Dictionary Decompression Architecture : une architecture de processeur à faible densité de code

Youssef FAHMI[1-2], Bertrand GRANADO[2], David KERR-MUNSLOW[1],

[1]Cortus S.A,97 rue de Freyr,Montpellier France

[2]ETIS - CNRS - ENSEA - Université Cergy Pontoise, France

youssef.fahmi@cortus.com, bertrand.granado@ensea.fr, david.kerr-munslow@cortus.com

Résumé – Dans cet article nous avons présenté un système de compression/décompression entiérement fonctionnel et transparent avec un gain moyen de 15% de densité de code. Notre mise en oeuvre est totalement compatible avec une technologie ASIC. Notre décompresseur valide l'intégralité des tests de regressions de gcc. L'innovation apportée par notre méthode de compression pré-link garde la compatibilité avec le format elf et facilite l'adaptation du reste de chaine de développement.

Abstract – In this paper we have presented a transparent and fully functional compression/decompression system with a code density average gain of 15%. Our implementation is fully compatible with an ASIC. Our decompressor validates all gcc regression tests. The innovation introduced by our prelink compression method is keeping compatibility with the elf format wich facilitate the adaptation of the rest of the toolchain.

1 Introduction

Les appareils nomades, basés sur des Système sur Puce [1] et fonctionnant à l'aide de batteries ont envahi notre quotidien. Ils ont un besoin toujours croissant de puissance de calcul tout en minimisant leur consommation, augmentant ainsi leur efficacité énergétique. Le coût de la mémoire induite par une partie logicielle de plus en plus importante et de plus en plus complexe est à prendre en compte pour accroître cette efficacité. Il faut modérer la quantité de mémoire nécessaire pour mettre en oeuvre une fonctionnalité désirée en augmentant la densification du code utilisé par le Système sur Puce. Cette densification permet de réaliser une meilleure adéquation de l'architecture aux applications s'exécutant en son sein, telles que des applications multi-média, de télécommunication, de contrôle, de sécurité, ...

Il existe des solutions qui offre des bonnes densités de code, telles que les architectures CISC [2] ou les architectures à base de pile. Cependant, ces solutions ont des inconvénients rédhibitoires. La complexité inhérente aux architectures CISC limite la vitesse d'horloge maximale qui peut être obtenue et le micro-codage utilisé dans ces architectures peut augmenter le nombre de cycles nécessaires par instruction. Les architectures à base de pile utilisent fréquemment de la mémoire externe à la puce pour tout ou partie de la mise en oeuvre de la pile de programme, ce qui peut avoir un impact majeur sur les performances. Cette utilisation peut en cas de débordement de la pile avoir de graves et imprévisibles pénalités de performance. Une comparaison plus complète est présentée dans [8].

Une solution pour répondre aux besoins de puissance de calcul est d'utiliser une architecture RISC [3]. Au sein de la société Cortus, il a été développé un processeur RISC pour les systèmes embarqués: l'APS3. Son architecture est de type Harvard, ayant deux chemins séparés pour l'accès aux données et aux instructions. Les instructions ont une taille de 16 bits, avec une possibilité de les prolonger d'une extension de 16 bits lorsque cela est nécessaire. Le processeur APS3 a une superficie de silicium très petite pour une consommation réduite tout en gardant de hautes performances. En technologie UMC 180 nm sa taille est de 8535 portes logiques, sa fréquence de fonctionnement de 188,6 MHz et sa consommation de 38 $\mu W/MHz$.

TAB. 1: comparaison de processeurs

	APS3	Cortex-M0	ARM9	8051
DMIPS/MHZ	1.14 (1.59 w/ hwdiv)	0.9	1.1 (0.9 Thumb)	0.07
Gate Count	8k	12k	117k	5k
Area 180nm	$0.11mm^2$	$0.25mm^2$	$1.1mm^2$	$0.08mm^2$
MAX Freq	175MHZ (wc180nm)		160MHZ (wc180nm)	50MHZ (typ180nm)
	400 MHZ (wc90nm)	270MHZ (typ90nm)		

APS3 vs chiffres publiquement disponibles pour ARM9, Cortex-M0 et le 8051 haute performance

Le problème majeur de l'architecture RISC, est sa consom-

[1]SoC : System on Chip en anglais

[2]Complex Instruction Set Computer

[3]Reduce Instruction Set Computer

mation de mémoire, environ deux fois plus que celle de l'architecture CISC.

