Estudo e implementação de algoritmos de resumo (*hash*) criptográfico na plataforma Intel® XScale®

Paulo Henrique Nóbrega Tavares

Trabalho Final de Mestrado Profissional em Computação

Instituto de Computação
Universidade Estadual de Campinas

Estudo e implementação de algoritmos de resumo (*hash*) criptográfico na plataforma Intel® XScale®

Paulo Henrique Nóbrega Tavares

21 de fevereiro de 2006

Banca Examinadora:

- Prof. Dr. Julio César López Hernández
  Instituto de Computação - Unicamp
- Prof. Dr. Marco Aurélio Amaral Henriques
  Departamento de Engenharia de Computação e Automação Industrial - Unicamp
- Prof. Dr. Paulo Sérgio Licciardi Messeder Barreto
  Departamento de Engenharia de Computação e Sistemas Digitais - USP
- Prof. Dr. Ricardo Dahab (Orientador)
  Instituto de Computação - Unicamp
- Prof. Dr. Rodolfo Jardim de Azevedo (Suplente)
  Instituto de Computação - Unicamp

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Tavares, Paulo Henrique Nóbrega
T197e Estudo e implementação de algoritmos de resumo (hash) criptográfico na plataforma Intel XScale / Paulo Henrique Nóbrega Tavares -- Campinas, [S.P. :s.n.], 2006.

Orientador : Ricardo Dahab
Trabalho final (mestrado profissional) - Universidade Estadual de Campinas, Instituto de Computação.

1. Criptografia. 2. Hashing (Computação). 3. Arquitetura de computadores. I. Dahab, Ricardo. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Study and implementation of cryptographic hash algorithms on the Intel XScale platform.

Palavras-chave em inglês (Keywords): 1. Cryptography. 2. Hashing (Computer science). 3. Computer architecture.

Área de concentração: Engenharia de Computação

Titulação: Mestre em Computação

Banca examinadora: Prof. Dr. Ricardo Dahab (IC-UNICAMP)
Prof. Dr. Julio César López Hernández (IC-UNICAMP)
Prof. Dr. Paulo Sérgio Licciardi Messeder Barreto (IME-USP)
Prof. Dr. Marco Aurélio Amaral Henriques (FEEC-UNICAMP)

Data da defesa: 21/02/2006

Programa de Pós-Graduação: Mestrado Profissional em Computação

Estudo e implementação de algoritmos de resumo (*hash*) criptográfico na plataforma Intel® XScale®

Este exemplar corresponde à redação final do Trabalho Final devidamente corrigido e defendido por Paulo Henrique Nóbrega Tavares e aprovado pela Banca Examinadora.

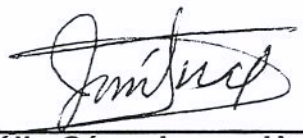
Campinas, 21 de fevereiro de 2006.

Prof. Dr. Ricardo Dahab (Orientador)

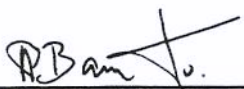
Trabalho Final apresentado ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Computação na área de Engenharia de Computação .

TERMO DE APROVAÇÃO

Trabalho Final Escrito defendido e aprovado em 21 de fevereiro de 2006, pela Banca Examinadora composta pelos Professores Doutores:



Prof. Dr. Júlio César Lopez Hernandes
IC - UNICAMP



Prof. Dr. Paulo Sérgio Licciardi Messeder Barreto
USP

Prof. Dr. Marco Aurélio Amaral Henriques
FEEC - UNICAMP

Prof. Dr. Ricardo Dahab
IC - UNICAMP

© Paulo Henrique Nóbrega Tavares, 2006.
Todos os direitos reservados.

Agradecimentos

Agradeço à minha esposa, Katya, e ao meu filho, Ricardo, pela generosidade e compreensão com que suportaram a minha ausência.
Aos meus pais, Arnaldo e Otília (*in memoriam*), pela formação que tornou um dia este trabalho possível.

Resumo

Nos últimos anos, a quantidade de dispositivos móveis tem crescido dramaticamente assim como a complexidade das aplicações que os usuários desejam executar neles. Ao mesmo tempo, a preocupação com a segurança do grande público também tem aumentado, criando uma demanda maior por aplicações criptográficas, como assinaturas digitais, que dependem de funções de resumo (hash) criptográfico. Neste contexto tornou-se importante estudar o desempenho de funções de resumo nesta nova geração de processadores, desenvolvidos para estes dispositivos. Neste trabalho estudamos a família de funções de resumo SHA (Secure Hash Algorithm) e *Whirlpool*, algumas de suas implementações, as características dos processadores Intel XScale que podem ser usadas para melhorar o desempenho de tais funções, com atenção especial para as novas extensões Wireless MMX. Também aplicamos algumas dessas técnicas e apresentamos os resultados dos testes de desempenho executados.

Abstract

In recent years, the number of mobile devices has grown dramatically and so has the complexity of applications their users wish to run. At the same time, security concerns of the general public have also increased, creating a greater demand for cryptographic applications such as digital signatures, which use hash functions. In this context it has become very important to study the performance of hash functions on the new generation of processors developed for these devices. In this work we study the SHA (Secure Hash Algorithm) family of hash functions and some of their implementations, the Intel XScale processors characteristics that can be used to improve the performance of those functions, with special attention to the new Wireless MMX extensions. We also applied some of these techniques and report the results of the performance tests executed.

Índice

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Neste capítulo apresentaremos a motivação para o desenvolvimento deste trabalho, os objetivos alcançados e como ele está organizado.

1.1 Motivação

Nos últimos anos, a quantidade de dispositivos móveis (como telefones celulares, agendas eletrônicas, tocadores de vídeo e música, etc.) tem crescido dramaticamente. Ao mesmo tempo, a sofisticação das aplicações que os usuários desejam executar nestes aparelhos também tem aumentado, criando preocupações ainda maiores com a segurança, seja do conteúdo, seja dos dados pessoais armazenados e trocados, e criando uma demanda maior por aplicações criptográficas (como assinaturas digitais) que dependem de funções de resumo (*hash*) criptográfico. Além disso, ataques desenvolvidos recentemente contra funções de resumo tradicionalmente utilizadas têm incentivado a utilização de funções com tamanho maior de resumo, que tendem a ter desempenho inferior.

Neste cenário, tornou-se importante estudar o desempenho destas funções nos processadores que equipam estes novos equipamentos e as oportunidades de otimização existentes. Como foco de estudo foi escolhida a família de algoritmos SHA e o processador Intel® XScale®.

Entre os algoritmos de resumo existentes, destacam-se pelo seu uso o MD5 e a família SHA. Devido aos ataques que têm sofrido nos últimos anos, o MD5 está em desuso e o SHA-1 está entrando em desuso e o próprio NIST reconhece que a migração para um algoritmo mais forte é necessária, embora não a considere urgente ([34, p. 4],[11],[45]). Uma alternativa que tem ganhado aceitação neste cenário é a função *Whirlpool* ([21, p.69]). Escolhemos então a família SHA e a função *Whirlpool* para a nossa análise.

1.2 Objetivos

Os objetivos estão dispostos da seguinte forma:

- Estudo geral das funções de resumo criptográfico, com foco na família SHA (SHA-1, SHA-224, SHA-256, SHA-384 e SHA-512) e *Whirlpool*.
- Implementação das funções da família SHA em ANSI C, tendo em vista a execução na plataforma Intel® XScale®.
- Identificação das técnicas que podem ser utilizadas para a otimização de uma implementação de funções de resumo na plataforma Intel® XScale®, especialmente as extensões Wireless MMX™, tais como o processamento de duas iterações do laço em paralelo.
- Aplicação de algumas destas técnicas e avaliação dos resultados obtidos.

1.3 Organização

Neste capítulo foram apresentados a motivação e os objetivos deste trabalho. O capítulo 2 apresenta as funções de resumo que são objeto do trabalho, discute a implementação destas funções e as principais análises de desempenho disponíveis na literatura. No capítulo 3 serão descritas as principais características dos processadores Intel® XScale®, algumas técnicas de otimização aplicáveis a estes processadores e a sua utilização em funções de resumo. A seguir (no capítulo 4) são mostrados os resultados obtidos nos testes realizados e as conclusões a que chegamos (capítulo 5).

Capítulo 2

Funções de resumo criptográfico

Neste capítulo falaremos sobre o que são as funções de resumo (*hash*) criptográfico em geral, quais são as suas principais propriedades e descreveremos as funções estudadas neste trabalho: a família de algoritmos SHA e a função *Whirlpool*. Também serão mostradas algumas das implementações encontradas para essas funções assim como um breve histórico da análise de desempenho disponível na literatura.

Em [40, cap. 9], Menezes et al. descrevem detalhadamente as funções de resumo e os conceitos e a notação utilizados nesta seção são baseados nessa referência.

2.1 Conceitos básicos

Funções de resumo (*hash*) recebem uma entrada e produzem uma saída, normalmente chamada de *hash-code*, resultado (ou valor) do resumo ou simplesmente resumo. Uma função de resumo h mapeia cadeias de bits de comprimento arbitrário (embora finito) em cadeias de bits de tamanho fixo (n). Para um dado domínio D e um contra-domínio R, com $h : D \rightarrow R$ e $\mid D \mid > \mid R \mid$, a função é de muitos para um, o que implica necessariamente na existência de colisões, isto é, valores de $x \neq x'$ tais que $f(x) = f(x')$.

As funções de resumo são utilizadas para muitas aplicações não relacionadas à criptografia (métodos de busca, por exemplo) e, neste escopo, trataremos apenas das funções de resumo (*hash* resumo) criptográfico (cujas propriedades serão vistas adiante) e serão chamadas simplesmente de funções de resumo.

Se considerarmos que o domínio de h consiste de entradas de t bits $(t > n)$ e que a função h é aleatória (no sentido de que todas as saídas são igualmente prováveis) então cerca de 2^{t-n} entradas seriam mapeadas para uma mesma saída e duas entradas escolhidas aleatoriamente gerariam a mesma saída com probabilidade 2^{-n} (independente de t).

A idéia é que o valor do resumo serve como uma representação compacta (às vezes chamada de

resumo criptográfico, *imprint*, *digital fingerprint* ou *message digest*) da cadeia de entrada e pode ser usada como se identificasse unicamente aquela cadeia.

Em *The Hashing Function Lounge* ([5]) Barreto mostra uma relação das principais funções de resumo existentes e suas características, referências a ataques conhecidos, etc.

2.2 Aplicações

Entre outras aplicações, as funções de resumo são usadas em criptografia para:

- Garantia de integridade dos dados.
- Garantia de origem de uma mensagem.
- Cálculo de respostas que são função de uma chave secreta e de uma mensagem de desafio (em protocolos de identificação através de desafio).
- Confirmação de chave.
- Confirmação de conhecimento (habilidade de comprovar conhecimento prévio de algo sem a necessidade de expor os dados previamente).
- Derivação de chave.
- Geração de números pseudo-aleatórios.

Na próxima seção serão descritas as propriedades necessárias a essas aplicações.

2.3 Propriedades

Uma **função de resumo** h é uma função que possui, no mínimo, as seguintes propriedades:

compressão h mapeia uma entrada x de comprimento (número de bits) arbitrário em uma saída $h(x)$ de comprimento n.

eficiência computacional dados h e a entrada x, $h(x)$ é fácil de calcular.

Além dessas, as funções de resumo criptográfico em geral possuem as seguintes propriedades adicionais:

resistência a inversão (*preimage resistance*) Para (quase) todas as saídas é computacionalmente difícil encontrar uma entrada que leve àquela saída, ou seja, encontrar uma pré-imagem x' tal que $h(x') = y$ quando é dado um y para o qual a entrada é desconhecida. Esta propriedade também é chamada de mão única (*one-way*).

resistência a segunda inversão (*2nd preimage resistance*) é computacionalmente difícil encontrar uma segunda entrada que tenha a mesma saída que qualquer entrada especificada; ou seja, dado x encontrar uma segunda pré-imagem $x' \neq x$ tal que $h(x) = h(x')$. Esta propriedade também é chamada de resistência fraca à colisão (*weak collision resistance*).

resistência a colisão (*collision resistance*) é computacionalmente difícil encontrar quaisquer duas entradas distintas que levem à mesma saída, ou seja tais que $h(x) = h(x')$ Esta propriedade também é chamada de resistência forte à colisão (*strong collision resistance*).

A resistência a colisões torna as funções de resumo excelentes para a deteção de modificações (acidentais ou não) em arquivos: é muito pouco provável que uma alteração no arquivo não resulte em modificação do valor do resumo.

Algumas observações importantes a respeito daquelas propriedades são:

- Resistência a colisão implica em resistência à segunda inversão.
- Resistência à segunda inversão não implica em resistência à inversão (nem resistência à inversão implica em resistência à segunda inversão.
- Resistência a colisão não implica em resistência a inversão.

Computacionalmente impraticável (ou difícil) deve ser entendido dentro de um contexto de referência, que varia de acordo com a evolução da tecnologia, recursos à disposição de um (potencial) atacante, etc.

2.4 Funções de resumo estudadas

2.4.1 A família MD5

A família MD5 foi projetada pela RSA (MD2) e por Ron Rivest (MD4 e MD5) e compreende:

- MD2 - RFC 1319 ([33]).
- MD4 - RFC 1320 ([49]).
- MD5 - RFC 1321 ([50]).

Todos os membros da família geram um resumo de 128 bits.

Durante muito tempo o MD5 foi largamente usado, chegando a ser considerado quase um sinônimo de função de resumo. No entanto, devido aos ataques, esta função tem caído em desuso, sendo substituída por outras que têm tamanhos de resumo maiores, como o SHA-1. Em [57], Wang et al. descrevem um dos ataques mais recentes ao MD5.

2.4.2 A família SHA

Os algoritmos da família SHA (*Secure Hash Algorithm*) foram especificados pelo padrão FIPS (Federal Information Processing Standards) 180-2 do US National Institute of Standards and Technology (órgão do governo americano). O padrão original (FIPS 180-1) especificava apenas o algoritmo conhecido como SHA-1, que tem um resumo (*message digest*) de 160 bits. O FIPS 180-2 (publicado em agosto de 2002) acrescentou três novos algoritmos (SHA-256, SHA-384 e SHA-512) e uma mudança (publicada em fevereiro de 2004) adicionou o algoritmo SHA-224. A RFC 3174 ([1]) também descreve o SHA-1, trazendo inclusive uma implementação em linguagem C do algoritmo.

Os algoritmos da família SHA estão relacionados ao MD5 mas utilizam tamanhos de resumo maiores (a partir de 160). Esta semelhança é, inclusive, fonte de críticas, visto que ataques ao MD5 podem eventualmente servir de "entrada" para possíveis ataques às funções da família SHA.

Os algoritmos hoje aceitos pelo padrão são:

- SHA-1.
- SHA-224.
- SHA-256.
- SHA-384.
- SHA-512.

O padrão descreve também uma forma de truncar o resultado para obter resumos de outros comprimentos (por exemplo 96 bits), como conveniência para facilitar a interoperabilidade de aplicações utilizando tais resumos. No entanto, o padrão não faz nenhuma alegação quanto à segurança de tais resumos truncados e proíbe a utilização deles em aplicações padronizadas que referenciam aquele padrão.

A descrição e a notação usadas nesta seção e nas seguintes seguem a especificação oficial do algoritmo ([44]). Os valores das constantes (K_t) e dos valores iniciais do resumo (H^0) não são mostrados aqui mas também podem ser encontrados naquela referência.

A tabela 2.1 (baseada em [44, seções 4.1 e 4.2, seção 1 — figura 1 e *change notice 1*]) resume as principais características dos algoritmos da família SHA. É interessante observar que:

- Os valores de inicialização são diferentes para cada um dos algoritmos.
- O algoritmo SHA-224 é idêntico ao algoritmo SHA-256 (exceto pelo valor de inicialização) — o valor do resumo final (de 256 bits) é truncado para obter um valor de 224 bits.

- Um mecanismo análogo relaciona o SHA-384 e o SHA-512.
- Conforme explica a especificação [44, seção 1, nota de rodapé 2], segurança neste contexto refere-se ao fato de que um "ataque de aniversário" (*birthday attack*) em uma mensagem com sumário de tamanho n produz uma colisão com complexidade $2^{n/2}$. Em, [58], no entanto, Wang et al. mostram um ataque de complexidade 2^{69} (inferior ao limite teórico de 2^{80}).

Tabela 2.1: Comparação entre os algoritmos da família SHA

Algoritmo	**Tamanho da mensagem (bits)**	**Tamanho do bloco (bits)**	**Tamanho do resumo (bits)**	**Segurança (bits)**	**Funções lógicas**	**Constantes**
SHA-1	$< 2^{64}$	512 (16 palavras de 32 bits)	160	80	Uma seqüência de 80 funções lógicas de 4 tipos diferentes. Cada função opera em 3 palavras de 32 bits.	80 constantes de 32 bits (apenas 4 valores diferentes).
SHA-224	$< 2^{64}$	512 (16 palavras de 32 bits)	224	112	6 funções lógicas. Cada função opera em 3 palavras de 32 bits.	64 constantes de 32 bits (primeiros 32 bits da parte fracional das raízes cúbicas dos primeiros 64 primos).
SHA-256	$< 2^{64}$	512 (16 palavras de 32 bits)	256	128	6 funções lógicas. Cada função opera em 3 palavras de 32 bits.	64 constantes de 32 bits (primeiros 32 bits da parte fracional das raízes cúbicas dos primeiros 64 primos).
SHA-384	$< 2^{128}$	1024 (16 palavras de 64 bits)	384	192	6 funções lógicas. Cada função opera em 3 palavras de 64 bits.	80 constantes de 64 bits (primeiros 64 bits da parte fracional das raízes cúbicas dos primeiros 64 primos).
SHA-512	$< 2^{128}$	1024 (16 palavras de 64 bits)	512	256	6 funções lógicas. Cada função opera em 3 palavras de 64 bits.	80 constantes de 64 bits (primeiros 64 bits da parte fracional das raízes cúbicas dos primeiros 64 primos).

2.4.2.1 O algoritmo SHA-1

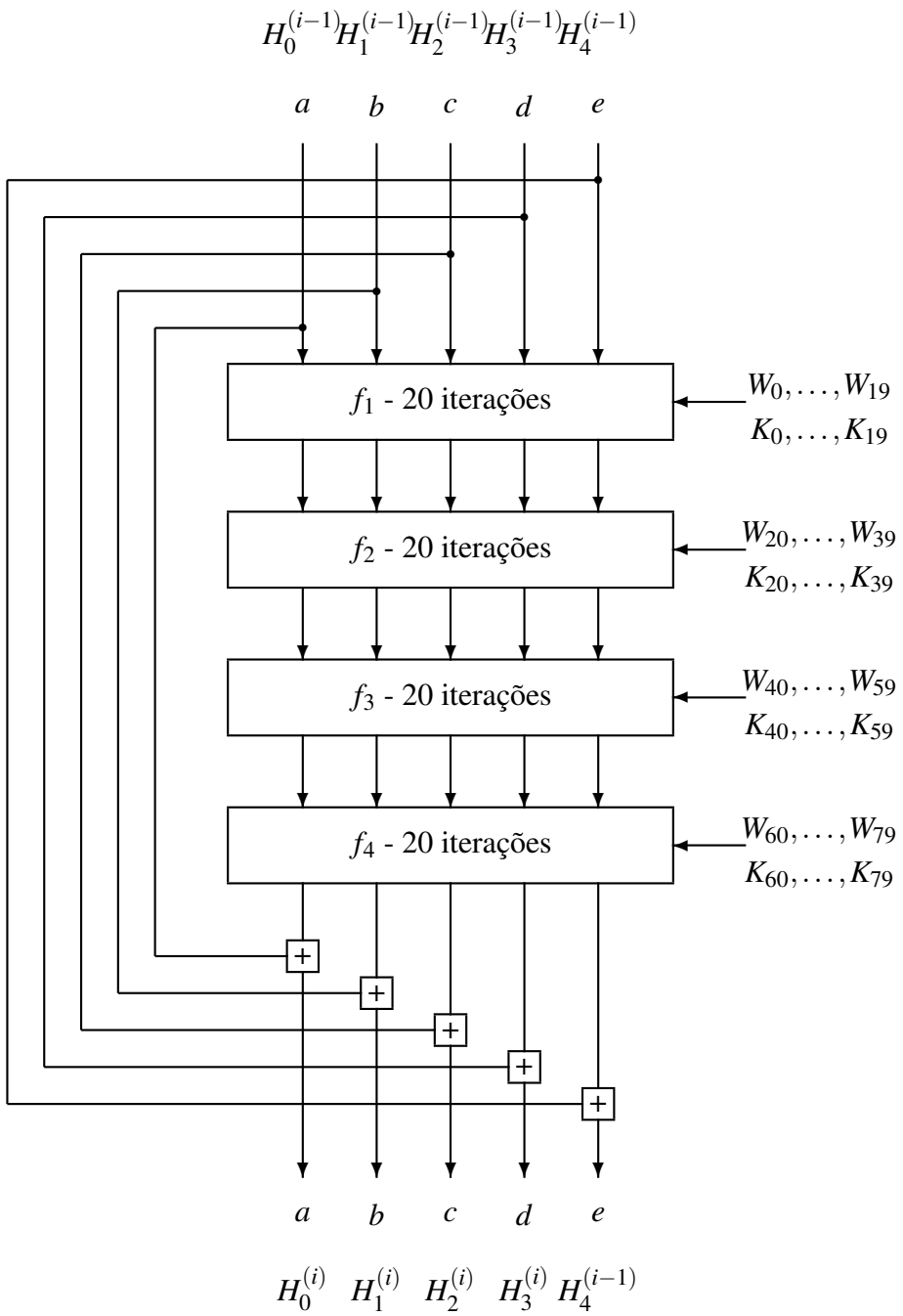
Este algoritmo pode ser usado para processar uma mensagem M de comprimento l bits (onde $0 \leqslant l < 2^{64}$) e utiliza:

- 80 chaves de 32 bits ($W_0 \ldots W_{79}$).
- 5 variáveis de trabalho de 32 bits (W_t) (a,b,c,d,e) e uma variável temporária de 32 bits (T).
- Um valor de resumo que consiste de 5 palavras de 32 bits ($H_0^{(i)} \ldots H_5^{(i)}$).

O valor final do resumo é um valor de 160 bits.

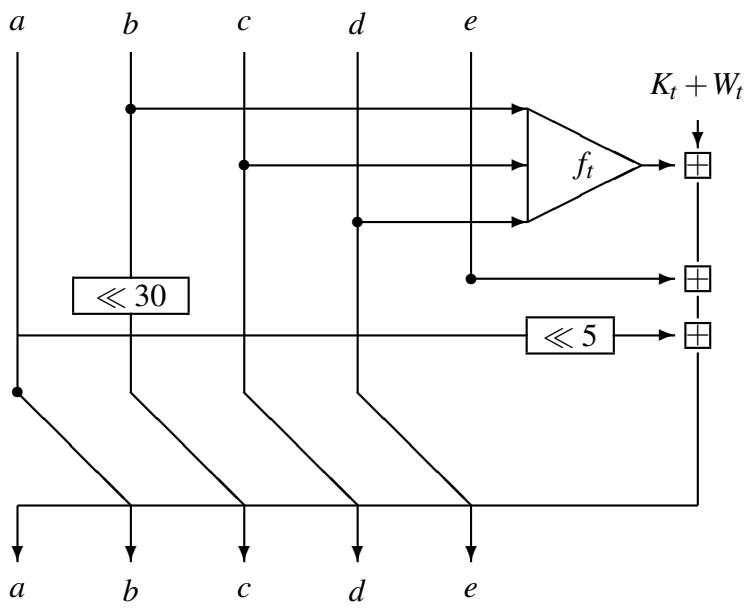
A figura 2.1 (baseada em [48, figura 6.12] e [54, figura 9.5]) mostra uma visão geral do processamento efetuado pelo algoritmo SHA-1. (A adição é efetuada módulo 2^{32}).

Figura 2.1: Visão geral do algoritmo SHA-1



A figura 2.2 (baseada em [48, figura 6.12]) mostra mais detalhadamente um passo do algoritmo SHA-1.

Figura 2.2: Passo do algoritmo SHA-1



Na especificação do algoritmo, a notação $\text{ROTR}^n(x)$ indica rotação do valor x à direita n bits, $\text{ROTL}^n(x)$ indica rotação do valor x à esquerda n bits e $\text{SHR}^n(x)$ indica deslocamento do valor x à direita n bits.

Inicialmente, o algoritmo executa um pré-processamento, que consiste em:

1. Complementar a mensagem fazendo com que o seu comprimento seja um múltiplo de 512 bits. Supondo que o comprimento da mensagem é l, é acrescentado um bit "1", seguido de k bits "0" e do valor de l representado em 64 bits. O valor de k é calculado como a menor solução não-negativa para a equação $l + 1 + k = 448 \mod 512$.

