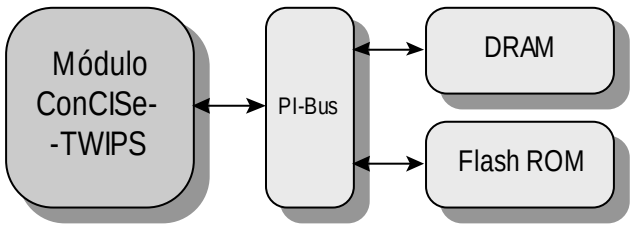

Fig. 7 Organização do sistema simulado

O sistema que finalmente foi instanciado para simulação (ver Fig. 7) é composto por um processador TWIPS/ConCISe, uma memória FLASH-ROM, uma memória DRAM e um barramento PI-Bus (estes 3 últimos componentes instanciados a partir de módulos da biblioteca padrão do TSS).

D. Um assembler para o ConCISe

Um *assembler* para o protótipo ConCISe foi elaborado a partir do *gas* (GNU *cross-assembler*, versão incluída no *binutils 2.8.1*) para código MIPS. Bastou acrescentar à *hash-table* que o *gas* utiliza para descrever o conjunto de instruções do MIPS uma nova linha, contendo a informação de codificação relativa à instrução *rfui*.

IV. ANÁLISE DOS RESULTADOS DAS SIMULAÇÕES

A. O algoritmo DES

Utilizámos a implementação da função *fcrypt* de Eric Young. De um total de 211 candidatos (incluindo sub-candidatos), obtidos pelo módulo detector de candidatos da cadeia de compilação apresentada na Fig. 4, dez candidatos foram manualmente seleccionados, tendo em conta informação obtida da análise em tempo de execução feita pelo *profiler*, e traduzidos numa só descrição de *hardware* em PHDL, juntamente com a lógica de descodificação de instruções.

Começamos o nosso estudo ajustando as *caches* do simulador de acordo com a configuração de um Philips PR3001, um processador MIPS de gama média. Assim, temos 1KB de *Cache* de Dados (D$) e 4KB de *Cache* de Instruções (I$). Para esta configuração, um *speed-up* de aproximadamente 35% é obtido na Contagem de Ciclos (CC) da arquitectura MIPS+UFC em relação a um MIPS normal.

Este valor fica áquem do *speed-up* obtido para a Contagem de Instruções (CI). O algoritmo DES utiliza *Look-Up Tables* (LUTs) e o tempo de processamento da aplicação é dominado pelos recarregamentos de D$ (cerca de 30%) devidos aos acessos às LUTs, e a vantagem de utilizar a UFC é diminuída.

Fazemos então a experiência de aumentar progressivamente o tamanho de D$ (ver Fig. 8). À medida que se reduz o número de ciclos de processamento desperdiçados em recarregamentos de D$ (Fig. 9), medem-se *speed-ups* em CC maiores, que estabilizam em ≈57%, quando a D$ atinge a capacidade de 8KB. Este valor ainda é inferior ao *speed-up* de 61% obtido para a CI, devido ao efeito do carregamento inicial de I$ e D$.

Fazemos em seguida uma análise comparativa entre o investimento na UFC e no aumento da capacidade de D$.

Na Fig. 10 podemos ver que um aumento de 2KB para 4KB de D$ (sem UFC) aumenta o desempenho em cerca de 8%. Se adicionarmos a UFC em vez dos 2KB extra de *cache*, uma diminuição em cerca de 25% da CC é observada. Contudo, isto implica uma troca: a UFC toma *2,5mm*^{2} num processo de *0,35*μ*m*, ao passo que 2KB de *cache* apenas gastam *0,85mm*^{2}, pelo mesmo processo de fabrico.