Bien que l'APS3 soit déjà performant au niveau de la densité de code grâce à son jeu d'instruction 16/32 bits, un gain non-négligeable (de l'ordre de 20%) peut encore être obtenu en compressant les programmes.

Pour tirer partie de cette méthode il est nécessaire de mettre en oeuvre matériellement au sein du processeur APS3 un décompresseur qui n'aie pas d'impact significatif sur ses performances. Cela implique qu'idéalement, il doit être possible de décompresser une instruction par cycle.

L'approche présentée dans cet article est basée sur une compression à base de dictionnaire afin d'accroitre la densité du code du processeur. Théoriquement, cette méthode fournit un gain plus faible que les méthodes statistiques, mais nos premiers travaux basées sur l'algorithme de Huffman ne nous ont pas permis d'atteindre les performances attendues. Le décompresseur réalisé à l'aide de cet algorithme introduit une latence inacceptable de 2 à 3 cycles par instruction.

Nous avons décidé de mettre en oeuvre une méthode utilisant un dictionnaire de compression afin d'avoir une décompression rapide avec une latence acceptable et une taille raisonnable.
Il existe deux principaux types de compression par dictionnaire:

- la compression d'instructions où une instruction est remplacée par une instruction plus courte.
- La compression de séquences où un indice ou une clé remplace un ensemble d'instructions fréquentes. Théoriquement, cette méthode donne un meilleur gain.

La réussite d'une compression peut être mesurée par le taux de compression qui est le rapport de la taille du programme compressé par la taille du programme d'origine, en prenant en compte dans la taille du fichier compressé la taille du dictionnaire qui est nécessaire pour décompresser le programme. Le taux de compression s'exprime ainsi :

$$CR = \frac{CPS + DS}{OPS}$$

CR est le taux de compression, CPS la taille du programme comprimé, DS la taille du dictionnaire et OPS la taille du programme d'origine.

2 Etat de l'art

Dans la littérature, la compression par dictionnaire pour augmenter la densité de code d'un processeur est utilisée par Netto et al [6] qui ont travaillé sur le processeur leon (sparc8) et Lekatsas et al [4] dans leur travail sur le Xtensa 1040. Liao et Devadas [5] ainsi que Seong et Prabhat [10] ont utilisé en plus de la méthode de compression à base de dictionnaire "classique" des bitmasks pour augmenter la densité du code.

Un exemple de compression de séquences d'instructions a été présenté par Lefurgy [3]. Nous avons également trouvé de la compression à base de dictionnaire dans le monde VLIW [4] dans les travaux de Nam Park, et Kyung [9].

3 Méthode de Compression

L'APS3 est un processeur 16/32 bits avec une forte présence d'instructions 16 bits dans les programmes, ce qui réalise déjà une compression d'instructions. Nous avons donc réalisé une compression matérielle de séquences d'instructions.

3.1 Compression post-link

Dans un premier temps nous avons compressé les exécutables, aprés l'édition de liens, sans prendre en compte les sauts. Cette compression consiste à remplacer des séquences par des instructions de 16 bits contenant un opcode non utilisé suivi de l'index de la séquence dans le dictionnaire. Les séquences présentes dans le dictionnaire sont issues d'une étude statistique qui est faite sur le programme, dans le cas d'une compression dynamique, ou d'un ensemble représentatif de programmes, dans le cas d'une compression statique.

Une fois cette première partie effectuée, nous avons considéré les sauts. En ce qui concerne les saut directs, nous remplaçons les adresses initiales de saut dans ces instructions par les adresses qui leur correspond après compression. Pour les sauts indirects, l'adresse présente dans le registre de saut peut parvenir d'un calcul qui ne prend pas ces racines dans la même fonction, notre méthode consiste à chercher toutes les adresses de sauts indirects et à créer des tableaux de correspondance pour ces adresses.