2. Dividir a mensagem em blocos de 512 bits, $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$.

3. Inicializar o valor do resumo H^0.

Em seguida, os blocos da mensagem ($M^{(1)} \ldots M^{(N)}$) são processados em ordem e para cada um deles as seguintes etapas são repetidas:

1. Cálculo das chaves:

$$W_t = \begin{cases} M_t^{(i)} & 0 \leqslant t \leqslant 15, \\ ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) & 16 \leqslant t \leqslant 79. \end{cases}$$

2. Inicialização das variáveis de trabalho a, b, c, d e e com o $(i-1)$-ésimo valor do resumo:

$$
\begin{aligned}
a &= H_0^{(i-i)} \\
b &= H_1^{(i-i)} \\
c &= H_2^{(i-i)} \\
d &= H_3^{(i-i)} \\
e &= H_4^{(i-i)}
\end{aligned}
$$

3. Repetição do laço principal 80 vezes ($0 \leqslant t \leqslant 79$):

$$
\begin{aligned}
T &= \text{ROTL}^5(a) + f_t(b,c,d) + e + K_t + W_t \\
e &= d \\
d &= c \\
c &= \text{ROTL}^{30}(b) \\
b &= a \\
a &= T
\end{aligned}
$$

onde

$$
f_t(x,y,z) = \begin{cases}
\text{Ch}(x,y,z) & 0 \leqslant t \leqslant 19 \\
\text{Parity}(x,y,z) & 20 \leqslant t \leqslant 39 \\
\text{Maj}(x,y,z) & 40 \leqslant t \leqslant 59 \\
\text{Parity}(x,y,z) & 60 \leqslant t \leqslant 79
\end{cases}
$$

e

$$
\begin{aligned}
\text{Ch}(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z) \\
\text{Parity}(x,y,z) &= (x \oplus y \oplus z) \\
\text{Maj}(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)
\end{aligned}
$$

4. Cálculo do i-ésimo valor intermediário do resumo $H^{(i)}$:

$$H_0^{(i)} = a + H_0^{(i-i)}$$
$$H_1^{(i)} = b + H_1^{(i-i)}$$
$$H_2^{(i)} = c + H_2^{(i-i)}$$
$$H_3^{(i)} = d + H_3^{(i-i)}$$
$$H_4^{(i)} = e + H_4^{(i-i)}$$

Depois do processamento de todos os blocos da mensagem, o resultado é obtido através de ($\|$ indica concatenação):

$$H_0^{(N)} \parallel H_1^{(N)} \parallel H_2^{(N)} \parallel H_3^{(N)} \parallel H_4^{(N)}.$$

2.4.2.2 Os algoritmos SHA-224 e SHA-256

Estes algoritmos podem ser usados para processar uma mensagem M de comprimento l bits (onde $0 \leqslant l < 2^{64}$) e utilizam:

- 64 chaves de 32 bits ($W_0 \ldots W_{63}$).
- 8 variáveis de trabalho de 32 bits (W_t) (a,b,c,d,e,f,g,h) e duas variáveis temporárias de 32 bits (T_1 e T_2).
- Um valor de resumo que consiste de 8 palavras de 32 bits ($H_0^{(i)} \ldots H_7^{(i)}$).

O valor final do resumo tem 256 bits para o SHA-256 e de 224 bits para o SHA-224.

Inicialmente, o algoritmo executa um pré-processamento, que consiste em:

1. Complementar a mensagem fazendo com que o seu comprimento seja um múltiplo de 512 bits. Supondo que o comprimento da mensagem é l, é acrescentado um bit "1", seguido de k bits "0" e do valor de l representado em 64 bits. O valor de k é calculado como a menor solução não-negativa para a equação $l + 1 + k = 448 \mod 512$.
2. Dividir a mensagem em blocos de 512 bits, $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$.
3. Inicializar o valor do resumo H^0.

Em seguida, os blocos da mensagem ($M^{(1)} \ldots M^{(N)}$) são processados em ordem e para cada um deles as seguintes etapas são repetidas:

1. Cálculo das chaves:

$$W_t = \begin{cases} M_t^{(i)} & 0 \leqslant t \leqslant 15, \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \leqslant t \leqslant 63. \end{cases}$$

onde

$$\begin{aligned} \mathrm{Ch}(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z) \\ \mathrm{Maj}(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\ \Sigma_0^{\{256\}}(x) &= \mathrm{ROTR}^2(x) \oplus \mathrm{ROTR}^{13}(x) \oplus \mathrm{ROTR}^{22}(x) \\ \Sigma_1^{\{256\}}(x) &= \mathrm{ROTR}^6(x) \oplus \mathrm{ROTR}^{11}(x) \oplus \mathrm{ROTR}^{25}(x) \\ \sigma_0^{\{256\}}(x) &= \mathrm{ROTR}^7(x) \oplus \mathrm{ROTR}^{18}(x) \oplus \mathrm{SHR}^3(x) \\ \sigma_1^{\{256\}}(x) &= \mathrm{ROTR}^{17}(x) \oplus \mathrm{ROTR}^{19}(x) \oplus \mathrm{SHR}^{10}(x) \end{aligned}$$

2. Inicialização das variáveis de trabalho a, b, c, d, e, f, g e h com o $(i-1)$-ésimo valor do resumo:

$$\begin{aligned} a &= H_0^{(i-i)} \\ b &= H_1^{(i-i)} \\ c &= H_2^{(i-i)} \\ d &= H_3^{(i-i)} \\ e &= H_4^{(i-i)} \\ f &= H_5^{(i-i)} \\ g &= H_6^{(i-i)} \\ h &= H_7^{(i-i)} \end{aligned}$$

3. Repetição do laço principal 64 vezes ($0 \leqslant t \leqslant 63$):

$$\begin{aligned} T_1 &= h + \Sigma_1^{\{256\}}(e) + \mathrm{Ch}(e,f,g) + K_t^{\{256\}} + W_t \\ T_2 &= \Sigma_0^{\{256\}}(a) + \mathrm{Maj}(a,b,c) \\ h &= g \end{aligned}$$

$$g = f$$
$$f = e$$
$$e = d + T_1$$
$$d = c$$
$$c = b$$
$$b = a$$
$$a = T_1 + T_2$$

(as funções σ, Σ, Ch e Maj são as mesmas descritas para o cálculo das chaves).

4. Cálculo do i-ésimo valor intermediário do resumo $H^{(i)}$:

$$H_0^{(i)} = a + H_0^{(i-i)}$$
$$H_1^{(i)} = b + H_1^{(i-i)}$$
$$H_2^{(i)} = c + H_2^{(i-i)}$$
$$H_3^{(i)} = d + H_3^{(i-i)}$$
$$H_4^{(i)} = e + H_4^{(i-i)}$$
$$H_5^{(i)} = f + H_5^{(i-i)}$$
$$H_6^{(i)} = g + H_6^{(i-i)}$$
$$H_7^{(i)} = h + H_7^{(i-i)}$$

Depois do processamento de todos os blocos da mensagem, o resultado é obtido através de:

$$H_0^{(N)} \parallel H_1^{(N)} \parallel H_2^{(N)} \parallel H_3^{(N)} \parallel H_4^{(N)} \parallel H_5^{(N)} \parallel H_6^{(N)} \parallel H_7^{(N)}.$$

para o SHA-256 e

$$H_0^{(N)} \parallel H_1^{(N)} \parallel H_2^{(N)} \parallel H_3^{(N)} \parallel H_4^{(N)} \parallel H_5^{(N)} \parallel H_6^{(N)}$$

para o SHA-224.

2.4.2.3 Os algoritmos SHA-384 e SHA-512

Estes algoritmos podem ser usados para processar uma mensagem M de comprimento l bits (onde $0 \leqslant l < 2^{128}$) e utilizam:

- 80 chaves de 64 bits ($W_0 \ldots W_{79}$).
- 8 variáveis de trabalho de 64 bits (W_t) (a,b,c,d,e,f,g,h) e duas variáveis temporárias de 64 bits (T_1 e T_2).
- Um valor de resumo que consiste de 8 palavras de 64 bits ($H_0^{(i)} \ldots H_7^{(i)}$).

O valor final do resumo tem 512 bits para o SHA-512 e de 384 bits para o SHA-384.

Inicialmente, o algoritmo executa um pré-processamento, que consiste em:

1. Complementar a mensagem fazendo com que o seu comprimento seja um múltiplo de 1024 bits. Supondo que o comprimento da mensagem é l, é acrescentado um bit "1", seguido de k bits "0" e do valor de l representado em 128 bits. O valor de k é calculado como a menor solução não-negativa para a equação $l + 1 + k = 896 \mod 1024$.
2. Dividir a mensagem em blocos de 512 bits, $M^{(1)}$, $M^{(2)}$, ..., $M^{(N)}$.
3. Inicializar o valor do resumo H^0.

Em seguida, os blocos da mensagem ($M^{(1)} \ldots M^{(N)}$) são processados em ordem e para cada um deles as seguintes etapas são repetidas:

1. Cálculo das chaves:

$$W_t = \begin{cases} M_t^{(i)} & 0 \leqslant t \leqslant 15, \\ \sigma_1^{\{512\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{512\}}(W_{t-15}) + W_{t-16} & 16 \leqslant t \leqslant 79. \end{cases}$$

onde

$$\begin{aligned} \mathrm{Ch}(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z) \\ \mathrm{Maj}(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\ \Sigma_0^{\{512\}}(x) &= \mathrm{ROTR}^{28}(x) \oplus \mathrm{ROTR}^{34}(x) \oplus \mathrm{ROTR}^{39}(x) \\ \Sigma_1^{\{512\}}(x) &= \mathrm{ROTR}^{14}(x) \oplus \mathrm{ROTR}^{18}(x) \oplus \mathrm{ROTR}^{41}(x) \\ \sigma_0^{\{512\}}(x) &= \mathrm{ROTR}^{1}(x) \oplus \mathrm{ROTR}^{8}(x) \oplus \mathrm{SHR}^{7}(x) \\ \sigma_1^{\{512\}}(x) &= \mathrm{ROTR}^{19}(x) \oplus \mathrm{ROTR}^{61}(x) \oplus \mathrm{SHR}^{6}(x) \end{aligned}$$

2. Inicialização das variáveis de trabalho a, b, c, d, e, f, g e h com o $(i-1)$-ésimo valor do resumo:

$$
\begin{aligned}
a &= H_0^{(i-i)} \\
b &= H_1^{(i-i)} \\
c &= H_2^{(i-i)} \\
d &= H_3^{(i-i)} \\
e &= H_4^{(i-i)} \\
f &= H_5^{(i-i)} \\
g &= H_6^{(i-i)} \\
h &= H_7^{(i-i)}
\end{aligned}
$$

3. Repetição do laço principal 80 vezes ($0 \leqslant t \leqslant 79$):

$$
\begin{aligned}
T_1 &= h + \Sigma_1^{\{512\}}(e) + \mathrm{Ch}(e,f,g) + K_t^{\{256\}} + W_t \\
T_2 &= \Sigma_0^{\{512\}}(a) + \mathrm{Maj}(a,b,c) \\
h &= g \\
g &= f \\
f &= e \\
e &= d + T_1 \\
d &= c \\
c &= b \\
b &= a \\
e &= T_1 + T_2
\end{aligned}
$$

(as funções σ, Σ, Ch e Maj são as mesmas descritas para o cálculo das chaves).

4. Cálculo do i-ésimo valor intermediário do resumo $H^{(i)}$:

$$
\begin{aligned}
H_0^{(i)} &= a + H_0^{(i-i)} \\
H_1^{(i)} &= b + H_1^{(i-i)} \\
H_2^{(i)} &= c + H_2^{(i-i)}
\end{aligned}
$$

$$
\begin{aligned}
H_3^{(i)} &= d + H_3^{(i-i)} \\
H_4^{(i)} &= e + H_4^{(i-i)} \\
H_5^{(i)} &= f + H_5^{(i-i)} \\
H_6^{(i)} &= g + H_6^{(i-i)} \\
H_7^{(i)} &= h + H_7^{(i-i)}.
\end{aligned}
$$

Depois do processamento de todos os blocos da mensagem, o resultado é obtido através de:

$$H_0^{(N)} \parallel H_1^{(N)} \parallel H_2^{(N)} \parallel H_3^{(N)} \parallel H_4^{(N)} \parallel H_5^{(N)} \parallel H_6^{(N)} \parallel H_7^{(N)}$$

para o SHA-512 e

$$H_0^{(N)} \parallel H_1^{(N)} \parallel H_2^{(N)} \parallel H_3^{(N)} \parallel H_4^{(N)} \parallel H_5^{(N)}$$

para o SHA-384.

2.4.3 *Whirlpool*

Whirlpool foi desenvolvida por Vincent Rijmen e Paulo S. L. M. Barreto ([4]) e submetida ao NESSIE — New European Schemes for Signatures, Integrity, and Encryption ([43]), tendo sido selecionada para o portfólio de primitivas criptográficas. Também foi proposta para o padrão ISO/IEC 10118-3:2003(E) ([32]). Trata-se de uma função de resumo que processa mensagens de comprimento menor do que 2^{256} bits para gerar um resumo de 512 bits. Ela consiste essencialmente na aplicação repetida de uma função de compressão, W, que usa uma chave gerada a partir da entrada ([35],[36]).

Tanto a descrição quanto a notação utilizadas nas seções seguintes são baseadas no artigo em que os autores descrevem o algoritmo ([4, especialmente as seções 3 e 7]).

A função *Whirlpool* opera sobre mensagens M cujo comprimento $L < 2^{256}$ (em bits). Antes de executar a operação de resumo sobre uma mensagem M, ela deve ser complementada com um bit "1" seguido de tantos bits "0" quantos forem necessários para chegar a uma cadeia de bits cujo tamanho seja um múltiplo ímpar de 256. Em seguida, a representação do comprimento da mensagem original (em 256 bits e justificada à direita) é acrescentada, de forma a obter a mensagem complementada m a qual é então particionada em t blocos $(m_1, \ldots, m_t)$ de 512 bits cada. Estes blocos são tratados como seqüências de 64 bytes agrupados seqüencialmente em uma matriz quadrada de 8×8 bytes. Cada um dos blocos é então processado em seqüência ([40, algoritmo

9.43]):

$$\eta_i = \mu(m_i),$$
$$H_0 = \mu(\mathrm{IV}),$$
$$H_i = W[H_{i-1}](\eta_i) \oplus H_{i-1} \oplus \eta_i, 1 \leqslant i \leqslant t,$$

onde

- μ é a função que mapeia a entrada, descrita na seção 2.4.3.1.

- IV (*initialization vector*) é uma cadeia fixa de 512 bits.

O valor final do resumo é definido como a saída H_t da função de compressão mapeada de volta para uma cadeia de bits:

$$\text{WHIRLPOOL}(M) \equiv \mu^{-1}(H_t).$$

A função W é definida como:

$$W[K] = \left(\bigcirc_{1}^{r=R} \rho[K^r] \right) \circ \sigma[K^0] \tag{2.1}$$

onde

$$\rho[k] \equiv \sigma[k] \circ \theta \circ \pi \circ \gamma$$

e K^r, o cálculo das chaves, é:

$$K^0 \equiv K$$
$$K^r \equiv \rho[c^r](K^{r-1}), r > 0.$$

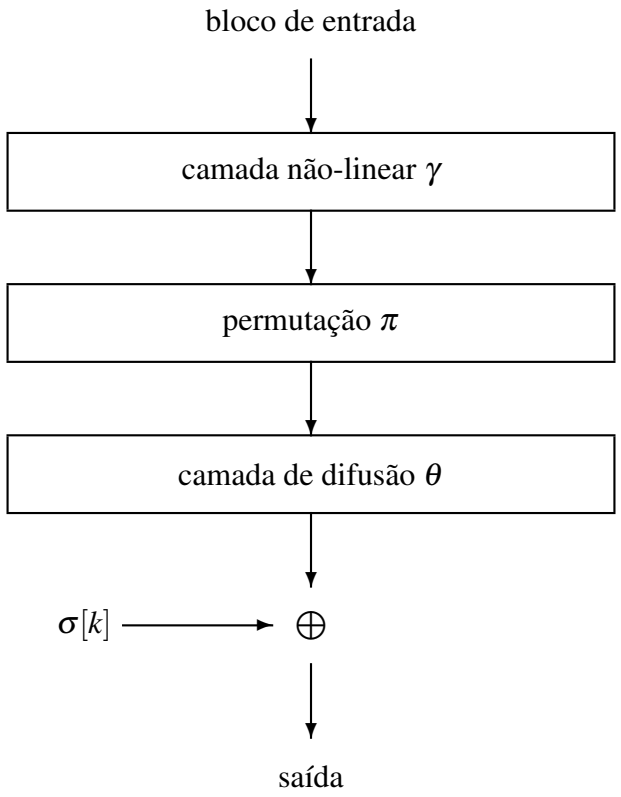
Dada uma seqüência de funções $f_m, f_{m+1}, \ldots, f_{n-1}, f_n, m \leqslant n$, a notação $\bigcirc$ é usada para indicar a composição de funções:

$$\bigcirc_{r=m}^{n} f_r \equiv f_m \circ f_{m+1} \circ \cdots \circ f_{n-1} \circ f_n$$
$$\bigcirc_{m}^{r=n} f_r \equiv f_n \circ f_{n-1} \circ \cdots \circ f_{m+1} \circ f_m$$

Para o *Whirlpool*, $R = 10$. As funções μ, σ, θ, π e γ, utilizadas no cálculo de W mostrado acima, são descritas nas seções a seguir. As operações são feitas em $\mathscr{M}_{8\times8}[\mathrm{GF}(2^8)] \rightarrow \mathscr{M}_{8\times8}[\mathrm{GF}(2^8)]$. $\mathscr{M}_{8\times8}[\mathrm{GF}(2^8)]$ denota o conjunto de matrizes 8×8 sobre o corpo finito $\mathrm{GF}(2^8)$.

A figura 2.3 (baseada em [35, figura 1] e [36, figura 1]) mostra um passo do algoritmo *Whirlpool*.

Figura 2.3: Passo do algoritmo *Whirlpool*



2.4.3.1 Mapeamento da entrada e saída

O estado do resumo é visto internamente como uma matriz $\mathscr{M}_{8\times8}[\mathrm{GF}(2^8)]$ e a função que faz o mapeamento $\mathscr{M}_{8\times8}[\mathrm{GF}(2^8)] \rightarrow \mathscr{M}_{8\times8}[\mathrm{GF}(2^8)]$ é definida como:

$$\mu(a) = b \Leftrightarrow b_{ij} = a_{8i+j}, 0 \leqslant i, j \leqslant 7$$

Para exemplificar, seja a entrada um bloco de 64 bytes (512 bits)

$$a_{00}\ a_{01}\ a_{02}\ a_{03}\ a_{04}\ a_{05}\ a_{06}\ \ldots\ a_{76}\ a_{77}$$

(os índices dos bits estão expressos em notação octal)

A matriz resultante, a, será:

$$a = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} & a_{06} & a_{07} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} \\ a_{60} & a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} \\ a_{70} & a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} \end{bmatrix}$$

2.4.3.2 Camada não-linear γ

Esta camada consiste na aplicação da matriz de substituição (*S-box*),

$$\gamma(a) = b \Leftrightarrow b_{ij} = S[a_{ij}], 0 \leqslant i, j \leqslant 7$$

2.4.3.3 Permutação cíclica π

Esta camada consiste essencialmente em deslocar a coluna j para baixo j posições:

$$\pi(a) = b \Leftrightarrow b_{ij} = a_{(i-j) \bmod 8, j}, 0 \leqslant i, j \leqslant 7$$

Entrada:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} & a_{06} & a_{07} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} \\ a_{60} & a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} \\ a_{70} & a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} \end{bmatrix}$$

Saída:

$$\pi(a) = \begin{bmatrix} a_{00} & a_{71} & a_{62} & a_{53} & a_{44} & a_{35} & a_{26} & a_{17} \\ a_{10} & a_{01} & a_{72} & a_{63} & a_{54} & a_{45} & a_{36} & a_{27} \\ a_{20} & a_{11} & a_{02} & a_{73} & a_{64} & a_{55} & a_{46} & a_{37} \\ a_{30} & a_{21} & a_{12} & a_{03} & a_{74} & a_{65} & a_{56} & a_{47} \\ a_{40} & a_{31} & a_{22} & a_{13} & a_{04} & a_{75} & a_{66} & a_{57} \\ a_{50} & a_{41} & a_{32} & a_{23} & a_{14} & a_{05} & a_{76} & a_{67} \\ a_{60} & a_{51} & a_{42} & a_{33} & a_{24} & a_{15} & a_{06} & a_{77} \\ a_{70} & a_{61} & a_{52} & a_{43} & a_{34} & a_{25} & a_{16} & a_{07} \end{bmatrix}$$

2.4.3.4 Camada de difusão linear θ

Esta camada consiste na multiplicação por uma matriz circulante (C):

$$\theta(a) = b \Leftrightarrow b = a \,.\, C$$

Ou seja, em $b = \theta(a)$:

$$b_{ij} = \sum_{l=0}^{7} a_{il} . c_{li}$$

De acordo com os valores definidos para o *Whirlpool*, a matriz C é [1] :

[1] O $_x$ indica que o valor está expresso em notação hexadecimal.

$$C = \begin{bmatrix} 01_x & 01_x & 04_x & 01_x & 08_x & 05_x & 02_x & 09_x \\ 09_x & 01_x & 01_x & 04_x & 01_x & 08_x & 05_x & 02_x \\ 02_x & 09_x & 01_x & 01_x & 04_x & 01_x & 08_x & 05_x \\ 05_x & 02_x & 09_x & 01_x & 01_x & 04_x & 01_x & 08_x \\ 08_x & 05_x & 02_x & 09_x & 01_x & 01_x & 04_x & 01_x \\ 01_x & 08_x & 05_x & 02_x & 09_x & 01_x & 01_x & 04_x \\ 04_x & 01_x & 08_x & 05_x & 02_x & 09_x & 01_x & 01_x \\ 01_x & 04_x & 01_x & 08_x & 05_x & 02_x & 09_x & 01_x \end{bmatrix}$$

Em termos da matriz original (a), a aplicação seqüêncial de $\theta \circ \pi \circ \gamma$ produz uma saída b onde:

$$b_{00} = S[a_{00}].c_{00} + S[a_{71}].c_{10} + S[a_{62}].c_{20} + S[a_{53}].c_{30} + S[a_{44}].c_{40} + S[a_{35}].c_{50} + S[a_{26}].c_{60} + S[a_{17}].c_{70}$$
$$b_{01} = S[a_{00}].c_{01} + S[a_{71}].c_{11} + S[a_{62}].c_{21} + S[a_{53}].c_{31} + S[a_{44}].c_{41} + S[a_{35}].c_{51} + S[a_{26}].c_{61} + S[a_{17}].c_{71}$$
$$b_{02} = S[a_{00}].c_{02} + S[a_{71}].c_{12} + S[a_{62}].c_{22} + S[a_{53}].c_{32} + S[a_{44}].c_{42} + S[a_{35}].c_{52} + S[a_{26}].c_{62} + S[a_{17}].c_{72}$$
$$b_{03} = S[a_{00}].c_{03} + S[a_{71}].c_{13} + S[a_{62}].c_{23} + S[a_{53}].c_{33} + S[a_{44}].c_{43} + S[a_{35}].c_{53} + S[a_{26}].c_{63} + S[a_{17}].c_{73}$$
$$b_{04} = S[a_{00}].c_{04} + S[a_{71}].c_{14} + S[a_{62}].c_{24} + S[a_{53}].c_{34} + S[a_{44}].c_{44} + S[a_{35}].c_{54} + S[a_{26}].c_{64} + S[a_{17}].c_{74}$$
$$b_{05} = S[a_{00}].c_{05} + S[a_{71}].c_{15} + S[a_{62}].c_{25} + S[a_{53}].c_{35} + S[a_{44}].c_{45} + S[a_{35}].c_{55} + S[a_{26}].c_{65} + S[a_{17}].c_{75}$$
$$b_{06} = S[a_{00}].c_{06} + S[a_{71}].c_{16} + S[a_{62}].c_{26} + S[a_{53}].c_{36} + S[a_{44}].c_{46} + S[a_{35}].c_{56} + S[a_{26}].c_{66} + S[a_{17}].c_{76}$$
$$b_{07} = S[a_{00}].c_{07} + S[a_{71}].c_{17} + S[a_{62}].c_{27} + S[a_{53}].c_{37} + S[a_{44}].c_{47} + S[a_{35}].c_{57} + S[a_{26}].c_{67} + S[a_{17}].c_{77}$$
$$\ldots$$
$$b_{70} = S[a_{70}].c_{00} + S[a_{61}].c_{10} + S[a_{52}].c_{20} + S[a_{43}].c_{30} + S[a_{34}].c_{40} + S[a_{25}].c_{50} + S[a_{16}].c_{60} + S[a_{07}].c_{70}$$
$$b_{71} = S[a_{70}].c_{01} + S[a_{61}].c_{11} + S[a_{52}].c_{21} + S[a_{43}].c_{31} + S[a_{34}].c_{41} + S[a_{25}].c_{51} + S[a_{16}].c_{61} + S[a_{07}].c_{71}$$
$$b_{72} = S[a_{70}].c_{02} + S[a_{61}].c_{12} + S[a_{52}].c_{22} + S[a_{43}].c_{32} + S[a_{34}].c_{42} + S[a_{25}].c_{52} + S[a_{16}].c_{62} + S[a_{07}].c_{72}$$
$$b_{73} = S[a_{70}].c_{03} + S[a_{61}].c_{13} + S[a_{52}].c_{23} + S[a_{43}].c_{33} + S[a_{34}].c_{43} + S[a_{25}].c_{53} + S[a_{16}].c_{63} + S[a_{07}].c_{73}$$
$$b_{74} = S[a_{70}].c_{04} + S[a_{61}].c_{14} + S[a_{52}].c_{24} + S[a_{43}].c_{34} + S[a_{34}].c_{44} + S[a_{25}].c_{54} + S[a_{16}].c_{64} + S[a_{07}].c_{74}$$
$$b_{75} = S[a_{70}].c_{05} + S[a_{61}].c_{15} + S[a_{52}].c_{25} + S[a_{43}].c_{35} + S[a_{34}].c_{45} + S[a_{25}].c_{55} + S[a_{16}].c_{65} + S[a_{07}].c_{75}$$
$$b_{76} = S[a_{70}].c_{06} + S[a_{61}].c_{16} + S[a_{52}].c_{26} + S[a_{43}].c_{36} + S[a_{34}].c_{46} + S[a_{25}].c_{56} + S[a_{16}].c_{66} + S[a_{07}].c_{76}$$
$$b_{77} = S[a_{70}].c_{07} + S[a_{61}].c_{17} + S[a_{52}].c_{27} + S[a_{43}].c_{37} + S[a_{34}].c_{47} + S[a_{25}].c_{57} + S[a_{16}].c_{67} + S[a_{07}].c_{77}$$

2.4.3.5 Adição de chaves $\sigma[k]$

$\sigma[k](a) = b \Leftrightarrow b_{ij} = a_{ij} \oplus k_{ij}, 0 \leqslant i, j \leqslant 7$

2.4.3.6 As constantes c^r

As constantes utilizadas são definidas como:

$$\begin{aligned} c^r_{0j} &\equiv S[8(r-1)+j], \quad 0 \leqslant j \leqslant 7 \\ c^r_{ij} &\equiv 0, \qquad\qquad\quad 1 \leqslant i \leqslant 7, 0 \leqslant j \leqslant 7 \end{aligned}$$

2.5 Implementações das funções estudadas

2.5.1 Implementações da família SHA existentes

A seguir são levantados alguns dos pontos mais interessantes de algumas das implementações encontradas, sem pretensão de fazer uma análise completa de cada uma delas. Além das implementações mais conhecidas (como OpenSSL e MIRACL), foram incluídas outras implementações que apresentaram desempenho diferenciado em alguns casos (por exemplo Devine, Gifford) ou opções úteis para comparação (como o desenrolamento do laço em Saddi).