Mas, além disso, verifica-se que o *speed-up* permitido pelo aumento da capacidade de D$ estabiliza em 4KB. O *speed-up* possibilitado pela UFC pode ser visto como uma

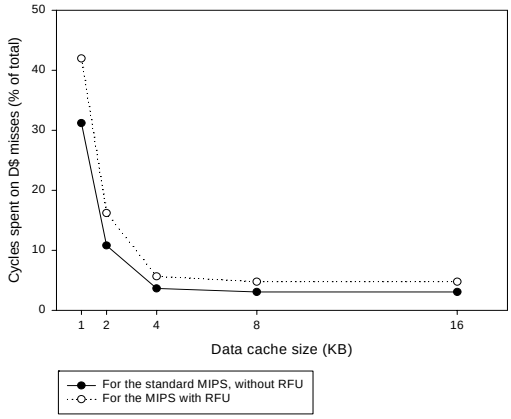

Fig. 9 Percentagem de ciclos gastos em recarregamentos de D$ para o algoritmo DES.

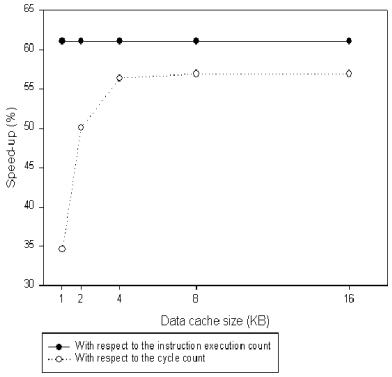

Fig. 8 Capacidade de D$ vs *speed-up* para o algoritmo DES.

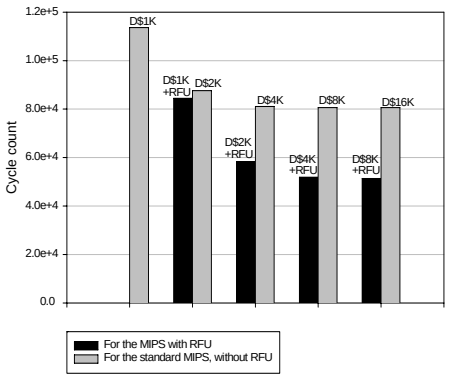

Fig. 10 Investimento em capacidade de D$/UFC para o algoritmo DES.

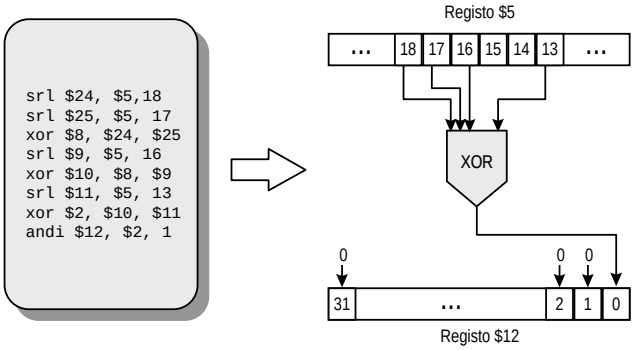

Fig. 11 Transformação de uma sequência de instruções nativas MIPS numa instrução *rfui*, para o algoritmo A5.

forma de aumentar o desempenho do processador, sem utilizar velocidades mais altas de relógio (e o consumo de energia acrescido inerente), depois de técnicas tradicionais como o aumento da capacidade das *caches* se tornarem inúteis[2].

B. O algoritmo A5

Utilizámos a implementação de Bruce Schneier [13]. O A5 é um algoritmo *stream cypher*. Como são normalmente baseados em *shift registers*, estes algoritmos envolvem muita manipulação de *bits*, bastante inapropriada para implementação em *software*. É previsível que a UFC possa ajudar a minorar significativamente este problema.

Dos blocos candidatos encontrados, nove foram manualmente seleccionados e automaticamente traduzidos para uma só configuração da UFC.