Pour son implantation, le décompresseur peut être post-cache ou pré-cache. Une implantation pré-cache permet d'avoir plus de flexibilité si le décompresseur n'est pas sur la même puce que le processeur. Une implantation post-cache est souvent moins flexible, par contre elle permet d'avoir des données compressées dans le cache.

Nous avons choisi une implantation post-cache, est avons considéré son lieu d'intégration : implanter la décompression hors du pipeline (extra-pipeline) ou à l'intérieur pipeline (intra-pipeline).

Le premier choix permet de ne pas modifier l'architecture du processeur, mais oblige à gérer deux domaines d'adresses différents en augmentant la taille et la latence du décompresseur.
Le second choix ne pose pas ces problèmes mais modifie l'architecture interne du processeur. Notre but étant d'avoir un processeur ayant la plus haute densité de code possible avec des grandes performances, nous avons opté pour la décompression intra-

[4] Very Long Instruction Word

pipeline présentée dans la figure 1.

Plusieurs travaux considèrent la compression intra-pipeline. Dans Brorsson, Mats and Collin, Mikael [1] il est question de compresser des instructions en se basant sur du profilage. Corliss et al. "DISE" [2] réalise une mise en œuvre assez complexe offrant de bons gain allant jusqu'à 25%, cependant l'impact sur la performance n'a pas été pris en compte dans cette étude. Haluk Ozaktas et Karine Heydemann [7] compressent des séquences d'instructions dans une instruction de 32 bits qui contient un opcode de 6 bits, 2 bits d'informations contenant le nombre de slots utilisé et 3 slots de 8 bits en considérant des suites de séquences compressables ce que nous n'avons pas constaté dans les programmes que nous considérons.

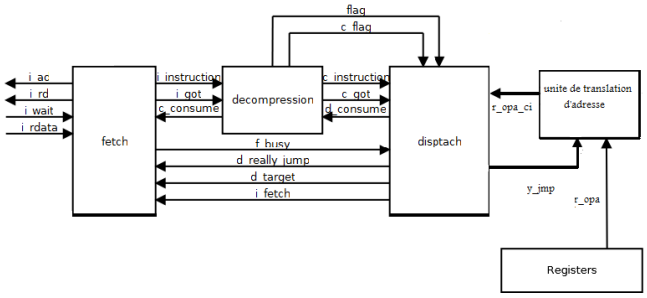

FIG. 1: schéma décompression post-link

3.2 Compression pre-link

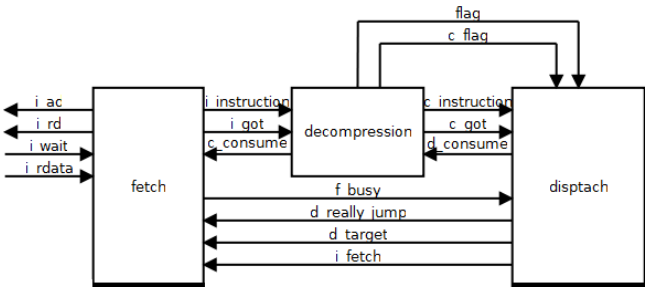

FIG. 2: schéma décompression pre-link

La compression post-link que nous avons réalisée est fonctionnelle, cependant l'unité de translation d'adresse se trouve sur un chemin critique. Nous avons décidé, dans un soucis de performance, d'éliminer cette unité en faisant compression pré-link. Dans ce cas on compresse les différents fichiers objets qui rentrent dans la génération de l'exécutable avant d'appeler l'éditeur de lien, voir la figure 2.

La compression des fichiers objets est un peu plus compliquée que la compression d'un exécutable. Il faut veiller à garder le même format, format elf ici, pour le fichier objet compressé et le fichier objet original.