2.5.1.1 Crypto++

Crypto++ ([12]) é uma biblioteca que implementa diversas funções relacionadas à criptografia e inclui uma implementação da família SHA. Trata-se de uma implementação em C++ bastante otimizada. Utiliza um laço completamente desenrolado para o SHA-1 e parcialmente desenrolado para o SHA-256 e SHA-512. O cálculo das chaves é feito junto ao laço principal.

2.5.1.2 Christophe Devine

Devine ([14] e [15]) fez uma implementação de SHA-1 e SHA-256, que não suporta bits, apenas bytes.

Usando artifícios como

```
#define P(a,b,c,d,e,x)                                  \
{                                                       \
    e += S(a,5) + F(b,c,d) + K + x; b = S(b,30);        \
}
```

a implementação é bastante concisa e legível.

Os laços (tanto o de cálculo das chaves quanto o laço principal) são completamente desenrolados.

2.5.1.3 Aaron D. Gifford

A implementação de Gifford ([19]) suporta todas as funções da família SHA (SHA-1, SHA-224, SHA-256, SHA-384 e SHA-512) mas também não suporta bits, apenas bytes.

Traz boas sugestões na definição de tipos, usando os tipos definidos pelo ANSI C em `inttypes.h` (como `uint8_t`, `uint32_t`, etc.), definição de macros para manipulação de bits em plataformas *big endian* [2] e *little endian* [3].

Utiliza um `#define` para controlar se o laço principal é desenrolado. O cálculo das chaves é feito no meio enquanto o laço principal e os valores são usados diretamente no cálculo — em vez de calcular todas as chaves antes do laço principal.

2.5.1.4 GNU Crypto

GNU Crypto [20] é uma implementação em linguagem java, sem grandes artifícios visando a otimização.

2.5.1.5 Intel® *Integrated Performance Primitives* (IPP)

A Intel® oferece uma biblioteca, chamada de *Integrated Performance Primitives* ([22]), que implementa de forma otimizada para diversas plataformas muitas operações comumente utilizadas pelas aplicações, incluindo aquelas necessárias para criptografia. As funções da família SHA (exceto SHA-224) estão disponíveis mas como se trata de uma implementação proprietária o seu código não foi analisado.

2.5.1.6 *Multiprecision Integer and Rational Arithmetic C/C++ Library* (MIRACL)

Embora não seja o objetivo principal da biblioteca MIRACL ([37]), ela também inclui suporte às funçoes de resumo da família SHA.

2.5.1.7 LibTomCrypt

Libtomcrypt ([13]) é uma biblioteca de funções ligadas à criptografia, incluindo funções de resumo, que foi criada por Tom St. Denis. Traz uma implementação de SHA-1, SHA-224, SHA-256, SHA-384 e SHA-512. Também não suporta bits, apenas bytes.

O cálculo das chaves é feito no início e não tem opção de desenrolar.

[2] Sistema em que o byte mais significativo é armazenado primeiro (isto é, no endereço de memória mais baixo), comum à maioria dos processadores RISC ([59])

[3] Sistema em que o byte menos significativo é armazenado primeiro (isto é, no endereço de memória mais baixo), como a família de processadores do PC ([59])

Um vetor (`S`) armazena o conteúdo dos registradores. Utiliza um `#define` para controlar o modo de operação. Se a opção de gerar código reduzido for utilizada, os laços são do tipo:

```
#define RND(a,b,c,d,e,f,g,h,i)                          \
     t0 = h + Sigma1(e) + Ch(e, f, g) + K[i] + W[i];    \
     t1 = Sigma0(a) + Maj(a, b, c);                     \
     d += t0;                                           \
     h  = t0 + t1;

     for (i = 0; i < 64; ++i) {
         RND(S[0],S[1],S[2],S[3],S[4],S[5],S[6],S[7],i);
         t = S[7]; S[7] = S[6]; S[6] = S[5]; S[5] = S[4];
         S[4] = S[3]; S[3] = S[2]; S[2] = S[1]; S[1] = S[0]; S[0] = t;
     }
```

Caso contrário os laços são desenrolados usando o vetor para armazenamento temporário, conforme mostrado abaixo:

```
#define RND(a,b,c,d,e,f,g,h,i,ki)                       \
     t0 = h + Sigma1(e) + Ch(e, f, g) + ki + W[i];      \
     t1 = Sigma0(a) + Maj(a, b, c);                     \
     d += t0;                                           \
     h  = t0 + t1;

    RND(S[0],S[1],S[2],S[3],S[4],S[5],S[6],S[7],0,0x428a2f98);
    RND(S[7],S[0],S[1],S[2],S[3],S[4],S[5],S[6],1,0x71374491);
    RND(S[6],S[7],S[0],S[1],S[2],S[3],S[4],S[5],2,0xb5c0fbcf);
    ...
```

A descrição acima é válida apenas para algumas funções (não vale para SHA-1, por exemplo).

2.5.1.8 OpenSSL

A biblioteca do OpenSSL ([46]) inclui diversas rotinas ligadas à criptografia, incluindo as funções da família SHA.

A implementação do SHA-1 não utiliza as extensões MMX™/SSE. A implementação do SHA-512 utiliza as extensões MMX™/SSE essencialmente para executar operações de 64 bits, que são bem mais lentas quando são usados apenas os registradores de 32 bits. Aparentemente apenas o cálculo das chaves explora os registradores de 128 bits para executar operações em paralelo. Tanto para o SHA-1 quanto para o SHA-512, a abordagem escolhida pelo autor foi otimizar o código *assembly* gerado pelo compilador e, segundo ele, houve um ganho considerável no desempenho. O código é bastante documentado e pode ser uma boa referência para uma implementação otimizada para o Pentium®.

2.5.1.9 Allan Saddi

A implementação de Saddi ([51]) suporta toda a família SHA (SHA-1, SHA-224, SHA-256, SHA-384 e SHA-512) mas, assim como outras implementações, não suporta bits, apenas bytes.

Utiliza um `#define` para controlar quantas vezes o laço principal é desenrolado. As opções são 1, 2, 4, 5, 10 ou 20 para o SHA-1 e 1, 2, 3, 4, 8, 16, 32 ou 64 para o SHA-256 mas não há opção de desenrolar para o SHA-512. O cálculo das chaves é feito no início e este laço não tem opção de desenrolamento.

2.5.2 Implementações de *Whirlpool* existentes

Provavelmente por se tratar de um algoritmo mais recente, o número de implementações de *Whirlpool* encontradas foi bem menor. A implementação como parte do *kernel* do Linux só foi identificada no final do trabalho e não foi incluída nesta análise por limitações de tempo.

2.5.2.1 Paulo S. L. M. Barreto e Vincent Rijmen

A implementação dos autores do algoritmo ([6]) utiliza tabelas de busca de 64 bits para tornar o cálculo mais rápido. A seção 3.5.2.1 descreve de forma mais detalhada como esta otimização é possível.

2.5.2.2 Crypto++

Conforme comentário no arquivo, a implementação que faz parte da biblioteca Crypto++ ([12]) é baseada na implementação de Barreto e Rijmen ([6]) e foi adaptada por Kevin Springle. Além de utilizar 4 (em vez de 8) tabelas de busca, observa-se um grande emprego de macros para efetuar o processamento

2.5.2.3 LibTomCrypt

Esta implementação ([13]), que faz parte da mesma biblioteca citada na seção 2.5.1.7 também utiliza 8 tabelas de busca, assim como a implementação de Barreto e Rijmen ([6]).

Como alternativa, ela utiliza rotações da mesma tabela em vez de tabelas separadas, como mostrado pelo trecho de código abaixo:

```
#ifdef LTC_SMALL_CODE

#define SB0(x) sbox0[x]
#define SB1(x) ROR64c(sbox0[x], 8)
#define SB2(x) ROR64c(sbox0[x], 16)
```

```
#define SB3(x) ROR64c(sbox0[x], 24)
#define SB4(x) ROR64c(sbox0[x], 32)
#define SB5(x) ROR64c(sbox0[x], 40)
#define SB6(x) ROR64c(sbox0[x], 48)
#define SB7(x) ROR64c(sbox0[x], 56)

#else

#define SB0(x) sbox0[x]
#define SB1(x) sbox1[x]
#define SB2(x) sbox2[x]
#define SB3(x) sbox3[x]
#define SB4(x) sbox4[x]
#define SB5(x) sbox5[x]
#define SB6(x) sbox6[x]
#define SB7(x) sbox7[x]
```

2.5.3 Histórico da análise de desempenho de funções de resumo

O relato a seguir é focado na família SHA e *Whirlpool*.

2.5.3.1 Implementação de SHA

Na análise da implementação do SHA-1 feita for McCurley ([39, seção 5]) encontramos as seguintes sugestões principais:

- Desenrolamento dos laços, tanto do cálculo das chaves quanto do corpo principal.
- Tentativa de manutenção de W em registradores, fazendo o cálculo das chaves imediatamente antes de elas serem necessárias em vez de fazer o cálculo no início.
- Redefinição dos registradores, ou seja, em vez de fazer o deslocamento (copiar B para A, por exemplo) simplesmente tratar A como se fosse B na iteração seguinte do laço.
- Reescrita das funções $f_n(x,y,z)$ definidas em [44, 4.1.1]:
  - A função

    $$\mathrm{Ch}(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$$

    (usada para $0 \leqslant t \leqslant 19$) pode ser reescrita como

$$(z \oplus (x \wedge (y \oplus z)))$$

(que usa três operações em vez de quatro).

– A função

$$\text{Maj}(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

(usada para $40 \leqslant t \leqslant 59$) pode ser reescrita como

$$(x \wedge y) \vee (z \wedge (x \vee y))$$

(que usa quatro operações em vez de cinco). Esta poderia ainda ser reescrita como

$$x \wedge (y \oplus z) \oplus (z \wedge y)$$

que pode ser mais rápida em alguns processadores.

– A função

$$\text{Parity}(x,y,z) = x \oplus y \oplus z$$

(utilizada para $20 \leqslant t \leqslant 39$ e para $60 \leqslant t \leqslant 79$) não é discutida (certamente porque já utiliza um mínimo de operações sobre os valores).

- Utilização de suporte de hardware para a rotação.
- Alocação de registradores, de forma a manter todos os valores usados no corpo principal em registradores e evitar acessos (caros) à memória. Hoje esta recomendação se traduz em cuidado no acesso ao *cache*.

Em [10] Bosselaers faz uma análise de caminho crítico das operações de funções da família MD4 (incluindo SHA-1) e quanto do paralelismo existente nestas funções poderia ser explorado nas arquiteturas existentes, trazendo uma discussão teórica sobre o que seria o máximo de paralelismo possível no SHA-1.

2.5.3.2 Desempenho do MD5

Em [56] e [55] Touch analisa o desempenho do MD5 (precursor do SHA) e faz, entre outras, as seguintes sugestões:

- Gerência de *cache*.

- Otimização das operações de troca de ordem dos bytes, usando linguagem *assembly* quando necessário.
- Desenrolamento dos laços.

2.5.3.3 Funções de resumo no Pentium®

Em [9] e [8] Bosselaers discute implementações otimizadas de várias funções de resumo (incluindo SHA-1) no processador Pentium® da Intel®, incluindo a utilização dos *pipes U* e *V* do Pentium® de forma a conseguir o máximo de paralelismo na execução das instruções ([9, seção 3]).

Em [42] Nakajima et al. discutem a implementação de várias funções, incluindo SHA-1, SHA-256, SHA-512 e *Whirlpool*. Três métodos são comparados ([42, seção 5.1]):

- implementação direta, usando os registradores x86.
- dois blocos (processamento de duas mensagens em paralelo) para as funções de 32 bits, usando as instruções MMX™.
- três blocos em paralelo (combinação dos métodos anteriores), também para as funções de 32 bits.

Algumas conclusões interessantes incluem:

- SHA-512 — utilização de adição de 32 bits para suprir a falta da adição de 64 bits no Pentium® III ([42, seção 5.2]).
- *Whirlpool*— utilização de quatro tabelas de 2 KiB [4] (em vez da implementação padrão com oito tabelas descrita em [4]), utilizando a instrução **`pshufw`** (*packet shuffle word*) para gerar os dados quando necessário, de forma a evitar que as tabelas ocupem todo o *cache* (de 16 KiB) do Pentium® ([42, seção 5.2]). De acordo com a documentação da Intel® ([26, seção 2.4.1.1]) o tamanho do *cache* é de 32 KiB portanto o tamanho das tabelas não seria necessariamente um problema para o XScale®.

Em [38, seção 7] Matsui et al. comparam o desempenho de SHA512 e *Whirlpool* (não tratam das outras funções da família SHA), especialmente os ganhos que podem ser obtidos com a utilização das instruções MMX™ de 64 bits. Alguns comentários:

- O SHA-512 se beneficia fortemente do suporte a adição de 64 bits, visto que esta operação é parte importante daquela função. Também levanta a possibilidade de utilizar as instruções XMM™ (de 128 bits) para o processamento de dois fluxos de resumo em paralelo.

[4] 1 KiB = 2^{10} bytes

- A implementação de *Whirlpool*, é a mesma descrita em [42].

Embora fale do desempenho de funções criptográficas no Pentium®, em [52] Schneier não trata de funções de resumo.

Em [2] Aciicmez analisa a aplicabilidade de instruções SIMD (*Single-Instruction Multiple-Data*) à implementação das funções da família SHA, com foco no Intel® MMX™ mas não trata de *Whirlpool*.

Em [7] Bik et al. descrevem técnicas para exploração do paralelismo no Pentium®. Gerber ([18]), Fog ([17]), Bosselaers ([9] e [8]) e Nakajima et al. ([42]) também fizeram trabalhos que contém informações muito úteis para a otimização para o Pentium®, sendo que os três últimos são voltados para funções de resumo. Em [41], Mirceski também apresenta alguns resultados no Pentium®, embora o seu foco seja a proposta de novas funções de resumo que utilizam as extensões XMM™.

Capítulo 3

Otimização para o XScale®

Neste capítulo será apresentado o processador Intel® XScale® e suas principais características, com foco nas possibilidades de exploração visando a otimização dos algoritmos. Serão também mostradas as principais técnicas de otimização estudadas e sua aplicabilidade aos referidos algoritmos, com destaque para a utilização das extensões Wireless MMX™.

3.1 O processador Intel® XScale®

A tecnologia Intel® XScale® (veja `http://www.intel.com/design/intelxscale/`) foi projetada para processamento de alto desempenho e baixo consumo de energia, voltada para aplicações de rede sem-fio e serviços de alto valor. Tem sido bastante utilizada em dispositivos móveis (como PDAs).

3.1.1 Os processadores Intel® PCA

Os processadores Intel® PCA implementam o conjunto de instruções ARM versão 5TE ISA (excluindo as instruções de ponto flutuante). Além do núcleo ARM, a microarquitetura inclui também unidades de gerência de memória de dados e instruções, *caches* de instruções e dados, *buffers* de escrita, etc. O processador PXA270 implementa também as extensões Intel® Wireless MMX™.

Hoje a família compreende:

- Intel® PXA255.
- Intel® PXA270 ([28]).

Algumas das principais características da arquitetura Intel® PCA são ([47, p. 70-71]):

- *Pipeline* de sete a oito estágios.

- *Cache* de instruções de 32 KiB.
- *Cache* de dados de 32 KiB.
- 16 registradores de dados de 32 bits (*r0* a *r15*), sendo 13 deles (*r0* a *r12*) de uso geral.
- *Buffer* de predição de desvio com 128 entradas.
- Diversas funções de gerência de potência.
- Contadores para monitoração de desempenho.
- Suporte a depuração (incluindo pontos de parada em hardware).

3.1.2 O núcleo ARM

Algumas das características especiais da arquitetura ARM:

- Execução condicional ([3, seção 4] e [53, seção 6.5]).
- Instruções aritméticas podem atualizar o indicador de zero (*flag* Z) sem uma comparação explícita ([3, 5.2 e 6.1], [53, 5.3.1 e 5.3.2]).
- Deslocamento da entrada da unidade lógico-aritmética (*barrel shifter*) ([53, 3.1.2]).
- Execução de instruções em paralelo, explorando o *pipeline* ([53, 2.3 e 6.3], [27, A.2.2] e [26, 6.2.6]). É importante notar que embora o XScale® emita apenas uma instrução por ciclo, os três *pipelines* (“Multiply/Multiply Accumulate(MAC)”, memória e execução principal) podem executar instruções ao mesmo tempo e, se não houver dependência de dados, as instruções podem completar independentemente ([27, A.2.2]). A arquitetura do XScale®, no entanto, tem uma restrição muito maior do que pode ser executado em relaçao à arquitetura *superpipelined* do Pentium®, por exemplo.
- Carga de dados da memória em paralelo com execução de outras instruções (instrução de *preload*) ([27, A.4.4.11]) e utilização das instruções **`LDM`** e **`STM`**.
- Predição de desvio. Sloss et al. ([53, 2.3.1]) mostram informações genéricas para o ARM10 e a documentação da Intel ([27, 10.2]) descreve a implementação no XScale®.

3.1.3 A Extensão Intel® Wireless MMX™

Como explicam Paver et al. ([47, cap. 1]), os usuários de equipamentos móveis (como telefones celulares, organizadores pessoais e tocadores de música e vídeo) têm se tornado cada vez mais exigentes com relação à utilização de conteúdo multimídia, esperando encontrar nestes dispositivos uma experiência tão rica quanto a que está disponível nos seus equipamentos de mesa.

Neste contexto, torna-se muito importante que os processadores para este segmento (ao qual o Intel® XScale® se destina) ofereçam aos desenvolvedores de software opções de alto desempenho e baixo consumo de energia (que é um dos maiores limitantes, visto que a capacidade de fornecimento de energia das baterias ainda é relativamente reduzida).

As extensões Wireless MMX™ respondem a esta necessidade, facilitando a melhoria de desempenho de muitos tipos de aplicação, incluindo vídeo, processamento de imagem, síntese e compressão de voz, telefonia, etc.

Neste trabalho foi abordada apenas a versão Wireless MMX™ 1.0, conforme implementada no PXA270. A versão 2.0, que inclui instruções adicionais, foi anunciada pela Intel® mas não foi estudada.

Do ponto de vista da arquitetura ARM, a implementação das extensões Wireless MMX™ no processador PXA270 é tratada como um co-processador, utilizando os padrões daquela arquitetura para comunicação entre o núcleo e os co-processadores.

O Wireless MMX™ implementa o conceito de processamento "SIMD" (*Single Instruction Multiple Data*) ([47, cap. 2]). De forma simplificada, neste tipo de processamento o registrador é dividido logicamente em múltiplas partes, sobre as quais uma determinada instrução é executada de forma independente. Em um registrador de 64 bits, por exemplo, pode-se executar duas operações de adição de 32 bits (ou oito de 8 bits) simultaneamente. A definição lógica do tipo dos operandos permite que cada parte do registrador seja vista como independente - por exemplo uma adição de 8 bits é feita de forma independente em cada parte do registrador, de forma que o resultado de uma operação não afete o resultado da outra. As opções oferecidas incluem:

- 1 par de operandos de 64 bits.
- 2 pares de operandos de 32 bits.
- 4 pares de operandos de 16 bits.
- 8 pares de operandos de 8 bits.

(Nem todas as operações permitem todos os tamanhos de operando).

Para algoritmos que executam operações semelhantes para conjuntos de dados extensos, este paralelismo permite uma grande economia no número de instruções necessárias para completar uma tarefa e um grande aumento no desempenho.

Para o Wireless MMX™, a grande maioria das funções aritméticas e lógicas está disponível. Pode-se consultar Paver et al. ([47, cap. 3, p. 41]) para uma descrição geral ou a documentação da Intel® ([25, apêndice A]) para uma referência detalhada das instruções Wireless MMX™.

Algumas das principais instruções são:

- Operações aritméticas: **`WADD`**, **`WSWB`**. Uma limitação importante é a ausência de suporte a operações de 64 bits.

- Operações de deslocamento: **`WSLL`**, **`WSRA`**, **`WSRL`**, **`WROR`**. Embora não exista rotação à esquerda, essa pode ser implementada a partir da rotação à direita pelo complemento do número de bits desejado (módulo tamanho da palavra).

- Operações lógicas: **`WAND`**, **`WANDN`**, **`WOR`**, **`WXOR`**. As operações lógicas podem ser efetuadas apenas em 1 par de operandos de 64 bits (o que não é necessariamente uma limitação, visto que as operações são feitas bit a bit de qualquer maneira).

As operações são semelhantes (mas não idênticas) às da extensão MMX™ dos processadores Pentium®, por exemplo. Tanto Paver et al. ([47, cap. 11]) quanto a documentação da Intel® ([25, cap. 4]) tratam do porte de aplicações para plataformas baseadas no Intel® PCA, incluindo uma comparação entre os modelos de programação das tecnologias Wireless MMX™ e MMX™/SSE.

Paver et al. ([47, p. 71-81]) descrevem o hardware do co-processador (responsável pela execução das instruções Wireless MMX™), incluindo o funcionamento do *pipeline*.

O manual da Intel® dedicado às extensões Wireless MMX™ ([25]) descreve como essas podem ser aplicadas a muitas classes de algoritmos (processamento de áudio, processamento de imagem, gráficos, vídeo, etc.) incluindo exemplos de implementação e otimização.