Na Fig. 11 mostramos o exemplo de um bloco candidato e da sua tradução em *hardware*. Como se pode ver, uma operação cujo mapeamento no conjunto fixo de instruções

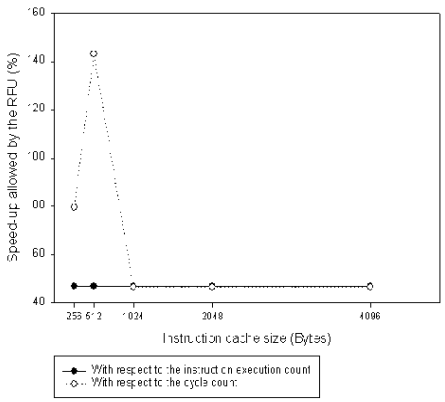

Fig. 12 Capacidade de I$ vs *speed-up* para o algoritmo A5.

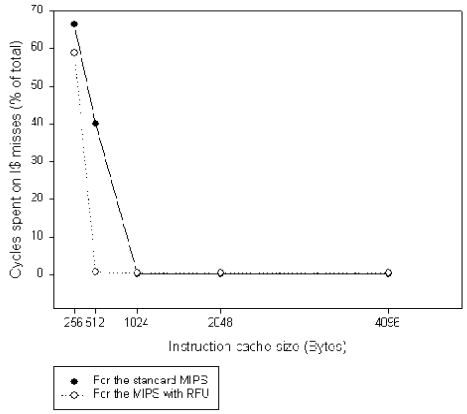

Fig. 13 Percentagem da CC gasta em recarregamentos de I$ para o algoritmo A5.

de um processador MIPS implica várias instruções (neste caso oito) pode-se resumir a uma porta XOR-4 e algumas conexões.

Para o algoritmo A5 completo, o *speed-up* em CI foi de aproximadamente 47%. Pode-se também observar nas Figs. 12 e 13 um efeito secundário interessante: ao reduzir blocos de instruções a apenas uma instrução, a UFC contribui para a redução dos recarregamentos de I$[3].

Assim, para uma capacidade de 512B da I$[4], a versão com UFC do código do A5 cabe completamente dentro de I$, o que não acontece com a versão MIPS. Nesse ponto, o *speed-up* permitido pela UFC atinge mais de 140%. Para uma I$ de 1KB, contudo, ambos os códigos cabem na *cache*.

C. O algoritmo LOKI97

O LOKI97 é um algoritmo de *block cypher* que cifra blocos de dados de 128 *bits*. Durante a cifragem e a decifragem, o LOKI97 faz uso extensivo de uma função altamente não-linear, a qual é implementada à custa de LUTs e de vários *and*s, *or*s, *xor*s, *shift*s e adições. A informação dada pelo *profiler* indicava que o cálculo desta função é a secção que maior parte do IC consome nas operações de cifragem/decifragem. Foi nesta função que encontrámos os 16 blocos candidatos que viriam a constituir a configuração da UFC.

Para os valores de *cache* de referência do PR3001 (I$: 4KB, D$:1KB), um *speed-up* em CC de 9,5% foi obtido, como pode ser visto na Fig 14.

Este resultado é modesto e explica-se pela quantidade de recarregamentos da D$ (ver Fig. 15). De facto, estes recarregamentos ocupam cerca de 40% da CC para a arquitectura MIPS+UFC com 1KB de *cache* (fig. 15). O

[2]Eventualmente, a partir de uma determinada capacidade de *cache*, o segmento crítico do código/dados de qualquer aplicação pode ser completamente armazenado em I$/D$, e *cache* extra é inútil.

[3] Este efeito de "compactação do código" não deve, contudo, ser visto como uma forma de reduzir custos em termos de memória, por causa da memória necessária para armazenar a configuração da UFC.

[4] Apesar da utilização de capacidades de *cache* irrealisticamente pequenas, a experiência é válida para verificar o efeito de "compactação de código" .

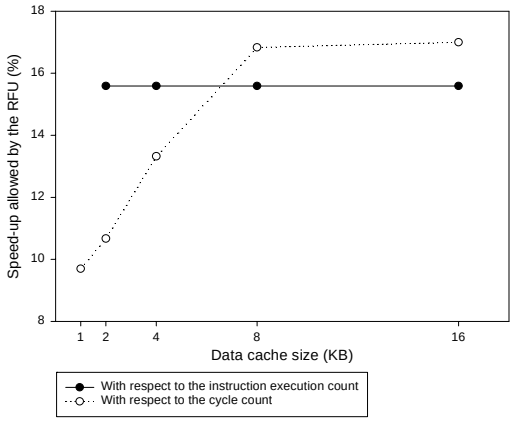

Fig. 14 Capacidade de D$ vs *speed-up* para o algoritmo LOKI97.