3.3 Unité de décompression

L'unité de décompression contient un buffer qui a la taille de la plus grande séquence possible, ici 192 bits. On vérifie tout d'abord si c'est une instruction compressée:

- Si l'instruction est compressée on met la séquence correspondante dans le buffer, on génère un signal flag qui va bloquer l'incrémentation du PC [5], à chaque coup d'horloge on décale les instructions présentes dans le buffer en fonction du signal d_consume qui est le signal indiquant le nombre de bits consommés par le bloc decode "0, 16 ou 32 bits", lorsque on transmet la dernière instruction présente dans le buffer on génère le signal c_flag qui va incrémenter le PC.
- Si l'instruction n'est pas compressée on la met dans le buffer sans générer de flag, ce qui fait que le décompresseur agit comme un transmetteur.

3.4 Test

Après avoir testé et validé le fonctionnement de notre décompresseur avec des programmes simples. Nous avons decidé de passer les tests de régressions de gcc afin de s'assurer que tous les programmes généré avec notre système de compression peuvent être exécuté par le processeur équipé de notre decompresseur.

L'APS3 est un des seuls processeurs à notre connaissance à passer tous les tests de régressions de gcc version 4.5 [6].

En ce qui concerne nos travaux, les résultats sont identiques avant et après l'insertion de l'unité de décompression ce qui démontre la transparence de notre unité de compression/décompression pour les programmes.

3.5 Impact sur les performances

L'ajout de notre décompresseur a un coût, du à l'ajout d'un étage de pipeline. La décompression en elle même est faite en temps réel ce qui n'ajoute pas de surcoût en terme de cycle.

Lors des tests de régressions un surplus de 33% de temps apparaît en présence d'aléas de contrôle lié aux branchements, du au gel du pipeline.

Toutefois ces test de régressions ne sont pas vraiment révélateurs, la répartition des sauts n'étant pas représentative des programmes embarqués.

Pour confronter notre solution à la réalité, nous avons utilisé, comme benchmarks, des programmes susceptibles d'être exécuté sur l'APS3 et avons trouvé une augmentation du temps d'exécution de 21% en moyenne lors d'aléas de contrôle.

[5] program counter
[6] http://gcc.gnu.org/gcc-4.5/buildstat.html

Avec certains programmes, tel que le programme de cryptographie "aes" le surplus de temps necessaire à l'execution est même de 0,31%. Ceci pouvant être expliqué par le fait que si nous compressons des grandes séquences ce qui est le cas ici on diminue les accès mémoire ce qui compense les cycles perdus lors des sauts.

Concernant la taille, en UMC 90nm avec une fréquence de 166 MHZ le decompresseur fais 2595 portes($18166\mu m^2$).

4 Conclusion

Dans cet article nous avons présenté un système de compression/décompression entièrement fonctionnel et transparent avec un gain moyen de 15% de densité de code. Notre mise en oeuvre est totalement compatible avec une technologie ASIC. Notre décompresseur valide l'intégralité des tests de régressions de gcc. L'innovation apportée par notre méthode de compression pré-link garde la compatibilité avec le format elf et facilite l'adaptation du reste de chaine de développement.

References

[1] Mats Brorsson and Mikael Collin. Adaptive and flexible dictionary code compression for embedded applications, 2006.

[2] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. A dise implementation of dynamic code decompression. *SIGPLAN Not.*, 38:232–243, June 2003.

[3] Charles Lefurgy, Peter Bird, I cheng Chen, and Trevor Mudge. *Improving Code Density Using Compression Techniques*. MICRO 30. IEEE Computer Society, 1997.

[4] Haris Lekatsas, J? Henkel, and Venkata Jakkula. Design of an one-cycle decompression hardware for performance increase in embedded systems. *Design Automation Conference*, 0:34, 2002.

[5] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steve Tjiang, and Albert Wang. Code optimization techniques for embedded dsp microprocessors, 1995.

[6] E. Wanderley Netto, R. Azevedo, P. Centoducatte, and G. Araujo. Multi-profile based code compression. *Design Automation Conference*, 0:244–249, 2004.

[7] Haluk Ozaktas and Karine Heydemann. Compression de code pour processeurs haute-performance. 2009.

[8] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[9] In-Cheol Park Sang-Joon Nam and Chong-Min Kyung. improving dictionnary based code compression vliw.

[10] Seok-Won Seong and Prabhat Mishra. An efficient code compression technique using application-aware bitmask and dictionary selection methods, 2007.