3.2 O processo de otimização

As recomendações descritas nesta seção foram obtidas principalmente de Sloss et al. ([53]), Gerber ([18]), Paver et al. ([47]) e da documentação da Intel® ([25], [24], [26], [27, apêndice A]) e da ARM ([3]) .

3.2.1 O acesso à memória

A utilização adequada do *cache* é crítica para o desempenho dos programas ([26, seção 2.4.1]). No XScale®, tanto o *cache* de dados quanto o *cache* de memória têm 32 KiB ([26, seção 2.4.1]) e a linha do *cache* tem 32 bytes ([26, seção 2.4]).

Considerando isso, as recomendações ([26, seção 2.4.1.1.2] e [27, seção A.4.2.6]) são que as estruturas de dados devem ser alinhadas em fronteiras de 32 bytes e o tamanho das estruturas de dados deve ser um múltiplo do tamanho da linha de *cache* para melhor utilização dessa. (Se um dado não está alinhado, por exemplo, será necessária a carga de duas linhas no *cache* para que ele possa ser acessado). No caso do Wireless MMX™, os dados só podem ser acessados em fronteiras de 64 bits portanto existe um trabalho adicional a ser feito caso este alinhamento não seja respeitado (veja [47, p.32]).

Além disso, como a capacidade do *cache* é finita, é importante tentar manter os acessos à memória localizados, acessando endereços próximos da mesma região de memória no mesmo trecho de código (para aproveitar o que foi carregado na linha do *cache*) e fazendo todo o processamento de determinado trecho antes de passar para outro (evitando que uma região que acabou de ser descarregada do *cache* tenha que ser carregada novamente). Em alguns casos isso requer rearranjar as estruturas de dados (ou os algoritmos) usados para o processamento.

Os manuais da Intel® podem ser consultados para mais informações sobre o *cache* e do *preload* ([27, A.4]) e explicações mais detalhadas de como os dados são carregados da memória ([26, seção 2.5]). Em [47, cap. 9] Paver et al. trata da otimização para os subsistemas de memória no XScale®, incluindo Wireless MMX™.

Como os acessos à memória podem ser muito demorados (já que a velocidade da memória tende a ser relativamente baixa), é comum o processador ter que esperar pelos dados para executar as instruções. Neste cenário, a utilização do *preload* (ou *prefetch*) pode ser uma das melhores técnicas de otimização disponíveis. Em [53, seção 6.3.1.1] Sloss et al. tratam do *preload* no ARM e na documentação da Intel® ([26, seção 6.2.7.1]) pode ser encontrado um resumo sobre a utilização do *preload* no XScale®, enquanto em [47, p. 189 ss.] Paver et al. tratam especificamente do *preload*.

De acordo com Gerber ([18]) , estas situações devem ser consideradas:

- Carga de dados obrigatória (*cache compulsory loads*) — caso em que o programa precisa processar uma grande quantidade de dados, a única solução é diminuir a quantidade de memória que o programa usa.

- Capacidade (*cache capacity loads*) — quando o programa processa mais dados do que é possível armazenar no *cache* ao mesmo tempo, uma possível solução é operar em blocos.

- Conflitos na carga (*cache conflict loads*) — devido às características do endereçamento do *cache*, pode não ser possível carregar um dado no *cache* mesmo que haja espaço e neste caso é recomendável analisar alinhamento dos dados e eventualmente alterá-los.

O *preload* não pode ser usado nem muito tarde nem muito cedo ([26]). Se usado muito tarde, não é eficaz para tornar os dados disponíveis antes de eles serem necessários para a execução da

instrução e, se feito prematuramente, pode ser que outros dados tenham que ser carregados em seu lugar no *cache* e o efeito do *preload* seja apenas poluir o *cache*.

Em alguns casos pode ser necessário, por exemplo, alterar a estrutura de dados usada para facilitar o *preload* ([26, p.6-24]). Paver et al. dizem que a distância ótima do *preload* deve ser igual à latência da memória ([47, p. 189 e seguintes]) e analisam o *preload* em laços ([47, p. 192]), incluindo o cálculo da distância de *preload* e a taxa de *preload* em um laço. Segundo a documentação da Intel® ([27, A.4.4.5]), se o laço for previsível e pequeno, pode ser mais vantajoso desenrolar o laço do que tentar fazer *preload*.

O *preload* sempre deve ser usado para carregar dados múltiplos de 32 bytes, que é o tamanho da linha do *cache* ([26, 6.2.6]). Também deve ser tomado cuidado para não exceder a capacidade de transferência de dados do sistema tentando fazer *preload* de uma quantidade excessiva de dados ([27, A.4.4.6]).

O manual da Intel ([27, A.3.1.4]) recomenda a utilização de **`MOV`** e **`MVN`** (em vez de **`LDR`**) para carregar valores imediatos, uma vez que além de poder gerar uma falta de *cache* (*cache miss*), **`LDR`** pode poluir o *cache*. Através de uma combinação das instruções **`MOV`**, **`MVN`**, **`ORR`**, **`BIC`** e **`ADD`**, é possível carregar qualquer valor de 32 bits usando uma seqüência de no máximo quatro instruções.

3.2.2 As instruções

Para a escolha das instruções é muito importante considerar fatores como número de ciclos necessários para a execução, latência e os efeitos do *pipeline* ([18, p. 63]). Em [53, seção 6.3] e [27, seção A.5, A.5.2 e A.5.3] os autores tratam de escalonamento de instruções (*instruction scheduling*) para XScale®, sendo que em [25, seção 3.2 a 3.4], a documentação da Intel trata especificamente da latência das instruções Wireless MMX™. Em [47, cap. 9] Paver et al. tratam da otimização para os *pipelines*. Ainda de acordo com as informações do XScale® fornecidas pela Intel® ([27, seção 10.4] e [27, tabela 10-5]), as instruções usadas nas funções de resumo já são simples e, portanto, parece não haver muitas oportunidades de otimização aí.

Os objetivos gerais da otimização são ([18, p. 57]):

- Disponibilizar muitas instruções para execução — quando há várias execuções que podem ser executadas ao mesmo tempo, ou seja, uma não depende do resultado da anterior, um processador com mais de um *pipeline* ou com várias unidades de execução pode paralelizar o processamento.

- Boa mistura de instruções — instruções que usam diferentes unidades de processamento podem manter todas as unidades do processador ocupado a maior parte do tempo, o que não seria possível se as instruções forem independentes mas necessitarem da mesma unidade de processamento.

- Desvios previsíveis — desvios que são previstos incorretamente forçam o processador a esvaziar o *pipeline* e atrasam o processamento.

O uso de funções expandidas *inline* pode melhorar o desempenho, evitando os desvios mas se forem usadas excessivamente podem poluir o *cache* de instruções e degradar o desempenho ([26, 2.4.1.1.1 e 6.2.5]). Muitos compiladores suportam o *inlining* automático (veja, por exemplo, [26, 6.2.5.6 e 6.2.5.7]), que pode ser controlado por uma heurística do próprio compilador, pelo tamanho das funções ou através de diretivas (como `inline` ou `#pragma inline`).

O objetivo geral do desenrolamento dos laços é reduzir a dependência entre os dados e prover uma mistura de instruções melhor, criando assim mais oportunidades de paralelização dentro do *pipeline* ([18, p. 120]). O benefício depende do custo da operação de desvio em relação ao custo da operação que está sendo expandida. Em [3, seção 6.2], [53, seção 5.3.3] e [53, seção 6.3.1.2] os autores tratam dos benefícios específicos para a arquitetura ARM. (Uma análise para processadores diversos no contexto de funções de resumo é feita em [42] por Nakajima et al.). Além disso, como sugere Gerber em [18, p. 125-127], é conveniente remover desvios de dentro dos laços.

A utilização de instruções condicionais pode ser bastante útil para eliminar desvios e otimizar expressões complexas (veja [3, seção 4], [53, 6.5] e [27, A.3.1.2 e A.3.1.3] — [18, p. 82-83] cita um uso semelhante da instrução **`CMOV`** no Pentium®). Em [27, 10-2 e A.3.1.2] e [26, 6.2.4.4] A Intel® trata especificamente dos desvios.

Algumas idéias para otimização de desvios incluem:

- Executar a operação incondicionalmente pode ser útil se o desvio for aleatório e o trabalho executado não for muito grande (e não tiver efeitos colaterais indesejados, obviamente) ([18, p. 85]).

- Se o desvio for aleatório, colocar os casos mais comuns no início (de um `switch`, por exemplo) pode melhorar o desempenho ([18, p. 86]).

No caso dos algoritmos de resumo estudados, os desvios são previsíveis estaticamente. Deve haver erro na predição apenas no último teste do laço.

3.2.3 Explorando as extensões MMX™

Segundo Gerber ([18, p. 23]) e Paver et al. ([47, p.148]) as principais opções para explorar as extensões MMX, em ordem decrescente de facilidade de uso, são:

- Conversão automática em vetores (feita pelo compilador) — os autores citados acima dão muitas dicas de como escrever o código de forma que o compilador reconheça mais facilmente as oportunidades de paralelização existentes.

- Biblioteca de classes C++, disponível para o compilador da Intel®.
- Funções intrínsecas, que permitem o acesso às instruções MMX™ sem que seja necessário escrever o código em linguagem *assembly*. A documentação da Intel ([26, seção 4.4]) descreve o uso de funções intrínsecas (*intrinsics* para o XScale® em geral e em [25, apêndice C] Paver et al. trazem uma referência sobre as funções disponíveis para o Wireless MMX™.
- Linguagem *assembly*.

Para a utilização das extensões Wireless MMX™ neste trabalho, foi escolhida a opção das funções intrínsecas. À exceção de parte da geração das chaves (cópia), o compilador não encontrou oportunidades de explorar as extensões automaticamente e, embora com menos flexibilidade do que a linguagem *assembly*, as funções intrínsecas permitem acesso às instruções Wireless MMX™.

3.3 Oportunidades de otimização

As áreas que elegemos como prioritárias para explorar oportunidades de otimização incluem:

- Características e opções (especialmente de otimização) dos compiladores.
- Técnicas que podem ser implementadas usando C padrão (como desenrolamento de laço) e que não estão restritas a uma arquitetura. Obviamente, a arquitetura tem grande influência nos eventuais ganhos observados, o que cria uma certa sobreposição com o próximo item.
- Implementações que exploram características específicas da arquitetura ARM/XScale®, como instruções especiais e arquitetura do *pipeline* e podem utilizar ou não C padrão.
- Extensão Intel® Wireless MMX™, que embora pudesse ser considerada como um conjunto de instruções especiais (e ser então incluída no item anterior) será tratada separadamente devido à importância especial desta extensão para as funções de resumo.

As funções de resumo são do tipo *compute-bound* e praticamente todo o tempo de execução se concentra no laço de geração das chaves e laço principal (especialmente neste último).

3.3.1 Características e opções do compilador

Nesta seção mencionaremos algumas técnicas que envolvem a utilização de opções específicas do compilador (otimização para tempo de execução, por exemplo) e/ou características que facilitam a geração de código mais eficiente. A grande vantagem dessas técnicas é que elas em geral não implicam em alteração do código fonte, o que facilita não apenas a sua utilização mas também a experimentação com opções diversas.

3.3.1.1 O compilador GCC

A documentação da Intel® ([26, seção 4.5]) descreve as chaves de compilação recomendadas pela Intel® para o gcc/XScale®. As principais chaves utilizadas foram `-mcpu=xscale` e `-mtune=xscale` que configuram a geração de código para o XScale® e `-O3` e `-O2`, que controlam o nível de otimização do compilador.

3.3.1.2 O compilador Intel®

Os principais documentos da Intel que podem ser usados como referências para o compilador da Intel® são:

- Descrição das ferramentas de desenvolvimento de software da Intel® para o XScale®, explicando qual compilador é compatível com cada plataforma ([16]).
- Visão geral das opções disponíveis, inclusive para o XScale® ([23]).
- Guia de otimização ([31]), que é mais voltado para o Pentium® mas fala de algumas opções que são comuns a outros compiladores também.
- Manual do compilador ([30]), que traz os detalhes das chaves disponíveis.

Em [26] é descrita a utilização da chave `/Qxscale`, bem como a geração de uma listagem *assembly*.

Em [47, p. 159-178] Paver et al. descrevem o que deve ser feito para que o compilador possa fazer a vetorização automaticamente. Chamamos de vetorização a paralelização das operações utilizando vetores para fazer o processamento de múltiplos dados simultaneamente.

Observou-se que na implementação original o compilador Intel® conseguiu vetorizar apenas parte do laço de geração das chaves que copiava os valores. Os outros laços não foram vetorizados.

Além disso, ele também não emprega automaticamente as instruções Wireless MMX™ para tratar operações de 64 bits.

De acordo com a Intel® ([29]), no ambiente do Microsoft Windows CE, não existe garantia de que os dados estarão alinhados em fronteiras de 8 bytes e, portanto, o compilador não gera instruções para carregar valores de 64 bits. Em vez disso dois valores de 32 bits são carregados nos registradores do ARM e então transferidos para os registradores Wireless MMX™.

3.3.1.3 O compilador Microsoft

De acordo com a documentação da Intel® ([26, seção 4.3]) o Microsoft embedded Visual C++ 3.0 não tem suporte específico para o XScale®. A versão 4.0 tem suporte a otimização para o XScale®

e suporta as funções intrínsecas. Naquele manual ([26, seção 4.4.1.2]) também são mencionados alguns problemas com PocketPC e eVC++ 3.0 e 4.0.

3.3.2 C padrão

Técnicas que podem ser exploradas utilizando o C padrão:

- Desenrolamento de laço.
- Remover desvios de dentro dos laços ([18, p. 125-127]).
- Eliminação de desvio, executando a operação incondicionalmente, pode ser útil caso o trabalho seja pequeno e o desvio seja aleatório ([18]).
- Otimização do desvio quando o desvio é aleatório, colocando o caso mais comum no início (de um *switch*, por exemplo) ([18]).
- Reescrita de operações lógicas (alguns exemplos dados por Bosselaers em [10, 5] e McCurley em [39]).

3.3.3 Explorando características da arquitetura do núcleo ARM

Algumas características do núcleo que podem ser exploradas:

- Memória (mais detalhes na seção 3.2.1):
  - Carregamento prévio de dados de (ou para) a memória (*preload*).
  - Alinhamento correto dos dados (32 bytes) ([26, 2.4.1.2.1]).
  - Melhor utilização do *cache*, tentando manter o máximo possível das instruções e dados no *cache*. (Em [53, cap. 12] Sloss et al. trazem maiores informações sobre o *cache* no ARM).
- Instruções (mais detalhes na seção 3.2.2):
  - Predição de desvio.
  - Eliminação de desvio, utilizando as operações condicionadas ao estado do registrador. Uma limitação é que, de acordo com a documentação da Intel® ([26, seção 4.4.1.2]), não é possível acessar as instruções condicionais usando funções intrínsecas, o que cria uma dependência do compilador e/ou utilização de linguagem *assembly*.
  - Desenrolamento de laço.

- Outros:
  - Suporte a rotações/deslocamentos dos operandos.
  - Escolha de operações alternativas mais "baratas", ou seja, com menor número de ciclos, latência mais baixa, etc. ([18, p. 63]).

Dependendo da técnica, ela pode ser empregada tanto em linguagem de alto nível quanto através de instruções em *assembly* (ou funções intrínsecas) inseridas no código de alto nível.

3.3.4 Extensão Intel® Wireless MMX™

- Suporte do Wireless MMX™ a operações de 64 bits.
- Suporte do Wireless MMX™ a execução paralela com dois pares de operandos de 32 bits.

3.3.5 Armadilhas a evitar

Pontos a serem evitados na implementação:

- As operações do ARM são de 32 bits, portanto operações com 8 bits e 16 bits devem ser evitadas ([53, 5.2.1]).
- Se possível, evitar usar um número de variáveis maior do que o de registradores disponíveis, já que as operações são sempre efetuadas com registradores ([53, 5.4])

3.4 Considerações sobre a implementação da família SHA

3.4.1 Operações usadas na família SHA

As operações e estruturas mais utilizadas pelas funções de resumo da família SHA são:

- Operações lógicas: E, OU-inclusivo, NÃO.
- Operações de deslocamento e rotação: deslocamento à direita e à esquerda e rotação à direita e à esquerda.
- Operações aritméticas (aritmética modular): adição.
- Laços.

- Operações com 32 (SHA-1, SHA-224, SHA-256) e 64 (SHA-384 e SHA-512) bits.

Outras informações sobre as características ideais de um processador para dar suporte a funções de resumo podem ser encontradas no artigo de Bosselaers ([10]).

3.4.2 XScale® e a família SHA - 1

- Operações lógicas
  - **`AND`** (E), **`EOR`** (OU-exclusivo), **`ORR`** (OU-inclusivo).
  - **`BIC`** (E NÃO).
  - **`RSB`** (subsititui o NÃO, **`NEG`** está disponível apenas no conjunto de instruções Thumb [1]).
- Operações de deslocamento e rotação
  - Em geral disponíveis (é possível utilizar **`MOV`** com deslocamento).
  - Rotação à direita está disponível mas à esquerda não. Isso não é um problema visto que essa pode ser implementada a partir da rotação à direita pelo complemento do número de bits desejado (módulo tamanho da palavra).
  - Operações de rotação não podem ser acessadas diretamente da linguagem C. Alguns compiladores disponibilizam funções intrínsecas para acesso a essas operações e/ou reconhecem a construção típica para implementar a rotação e otimizam o código para utilizar a instrução, quando disponível na arquitetura alvo.
- Operações aritméticas (aritmética modular): adição - está disponível (**`ADD`**).
- O custo dos desvios é baixo e eles podem ser otimizados utilizando-se uma contagem para baixo (**`SUBS`** em vez de **`ADD`**/**`CMP`**) ([53, seção 6.6] e [27, A.3.1.1]).
- Operações com 32 bits estão disponíveis (tamanho nativo de manipulação de dados).
- Operações com 64 bits não estão disponíveis - algumas operações podem se beneficiar do Wireless MMX™.

Em [47, p. 82] Paver et al. explicam a latência das instruções Wireless MMX™. Todas as instruções de interesse (**`WADD`**, **`WROR`**, etc.) têm R na coluna X1, o que significa que a latência é um ciclo e elas não causam atraso por dependência de dados. O **`WADD`** tem X/R mas conforme

[1] Instruções com comprimento de 16 bits, que implementam um subconjunto das instruções do ARM.

explicado ([47, p. 83]), o X vale apenas para o caso de operação com saturação — e portanto não se aplica aqui. Maiores detalhes sobre a latência das instruções Wireless MMX™ podem ser encontrados na documentação da Intel® ([25, seção 3.2, 3.3 e 3.4]).

3.5 Considerações sobre a implementação do algoritmo *Whirlpool*

Em [4, seção 7] os autores do algoritmo sugerem uma implementação do *Whirlpool* (tanto em arquiteturas de 64 bits quanto de 32 bits) baseada em tabelas de busca. A análise da implementação do *Whirlpool* feita a seguir considera esta sugestão.

3.5.1 Operações usadas pelo algoritmo *Whirlpool*

A implementação de 64 bits do *Whirlpool* utiliza basicamente:

- Durante o cálculo das chaves e cálculo das iterações:
  - Deslocamentos e E para extrair bytes específicos, que serão então utilizados como índices em uma tabela de busca contendo valores de 64 bits.
  - Ou-exclusivo entre os valores de 64 bits, que irão gerar um valor de 64 bits.
- Após a finalização do processamento de cada bloco:
  - Ou-exclusivo entre os valores do bloco anterior e os valores calculados no bloco corrente.

A implementação de 32 bits utiliza essencialmente os mesmos tipos de instrução.

3.5.2 XScale® e o algoritmo *Whirlpool*

A análise da disponibilidade de operações feita em 3.4.2 também se aplica ao *Whirlpool*. É interessante observar que sendo operações naturalmente realizadas em 64 bits, poderiam beneficiar-se da utilização dos registradores Wireless MMX™.

A seguir são descritas as sugestões dos autores em [4, seções 7.1 e 7.2], que permitem a utilização das tabelas de busca na implementação da função ρ, que foi descrita em 2.4.3.

3.5.2.1 Implementação em processadores de 64 bits

Se definirmos matrizes de 64 bits $T_i(x)$, $0 \leqslant i \leqslant 7$:

$$T_0(x) = S[x].c_{00}|S[x].c_{01}|S[x].c_{02}|S[x].c_{03}|S[x].c_{04}|S[x].c_{05}|S[x].c_{06}|S[x].c_{07}$$
$$T_1(x) = S[x].c_{10}|S[x].c_{11}|S[x].c_{12}|S[x].c_{13}|S[x].c_{14}|S[x].c_{15}|S[x].c_{16}|S[x].c_{17}$$
$$T_2(x) = S[x].c_{20}|S[x].c_{21}|S[x].c_{22}|S[x].c_{23}|S[x].c_{24}|S[x].c_{25}|S[x].c_{26}|S[x].c_{27}$$
$$T_3(x) = S[x].c_{30}|S[x].c_{31}|S[x].c_{32}|S[x].c_{33}|S[x].c_{34}|S[x].c_{35}|S[x].c_{36}|S[x].c_{37}$$
$$T_4(x) = S[x].c_{40}|S[x].c_{41}|S[x].c_{42}|S[x].c_{43}|S[x].c_{44}|S[x].c_{45}|S[x].c_{46}|S[x].c_{47}$$
$$T_5(x) = S[x].c_{50}|S[x].c_{51}|S[x].c_{52}|S[x].c_{53}|S[x].c_{54}|S[x].c_{55}|S[x].c_{56}|S[x].c_{57}$$
$$T_6(x) = S[x].c_{60}|S[x].c_{61}|S[x].c_{62}|S[x].c_{63}|S[x].c_{64}|S[x].c_{65}|S[x].c_{66}|S[x].c_{67}$$
$$T_7(x) = S[x].c_{70}|S[x].c_{71}|S[x].c_{72}|S[x].c_{73}|S[x].c_{74}|S[x].c_{75}|S[x].c_{76}|S[x].c_{77}$$

e aplicarmos estas matrizes aos valores $a_{00}\ a_{71}\ a_{62}\ a_{53}\ a_{44}\ a_{35}\ a_{26}\ a_{17}$

$$T_0(a_{00}) = S[a_{00}].c_{00}|S[a_{00}].c_{01}|S[a_{00}].c_{02}|S[a_{00}].c_{03}|S[a_{00}].c_{04}|S[a_{00}].c_{05}|S[a_{00}].c_{06}|S[a_{00}].c_{07}$$
$$T_1(a_{71}) = S[a_{71}].c_{10}|S[a_{71}].c_{11}|S[a_{71}].c_{12}|S[a_{71}].c_{13}|S[a_{71}].c_{14}|S[a_{71}].c_{15}|S[a_{71}].c_{16}|S[a_{71}].c_{17}$$
$$T_2(a_{62}) = S[a_{62}].c_{20}|S[a_{62}].c_{21}|S[a_{62}].c_{22}|S[a_{62}].c_{23}|S[a_{62}].c_{24}|S[a_{62}].c_{25}|S[a_{62}].c_{26}|S[a_{62}].c_{27}$$
$$T_3(a_{53}) = S[a_{53}].c_{30}|S[a_{53}].c_{31}|S[a_{53}].c_{32}|S[a_{53}].c_{33}|S[a_{53}].c_{34}|S[a_{53}].c_{35}|S[a_{53}].c_{36}|S[a_{53}].c_{37}$$
$$T_4(a_{44}) = S[a_{44}].c_{40}|S[a_{44}].c_{41}|S[a_{44}].c_{42}|S[a_{44}].c_{43}|S[a_{44}].c_{44}|S[a_{44}].c_{45}|S[a_{44}].c_{46}|S[a_{44}].c_{47}$$
$$T_5(a_{35}) = S[a_{35}].c_{50}|S[a_{35}].c_{51}|S[a_{35}].c_{52}|S[a_{35}].c_{53}|S[a_{35}].c_{54}|S[a_{35}].c_{55}|S[a_{35}].c_{56}|S[a_{35}].c_{57}$$
$$T_6(a_{26}) = S[a_{26}].c_{60}|S[a_{26}].c_{61}|S[a_{26}].c_{62}|S[a_{26}].c_{63}|S[a_{26}].c_{64}|S[a_{26}].c_{65}|S[a_{26}].c_{66}|S[a_{26}].c_{67}$$
$$T_7(a_{17}) = S[a_{17}].c_{70}|S[a_{17}].c_{71}|S[a_{17}].c_{72}|S[a_{17}].c_{73}|S[a_{17}].c_{74}|S[a_{17}].c_{75}|S[a_{17}].c_{76}|S[a_{17}].c_{77}$$