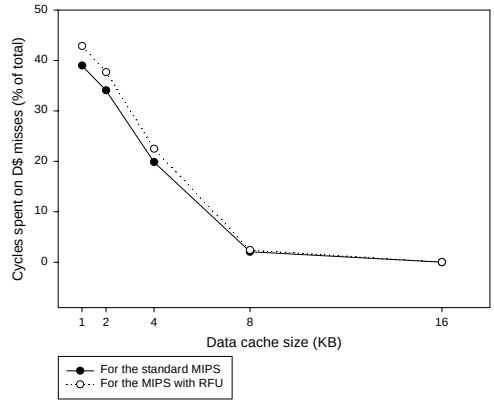

Fig. 15 Percentagem de ciclos gastos em recarregamentos de D$ para o LOKI97.

speed-up conseguido pela UFC é escondido por este valor.

Aumentando a capacidade de D$, podemos ver que o *speed-up* em CC do ConCISe aumenta até 17%. Verificamos que, para D$ de capacidade igual ou superior a 8KB, o *speed-up* em CC é superior em 1,5 pontos percentuais ao *speed-up* em CI. Como os recarregamentos da D$ deixam de dominar a CC (ver Fig. 15), a diminuição dos recarregamentos de I$ causada pelo "efeito de compactação de código" torna-se vísivel (Fig. 14).

Além disso, a diminuição dos recarregamentos de D$, causa uma redução de acessos à memória externa e consequentemente do tempo médio de acesso a esta, permitindo recarregamentos ligeiramente mais rápidos de I$.

A comparação do investimento em capacidade de D$ ou na UFC é representada na Fig. 16. A melhoria de desempenho proporcionada pela UFC (17%) a um sistema com 8KB de D$ ultrapassa a melhoria de desempenho

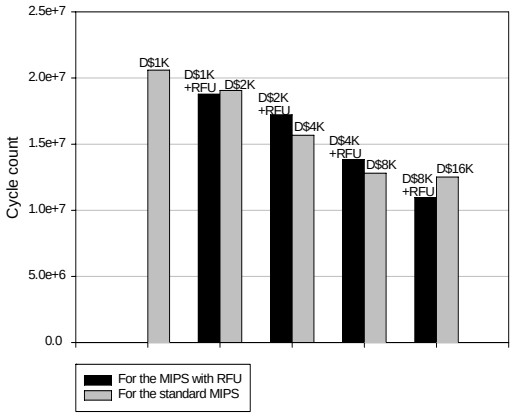

Fig. 16 Investimento em capacidade de D$ /UFC para o algoritmo LOKI97

obtida aumentando a capacidade de D$ de 8KB para 16KB. Este resultado é curioso, visto que o silicio extra (e portanto o custo acrescido) de adicionar 8KB de *cache* é aproximadamente o mesmo de acrescentar a UFC.

Após observar os efeitos da variação da capacidade de D$ para o LOKI97, quizemos saber mais sobre as possíveis contrapartidas de investimento em capacidade de I$/UFC, tendo em conta a influência do efeito de "compactação de código". Mantendo uma capacidade de D$ de 16KB (suficientemente alta para garantir a minização da influência dos recarregamentos desta sobre os resultados) fazemos variar a capacidade de I$ de 1KB a 16KB (Fig 17).