(Considere que | separa os bytes)

veremos que as linhas b_i serão da forma:

$$b_{00}|b_{01}|b_{02}|b_{03}|b_{04}|b_{05}|b_{06}|b_{07} = T_0(a_{00}) \oplus T_1(a_{71}) \oplus T_2(a_{62}) \oplus T_3(a_{53}) \oplus T_4(a_{44}) \oplus T_5(a_{35}) \oplus T_6(a_{26}) \oplus T_7(a_{17})$$

$$b_{10}|b_{11}|b_{12}|b_{13}|b_{14}|b_{15}|b_{16}|b_{17} =$$

$$\ldots$$

Assim, a partir dos $T_i(x)$ definidos acima podemos calcular os valores finais de ρ apenas fazendo indexações na tabela e ou-exclusivo, em vez de multiplicações:

$$b_i = \bigoplus_{k=0}^{7} T_k[a_{(i-k) \bmod 8,k}]$$

Usando os valores definidos pelo algoritmo *Whirlpool* para a matriz circulante C, as tabelas $T_k[x] \equiv S[x] \,.\, C_k, 0 \leqslant k \leqslant 7$ são:

$$T_0[x] = S[x].[01_x\ 01_x\ 04_x\ 01_x\ 08_x\ 05_x\ 02_x\ 09_x],$$
$$T_1[x] = S[x].[09_x\ 01_x\ 01_x\ 04_x\ 01_x\ 08_x\ 05_x\ 02_x],$$
$$T_2[x] = S[x].[02_x\ 09_x\ 01_x\ 01_x\ 04_x\ 01_x\ 08_x\ 05_x],$$
$$T_3[x] = S[x].[05_x\ 02_x\ 09_x\ 01_x\ 01_x\ 04_x\ 01_x\ 08_x],$$
$$T_4[x] = S[x].[08_x\ 05_x\ 02_x\ 09_x\ 01_x\ 01_x\ 04_x\ 01_x],$$
$$T_5[x] = S[x].[01_x\ 08_x\ 05_x\ 02_x\ 09_x\ 01_x\ 01_x\ 04_x],$$
$$T_6[x] = S[x].[04_x\ 01_x\ 08_x\ 05_x\ 02_x\ 09_x\ 01_x\ 01_x],$$
$$T_7[x] = S[x].[01_x\ 04_x\ 01_x\ 08_x\ 05_x\ 02_x\ 09_x\ 01_x].$$

3.5.2.2 Implementação em processadores de 32 bits

Se considerarmos que a matriz circulante original:

$$C = \begin{bmatrix} 01_x & 01_x & 04_x & 01_x & 08_x & 05_x & 02_x & 09_x \\ 09_x & 01_x & 01_x & 04_x & 01_x & 08_x & 05_x & 02_x \\ 02_x & 09_x & 01_x & 01_x & 04_x & 01_x & 08_x & 05_x \\ 05_x & 02_x & 09_x & 01_x & 01_x & 04_x & 01_x & 08_x \\ 08_x & 05_x & 02_x & 09_x & 01_x & 01_x & 04_x & 01_x \\ 01_x & 08_x & 05_x & 02_x & 09_x & 01_x & 01_x & 04_x \\ 04_x & 01_x & 08_x & 05_x & 02_x & 09_x & 01_x & 01_x \\ 01_x & 04_x & 01_x & 08_x & 05_x & 02_x & 09_x & 01_x \end{bmatrix}$$

equivale a

$$C = \begin{bmatrix} U & V \\ V & U \end{bmatrix}$$

onde

$$U = \begin{bmatrix} 01_x & 01_x & 04_x & 01_x \\ 09_x & 01_x & 01_x & 04_x \\ 02_x & 09_x & 01_x & 01_x \\ 05_x & 02_x & 09_x & 01_x \end{bmatrix}$$

e

$$V = \begin{bmatrix} 08_x & 05_x & 02_x & 09_x \\ 01_x & 08_x & 05_x & 02_x \\ 04_x & 01_x & 08_x & 05_x \\ 01_x & 04_x & 01_x & 08_x \end{bmatrix}$$

e se procedermos de forma semelhante à explicada em 3.5.2.1 e definirmos matrizes de 32 bits $T_{iA}(x)$ e $T_{iB}(x)$, $0 \leqslant i \leqslant 7$:

$$T_{0A}(x) = S[x].c_{00}|S[x].c_{01}|S[x].c_{02}|S[x].c_{03} \qquad T_{0B}(x) = S[x].c_{04}|S[x].c_{05}|S[x].c_{06}|S[x].c_{07}$$
$$T_{1A}(x) = S[x].c_{10}|S[x].c_{11}|S[x].c_{12}|S[x].c_{13} \qquad T_{1B}(x) = S[x].c_{14}|S[x].c_{15}|S[x].c_{16}|S[x].c_{17}$$
$$T_{2A}(x) = S[x].c_{20}|S[x].c_{21}|S[x].c_{22}|S[x].c_{23} \qquad T_{2B}(x) = S[x].c_{24}|S[x].c_{25}|S[x].c_{26}|S[x].c_{27}$$
$$T_{3A}(x) = S[x].c_{30}|S[x].c_{31}|S[x].c_{32}|S[x].c_{33} \qquad T_{3B}(x) = S[x].c_{34}|S[x].c_{35}|S[x].c_{36}|S[x].c_{37}$$
$$T_{4A}(x) = S[x].c_{40}|S[x].c_{41}|S[x].c_{42}|S[x].c_{43} \qquad T_{4B}(x) = S[x].c_{44}|S[x].c_{45}|S[x].c_{46}|S[x].c_{47}$$
$$T_{5A}(x) = S[x].c_{50}|S[x].c_{51}|S[x].c_{52}|S[x].c_{53} \qquad T_{5B}(x) = S[x].c_{54}|S[x].c_{55}|S[x].c_{56}|S[x].c_{57}$$
$$T_{6A}(x) = S[x].c_{60}|S[x].c_{61}|S[x].c_{62}|S[x].c_{63} \qquad T_{6B}(x) = S[x].c_{64}|S[x].c_{65}|S[x].c_{66}|S[x].c_{67}$$
$$T_{7A}(x) = S[x].c_{70}|S[x].c_{71}|S[x].c_{72}|S[x].c_{73} \qquad T_{7B}(x) = S[x].c_{74}|S[x].c_{75}|S[x].c_{76}|S[x].c_{77}$$

e relembrarmos que

$$b_{00} = S[a_{00}].c_{00} + S[a_{71}].c_{10} + S[a_{62}].c_{20} + \cdots + S[a_{26}].c_{60} + S[a_{17}].c_{70}$$
$$b_{01} = S[a_{00}].c_{01} + S[a_{71}].c_{11} + S[a_{62}].c_{21} + \cdots + S[a_{26}].c_{61} + S[a_{17}].c_{71}$$
$$b_{02} = S[a_{00}].c_{02} + S[a_{71}].c_{12} + S[a_{62}].c_{22} + \cdots + S[a_{26}].c_{62} + S[a_{17}].c_{72}$$
$$b_{03} = S[a_{00}].c_{03} + S[a_{71}].c_{13} + S[a_{62}].c_{23} + \cdots + S[a_{26}].c_{63} + S[a_{17}].c_{73}$$
$$b_{04} = S[a_{00}].c_{04} + S[a_{71}].c_{14} + S[a_{62}].c_{24} + \cdots + S[a_{26}].c_{64} + S[a_{17}].c_{74}$$

$$b_{05} = S[a_{00}].c_{05} + S[a_{71}].c_{15} + S[a_{62}].c_{25} + \cdots + S[a_{26}].c_{65} + S[a_{17}].c_{75}$$
$$b_{06} = S[a_{00}].c_{06} + S[a_{71}].c_{16} + S[a_{62}].c_{26} + \cdots + S[a_{26}].c_{66} + S[a_{17}].c_{76}$$
$$b_{07} = S[a_{00}].c_{07} + S[a_{71}].c_{17} + S[a_{62}].c_{27} + \cdots + S[a_{26}].c_{67} + S[a_{17}].c_{77}$$
$$\cdots$$
$$b_{70} = S[a_{70}].c_{00} + S[a_{61}].c_{10} + S[a_{52}].c_{20} + \cdots + S[a_{16}].c_{60} + S[a_{07}].c_{70}$$
$$b_{71} = S[a_{70}].c_{01} + S[a_{61}].c_{11} + S[a_{52}].c_{21} + \cdots + S[a_{16}].c_{61} + S[a_{07}].c_{71}$$
$$b_{72} = S[a_{70}].c_{02} + S[a_{61}].c_{12} + S[a_{52}].c_{22} + \cdots + S[a_{16}].c_{62} + S[a_{07}].c_{72}$$
$$b_{73} = S[a_{70}].c_{03} + S[a_{61}].c_{13} + S[a_{52}].c_{23} + \cdots + S[a_{16}].c_{63} + S[a_{07}].c_{73}$$
$$b_{74} = S[a_{70}].c_{04} + S[a_{61}].c_{14} + S[a_{52}].c_{24} + \cdots + S[a_{16}].c_{64} + S[a_{07}].c_{74}$$
$$b_{75} = S[a_{70}].c_{05} + S[a_{61}].c_{15} + S[a_{52}].c_{25} + \cdots + S[a_{16}].c_{65} + S[a_{07}].c_{75}$$
$$b_{76} = S[a_{70}].c_{06} + S[a_{61}].c_{16} + S[a_{52}].c_{26} + \cdots + S[a_{16}].c_{66} + S[a_{07}].c_{76}$$
$$b_{77} = S[a_{70}].c_{07} + S[a_{61}].c_{17} + S[a_{52}].c_{27} + \cdots + S[a_{16}].c_{67} + S[a_{07}].c_{77}$$

então podemos observar que

$$b_{00}|b_{01}|b_{02}|b_{03} = T_{0A}(a_{00}) \oplus T_{1A}(a_{71}) \oplus T_{2A}(a_{62}) \oplus T_{3A}(a_{53}) \oplus T_{4A}(a_{44}) \oplus T_{5A}(a_{35}) \oplus T_{6A}(a_{26}) \oplus T_{7A}(a_{17})$$

e

$$b_{04}|b_{05}|b_{06}|b_{07} = T_{0B}(a_{00}) \oplus T_{1B}(a_{71}) \oplus T_{2B}(a_{62}) \oplus T_{3B}(a_{53}) \oplus T_{4B}(a_{44}) \oplus T_{5B}(a_{35}) \oplus T_{6B}(a_{26}) \oplus T_{7B}(a_{17}).$$

O mesmo raciocínio se aplica às outras linhas de b e o cálculo final de ρ pode ser feito utilizando as tabelas de busca definidas e operações de ou-exclusivo.

3.6 Otimização das funções de resumo

As áreas de foco para otimização das funções de resumo estudadas foram identificadas através de:

- análise estática do algoritmo;
- recomendações obtidas a partir de análises disponíveis na literatura.

Outra fonte de informações útil seria a coleta de dados de execução com o VTune™ Performance Analyzer mas devido a limitações de recursos ela não pôde ser realizada.

3.6.1 Visão geral

Na tabela 3.1 é mostrado um resumo das técnicas de otimização estudadas, sua aplicabilidade aos algoritmos da família SHA e *Whirlpool* e também são indicadas quais dessas técnicas foram empregadas neste trabalho.

Como os algoritmos SHA-224 e SHA-384 são os mesmos que SHA-256 e SHA-512, respectivamente, SHA-224 e SHA-384 não são mencionados na tabela 3.1.

Tabela 3.1: Aplicabilidade das técnicas de otimização às funções de resumo estudadas

Técnica	**Aplica-se em *Whirlpool***	**Aplica-se em SHA**	**Utilizada neste trabalho**	**Comentários**
Uso de instrução nativa para a operação de rotação	Sim	SHA-1, SHA-256, SHA-512	Sim	Alguns compiladores já reconhecem automaticamente a construção em linguagem de alto nível e utilizam a instrução adequada, não sendo necessária a utilização de linguagem *assembly*.
Uso de instrução nativa para a conversão de formato *little-endian* para *big-endian*	Não	SHA-1, SHA-256, SHA-512	Não	Poderia ser empregado na fase de pré-processamento, quando ocorre esta conversão.
Definição alternativa para $\mathrm{Ch}(x,y,z)$ e $\mathrm{Maj}(x,y,z)$	Não	SHA-1, SHA-256, SHA-512	Sim	
Utilização de tabelas de 32 bits	Sim	Não	Não	

Continua na próxima página

Tabela 3.1: Aplicabilidade das técnicas de otimização às funções de resumo estudadas (continuação)

Técnica	**Aplica-se em *Whirlpool***	**Aplica-se em SHA**	**Utilizada neste trabalho**	**Comentários**
Cálculo da chave (W) junto ao laço principal	Não	SHA-1, SHA-256, SHA-512	Não	O objetivo seria fazer o cálculo imediatamente antes de sua utilização no laço principal, de forma a evitar novo acesso à memória para obter o valor da chave. Para o SHA, no pior caso (SHA-512), as chaves são 80 palavras de 64 bits (640 bytes), que é significativamente menor do que o *cache* de dados do XScale® (32 KiB). Como não há acesso a uma quantidade de dados grande após o cálculo das chaves, a probabilidade de que as chaves ainda estejam disponíveis no *cache* é muito alta e, portanto, o custo de acesso deve ser relativamente baixo — ainda que, obviamente, maior do que se as chaves já estivessem disponíveis em um registrador. A possibilidade de conflitos de acesso ao cache (*cache conflict loads*) não foi analisada.
Minimização da cópia de registrador ("renomeando-os" a cada iteração)	Sim	Sim	SHA-1, SHA-256, SHA-512	

Continua na próxima página

Tabela 3.1: Aplicabilidade das técnicas de otimização às funções de resumo estudadas (continuação)

Técnica	**Aplica-se em *Whirlpool***	**Aplica-se em SHA**	**Utilizada neste trabalho**	**Comentários**
Manutenção dos dados em registradores durante o cálculo do laço principal para evitar acesso à memória	Sim	SHA-1, SHA-256, SHA-512	Sim (sempre que possível)	O XScale® tem 16 registradores de dados de 32 bits (*r0* a *r15*), sendo 13 deles (*r0* a *r12*) de uso geral. Exceto para o SHA-1, é impossível manter todas as variáveis de trabalho, e valores de resumo intermediários em registradores, especialmente se considerarmos que eles também são necessários para armazenar outros valores auxiliares durante os cálculos. Uma alternativa seria usar os registradores do Wireless MMX™ (16 registradores de 64 bits). No entanto, como foi dito acima, o *cache* certamente minimiza o impacto desta falta de registradores. SHA-1: 11 palavras de 32 bits) [5 variáveis de trabalho (*a* a *e*), 1 variável temporária (T), 5 valores de resumo intermediários ($H_0^{(i)}$ a $H_4^{(i)}$)]. SHA-256: 18 palavras de 32 bits [8 variáveis de trabalho (*a* a *h*), 2 variáveis temporárias (T_1, T_2), 8 valores de resumo intermediários ($H_0^{(i)}$ a $H_7^{(i)}$)]. SHA-512: 18 palavras de 64 bits ou 36 palavras de 32 bits [8 variáveis de trabalho (*a* a *h*), 2 variáveis temporárias (T_1, T_2), 8 valores de resumo intermediários ($H_0^{(i)}$ a $H_7^{(i)}$)].

Continua na próxima página

Tabela 3.1: Aplicabilidade das técnicas de otimização às funções de resumo estudadas (continuação)

Técnica	**Aplica-se em *Whirlpool***	**Aplica-se em SHA**	**Utilizada neste trabalho**	**Comentários**
Uso de valores imediatos para constantes (em vez de acesso à memória)	Não	SHA-1, SHA-256, SHA-512	Não controlado diretamente (a cargo do compilador)	
Uso de *preload* (gerado automaticamente pelo compilador e/ou ajustado/inserido manualmente)	Sim	SHA-1, SHA-256, SHA-512	Não controlado diretamente (a cargo do compilador)	Se o algoritmo faz o cálculo das chaves inicialmente, todos os dados da mensagem já serão trazidos à memória *cache* neste instante e, como a quantidade de dados acessada não é tão grande, provavelmente o ganho do *preload* não será significativo (pode até ser que haja uma degradação do desempenho devido à execução de uma instrução adicional). A possibilidade de conflitos de acesso ao cache (*cache conflict loads*) não foi analisada.
Escolha de instruções mais rápidas	Sim	SHA-1, SHA-256, SHA-512	Sim (instrução AND NOT utilizada)	As instruções utilizadas já são simples (tanto no núcleo ARM quanto no Wireless MMX™), têm latência de um ciclo, etc.
Laços — Desenrolamento	Sim	SHA-1, SHA-256, SHA-512	Sim	

Continua na próxima página

Tabela 3.1: Aplicabilidade das técnicas de otimização às funções de resumo estudadas (continuação)

Técnica	**Aplica-se em *Whirlpool***	**Aplica-se em SHA**	**Utilizada neste trabalho**	**Comentários**
Laços — Alterar contador para contar para zero	Sim	SHA-1, SHA-256, SHA-512	Não	A expectativa é que o ganho não seja muito grande.
Laços — Eliminar desvio através do uso de instruções condicionais	Sim	SHA-1, SHA-256, SHA-512	Não	Instruções condicionais estão disponíveis para o Wireless MMX™ também (portanto em princípio poderiam ser aplicadas mesmo se os registradores Wireless MMX™ estiverem sendo usados para operações de 64 bits. No entanto, os laços são relativamente grandes e como a taxa de erro na previsão de desvio deve ser muito baixa, o custo do desvio pode não ser significativo. O "acerto" na predição do desvio não significa que o desvio tem custo zero mas sim que não haverá penalidade devido ao esvaziamento do *pipeline* e uma instrução já pode ser executada no ciclo seguinte. De acordo com a documentação da Intel ([27, seção 10.4.2, tabela 10-3]), o *issue latency* neste caso é um.
Extensões Wireless MMX™— Vetorização automática	Sim	SHA-1, SHA-256	Sim	

Continua na próxima página

Tabela 3.1: Aplicabilidade das técnicas de otimização às funções de resumo estudadas (continuação)

Técnica	**Aplica-se em *Whirlpool***	**Aplica-se em SHA**	**Utilizada neste trabalho**	**Comentários**
Extensões Wireless MMX™— Paralelização manual	Não	SHA-1, SHA-256	Sim	
Extensões Wireless MMX™— Operações de 64 bits	Sim	SHA-512	Sim	
Alinhamento dos dados em fronteiras de 32 bytes	Sim	SHA-1, SHA-256, SHA-512	Não controlado diretamente (a cargo do compilador)	Deveria ser feito automaticamente pelo compilador, — se for necessário é possível usar instruções de `align`.
Utilização de operações com deslocamento *barrel shifter*	Sim	Não	Não controlado diretamente (a cargo do compilador)	

.

3.6.2 Utilização do Wireless MMX™ no SHA-1

Para o SHA-1, tanto na geração das chaves quanto no laço principal, o objetivo da utilização das extensões Wireless MMX™ é utilizar os registradores de 64 bits para processar os dados relativos a duas iterações em paralelo, reduzindo assim o número de instruções executadas pelo processador e melhorando o desempenho da função.

3.6.2.1 Geração das chaves

O cálculo das chaves é definido como:

$$W_t = \begin{cases} M_t^{(i)} & 0 \leqslant t \leqslant 15 \\ ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) & 16 \leqslant t \leqslant 79 \end{cases}$$

Se armazenarmos os valores de W_t (que são de 32 bits) em um vetor auxiliar $W64$ com palavras de 64 bits, onde:

$$W64_i = W_{(i*2)+1} | W_{(i*2)}$$

teremos:

$$\begin{aligned} W64_0 &= W_1 | W_0 \\ W64_1 &= W_3 | W_2 \\ &\cdots \\ W64_{39} &= W_{79} | W_{78} \end{aligned}$$

Para o cálculo da chave W_t, precisamos dos valores de W_{t-3}, W_{t-8}, W_{t-14} e W_{t-16}. Consideremos $i = t*2$ e analisemos o cálculo da iteração i:

- $W_{(t+1)-8}$ e W_{t-8} estão disponíveis em $W64_{i-4}$.
- $W_{(t+1)-14}$ e W_{t-14} estão disponíveis em $W64_{i-7}$.
- $W_{(t+1)-16}$ e W_{t-16} estão disponíveis em $W64_{i-8}$.

- Como 3 não é múltiplo de 2, para o cálculo de $W_{(t+1)-3}$ e W_{t-3}, precisamos utilizar os 32 bits menos significativos de $W64_{i-2}$ e os 32 bits mais significativos de $W64_{i-1}$ no cálculo - o que nos força a fazer algumas operações a mais.

Assumindo que `wR0`, `wR1`, ..., `wRn` são os registradores Wireless MMX™ de 64 bits, uma implementação direta poderia ser a seguinte:

```
/* começamos em 8 visto que para 0 ⩽ t ⩽ 15 */
/* as chaves são apenas cópias da mensagem */
for (i = 8; i ⩽ 39; i++)
{
  /* t = i*2 */

  wR0 ← 32 bits  menos significativos de W64[i - 2] |
      32 bits  mais significativos de W64[i - 1]
  wR1 ← W64[i - 4] /* wR1 armazena W(t+1)-8 e Wt-8 */
  wR2 ← W64[i - 7] /* wR2 armazena W(t+1)-14 e Wt-14 */
  wR3 ← W64[i - 8] /* wR3 armazena W(t+1)-16 e Wt-16 */
  wR0 ← wR0 ⊕ wR1
  wR0 ← wR0 ⊕ wR2
  wR0 ← wR0 ⊕ wR3
  /* como não existe rotação à esquerda, trocamos o ROTL1 por ROTR31 */
  wR0 ← wR0 rotacionado à direita 31 bits
  W64[i] ← wR0
}
```

Como o XScale® dispõe de 16 registradores de 64 bits (`wR0` ... `wR15`), podemos utilizar mais registradores e criar uma "janela deslizante" para evitar a cópia de novos valores **da** memória a cada passo (os valores ainda têm que ser copiados **para** a memória, a não ser que seja feita uma otimização para calcular os valores de W somente quando eles são necessários para o algoritmo de processamento do bloco).

O trecho abaixo mostra como poderia ser uma implementação usando esta idéia:

```
wR0 ← W64[0] /* wR0 armazena W1 e W0 */
...
wR7 ← W64[7] /* wR7 armazena W15 e W14 */
/* wR8 é usado como temporário */
for (i = 8; i ⩽ 39; i++)
{
  /* t = i*2 */
  /* wR0 já contém o valor W(t+1)-16 e Wt-16 */
  wR0 ← wR0 ⊕ wR1 /* wR1 contém W(t+1)-14 e Wt-14 */
  wR0 ← wR0 ⊕ wR4 /* wR4 contém W(t+1)-8 e Wt-8 */
```

```
  wR8 ← 32 bits  menos significativos de wR7 |
      32 bits  mais significativos de wR6
  wR0 ← wR0 ⊕ wR8 /* wR8 contém W_(t+1)-3 e W_t-3 */
  /* como não existe rotação à esquerda, trocamos o ROTL^1 por ROTR^31 */
  wR0 ← wR0 rotacionado à direita 31 bits /* wR8 agora contém o valor final */
  W64[i] ← wR0
  /* prepara a próxima iteração */
  wR8 ← wR0 /* salva o valor de wR0 */
  wR0 ← wR1
...
  wR7 ← wR8
}
```

As cópias no final do laço podem ser evitadas se o laço for desenrolado, de forma que a cada iteração em vez de copiar o valor, apenas o significado do registrador seja alterado. Este é o código mostrado abaixo, já utilizando as funções intrínsecas apropriadas:

```
  /* w64 armazena as chaves em variáveis de 64 bits  */
  WORD64 w64[SHA_1_N_STEPS/2];
  /* __m64 indica ao compilador que as variáveis devem ser alocadas */
  /* em registradores Wireless MMX se possível                      */
  WORD64 m64[SHA_1_WORDS_PER_BLOCK/2];
  __m64 wR0, wR1, wR2, wR3, wR4, wR5, wR6, wR7, wR8;

...
        for (i = 0; i <= 7; i++)
          {
            int j = i * 2;

            w64[i] = _mm_cvtm64_si64(_mm_setr_pi32(m[j],m[j+1]));
          }
        wR0 = _mm_cvtsi64_m64(w64[0]);
...
        wR7 = _mm_cvtsi64_m64(w64[7]);
#define SHA1_KEY_SCHEDULE_ROUND(wR0, wR1, wR2, wR3, wR4, wR5, wR6, wR7, wR8, i) \
            wR0 = _mm_xor_si64(wR0,wR1);\
            wR0 = _mm_xor_si64(wR0,wR4);\
            wR8 = _mm_align_si64(wR6,wR7,4);\
            wR0 = _mm_xor_si64(wR0,wR8);\

            wR0 = _mm_rori_pi32(wR0, 31);\
            w64[i] = _mm_cvtm64_si64(wR0);\
            i++
```

```
        for (i = 8; i <= 39;)
          {
            int t;
            SHA1_KEY_SCHEDULE_ROUND(wR0, wR1, wR2, wR3, wR4, wR5, wR6, wR7, wR8, i);
            SHA1_KEY_SCHEDULE_ROUND(wR1, wR2, wR3, wR4, wR5, wR6, wR7, wR0, wR8, i);
...
            SHA1_KEY_SCHEDULE_ROUND(wR7, wR0, wR1, wR2, wR3, wR4, wR5, wR6, wR8, i);

          }
```

Algumas observações:

- Não existe uma função intrínseca (disponível em linguagem C) para **WLDR** e o compilador não a utiliza automaticamente já que por padrão os dados não estão necessariamente alinhados adequadamente (em uma fronteira de 64 bits).
- A rotação dos registradores (alterando o seu significado a cada iteração) faz o equivalente $wR8 = wR0$, $wR0 = wR1$, ..., $wR7 = wR8$.

Veja também a análise feita por Aciicmez em [2, seção V.B.1].

3.6.2.2 Laço principal

O laço principal é definido como:

$$T = \text{ROTL}^5(a) + f_t(b,c,d) + e + K_t + W_t$$
$$e = d$$
$$d = c$$
$$c = \text{ROTL}^{30}(b)$$
$$b = a$$
$$a = T$$

As definições alternativas (descritas em 2.5.3.1) também podem ser usadas.

O objetivo da implementação é utilizar os registradores de 64 bits do Wireless MMX™ para efetuar o cálculo de duas iterações em paralelo. Veja também a análise feita por Aciicmez em [2, seção V.B.1]).

Considerando que:

- a depende do resultado da iteração anterior,

- b é igual ao valor de a da iteração anterior,
- c é igual ao valor de b da iteração anterior, deslocado de 30 bits à esquerda,
- d é igual ao valor de c da iteração anterior,
- e é igual ao valor de d da iteração anterior.

podemos observar que:

- Como a tem uma dependência em relação à iteração anterior, o cálculo de $\text{ROTL}^5(a)$ não pode ser feito em paralelo.
- O cálculo de $\text{ROTL}^{30}(b)$ pode ser feito em paralelo.
- O cálculo de $f_1(b,c,d)$ pode ser feito em paralelo, desde que seja usado um artifício para o cálculo de c, visto que ele depende de uma rotação do valor de b.
- $e + K_t + W_t$ pode ser calculado em paralelo sem dificuldades.

Faremos então o processamento das iterações t e $t+1$ em paralelo, usando como contador auxiliar a variável $i = t*2$ e as variáveis de 64 bits $b64$, $c64$, $d64$ e $e64$ (como o valor de a depende do resultado da iteração anterior, não é utilizada uma variável $a64$):

$$b64_i = b_{(i*2)+1} | b_{i*2}$$
$$c64_i = c_{(i*2)+1} | c_{i*2}$$
$$d64_i = d_{(i*2)+1} | d_{i*2}$$
$$e64_i = e_{(i*2)+1} | e_{i*2}$$

O trecho de código a seguir, escrito como parte deste trabalho, mostra a implementação do laço, usando as funções intrínsecas do Wireless MMX™ (as funções `CH_WMMX`, etc. são descritas na seção 3.6.2.3):

```
WORD32 ah0, th0, th1, x;
/* w64 armazena a chave calculada anteriormente */
WORD64 w64[SHA_1_N_STEPS/2];


__m64 a64, b64, c64, d64, e64;
__m64 tmpm64;
```

```
...

        /* a, b, c, d e e já continham o valor do resumo */
        /* após o processamento do bloco anterior        */
        b64 = _mm_setr_pi32(b           , a            );
        c64 = _mm_setr_pi32(c           , ROTRC32(b,2) );
        d64 = _mm_setr_pi32(d           , c            );
        e64 = _mm_setr_pi32(e           , d            );

        ah0 = a;

#define SHA1_ROUND_FCT_1(b64,c64,d64) CH_WMMX(tmpm64,b64,c64,d64)
#define SHA1_ROUND_FCT_2(b64,c64,d64) PARITY_WMMX(tmpm64,b64,c64,d64)
#define SHA1_ROUND_FCT_3(b64,c64,d64) MAJ_WMMX(tmpm64,b64,c64,d64)

#define SHA1_ROUND_FCT_4(b64,c64,d64) SHA1_ROUND_FCT_2(b64,c64,d64)


#define SHA1_ROUND(a64, b64, c64, d64, e64, k, i) \
            tmpm64 = _mm_add_pi32(tmpm64, e64);\
            tmpm64 = _mm_add_pi32(tmpm64, k);\
            tmpm64 = _mm_add_pi32(tmpm64, _mm_cvtsi64_m64(w64[i]));\
            th0 = _mm_extract_pi32(tmpm64,0);\
            th1 = _mm_extract_pi32(tmpm64,1);\
            x = ROTRC32(ah0,27);\
            th0 += x;\
            x = ROTRC32(th0,27);\
            th1 += x;\

            /* prepara as variáveis b64, c64, d64 e e64 com os */
            /* valores apropriados para a próxima iteração     */
            e64 = c64;\
            x = ROTRC32(_mm_extract_pi32(b64,1),2);\
            d64 = _mm_setr_pi32(_mm_extract_pi32(c64,1), x);\
            c64 = _mm_setr_pi32(x, ROTRC32(th0,2));\
            b64 = _mm_setr_pi32(th0, th1);\
            ah0 = th1;

#define SHA1_ROUND_1(a64, b64, c64, d64, e64, i) \
    SHA1_ROUND_FCT_1(b64,c64,d64);SHA1_ROUND(a64,b64,c64,d64,e64,SHA_1_K_1_W64,i)
#define SHA1_ROUND_2(a64, b64, c64, d64, e64, i) \
    SHA1_ROUND_FCT_2(b64,c64,d64);SHA1_ROUND(a64,b64,c64,d64,e64,SHA_1_K_2_W64,i)
#define SHA1_ROUND_3(a64, b64, c64, d64, e64, i) \
```

```
    SHA1_ROUND_FCT_3(b64,c64,d64);SHA1_ROUND(a64,b64,c64,d64,e64,SHA_1_K_3_W64,i)
#define SHA1_ROUND_4(a64, b64, c64, d64, e64, i) \
    SHA1_ROUND_FCT_4(b64,c64,d64);SHA1_ROUND(a64,b64,c64,d64,e64,SHA_1_K_4_W64,i)

        for (i = 0; i <= 9; i++)
          {
            SHA1_ROUND_1(a64, b64, c64, d64, e64, i);
          }
...
        for (i = 30; i <= 39; i++)
          {
            SHA1_ROUND_4(a64, b64, c64, d64, e64, i);
          }

        a = ah0;
        b = _mm_extract_pi32(b64, 0);
        c = _mm_extract_pi32(c64, 0);
        d = _mm_extract_pi32(d64, 0);
        e = _mm_extract_pi32(e64, 0);
...
```

No início do laço, o compilador poderia usar a instrução **TMCRR** para transferir o conteúdo das variáveis *a*, *b*, *c*, *d* e *e* (que provavelmente estariam nos registradores de 32 bits) para os registradores de 64 bits (Wireless MMX™)

A cópia de valores entre os registradores a cada iteração pode ser eliminada observando-se a rotação entre os valores (o que é $a64$ na iteração i será $b64$ na iteração $i+1$, etc.). Esta implementação não é mostrada aqui.

3.6.2.3 Definição das funções

As funções utilizadas no processamento do SHA-1 são definidas como:

$$\begin{aligned} \mathrm{Ch}(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z) \\ \mathrm{Parity}(x,y,z) &= (x \oplus y \oplus z) \\ \mathrm{Maj}(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \end{aligned}$$

O cálculo das funções em si não depende de nenhuma outra variável e pode ser feito para duas iterações em paralelo desde que os valores apropriados das variáveis correspondentes às duas iterações estejam armazenados em registradores de 64 bits.

Este é o código que implementa estas funções, utilizando as funções intrínsecas do Wireless MMX™:

```
        /* podemos usar a instrução AND NOT nativa do processador */
#define CH_WMMX(_mm_result64,_mm_x64,_mm_y64,_mm_z64) \

        do {\
            __m64 tmp64;\
            _mm_result64 = _mm_andnot_si64(_mm_x64,_mm_z64);\
            tmp64 = _mm_and_si64(_mm_x64,_mm_y64);\
            _mm_result64 = _mm_xor_si64(_mm_result64,tmp64);\
        } while (0)

#define MAJ_WMMX(_mm_result64,_mm_x64,_mm_y64,_mm_z64) \

        do {\
            __m64 tmp64;\
            _mm_result64 = _mm_or_si64(_mm_x64,_mm_y64);\
            _mm_result64 = _mm_and_si64(_mm_result64,_mm_z64);\
            tmp64 = _mm_and_si64(_mm_x64,_mm_y64);\
            _mm_result64 = _mm_or_si64(tmp64,_mm_result64);\
        } while (0)

#define PARITY_WMMX(_mm_result64,_mm_x64,_mm_y64,_mm_z64) \

        do {\
            _mm_result64 = _mm_xor_si64(_mm_x64,_mm_y64);\
            _mm_result64 = _mm_xor_si64(_mm_result64,_mm_z64);\
        } while (0)
```

3.6.3 Utilização do Wireless MMX™ no SHA-256

Como a estrutura do SHA-256 é semelhante à do SHA-1 e também utiliza palavras de 32 bits, uma abordagem semelhante à adotada para o SHA-1 poderia ser usada para o SHA-256. Embora semelhantes, os algoritmos SHA-1 e SHA-256 têm algumas diferenças importantes, como complexidade de cálculo (as funções σ utilizadas no SHA-256, por exemplo, são mais complexas do que as rotações simples utilizadas no SHA-1) e número de variáveis (que tem implicações nas dependências entre duas iterações do laço principal). Devido a limitações de tempo, este caminho não foi explorado.

3.6.4 Utilização do Wireless MMX™ no SHA-512

Tanto para a geração das chaves quanto no laço principal, o algoritmo do SHA-512 utiliza palavras de 64 bits. Como os registradores Wireless MMX™ do XScale® têm 64 bits, a melhor utilização da extensão Wireless MMX™ é a realização de operações de 64 bits e não há oportunidade para explorar a paralelização de operações, como no caso do SHA-1 e SHA-256. A principal limitação é que a instrução de adição de 64 bits não está disponível e estas operações têm que ser implementadas usando instruções de 32 bits, o que impacta negativamente o desempenho. Nakajima et al. (em [42]) e Matsui et al. (em [38, seção 7]) analisam a limitação semelhante no Pentium® e suas extensões MMX™. Em [2, seção V.B.3] Aciicmez também analisa a implementação do SHA-512 usando MMX™.

3.6.4.1 Funções gerais

A implementação das funções $\sigma^{\{512\}}$ e $\Sigma^{\{512\}}$ foi feita da seguinte forma:

```
#define UPPERC_SIGMA_512_WMMX(_mm_result64,_mm_x64,rot1,rot2,rot3) \
        do {\
            __m64 tmp64;\
            _mm_result64 = _mm_rori_si64(_mm_x64,rot1);\
            tmp64 = _mm_rori_si64(_mm_x64,rot2);\
            _mm_result64 = _mm_xor_si64(_mm_result64,tmp64);\
            tmp64 = _mm_rori_si64(_mm_x64,rot3);\
            _mm_result64 = _mm_xor_si64(_mm_result64,tmp64);\
        } while(0)

#define LOWERC_SIGMA_512_WMMX(_mm_result64,_mm_x64,rot1,rot2,shr3) \
        do {\
            __m64 tmp64;\
            _mm_result64 = _mm_rori_si64(_mm_x64,rot1);\
            tmp64 = _mm_rori_si64(_mm_x64,rot2);\
            _mm_result64 = _mm_xor_si64(_mm_result64,tmp64);\
            tmp64 = _mm_srli_si64(_mm_x64,shr3);\
            _mm_result64 = _mm_xor_si64(_mm_result64,tmp64);\
        } while(0)

#define UPPERC_SIGMA_512_0_WMMX(_mm_result64,_mm_x64) \
    UPPERC_SIGMA_512_WMMX(_mm_result64,_mm_x64,28,34,39)
#define UPPERC_SIGMA_512_1_WMMX(_mm_result64,_mm_x64) \
    UPPERC_SIGMA_512_WMMX(_mm_result64,_mm_x64,14,18,41)
#define LOWERC_SIGMA_512_0_WMMX(_mm_result64,_mm_x64) \
    LOWERC_SIGMA_512_WMMX(_mm_result64,_mm_x64, 1, 8, 7)
#define LOWERC_SIGMA_512_1_WMMX(_mm_result64,_mm_x64) \
```

```
LOWERC_SIGMA_512_WMMX(_mm_result64,_mm_x64,19,61, 6)
```

3.6.4.2 Geração das chaves

O cálculo das chaves é definido como:

$$W_t = \begin{cases} M_t^{(i)} & 0 \leqslant t \leqslant 15 \\ \sigma_1^{\{512\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{512\}}(W_{t-15}) + W_{t-16} & 16 \leqslant t \leqslant 79 \end{cases}$$

onde:

$$\begin{aligned} \mathrm{Ch}(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z) \\ \mathrm{Maj}(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\ \Sigma_0^{\{512\}}(x) &= \mathrm{ROTR}^{28}(x) \oplus \mathrm{ROTR}^{34}(x) \oplus \mathrm{ROTR}^{39}(x) \\ \Sigma_1^{\{512\}}(x) &= \mathrm{ROTR}^{14}(x) \oplus \mathrm{ROTR}^{18}(x) \oplus \mathrm{ROTR}^{41}(x) \\ \sigma_0^{\{512\}}(x) &= \mathrm{ROTR}^{1}(x) \oplus \mathrm{ROTR}^{8}(x) \oplus \mathrm{SHR}^{7}(x) \\ \sigma_1^{\{512\}}(x) &= \mathrm{ROTR}^{19}(x) \oplus \mathrm{ROTR}^{61}(x) \oplus \mathrm{SHR}^{6}(x) \end{aligned}$$

A implementação é bastante direta. O trecho de código abaixo mostra parte da geração das chaves:

```
  /* w armazena as chaves */
  WORD64 w[SHA_512_N_STEPS];

  __m64 tmpm64, tmpx64;
...

  for (/* a partir do valor anterior de i */; i < SHA_512_N_STEPS; i++)
    {
      w[i] = w[i-7] + w[i-16];
      tmpx64 = _mm_cvtsi64_m64(w[i-2]);
      LOWERC_SIGMA_512_1_WMMX(tmpm64,tmpx64);
      w[i] += _mm_cvtm64_si64(tmpm64);
      tmpx64 = _mm_cvtsi64_m64(w[i-15]);
      LOWERC_SIGMA_512_0_WMMX(tmpm64,tmpx64);
      w[i] += _mm_cvtm64_si64(tmpm64);
    }
```

3.6.4.3 Laço principal

O laço principal é definido como:

$$
\begin{aligned}
T_1 &= h + \Sigma_1^{512}(e) + \mathrm{Ch}(e,f,g) + K_t^{512} + W_t \\
T_2 &= \Sigma_0^{512}(a) + \mathrm{Maj}(a,b,c) \\
h &= g \\
g &= f \\
f &= e \\
e &= d + T_1 \\
c &= b \\
b &= a \\
a &= T_1 + T_2
\end{aligned}
$$

O código é bastante direto, sendo necessário salientar apenas que as variáveis $T1$ e $T2$ não são armazenadas em registradores Wireless MMX™ porque como não existe um **`ADD`** de 64 bits, elas teriam que ser convertidas para 32 bits de qualquer maneira.

O corpo do laço principal da função foi implementado desta forma:

```
#define DO_ROUND_SHA512() \
        T1 = _mm_cvtm64_si64(h);\
        UPPERC_SIGMA_512_1_WMMX(tmpm64,e);\
        T1 += _mm_cvtm64_si64(tmpm64);\
        CH_WMMX(tmpm64,e, f, g);\
        T1 += _mm_cvtm64_si64(tmpm64);\
        T1 += SHA_512_K[i];\
        T1 += w[i];\
        UPPERC_SIGMA_512_0_WMMX(tmpm64,a);\
        T2 = _mm_cvtm64_si64(tmpm64);\
        MAJ_WMMX(tmpm64,a, b, c);\
        T2 += _mm_cvtm64_si64(tmpm64);\
        h = g;\
        g = f;\
        f = e;\
        e = _mm_cvtsi64_m64(_mm_cvtm64_si64(d) + T1);\

        d = c;\
```

```
        c = b;\
        b = a;\
        a = _mm_cvtsi64_m64(T1 + T2);\
        i++
```

Com o desenrolamento, o corpo e o laço podem ser reescritos da seguinte maneira:

```
#define DO_ROUND_SHA512(a,b,c,d,e,f,g,h,i) \
        T1 = _mm_cvtm64_si64(h);\
        UPPERC_SIGMA_512_1_WMMX(tmpm64,e);\
        T1 += _mm_cvtm64_si64(tmpm64);\
        CH_WMMX(tmpm64,e, f, g);\
        T1 += _mm_cvtm64_si64(tmpm64);\
        T1 += SHA_512_K[i];\
        T1 += w[i];\
        UPPERC_SIGMA_512_0_WMMX(tmpm64,a);\
        T2 = _mm_cvtm64_si64(tmpm64);\
        MAJ_WMMX(tmpm64,a, b, c);\
        T2 += _mm_cvtm64_si64(tmpm64);\
        d = _mm_cvtsi64_m64(_mm_cvtm64_si64(d) + T1);\
        h = _mm_cvtsi64_m64(T1 + T2);\
        i++
...
        for (; i < SHA_512_N_STEPS; )
          {
            DO_ROUND_SHA512(a,b,c,d,e,f,g,h,i);
            DO_ROUND_SHA512(h,a,b,c,d,e,f,g,i);
...
            DO_ROUND_SHA512(b,c,d,e,f,g,h,a,i);
          }
```

É interessante observar a limitação devida à ausência da instrução de adição de 64 bits nas extensões Wireless MMX™. As operações de adição envolvendo $T1$ e $T2$ são implementadas pelo compilador usando operações de 32 bits.

3.6.5 Utilização de operações com 32 bits no *Whirlpool*

Para arquiteturas de 32 bits, Barreto e Rijmen ([4, 7.2]) sugerem a utilização de tabelas de busca desse tamanho. As operações utilizadas seriam essencialmente as mesmas que as de uma implementação de 64 bits, que estão disponíveis no núcleo ARM.

3.6.6 Utilização do Wireless MMX™ no *Whirlpool*

Assim como ocorre com o algoritmo SHA-512 (veja seção 3.6.4), o algoritmo do *Whirlpool* utiliza palavras de 64 bits. Como os registradores Wireless MMX™ do XScale® têm 64 bits, a melhor utilização da extensão Wireless MMX™ neste caso também seria para a realização de operações de 64 bits.

Capítulo 4

Implementações e análise de resultados

Neste capítulo falaremos sobre a implementação feita para os algoritmos da família SHA, dos testes realizados com esta implementação e com a implementação do *Whirlpool*. Serão também apresentados os resultados obtidos a partir dos testes realizados no Pentium®, PXA255 e PXA270.

4.1 Implementações

Para a família SHA, foi feita uma nova implementação em linguagem C padrão (exceto pela utilização de funções intrínsecas para explorar as extensões Wireless MMX™). Embora existissem diversas implementações disponíveis, uma nova implementação a partir da especificação permitiu uma melhor compreensão das dificuldades envolvidas e das possibilidades de otimização. Os `#defines` foram extensamente usados para permitir o controle das opções utilizadas e facilitar os testes dessas variações.

Devido a limitações de tempo, não foi feita uma nova implementação da função *Whirlpool*. A versão disponibilizada pelos autores do algoritmo ([6]) foi utilizada para os testes.

A implementação inicial foi desenvolvida em um ambiente Windows XP/Cygwin e o código foi posteriormente portado para o Linux e para o Windows Mobile. Apesar da preocupação inicial com a portabilidade, a mudança do ambiente Cygwin/Linux para o Windows revelou muitas diferenças entre os compiladores, especialmente na definição de tipos de tamanho fixo.

.

4.2 Ambiente de desenvolvimento

Foram utilizados software e equipamentos do laboratório Intel na Unicamp para o desenvolvimento deste trabalho.

4.2.1 Ambiente de desenvolvimento Windows/Cygwin

No Windows XP/Cygwin 1.5.18, foi utilizado o compilador GNU gcc versão 3.4.4.

4.2.2 Ambiente de desenvolvimento Linux

No Linux (Fedora) foi utilizado o compilador GNU gcc versão 4.0.1 para geração de código para o próprio Linux. A geração de código para o PXA255 foi feita utilizando o gcc arm toolchain versão 3.4.2.

4.2.3 Ambiente de desenvolvimento Windows/Windows Mobile

Para a geração de código para o Windows Mobile foi utilizado o Windows Professional 2000 e o o compilador Intel® *Intel® C++ Compiler For Microsoft eMbedded Visual C++* versão *2.0.3 For Windows CE, Standard*.

4.3 Interfaces

4.3.1 Interface - família SHA

Em seguida é mostrado um trecho do arquivo `sha.h`, que descreve a interface utilizada para as funções de resumo da família SHA.

```
typedef struct SHA1_CONTEXT SHA1_CONTEXT;
typedef struct SHA224_CONTEXT SHA224_CONTEXT;
typedef struct SHA256_CONTEXT SHA256_CONTEXT;
typedef struct SHA384_CONTEXT SHA384_CONTEXT;
typedef struct SHA512_CONTEXT SHA512_CONTEXT;

/* SHA-1 */

HASH_STATUS sha1_context_size (int *size_ptr);
HASH_STATUS sha1_initialize (SHA1_CONTEXT * state_ptr);
HASH_STATUS sha1_update (const uint8_t * message_ptr,
                         int msg_len,
                         SHA1_CONTEXT *state_ptr);
HASH_STATUS sha1_finalize (uint8_t * digest_ptr,
                           SHA1_CONTEXT *state_ptr);
HASH_STATUS sha1_digest (const uint8_t * message_ptr,
                         int msg_len,
```

```
                        uint8_t * digest_ptr);

/* SHA-224 */

HASH_STATUS sha224_context_size (int *size_ptr);
HASH_STATUS sha224_initialize (SHA224_CONTEXT * state_ptr);
HASH_STATUS sha224_update (const uint8_t * message_ptr,
                           int msg_len,
                           SHA224_CONTEXT *state_ptr);
HASH_STATUS sha224_finalize (uint8_t * digest_ptr,
                             SHA224_CONTEXT *state_ptr);
HASH_STATUS sha224_digest (const uint8_t * message_ptr,
                           int msg_len,
                           uint8_t * digest_ptr);

/* SHA-256 */

HASH_STATUS sha256_context_size (int *size_ptr);
HASH_STATUS sha256_initialize (SHA256_CONTEXT * state_ptr);
HASH_STATUS sha256_update (const uint8_t * message_ptr,
                           int msg_len,
                           SHA256_CONTEXT *state_ptr);
HASH_STATUS sha256_finalize (uint8_t * digest_ptr,
                             SHA256_CONTEXT *state_ptr);
HASH_STATUS sha256_digest (const uint8_t * message_ptr,
                           int msg_len,
                           uint8_t * digest_ptr);

/* SHA-384 */

HASH_STATUS sha384_context_size (int *size_ptr);
HASH_STATUS sha384_initialize (SHA384_CONTEXT * state_ptr);
HASH_STATUS sha384_update (const uint8_t * message_ptr,
                           int msg_len,
                           SHA384_CONTEXT *state_ptr);
HASH_STATUS sha384_finalize (uint8_t * digest_ptr,
                             SHA384_CONTEXT *state_ptr);
HASH_STATUS sha384_digest (const uint8_t * message_ptr,
                           int msg_len,
                           uint8_t * digest_ptr);
/* SHA-512 */

HASH_STATUS sha512_context_size (int *size_ptr);
```

```
HASH_STATUS sha512_initialize (SHA512_CONTEXT * state_ptr);
HASH_STATUS sha512_update (const uint8_t * message_ptr,
                           int msg_len,
                           SHA512_CONTEXT *state_ptr);
HASH_STATUS sha512_finalize (uint8_t * digest_ptr,
                             SHA512_CONTEXT *state_ptr);
HASH_STATUS sha512_digest (const uint8_t * message_ptr,
                           int msg_len,
                           uint8_t * digest_ptr);
```

A interface das outras funções analisadas é bastante semelhante: uma função de inicialização, uma função de processamento e uma função de finalização. Esta abordagem tem a vantagem de permitir o processamento por partes, dando flexibilidade ao usuário. A essas funções às vezes é acrescentada uma função de conveniência que faz todo o trabalho e é útil nos casos em que a mensagem inteira a ser processada está disponível imediatamente e tem um tamanho razoável. A biblioteca MIRACL utiliza uma abordagem um pouco diferente, na medida em que a entrada deve ser fornecida byte a byte.