Como pode ser visto na Fig 17, para uma I$ de 1KB, o *speed-up* em CC (acima de 50%) é bastante superior ao *speed-up* em CI (cerca de 16%). O código executável para MIPS+UFC, sendo mais curto, requer menos recarregamentos de I$, melhorando o desempenho, de acordo com o esperado. Na Fig 18 pode-se ver que para uma capacidade de 1KB de I$, a configuração MIPS+UFC gasta 19% da CC total em recarregamentos de I$, contra 50% da CC total para a arquitectura MIPS normal. Para I$ de 2KB, a arquitectura MIPS gasta muito menos ciclos em recarregamentos, o que leva a uma queda abrupta do *speed-up* em CC, obtido com o uso da UFC. Contudo, podemos ver que, para I$ de 4KB, o *speed-up* em CC está de novo acima do *speed-up* em CI. Este resultado é consistente com a maior diminuição da percentagem de recarregamentos de I$ para a arquitectura MIPS+UFC, mostrada na Fig 18. A partir deste ponto, a I$ é suficiente para albergar todo o segmento crítico do código, e o *speed-up* em CC é aproximadamente o mesmo que o *speed-up* em CI.

Numa análise de investimento em capacidade de I$/UFC, (Fig. 19) verificamos que 4KB de I$ são preferíveis em termos de desempenho a 2KB+UFC. Contudo, a partir de 4KB de I$, mais *cache* é inútil, e a UFC continua a ser uma mais-valia.

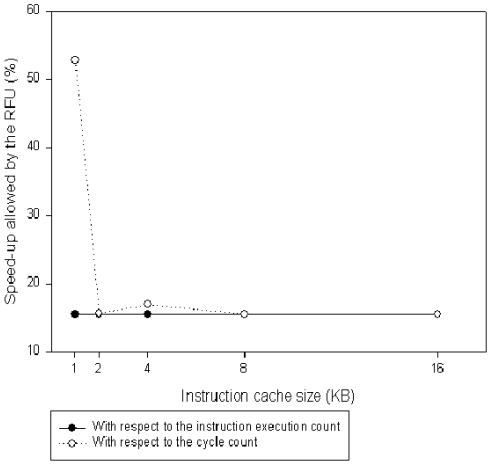

Fig. 17 Capacidade de I$ vs *speed-up* para o algoritmo LOKI97.

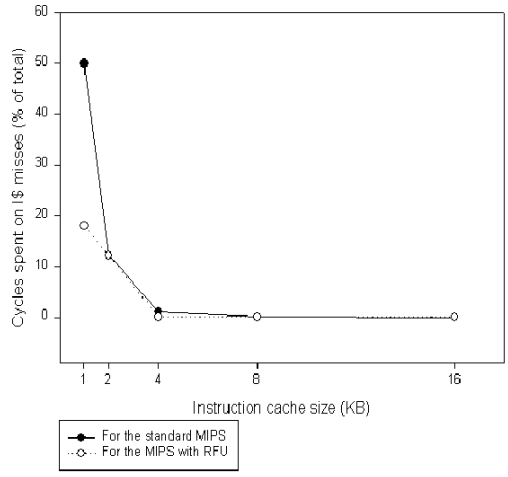

Fig. 18 Percentagem de CC gasta em recarregamentos de I$. para o algoritmo LOKI97.

D. O algoritmo MD5

O MD5 é um algoritmo de *compressão de mensagens*. Os primeiros resultados de simulação para este algoritmo, tomados para os valores de *cache* de referência, mostram um *speed-up* em CC de 26%, bastante acima dos 9% obtidos para o *speed-up* em CI, e uma percentagem negligenciável de recarregamentos de D$ (abaixo de 1% do tempo de execução). Obviamente, este elevado valor de *speed-up* em CC é explicado pelo efeito de "compactação de código".

Variando a capacidade de I$ (Figs. 20, 21), obtemos resultados cuja análise é semelhante à que fizemos para o algoritmo A5. Neste caso, contudo, o pico do efeito surge para um valor mais realista de *cache* (4KB).

Fazendo um estudo de investimento I$/UFC (Fig 22), verifica-se que a UFC é melhor investimento que *cache* extra a partir do momento em que se tem um mínimo de 4KB de capacidade em I$

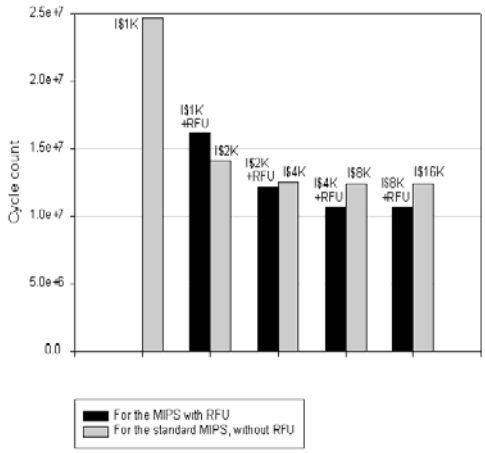

Fig. 19 Investimento em capacidade de I$/UFC para o algoritmo LOKI97.

E. O algoritmo Magenta

O Magenta é um algoritmo de criptografia desenvolvido pela Deutsche Telekom.

Apenas foram encontrados seis blocos candidatos dentro do segmento crítico do código, mas cada um deles tinha múltiplas ocurrências. Além disso, dois desses blocos candidatos eram compostos por mais de 10 instruções.

Os resultados obtidos para as *caches* de referência indicam um *speed-up* em CC de 24%, bastante abaixo do *speed-up* em CI de quase 45% (Fig. 23). Esta diferença é explicada pela quantidade de recarregamentos de D$ (cerca de 40% da CC, como pode ser visto na Fig. 24). Esperamos que aumentando a capacidade de D$ o *speed-up* em CC tenda a convergir para o valor do *speed-up* em IC.

Como podemos ver na Fig. 23, mesmo quando D$ tem capacidade de 16KB, o *speed-up* em CC fica 5 pontos percentuais abaixo do *speed-up* em CI. Na Fig. 24 encontramos uma explicação para isto: os recarregamentos de D$ ainda representam cerca de 7% da CC (o algoritmo está a correr para apenas uma ronda de cifragem/decifragem, e o carregamento inicial das *caches* tem um efeito não desprezável).

No gráfico de investimento D$/UFC (Fig 25), vemos que duplicar uma D$ de 4KB apenas melhora o desempenho em 4%, ao passo que acrescentando a UFC aos 4KB se obtém uma melhoria de 38%. Não esquecer, todavia, que a UFC ocupa o dobro da superfície em silício que 4KB extra de capacidade de D$. Um aumento da capacidade de D$ de 8KB para 16KB também não bate uma D$ de 8KB com UFC: o máximo da melhoria de desempenho obtida com o aumento de D$ já foi alcançado com 8KB, e o investimento em cache extra é inútil. De notar também que a configuração 2KB+UFC tem melhor desempenho que 16KB de capacidade de D$.

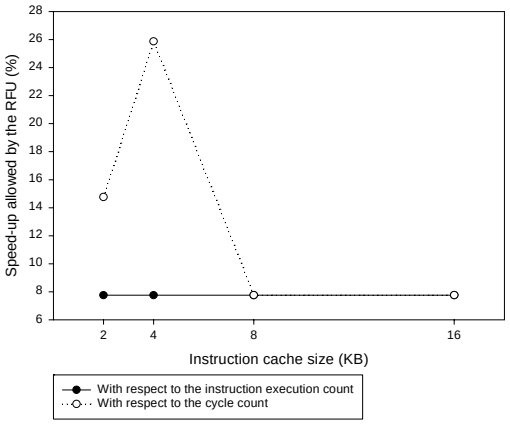

Fig. 20 Capacidade de I$ vs *speed-up* para MD5

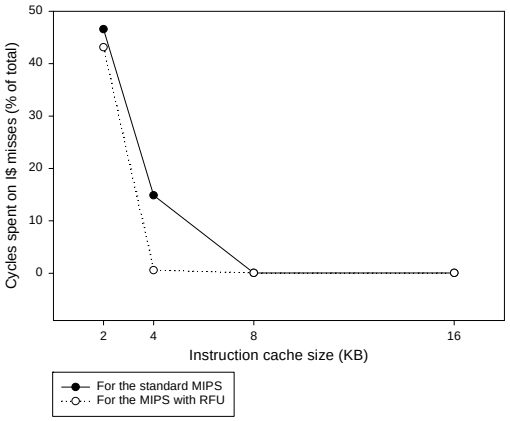

Fig. 21 Capacidade de I$ vs percentagem de recarregamentos de I$ para o algoritmo MD5