Apesar de as funções de resumo definirem suporte a um número arbitrário de bits, em geral as implementações suportam apenas um número inteiro de bytes. A implementação feita inicialmente para a família SHA suportava bits mas, para tornar mais significativa a comparação com outras implementações, as otimizações foram feitas em cima de uma simplificação que suporta apenas bytes. De qualquer forma, o suporte ou não a bits não altera o núcleo do processamento dos blocos, que é sempre feito em um número inteiro de bytes.

4.3.2 Interface - *Whirlpool*

Para os testes do *Whirlpool* foi utilizada a implementação de Barreto e Rijmen ([6].

4.4 Testes

Os testes realizados podem ser divididos basicamente em testes de correção, cujo objetivo é garantir que a implementação funciona corretamente, e testes de desempenho, que buscam medir a velocidade de processamento das funções.

4.4.1 Testes de correção

Nos testes para garantir que a implementação implementa corretamente as funções pretendidas foram utilizados os seguintes tipos de vetores:

- Vetores menores do que o tamanho mínimo do bloco (testar efeitos do *padding*).
- Vetores que resultem no tamanho exato do bloco (testar cálculo do *padding*).
- Vetores que geram dois blocos (lógica de geração e preenchimento do segundo bloco).
- Vetores "grandes", ou seja, com milhares de bytes.

Os vetores utilizados foram essencialmente os que estão apresentados como exemplo na especificação do SHA ([44]) e os que estão disponíveis no *site* do NIST.

Se for considerada a implementação de suporte a bits:

- Vetores cujo número de bits não é múltiplo de 8.

Usando um fonte .C comum, a maioria dos testes de correção pode ser executada (e o código depurado) numa plataforma de acesso mais fácil (Windows/Linux).

4.4.2 Desempenho

O processamento de uma função de resumo compreende três partes:

inicialização em geral contém apenas a inicialização das variáveis de controle do processamento e não tem um impacto significativo no desempenho;

processamento é nesta parte onde ocorre praticamente todo o trabalho e, pela sua natureza iterativa, é onde o desempenho é mais afetado;

finalização tipicamente copia o resultado final do resumo de volta para uma área passada para o chamador da função e, portanto, a sua influência no desempenho também não é relevante.

Para avaliação de desempenho, a verificação da lógica de preenchimento do *padding*, por exemplo, não é tão importante. Supõe-se que a implementação funciona corretamente. Após cada alteração, a implementação deve ser testada para verificar a correção, obviamente. No entanto, devido às particularidades de manipulação de bit, ordem dos bytes (*little-endian* e *big-endian*), etc. idealmente os testes de correção têm que ser executados na plataforma alvo também.

Possíveis vetores de teste:

pequenos vetores com tamanho menor do que o tamanho do bloco. Neste caso, o impacto da carga das funções em memória (efeitos do *cache* vazio, por exemplo) pode representar uma parte significativa do tempo mas em aplicações reais isto é pouco provável; Se a função for usada pouco freqüentemente ou para vetores pequenos o impacto no desempenho será proporcionalmente pequeno;

médios vetores com tamanho de 30 KiB (30.720 bytes), representativos de um e-mail ou pequeno arquivo texto.

grandes vetores com 1 MiB[1], representativos de arquivos. Para este tamanho, os efeitos da inicialização devem ser desprezíveis e a taxa de processamento de bytes poderia ser extrapolada para vetores de tamanho maior.

Como o trabalho de processamento de cada bloco em uma função de resumo é essencialmente o mesmo, optou-se por utilizar apenas um tamanho de vetor nos testes.

No Cygwin/Linux, a opção escolhida foi processar 1 MiB por vez, passando 16 KiB a cada chamada da função de processamento do resumo e repetir cada função 100 vezes para determinar o tempo médio de execução. No Windows Mobile foi usada uma estratégia semelhante, reduzindo apenas o número de repetições de 100 para 50. Em ambos os casos o valor `0x10` foi utilizado para preencher os vetores de teste. Os resultados estão sendo mostrados em MiB/s.

4.4.2.1 Geração dos testes

No Cygwin/Linux, a partir de um arquivo texto contendo informações como:

- Autor
- Função a ser testada (SHA-1, SHA-224, etc.)
- Chaves de compilação para controlar otimização
- Chaves de compilação especiais (para ativar desenrolamento de laços, selecionar opções de implementação, etc.)

um programa gera uma série de arquivos de comandos do *shell* para compilar, executar um teste mínimo (processamento dos exemplos do padrão FIPS e verificação do resultado) e executar os testes de desempenho.

No Windows Mobile, como o foco era a geração de testes apenas para a nossa implementação (SHA) e Barreto e Rijmen (*Whirlpool*), foi adotada uma abordagem mais simples e a geração do arquivo para execução de testes foi feita manualmente. O *shell script* utilizado no Cygwin/Linux não pôde ser inteiramente reaproveitado devido à inexistência do interpretador no ambiente de execução Windows Mobile.

[1] 1 MiB = 2^{20} bytes, ou seja 1.048.576 bytes

4.4.3 Execução dos testes

O resultado dos testes é escrito em um arquivo com campos separados por vírgulas (formato CSV — *comma-separated values*) de forma a facilitar a sua importação em uma planilha para análise.

4.4.3.1 Medição do tempo de execução

No Linux foi utilizada a função `clock()` e no Windows Mobile a função `GetTickCount()`. Em todos os casos o teste foi efetuado quando não havia nenhuma outra aplicação sendo executada na máquina, além das funções básicas do sistema operacional.

4.4.3.2 Execução dos testes

Na execução dos testes foram estudados os efeitos de:

- utilização de opções de otimização (sem otimização, O2 e O3 — conforme aplicável);
- alvo de geração de código do compilador (código para PXA255 e PXA270);
- desenrolamento de laços;
- escrita de trechos utilizando funções intrínsecas para explorar o suporte ao Wireless MMX™;
- combinação de algumas das opções acima.

Os resultados apresentados a seguir foram obtidos a partir de executáveis compilados pelo autor (e não pelos autores originais) e, portanto, podem não refletir o melhor desempenho possível de cada uma das funções.

4.5 Resultados no Pentium®

4.5.1 Comparação de outras implementações no Pentium®

Os testes foram executados em uma máquina com processador Pentium® IV 2.4 GHz e 512 MiB de RAM. O compilador usado foi o GNU C compiler (gcc) versão 4.0.1 rodando sobre o Linux (Fedora Core 4).

Nas tabelas de resultados mostradas a seguir, uma célula em branco indica que uma determinada opção não estava disponível para aquela implementação (por exemplo, um determinado nível de desenrolamento do laço, utilização de trechos em “assembly”, etc). “N/T” (não testado) é utilizado para os casos em que a implementação poderia ter sido feita e/ou testada mas não o foi devido a uma limitação de tempo.

O significado das opções mostradas é o seguinte:

O2 compilação com a chave de compilação `-O2`

O3 compilação com a chave de compilação `-O3`

O3ASM para LibTomCrypt, compilação com a chave de compilação `-O3` e habilitação de otimizações em linguagem *assembly*.

O3UNROLL01 o laço principal não foi desenrolado

O3UNROLLXX SHA-1 cada uma das partes do laço principal (0 a 19, 20 a 39, etc.) foi desenrolada XX (02, 04, 05, 10 ou 20) vezes.

SHA-256 o laço principal foi desenrolado XX (02, 04, 08, 16, 32 ou 64) vezes

SHA-512 o laço principal foi desenrolado XX (04, 08, 80) vezes

O3UNROLLXXASM para LibTomCrypt, o laço foi desenrolado XX vezes (05 para o SHA-1, 08 para o SHA-256 e 08 para o SHA-512) as otimizações em linguagem *assembly* foram habilitadas.

O3UNROLLASM para LibTomCrypt, a implementação de *Whirlpool* foi compilada sem a chave `LTC_SMALL_CODE`, ou seja, utilizando múltiplas tabelas de busca.

4.5.1.1 SHA-1

Tabela 4.1: Comparação de desempenho do SHA-1 no Pentium®

Identificação	Devine	Gifford	LibTomCrypt	MIRACL	OpenSSL	Saddi	Tavares
O2	56.5	64.2	51.0	30.8		40.5	32.4
O3	56.6	64.4	87.9		220.0	42.4	32.3
O3ASM			57.1				
O3UNROLL01						42.6	45.3
O3UNROLL02						41.4	N/T
O3UNROLL04						41.2	N/T
O3UNROLL05			56.1			41.0	45.5
O3UNROLL05ASM			89.3				
O3UNROLL10						41.3	N/T
O3UNROLL20		77.4				41.5	35.1
sem otimização	33.1	38.2	40.2	18.9	223.0	29.2	27.6

Os valores estão expressos em MiB/s. Células em branco indicam opção não disponível.

Avaliação geral:

- Entre as implementações em C, a LibTomCrypt ([13]) é a mais rápida, tanto sem otimização quanto com otimização máxima (`-O3` no gcc). No entanto, a implementação do OpenSSL ([46]), que foi feita em linguagem *assembly*, é muito mais rápida. Observa-se que mesmo utilizando as opções de otimização do compilador, uma implementação otimizada em linguagem *assembly* ainda pode ser muito mais rápida.

- Em todos os casos houve um ganho muito significativo com a utilização de otimização pelo compilador chegando a mais do que dobrar a velocidade em relação à versão não otimizada (LibTomCrypt).

- Para as implementações avaliadas, o efeito do desenrolamento do laço foi variável de uma implementação para outra. Em alguns casos (Saddi, LibTomCrypt) ele não foi muito perceptível, em outros (Gifford) houve uma melhora e em alguns casos provocou queda no desempenho (nossa implementação, com o laço desenrolado completamente). É importante notar que a quantidade de vezes que o laço é desenrolado pode fazer muita diferença. Para a nossa implementação, por exemplo, houve um ganho quando o laço foi desenrolado 5 vezes mas uma queda significativa quando houve um desenrolamento maior.

4.5.1.2 SHA-256

Tabela 4.2: Comparação de desempenho do SHA-256 no Pentium®

Identificação	**Devine**	**Gifford**	**LibTomCrypt**	**MIRACL**	**OpenSSL**	**Saddi**	**Tavares**
O2	27.7	34.4	37.2	31.4		38.0	35.9
O3	27.7	34.9	36.9		45.4	38.0	36.4
O3ASM			39.9				
O3UNROLL01						35.2	36.2
O3UNROLL02						37.9	N/T
O3UNROLL04						38.1	36.3
O3UNROLL08		22.7	33.8			38.1	36.4
O3UNROLL08ASM			42.8				
O3UNROLL16						38.1	N/T
O3UNROLL32						38.1	N/T
O3UNROLL64						38.1	36.4

Continua na próxima página

Tabela 4.2: Comparação de desempenho do SHA-256 no Pentium® (continuação)

Identificação	Devine	Gifford	LibTomCrypt	MIRACL	OpenSSL	Saddi	Tavares
sem otimização	28.5	28.9	27.0	16.8	33.3	30.1	27.3

Os valores estão expressos em MiB/s. Células em branco indicam opção não disponível.

Avaliação geral:

- Para o SHA-256 o desempenho das implementações foi bem mais próximo do que no caso do SHA-1, embora ainda se destaque a implementação do OpenSSL.
- Embora menos dramáticos do que no caso do SHA-1, os efeitos da otimização (`-O3`) ainda são bastante significativos.
- Os efeitos do desenrolamento do laço praticamente não se notam, exceto por uma pequena melhora no desempenho de Saddi (com desenrolamento igual a 2, em relação à versão sem desenrolamento).

4.5.1.3 SHA-512

Tabela 4.3: Comparação de desempenho do SHA-512 no Pentium®

Identificação	Gifford	LibTomCrypt	MIRACL	Saddi	Tavares
O2	16.0	19.5	12.8	20.0	13.2
O3	15.8	20.7		20.7	13.0
O3ASM		20.7			
O3UNROLL01					13.0
O3UNROLL04					13.3
O3UNROLL08	19.9	20.9			13.4
O3UNROLL08ASM		20.7			
O3UNROLL80					13.3
sem otimização	15.8	14.4	11.6	16.9	15.9

Os valores estão expressos em MiB/s. Células em branco indicam opção não disponível.

Avaliação geral:

- Diferentemente do SHA-1 e SHA-256, a utilização de otimização causou queda no desem-

penho da nossa implementação e não fez diferença para a implementação de Gifford. Para outras implementações ainda foram observados ganhos de desempenho.

- Para Gifford houve um aumento signficativo do desempenho com a utilização do desenrolamento, que não foi visto nem no caso da nossa implementação nem no de LibTomCrypt.
- Para o SHA-512 não foi possível utilizar a implementação do OpenSSL devido aos erros de compilação.
- Não havia implementação de Devine para o SHA-512.

4.5.1.4 *Whirlpool*

Tabela 4.4: Comparação de desempenho do *Whirlpool* no Pentium®

Identificação	**Barreto**	**LibTomCrypt**
O2	107.0	7.6
O3	106.0	7.9
O3ASM		7.9
O3UNROLL		18.7
O3UNROLLASM		14.1
sem otimização	98.3	6.0

Os valores estão expressos em MiB/s. Células em branco indicam opção não disponível.

Avaliação geral:

- A implementação de Barreto e Rijmen ([6]) é muitas vezes mais rápida do que a implementação de LibTomCrypt ([13]).
- No caso de Barreto e Rijmen ([6]), a utilização de otimização pelo compilador causou uma degradação significativa no desempenho.
- *Whirlpool* é bem mais rápida no Pentium® do que a função SHA-512 (que tem comprimento equivalente).
- quando compilado com LTC_SMALL_CODE, o desempenho da implementação de LibTomCrypt ([13]) melhorou bastante o desempenho, o que pode sugerir que o ganho de velocidade no acesso ao *cache* ao trabalhar com uma única tabela pode compensar o custo de fazer a rotação.

- Para o *Whirlpool* não foi possível utilizar a implementação do OpenSSL devido aos erros de compilação.

4.6 Resultados no XScale®

4.6.1 Comparação de implementações no Linux/XScale®

A geração de código foi feita em um computador de mesa conforme descrito na seção 4.2.2. A plataforma-alvo utilizada foi uma plataforma Sitsang com processador PXA255, rodando o sistema operacional Linux. Devido aos erros na compilação, o desempenho do OpenSSL não foi analisado nesta plataforma.

4.6.1.1 SHA-1

Tabela 4.5: Comparação de desempenho do SHA-1 no PXA255

Identificação	**Devine**	**Gifford**	**LibTomCrypt**	**MIRACL**	**Saddi**	**Tavares**
O2	15.3	7.7	8.3	4.0	9.8	5.3
O3	15.0	7.6	8.3		11.1	5.3
O3ASM			8.3			
O3UNROLL01					10.9	7.2
O3UNROLL02					11.0	N/T
O3UNROLL04					11.0	N/T
O3UNROLL05			9.3		10.9	7.8
O3UNROLL05ASM			9.7			
O3UNROLL10					10.0	N/T
O3UNROLL20		7.8			9.4	8.9
sem otimização	4.8	2.1		1.3	2.2	1.4

Os valores estão expressos em MiB/s. Células em branco indicam opção não disponível.

Avaliação geral:

- A implementação de LibTomCrypt foi a mais rápida quando o código foi compilado sem otimização mas a de Devine revelou-se mais rápida com a utilização de `-O3`.
- O desenrolamento máximo do laço diminuiu o desempenho de Saddi mas trouxe um ganho significativo para a nossa implementação. Para Gifford o efeito foi praticamente nulo.

Novamente, um padrão muito diferente do encontrado no Pentium®.

- Em todos os casos houve um ganho muito significativo com a utilização de otimização pelo compilador chegado a quase quadruplicar a velocidade em relação à versão não otimizada. Este ganho foi significativamente maior do que o observado no Pentium®.
- O desempenho da implementação varia muito do Pentium® para o XScale®, não sendo necessariamente possível extrapolar os resultados obtidos no Pentium® para o XScale®.

4.6.1.2 SHA-256

Tabela 4.6: Comparação de desempenho do SHA-256 no PXA255

Identificação	**Devine**	**Gifford**	**MIRACL**	**Saddi**	**Tavares**
O2	5.6	4.6	3.2	5.1	5.0
O3	4.9	4.8		5.4	5.0
O3UNROLL01				5.4	5.0
O3UNROLL02				5.5	
O3UNROLL04				5.5	5.0
O3UNROLL08		3.0		5.4	5.1
O3UNROLL16				5.5	
O3UNROLL32				5.4	N/T
O3UNROLL64				5.4	5.0
sem otimização	2.8	1.6	1.0	1.5	1.3

Os valores estão expressos em MiB/s. Células em branco indicam opção não disponível.

Avaliação geral:

- A implementação de Devine foi a mais rápida quando compilada sem otimização e com a opção `-O2`. É interessante observar que houve uma queda no desempenho quando foi utilizado `-O3`.
- Praticamente não houve alteração no desempenho quando foram utilizadas as diversas opções de desenrolamento.
- Em todos os casos, a utilização de otimização pelo compilador mostrou efeitos significativos sobre o desempenho, chegado em alguns casos (Gifford, Saddi) a triplicar a velocidade em relação à versão não otimizada.

4.6.1.3 SHA-512

Tabela 4.7: Comparação de desempenho do SHA-512 no PXA255

Identificação	**Gifford**	**MIRACL**	**Saddi**	**Tavares**
O2	0.7	0.9	1.4	1.1
O3	0.8		0.8	1.3
O3UNROLL01				1.3
O3UNROLL04				1.3
O3UNROLL08	0.9			1.3
O3UNROLL80				1.3
sem otimização	0.8	0.6	0.8	1.3

Os valores estão expressos em MiB/s. Células em branco indicam opção não disponível.

Avaliação geral:

- O desenrolamento do laço não trouxe ganhos (ou prejuízos) visíveis para o desempenho
- Não havia implementação de Devine para o SHA-512.

4.6.1.4 *Whirlpool*

Tabela 4.8: Comparação de desempenho do *Whirlpool* no PXA255

Identificação	**Barreto**
O2	1.4
O3	1.4
sem otimização	4.3

Os valores estão expressos em MiB/s. Células em branco indicam opção não disponível.

Avaliação geral:

- No caso de Barreto e Rijmen ([6]), a utilização de otimização pelo compilador causou uma degradação significativa no desempenho.

4.6.2 Análise da nossa implementação e a de Barreto e Rijmen no Windows Mobile/XScale®

A geração de código foi feita em um computador de mesa conforme descrito na seção 4.2.3. As plataformas-alvo utilizadas foram PXA270 (Dell Axim X50v) e PXA255 (HP iPAQ5550), ambas rodando o sistema operacional Windows Mobile.

A opção de otimização do compilador utilizada foi `/O3`, que já inclui um conjunto de opções voltadas para melhoria de velocidade. O controle da geração de código foi feito através das chaves `/QTPxsc2` e `/QTPxsc3`. `/QTPxsc2` habilita a geração de código para os processadores PXA255 (e PXA270) e `/QTPxsc3` acrescenta o suporte às extensões Wireless MMX™, específicas do PXA270.

Análise da nossa implementação (família SHA) e de Barreto e Rijmen (*Whirlpool*), com hardware, sistema operacional e compiladores idênticos e diferentes:

- chaves de compilação.
- alternativas de implementação.
- alvos de geração de código (código gerado para o PXA255 executando no PXA255 comparado com código gerado para o PXA270 rodando no PXA270).

O significado das opções mostradas é o seguinte:

OPT_01 Para o SHA-1, significa que foi utilizada a implementação em que o laço principal está dividido em quatro partes (uma para cada tipo de f_t). Esta opção não faz diferença para as implementações do SHA-256 e SHA-512.

OPT_03 Indica que as definições alternativas para as funções Ch e Maj (veja seção 2.5.3.1.

OPT_02 Indica que o laço principal foi desenrolado: cinco vezes para cada uma das quatro faixas conforme o tipo de função f_t $(0\ldots19, 20\ldots39, 40\ldots59, 60\ldots79)$ no SHA-1, oito vezes no SHA-256 e quatro vezes no SHA-512.

OPT_04 Indica que foi usada uma "rotação" das variáveis para evitar cópias.

O3 A compilação foi feita com a chave `O3` (otimização máxima).

MMX (automática) É a versão que foi compilada com suporte às extensões Wireless MMX™, mas cuja utilização (ou não) foi deixada a cargo do compilador).

MMX (manual) É a versão que foi compilada com suporte às extensões Wireless MMX™, com a versão do código implementada manualmente (veja seção 3.6.2).

4.6.2.1 SHA-1

Tabela 4.9: Comparação de opções de implementação de SHA-1 no PXA255

Identificação	**Tavares**
O3	5.7
O3-_OPT_01_03	8.5
O3-_OPT_02_03_04	8.4
sem otimização	1.3

Os valores estão expressos em MiB/s. Células em branco indicam opção não disponível.

Tabela 4.10: Comparação de opções de implementação de SHA-1 no PXA270

Identificação	**Tavares**
MMX (automática) O3	8.2
MMX (automática) O3-_OPT_01_03	11.8
MMX (automática) O3-_OPT_02_03_04	11.9
MMX (automática) sem otimização	2.1
MMX (manual) O3	13.4
MMX (manual) sem otimização	2.5
O3	8.4
O3-_OPT_01_03	12.3
O3-_OPT_02_03_04	12.1
sem otimização	2.1

Os valores estão expressos em MiB/s. Células em branco indicam opção não disponível.

Avaliação geral:

- Sem utilizar as extensões Wireless MMX™, a melhor combinação foi obtida através da retirada do desvio de dentro do laço (OPT_01) combinada com a utilização das definições alternativas para as funções Ch(), etc.

- O desenrolamento parcial do laço (5 vezes para cada parte, OPT_04) combinado com o renomeamento dos registradores (OPT_02) e o uso das definições alternativas (OPT_03) apresentou resultado inferior mas bastante semelhante ao anterior, o que sugere que o desenrolamento não traz um ganho muito significativo neste caso.

- O melhor desempenho geral foi do código usando Wireless MMX™ (escrito manualmente).
- O desempenho da nossa implementação (com otimização) no Windows Mobile é bastante semelhante ao observado no Linux.

Extensões Wireless MMX™:

- Para o código sem Wireless MMX™, o padrão de comportamento no PXA255 e PXA270 foi semelhante.
- O código gerado para o PXA270 de forma automática (ou seja, habilitando as extensões Wireless MMX™ mas sem nenhuma codificação manual) apresentou desempenho inferior ao do código gerado sem esta habilitação. De qualquer forma, exceto pela parte do laço principal que faz a cópia das chaves, o compilador não encontrou oportunidades de vetorização.

Efeitos da otimização do compilador:

- Ganho significativo, velocidade praticamente quatro vezes maior no PXA255.
- A melhoria de velocidade foi mas significativa no PXA270 do que no PXA255.
- Quando são usadas as extensões MMX™, o ganho é ainda maior (mais de cinco vezes).

Alguns resultados são esperados (naturais), outros necessitariam de uma análise mais detalhada, utilizando, por exemplo, uma ferramentas de análise como a mencionada em 5.2.2.

4.6.2.2 SHA-256

Tabela 4.11: Comparação de opções de implementação de SHA-256 no PXA255

Identificação	**Tavares**
O3	5.5
O3-_OPT_01_03	5.5
O3-_OPT_02_03_04	5.7
sem otimização	1.2

Os valores estão expressos em MiB/s. Células em branco indicam opção não disponível.

Tabela 4.12: Comparação de opções de implementação de SHA-256 no PXA270

Identificação	**Tavares**
MMX (automática) O3	7.1
MMX (automática) O3-_OPT_01_03	6.9
MMX (automática) O3-_OPT_02_03_04	8.1
MMX (automática) sem otimização	1.8
O3	8.2
O3-_OPT_01_03	8.2
O3-_OPT_02_03_04	8.5
sem otimização	1.8

Os valores estão expressos em MiB/s. Células em branco indicam opção não disponível.

Avaliação geral:

- O desempenho da nossa implementação (com otimização) no Windows Mobile é bastante semelhante ao observado no Linux.
- No PXA270, a compilação com utilização automática do Wireless MMX™ teve desempenho inferior ao da outra versão.
- Com utilização automática de Wireless MMX™ o desenrolamento do laço melhorou o desempenho — embora o ganho tenha sido bem menor .

4.6.2.3 SHA-512

Tabela 4.13: Comparação de opções de implementação de SHA-512 no PXA255

Identificação	**Tavares**
O3	2.9
O3-_OPT_01_03	3.0
O3-_OPT_02_03_04	3.2
sem otimização	0.7

Os valores estão expressos em MiB/s. Células em branco indicam opção não disponível.