F. Sumário

A Fig 26, apresenta o *speed-up* em CI[5] para cada uma das *benchmarks*. O algoritmo DES teve a maior melhoria de desempenho com 62% de *speed-up*. O pior resultado coube ao MD5 (8% de *speed-up*).

V. CONCLUSÃO

Demostrámos a funcionalidade do ConCISe para algumas aplicações típicas do campo da criptografia. Apresentaram-se diversos resultados de simulação, sob uma perspectiva comparativa em relação a um sistema MIPS comum. Os resultados observados permitiram-nos concluir que o comportamento das *caches* tem uma influência considerável sobre a melhoria de desempenho possibilitada pela UFC do ConCISe: se a cache de dados tiver capacidade suficiente para conter todos os dados

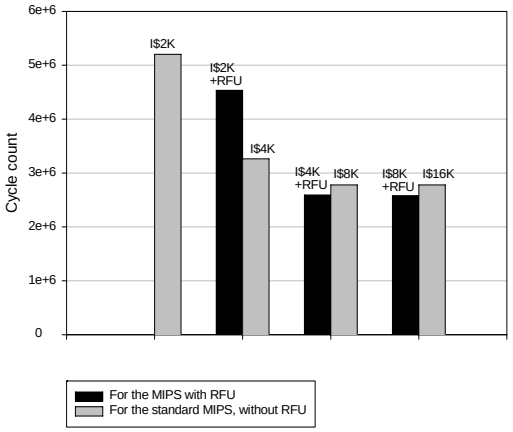

Fig. 22 Investimento capacidade de I$/UFC para o algoritmo MD5.

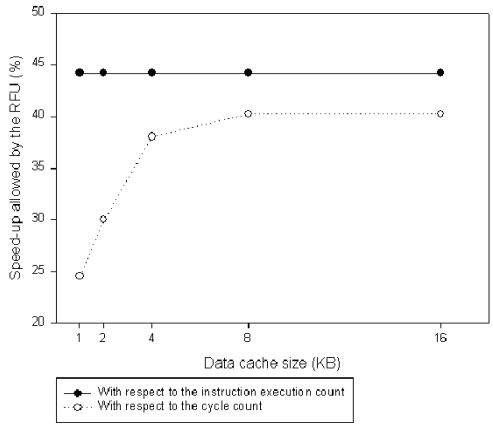

Fig. 23 Capacidade de D$ vs *speed-up* para o algoritmo Magenta.

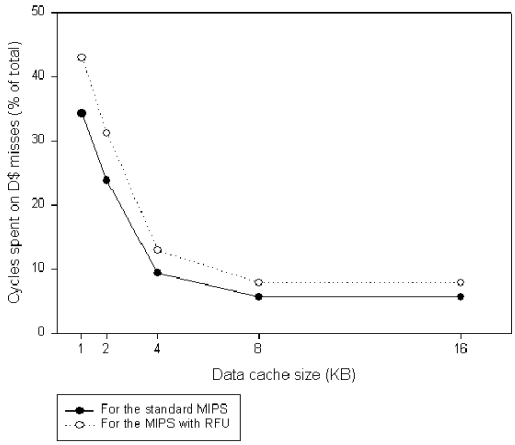

Fig. 24 Percentagem de CC gasta em recarregamentos de D$ para o algoritmo Magenta.

necessários ao segmento de código crítico de uma dada aplicação, então o *speed-up* obtido pela utilização do

[5] Note-se que, como mostram as simulações, o *speed-up* em CC tende a convergir para o *speed-up* em IC à medida que o tamanho das *caches* aumenta.