Tabela 4.14: Comparação de opções de implementação de SHA-512 no PXA270

Identificação	**Tavares**
MMX (automática) O3	4.4
MMX (automática) O3-_OPT_01_03	4.5
MMX (automática) O3-_OPT_02_03_04	4.8
MMX (automática) sem otimização	1.0
MMX (manual) O3	7.3
MMX (manual) sem otimização	1.2
O3	4.4
O3-_OPT_01_03	4.5
O3-_OPT_02_03_04	4.8
sem otimização	1.0

Os valores estão expressos em MiB/s. Células em branco indicam opção não disponível.

Avaliação geral:

- Sem utilizar as extensões Wireless MMX™, a melhor combinação foi obtida através do desenrolamento parcial do laço (4 vezes, OPT_04) combinado com o renomeamento dos registradores (OPT_02) e o uso das definições alternativas (OPT_03) para as funções Ch(), etc.

- A opção sem desenrolamento apresentou resultado inferior mas bastante semelhante ao anterior, o que sugere que o desenrolamento não traz um ganho muito significativo neste caso.

- O melhor desempenho geral foi do código usando Wireless MMX™ (escrito manualmente) (cerca de 50% superior à melhor opção sem Wireless MMX™).

Extensões Wireless MMX™:

- Para o código sem Wireless MMX™, o padrão de comportamento no PXA255 e PXA270 foi semelhante.

- O código gerado para o PXA270 de forma automática (ou seja, habilitando as extensões Wireless MMX™ mas sem nenhuma codificação manual) apresentou desempenho praticamente igual ao do código gerado sem esta habilitação, o que sugere que o compilador não explora as possibilidades de utilização dos registradores Wireless MMX™ para operações de 64 bits.

Efeitos da otimização:

- O padrão de ganho de velocidade com a otimização é semelhante ao observado para outras funções (cerca de quatro vezes).

4.6.2.4 *Whirlpool*

Tabela 4.15: Comparação de opções de implementação de *Whirlpool* no PXA255

Identificação	**Barreto**
O3	4.6
sem otimização	6.3

Os valores estão expressos em MiB/s. Células em branco indicam opção não disponível.

Tabela 4.16: Comparação de opções de implementação de *Whirlpool* no PXA270

Identificação	**Barreto**
MMX (automática) O3	6.9
MMX (automática) sem otimização	6.9
O3	6.9
sem otimização	9.3

Os valores estão expressos em MiB/s. Células em branco indicam opção não disponível.

Avaliação geral:

- Diferentemente do que ocorreu com as funções da família SHA, a utilização de otimização pelo compilador fez o desempenho cair significativamente.
- Conforme observado para o SHA-512, o compilador também não explorou as oportunidades de utilização do Wireless MMX™ para operações de 64 bits e o desempenho com (e sem) a habilitação destas extensões é praticamente idêntico.

4.6.3 Comparação do desempenho no Pentium® e XScale®

Conforme mencionado nos resultados, o padrão de comportamento das implementações não é o mesmo no XScale® e no Pentium®. Embora a diferença entre a velocidade do *Whirlpool* e do SHA-512 seja menor no XScale® do que no Pentium®, a função *Whirlpool* ainda é mais rápida do que SHA-512.

Capítulo 5

Conclusões e trabalhos futuros

Neste capítulo apresentamos as conclusões do trabalho e mostramos algumas possibilidades de aprofundamento que poderiam ser exploradas em trabalhos futuros.

5.1 Conclusões

Como parte do trabalho foram identificadas várias técnicas de otimização aplicáveis às funções de resumo estudadas e aos processadores da família XScale®, em particular aos que suportam as extensões Wireless MMX™. Muitas destas técnicas podem ser utilizadas em outros tipos de programa voltados para a mesma plataforma ou que possam fazer uso de instruções do tipo *SIMD*.

Em resumo:

- Existem muitas oportunidades de otimização das funções de resumo:
  - Embora os compiladores ofereçam suporte a otimização automática através de chaves para controlar a geração de código, nem sempre a otimização melhora realmente o desempenho, sendo necessária uma avaliação cuidadosa para determinar a combinação ideal. Em alguns casos a otimização pode degradar o desempenho ou uma chave aparentemente melhor (`-O3`) produzir um código menos rápido do que outra em princípio inferior (`-O2`). Além disso, a implementação em linguagem *assembly* pode ser muito mais rápida do que o código gerado pelo compilador.
  - Para as funções da família SHA (especialmente SHA-1), o desenrolamento é bastante importante; no entanto, a quantidade de vezes que o laço é desenrolado também pode fazer muita diferença. Além disso, o efeito é muito variável dependendo da implementação geral.

- O desempenho das implementações individuais varia muito do Pentium® para o XScale®, não sendo necessariamente possível extrapolar os resultados obtidos no Pentium® para o XScale®. Apesar disso, a razão entre a velocidade do SHA-256 e SHA-1 e do SHA-512 e SHA-256 é semelhante nas duas plataformas.
- Em relação às extensões Wireless MMX™, foi determinado que é possível explorá-las para melhorar o desempenho das funções estudadas mas na versão do compilador usada este benefício só pode ser obtido através de uma codificação semi-automática, já que compilador não explorou automaticamente as operações de 64 bits e praticamente não encontrou oportunidades de vetorização automaticamente.

5.2 Trabalhos futuros

A seguir são mostradas algumas atividades que poderiam complementar o estudo feito e que não foram realizadas devido a limitações de tempo ou disponibilidade de recursos (especialmente compiladores e máquinas).

5.2.1 Utilização de linguagem *assembly*

A transferência de dados entre os registradores Wireless MMX™ e os registradores de 32 bits talvez possa ser otimizada utilizando linguagem *assembly* diretamente. Como a implementação utilizou o suporte de funções intrínsecas, a transferência de valores é feita automaticamente pelo compilador. São usadas funções como `_mm_setr_pi32()` (para transferir dois valores de 32 bits para um registrador Wireless MMX™) e `_mm_cvtm64_si64()`/`_mm_cvtsi64_m64()` (para transportar valores de 64 bits de/para registradores Wireless MMX™) mas não há controle direto sobre a utilização da instrução **`TMCRR`** ou **`WLDR`** ou do reaproveitamento de valores que estão nos registradores. Portanto o código gerado pelo compilador não é necessariamente a alternativa mais rápida.

5.2.2 Utilização de ferramenta de análise

Durante o trabalho foram feitas algumas suposições sobre a execução do código. Entretanto, através da utilização de uma ferramenta de análise de execução mais detalhada como, por exemplo, o VTune™ Performance Analyzer, seria possível fazer medidas como as descritas abaixo e, assim, validar estas suposições:

- Determinação do número de *cache misses*, o que permitiria avaliar um possível ganho de desempenho com a utilização da instrução de *preload*.

- Taxa de erro na predição de desvio, identificando possíveis impactos dos laços.
- Paradas (*stalls*) durante a execução devido à indisponibilidade de dados e eventual necessidade de reordenamento de instruções para explorar melhor o *pipeline*.

5.2.3 Análise do desempenho no PXA270/Windows Mobile

A análise de desempenho feita para o PXA270 na seção 4.6.2 poderia ser estendida fazendo a avaliação de outras implementações existentes (como aquelas avaliadas no Linux na seção 4.5.1) e determinando os padrões de comportamento específicos do PXA270 (quando comparado com o Pentium®).

5.2.4 Análise de processamento de duas mensagens em paralelo usando Wireless MMX™

Em princípio a técnica usada por [42]Nakajima et al. em [42] para processar mensagens em paralelo também seria possível no XScale® e poderia ser explorada.

5.2.5 Implementação e análise do SHA-256 usando as extensões Wireless MMX™

Como explicado na seção 3.6.3, a utilização das extensões Wireless MMX™ para explorar o algoritmo SHA-256 não foi feita. Uma tal implementação poderia então determinar se na prática os ganhos possíveis para o SHA-256 são semelhantes aos observados para o SHA-1.

5.2.6 Implementação de 64 bits do *Whirlpool* usando extensões Wireless MMX™

A implementação do *Whirlpool* usando 64 bits muito provavelmente poderia ter o seu desempenho melhorado utilizando as operações nativas de 64 bits disponibilizadas pelas extensões Wireless MMX™. Devido à limitação de tempo esta alternativa não foi explorada neste trabalho.

5.2.7 Implementação de 32 bits do *Whirlpool*

Como sugerido pelos autores do algoritmo, uma implementação utilizando tabelas de busca de 32 bits poderia ser mais eficiente em uma arquitetura cujo comprimento de palavra fosse daquele tamanho, como é o caso do XScale®. Além da comparação da implementação de 64 bits com a

de 32 bits também seria interessante comparar o desempenho desta última com a implementação utilizando as extensões Wireless MMX™.

5.2.8 Comparação com IPP

Em se tratando de uma biblioteca desenvolvida pela própria Intel® (que é também fabricante dos processadores XScale®) otimizada para a plataforma e voltada para a eficiência, espera-se que o desempenho das Intel® (*Integrated Performance Primitives*) seja muito bom. Essa implementação poderia então ser tomada como referência (*benchmark*) e uma comparação com outras implementações existentes poderia trazer indicações de quanto espaço para otimização ainda haveria. Como não foi possível ter acesso à biblioteca em tempo para este trabalho esta comparação não foi feita aqui.

5.2.9 Outros

Além das já mencionadas acima, existem algumas técnicas mencionadas na tabela 3.1 que não foram utilizadas, como o cálculo das chaves junto ao laço principal, e que poderiam também ser objeto de estudo mais aprofundado.

Outras idéias, um pouco mais afastadas do foco central deste trabalho, mas ainda relacionadas incluem:

- Criação de funções de resumo especiais a partir de funções de criptografia simétrica (por exemplo AES) usando o método genérico e comparação com as funções estudadas para determinar se nesta plataforma as funções especiais poderiam ser uma alternativa de melhor desempenho.
- Proposta de extensões à plataforma XScale® que trariam ganho de desempenho para estas aplicações.

Apêndice A

Descrição das funções intrínsecas

A documentação da Intel® ([25]) traz informações detalhadas sobre todas as funções intrínsecas disponíveis para o Wireless MMX™. A descrição abaixo visa apenas facilitar a compreensão dos trechos de código incluídos no trabalho.

_mm_add_pi32(m1,m2) efetua a adição de dois valores de 32 bits armazenados em `m1` e `m2`.

_mm_align_si64(m1, m2, count) extrai um valor de 64 bits de dois valores de 64 bits (`count` indica qual o deslocamento).

_mm_and_si64(m1,m2) executa um **AND** bit-a-bit entre os valores de 64 bits armazenados em `m1` e `m2`.

_mm_andnot_si64 executa um **NOT** no valor de 64 bits armazenado em `m1` e usa o resultado para executar um **AND** bit-a-bit com o valor de 64 bits armazenado em `m2`.

_mm_cvtm64_si64 converte um objeto do tipo `__m64` (que representa um registrador Wireless MMX™ e pode ser manipulado pelas funções intrínsecas) em um inteiro de 64 bits.

_mm_cvtsi64_m64 converte um inteiro de 64 bits em um objeto do tipo `__m64` (que representa um registrador Wireless MMX™ e pode ser manipulado pelas funções intrínsecas).

_mm_extract_pi32(a,n) extrai um valor de 32 bits de um valor de 64 bits (`n` indica qual das duas palavras será extraída).

_mm_or_si64(m1,m2) executa um **OR** bit-a-bit entre os valores de 64 bits armazenados em `m1` e `m2`.

_mm_rori_pi32(m1,count) gira dois valores de 32 bits armazenados em `m1` pelo número de posições indicado em `count`.

_mm_rori_si64(m1,count) gira o valor de 64 bits armazenado em `m1` pelo número de posições indicado em `count`.

_mm_setr_pi32(i0,i1) combina os dois valores de 32 bits em um valor de 64 bits.

_mm_xor_si64(m1,m2) executa um **XOR** bit-a-bit entre os valores de 64 bits armazenados em `m1` e `m2`.

Referências Bibliográficas

[1] 3rd, D. Eastlake e P. Jones: *US secure hash algorithm 1 (SHA1)*, setembro 2001. `www.ietf.org/rfc/rfc3174.txt`.

[2] Aciicmez, O.: *Analysis of SIMD applicability to SHA algorithms*, 2003. `http://islab.oregonstate.edu/koc/ece679/project/2003/aciicmez.pdf`, acessado em 21 de novembro de 2005.

[3] ARM: *Application note 34 writing efficient C for ARM*. Relatório Técnico ARM DAI 0034A, Advanced RISC Machines Ltd (ARM), janeiro 1998. `http://www.arm.com`.

[4] Barreto, P. S. L. M. e V. Rijmen: *The whirlpool hashing function*, 2003. `http://planeta.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html`.

[5] Barreto, Paulo S. L. M.: *The hashing function lounge*, 2005. `http://paginas.terra.com.br/informatica/paulobarreto/hflounge.html`, acessado em 15 de agosto de 2005.

[6] Barreto, Paulo S. L. M. e Vincent Rijmen: *The whirlpool hash function*. `http://planeta.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html`, acessado em 1 de maio de 2005, versão 3.0.

[7] Bik, Aart, Milind Girkar, Paul Grey, e Xinmin Tian: *Efficient exploitation of parallelism on Pentium® III and Pentium® 4 processor-based systems*. Intel Technology Journal, Q1, 2001. `http://developer.intel.com`.

[8] Bosselaers, Antoon: *Even faster hashing on the Pentium®*, novembro 1997. `http://www.esat.kuleuven.ac.be/~cosicart/pdf/AB-9701.pdf`, acessado em 11 de julho de 2004, publicado em 13 de maio de 1997, atualizado em 13 de novembro de 1997.

[9] Bosselaers, Antoon, René Govaerts, e Joos Vandewalle: *Fast hashing on the Pentium®*. In Koblitz, N. (editor): *Advances in Cryptology - CRYPT0 '96*, volume 1109 de *Lecture Notes on Computer Science*, páginas 298–312, Berlin, 1996. Springer-Verlag.

[10] Bosselaers, Antoon, René Govaerts, e Joos Vandewalle: *SHA: A design for parallel architectures?* In Fumy, W. (editor): *Advances in Cryptology - EUROCRYPT '97: International Conference on the Theory and Application of Cryptographic Techniques*, volume 1233 de *Lecture Notes on Computer Science*, páginas 348–362, Berlin, maio 1997. Springer-Verlag. `http://www.esat.kuleuven.ac.be/~cosicart/pdf/AB-9700.pdf`, acessado em 11 de julho de 2004.

[11] Burr, William E.: *NIST brief comment on recent cryptanalytic attacks on SHA 1 and NIST plans*, abril 2005. `http://www.csrc.nist.gov/pki/HashWorkshop/NIST\%20Statement/Burr_Mar2005.html`, acessado em 23 de janeiro de 2006, Última atualização em 5 de abril de 2005.

[12] Dai, Wei: *Crypto++® library*. `http://www.eskimo.com/~weidai/cryptlib.html`, acessado em 7 de novembro de 2005, versão 5.2.1.

[13] Denis, Tom St.: *LibTomCrypt*. `http://libtomcrypt.org`, acessado em 01 de maio de 2005, versão 1.0.2.

[14] Devine, Christophe: *Implementação do algoritmo SHA-1*. `http://www.cr0.net:8040/code/crypto/sha1/`, acessado em 1 de maio de 2005, sem informação de versão.

[15] Devine, Christophe: *Implementação do algoritmo SHA-256*. `http://www.cr0.net:8040/code/crypto/sha256/`, acessado em 1 de maio de 2005, sem informação de versão.

[16] Dumschat, Uli: *Intel® software development tools for Intel XScale® microarchitecture*, julho 2005. `http://developer.intel.com`, acessado em 21 de novembro de 2005.

[17] Fog, Agner: *How to optimize for the Pentium® family of microprocessors*, abril 2004. Última atualização em 16 de abril de 2004.

[18] Gerber, Richard: *The Software Optimization Cookbook: High-performance Recipes for the Intel® Architecture*. Intel Press, 2002, ISBN 0-9712887-1-2.

[19] Gifford, Aaron D.: *Implementação dos algoritmos da família SHA*. `http://www.aarongifford.com/computers/sha.html`, acessado em 1 de maio de 2005, versão 2.1.1.

[20] GNU: *Gnu crypto*. `http://www.gnu.org/software/gnu-crypto/gnu-crypto.html`, acessado em 01 de maio de 2005, versão 2.0.1.

[21] Gutmann, Peter, David Naccache, e Charles C. Palmer: *When hashes collide*. IEEE Security and Privacy, páginas 68–71, maio 2005, ISSN 1540-7993.

[22] *Intel® integrated performance primitives*, 2004. `http://developer.intel.com/design/pca/applicationsprocessors/swsup/IPP.htm`.

[23] *Optimizing applications with the Intel® C++ and Fortran compilers for Windows, Windows CE .NET, and Linux - updated for intel® 8.1 compilers*, novembro 2004. `http://developer.intel.com`, acessado em 21 de novembro de 2005.

[24] Intel® Corporation: *Intel® PXA250 and PXA210 Processors Optimization Guide*, outubro 2002. `http://developer.intel.com`, Order Number: 278552-002.

[25] Intel® Corporation: *Intel® Wireless MMX™ Technology Developer Guide*, agosto 2002. `http://developer.intel.com`, Order Number: 251793-001.

[26] Intel® Corporation: *Intel XScale® PXA250 and PXA210 Application Processors Optimization Guide for Application Developers*, fevereiro 2003. `http://developer.intel.com`, Order Number: 278779.

[27] Intel® Corporation: *Intel XScale® Core Developer's Manual*, janeiro 2004. `http://developer.intel.com`, Order Number: 273473-002.

[28] Intel® Corporation: *Intel® PCA 27x Product Brief*, fevereiro 2004. `http://www.intel.com/design/pca/prodbref/253820pb.htm`, acessado em 29 de maio 2004, Order Number: 253820-002.

[29] Intel® Corporation: *Intel® C++ Compiler 2.0.3 For Windows CE, Standard - Release Notes*, 2005. `http://developer.intel.com`, Document Number: 280085-025US.

[30] Intel® Corporation: *Intel® C++ Compiler User's Manual*, 2005. `http://developer.intel.com`, Document Number: 278496-015.

[31] Intel® Corporation: *Quick-Reference Guide to Optimization with Intel® Compilers: "A Step-by-Step Approach to Application Tuning with Intel Compilers"*, junho 2005. `www.intel.com/software/products`, Order Number: 254349-004.

[32] ISO: *Iso/iec 10118-3:2004 information technology – security techniques – hash-functions – part 3: Dedicated hash-functions*, 2004.

[33] Kaliski, B.: *The MD2 message-digest algorithm*, abril 1992. http://www.ietf.org/rfc/rfc1321.txt.

[34] Kelsey, John: *Hash functions - perspective from the United States*, 2005. http://www.ecrypt.eu.org/stvl/hfw/Kelsey.ppt, acessado em 15 de agosto de 2005, Apresentado na ECRYPT Conference on Hash Functions 2005 (por Rich Schroeppel).

[35] Kitsos, P. e O. Koufopavlou: *Efficient architecture and hardware implementation of the Whirlpool hash function*. IEEE Transactions on Consumer Electronics, 50(1):208–213, fevereiro 2004.

[36] Kitsos, P. e O. Koufopavlou: *Whirlpool hash function: Architecture and VLSI implementation*. In *ISCAS 2004*, páginas II 893–896. IEEE, 2004, ISBN 0-7803-8251-X.

[37] Ltd., Shamus Software: *Multiprecision integer and rational arithmetic c/c++ library*. http://indigo.ie/~mscott/, acessado em 7 de novembro de 2005, versão 4.85.

[38] Matsui, Mitsuru e Sayaka Fukuda: *How to maximize software performance of symmetric primitives on Pentium III and Pentium 4 processors*. In Gilbert, H. e H. Handschuh (editores): *FSE 2005*, volume 3557 de *Lecture Notes on Computer Science*, páginas 398–412, Berlin, 2005. International Association for Cryptologic Research, Springer-Verlag.

[39] McCurley, Kevin: *A fast portable implementation of the secure hash algorithm, iii*. Relatório Técnico SAND93-2591, Sandia National Laboratories, junho 1994. http://mccurley.org/papers, acessado em 30 de maio de 2004, Revisado em 22 de junho de 1994.

[40] Menezes, Alfred J., Paul C. van Oorschot, e Scott A. Vanstone: *Handbook of Applied Cryptography*. CRC Press, primeira edição, agosto 1996, ISBN 0849385237. http://www.cacr.math.uwaterloo.ca/hac/.

[41] Mirceski, Emilijan: *Fast hashing on Pentium® 4*. http://cpuedge.com/, acessado em 7 de novembro de 2005.

[42] Nakajima, Junko e Mitsuru Matsui: *Performance analysis and parallel implementation of dedicated hash functions*. In Knudsen, L.R. (editor): *EUROCRYPT 2002*, volume 2332 de *Lecture Notes on Computer Science*, páginas 165–180, Berlin, 2002. Springer-Verlag.

[43] NESSIE: *NESSIE. New European Schemes for Signatures, Integrity, and Encryption*, 2003. http://www.cosic.esat.kuleuven.ac.be/nessie.

[44] NIST: *Secure hash standard*, 2002. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf.

[45] *Cryptographic hash workshop*, outubro 2005. `http://www.csrc.nist.gov/pki/HashWorkshop/program.htm`, acessado em 23 de janeiro de 2006.

[46] OpenSSL Project: *OpenSSL library*. `http://www.openssl.org/`, acessado em 7 de novembro de 2005.

[47] Paver, Nigel C., Bradley C. Aldrich, e Moinul H. Khan: *Programming with Intel® Wireless MMX™ Technology: A Developer's Guide to Mobile Multimedia Applications*. Intel Press, primeira edição, abril 2004, ISBN 0974364916.

[48] Pieprzyk, Josef, Thomas Hardjono, e Jennifer Seberry: *Fundamentals of Computer Security*. Springer-Verlag, Berlin, 2003, ISBN 3540431012.

[49] Rivest, R.: *The MD4 message-digest algorithm*, abril 1992. `http://www.ietf.org/rfc/rfc1321.txt`.

[50] Rivest, R.: *The MD5 message-digest algorithm*, abril 1992. `http://www.ietf.org/rfc/rfc1321.txt`.

[51] Saddi, Allan: *Implementação dos algoritmos da família SHA*. `http://www.saddi.com/software/sha/`, acessado em 1 de maio de 2005, versão 1.0.4.

[52] Schneier, Bruce e Doug Whiting: *Fast software encryption: Designing encryption algorithms for optimal software speed on the Intel® Pentium® processor*. In Biham, E. (editor): *Fast Software Encryption: 4th International Workshop, FSE'97*, volume 1267 de *Lecture Notes on Computer Science*, página 242, Berlin, janeiro 1997. Springer-Verlag. `http://www.schneier.com/paper-fast-software-encryption.html`, acessado em 11 de julho de 2004.

[53] Sloss, Andrew N., Dominic Symes, e Chris Wright: *ARM System Developer's Guide: Designing and Optimizing System Software*. Morgan Kaufmann, primeira edição, 2004, ISBN 1558608745.

[54] Stallings, William: *Cryptography and Network Security: Principles and Practice*. Prentice-Hall, segunda edição, 1998, ISBN 0138690170.

[55] Touch, J.: *Report on MD5 performance*, junho 1995. `http://www.ietf.org/rfc/rfc1810.txt`.

[56] Touch, Joseph D.: *Performance analysis of MD5*. In *SIGCOMM '95*, 1995, ISBN 0-89791-711-1.

[57] Wang, X. e H. Yu: *How to break MD5 and other hash functions*. In Cramer, R. (editor): *EUROCRYPT 2005*, volume 3494 de *Lecture Notes on Computer Science*, páginas 19–35, Berlin, 2005. International Association for Cryptologic Research, Springer-Verlag. Também disponível em `http://www.infosec.sdu.edu.cn/people/wangxiaoyun.htm`.

[58] Wang, Xiaoyun, Yiqun Lisa Yin, e Hongbo Yu: *Finding collisions in the full SHA1*. In Shoup, Victor (editor): *Advances in Cryptology CRYPTO 2005*, volume 3621 de *Lecture Notes on Computer Science*, página 17, Berlin, agosto 2005. International Association for Cryptologic Research, Springer-Verlag, ISBN 3-540-28114-2. `http://www.infosec.sdu.edu.cn/people/wangxiaoyun.htm`, acessado em 15 de agosto de 2005.

[59] *Webopedia*. `http://www.webopedia.com/TERM/b/big\_endian.html`, acessado em 11 de dezembro de 2005.