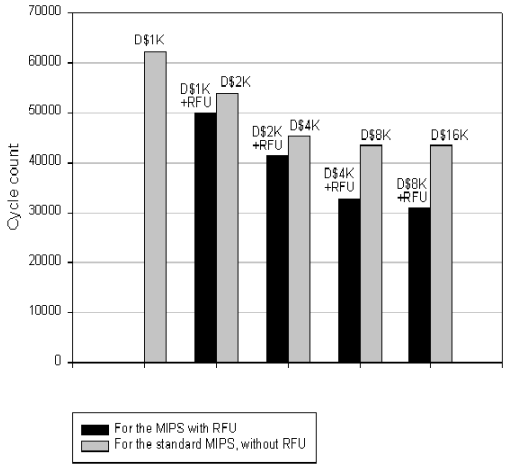

Fig. 25 Investimento em D$/UFC para o algoritmo Magenta .

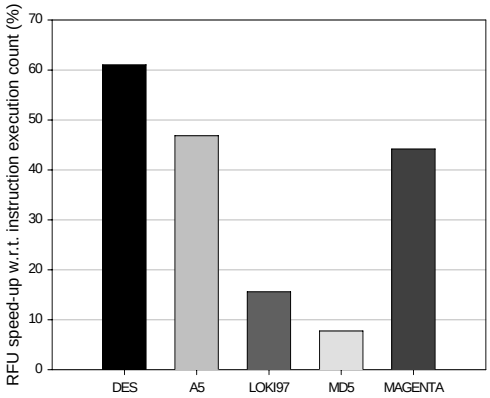

Fig. 26 *Speed-up*s em IC para todos os algoritmos.

ConCISe é mais significativo, mesmo que, com capacidades de D$ menores, o investimento na UFC possa ainda ser vantajoso em relação ao aumento da capacidade de *D$*.

Por outro lado, a compactação do código executável causada pela síntese de várias instruções MIPS numa só instrução *rfui* do ConCISe provoca como efeito secundário a melhoria de desempenho da cache de instruções (I$).

REFERÊNCIAS

[1] P. M. Athanas e H.F. Silverman, “Processor Reconfiguration Through Instruction Set Metamorphosis”, *Computer*, 26, 3, pp 11-18, Março, 1993

[2] R. Razdan e M. D. Smith, “A High Performance Microarchitecture with Hardware-Programmable Functional Units”, *Proceedings of the 27th Annual IEEE/ACM Intl. Symposium On Microarchitecture*, pp 172-180, Novembro, 1998

[3] M. J. Wirthlin e B. L. Hutchings, ”A Dynamic Instruction Set Computer”, *IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, Abril, 1995

[4] R. D. Wittig e P. Chow, “OneChip: An FPGA Processor with Reconfigurable Logic”, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp 126-135, Los Alamitos, CA, Abril, 1996

[5] J.R. Hauser e J. Wawrzynek, ”Garp: A MIPS Processor with a Reconfigurable Coprocessor”, *Proccedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp 24-33, 1997

[6] S. Hauck *et al*, “The Chimaera Reconfigurable Functional Unit”, *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997

[7] B. Kastrup *et al*, ”ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator”, *Proc. of the IEEE on Field-Programmable Custom Computing Machines*, Napa, EUA, 1997

[8] J. Hennessy, D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 1996

[9] J. Kane e J. Heinrich, *MIPS RISC Architecture*, Prentice-Hall PTR. 1992

[10] XPLA Designer version 2.1 Users Manual, http://www.coolpld.com

[11] T. Niemann, *A Guide to Lex and Yacc*, http://members.xoom.com/thomasn/y_man.htm

[12] B. Schneier, *Applied Cryptography 2nd Edition: Protocols, Algorithms and Source Code in C*, John Wiley and Sons Inc., 1996

[13] L. Brown, J. Pieprzyk, *Introducing the new LOKI97 Block Cypher*, http://www.adfa.oz.au/lpb/research/loki97/

[14] http://www.gl.umbc.edu/mabzug/cs/md5/md5.html