Compilation et faible consommation

Olivier Zendra

Chronique: Compilation et faible consommation

Olivier Zendra

INRIA-Lorraine / LORIA
615 Rue du Jardin Botanique, BP 101,
54602 VILLERS-LES-NANCY Cedex, FRANCE
Olivier.Zendra@loria.fr

RÉSUMÉ. *Dans les systèmes embarqués autonomes, de plus en plus présents dans notre vie quotidienne, la gestion de l'énergie est un problème capital. Si ce domaine est étudié depuis longtemps au niveau du matériel, il est en revanche moins exploré en ce qui concerne le logiciel et notamment le support au niveau du compilateur, où, historiquement, les optimisations ont plutôt concerné la vitesse. Cet article a donc pour but de faire un survol de l'état de l'art des techniques de compilation orientée faible consommation énergétique. Il présente brièvement des techniques bas niveau, puis plus largement des techniques de plus haut niveau liées notamment à la gestion mémoire et aux modes de fonctionnement des ressources, avant de dégager les perspectives du domaine.*

ABSTRACT. *In autonomous embedded systems — which become more and more present in our daily life — energy management is a crucial issue. Although it has been widely studied from a hardware point of view, this domain remains less explored in software, especially with respect to compiler support where historically optimizations generally dealt with speed. This paper thus aims at providing a survey of low-power and low-energy compilation techniques. It briefly introduces low-level techniques, then presents more largely higher-level techniques, especially those pertaining to memory management and operation modes of resources, before drawing the domain perspectives.*

MOTS-CLÉS : *compilation, optimisation, énergie, puissance, consommation, coalescing, boucle, mémoire, scratch-pad, cache, registre, DVS, DFS, hibernation*

KEYWORDS: *compilation, optimization, energy, power, consumption, coalescing, loop, memoiry, scratch-pad, cache, register, DVS, DFS, hibernation*

1. Introduction et problématique

Dans les systèmes embarqués autonomes, qui sont de plus en plus présents dans notre vie quotidienne, la gestion de l'énergie disponible est un problème capital. Si ce domaine est étudié depuis longtemps au niveau du matériel, il est en revanche moins exploré en ce qui concerne le logiciel et notamment le support au niveau du compilateur, où, historiquement, les optimisations ont plutôt concerné la vitesse.

Ces dernières peuvent présenter des points communs avec les optimisations pour la puissance ou l'énergie, mais ne sont pas forcément les mêmes. Par exemple, déplacer des instructions sur le chemin non critique (temporellement) fait gagner du temps, mais pas nécessairement de l'énergie. Les techniques d'optimisations classiques initialement conçues pour la vitesse doivent donc être réévaluées dans le contexte de l'énergie.

Ces optimisations en vitesse sont souvent meilleures du point de vue de l'énergie quand elles sont faites au niveau du compilateur plutôt qu'au niveau du matériel. En effet, dans le premier cas, la logique de l'optimisation est en partie faite à la compilation, sans coût d'exécution, alors que dans le matériel, cette logique se retrouve dans des circuits qui consomment de l'énergie, même s'ils ne prennent pas de temps car sont hors du chemin critique (Graybill *et al.*, 2002).

De plus, les compilateurs (hors ligne) permettent de dédier beaucoup de ressources (temps, CPU, mémoire) à des optimisations, ce qui est impossible dans le matériel. Ceci autorise la prise en compte d'un contexte beaucoup plus fourni. Les compilateurs peuvent ainsi "connaître le futur" d'un programme, via des analyses puissantes (analyses globales). Cependant, les compilateurs manquent d'informations sur le comportement exact des programmes lors de l'exécution, ce qui peut nuire à la qualité du code. En revanche, un environnement d'exécution de type machine virtuelle optimisante peut concilier les avantages des deux mondes. Le problème en est la consommation de ressources.

Enfin, notons qu'optimiser pour diminuer l'énergie n'est pas tout à fait équivalent à optimiser pour diminuer la densité de puissance (et donc les points chauds).

Malgré sa relative jeunesse, ce domaine est suffisamment vaste pour qu'il soit impossible de mentionner toutes les techniques et travaux effectués. Cet article vise donc à fournir un survol de solutions qui existent dans le domaine de la compilation de code optimisé pour la basse consommation énergétique.

La section 2 présente quelques optimisations bas niveau et leur impact en terme d'énergie. La section 3 aborde les optimisations de plus haut niveau liées à la gestion de la mémoire. Celles liées aux modes de fonctionnement des ressources sont présentées en section 4. Enfin, la section 5 conclut et présente les perspectives du domaine.

2. Optimisations bas niveau

Les optimisations bas niveau sont anciennes, nombreuses et d'un usage commun dans les compilateurs. Certaines peuvent avoir un effet bénéfique, quoiqu'en général relativement limité, sur la consommation énergétique. Notons qu'elles sont souvent très dépendantes du processeur. Nous en présentons ici quelques unes.

2.1. *Énergie de transition et commutation (switching activity)*

En réordonnant les instructions, il est possible de diminuer les commutations entre les instructions successives, ce qui diminue la puissance consommée (Graybill *et al.*, 2002). De même, des encodages judicieux des données (voire des instructions) permettent de diminuer l'énergie consommée.

On peut par exemple économiser l'énergie dûe aux changements dans les champs registres des instructions, en effectuant, après compilation et allocation de registres, une passe de renommage de registres (Kandemir *et al.*, 2000). Ceci résulte en une réduction de 11% de l'activité de commutation sur le champ registre de l'instruction (Woo *et al.*, 2001).

2.2. *Dépliage de boucles*

Le *loop unrolling* permet de diminuer le surcoût dû au contrôle de boucle et donc d'exécuter globalement moins d'instructions. Cette optimisation classique pour la vitesse se traduit donc aussi en des gains d'énergie, y compris au niveau du coeur d'exécution (ce qui est très positif pour y éviter les points chauds). Bien entendu, il convient d'être vigilant à l'accroissement de la taille du code, qui augmente la consommation et peut ne pas être acceptable dans un système embarqué.

3. Optimisations liées à la mémoire

Les optimisations de haut niveau ont souvent un impact bien plus fort que les optimisations bas niveau. Les optimisations de compilation liées à la mémoire sont potentiellement très efficaces du point de vue de la consommation.

3.1. *Compactage*

La fusion de variables (*variable coalescing*), qui fait coexister dans un même emplacement mémoire plusieurs (petites) données permet d'utiliser moins de place, ce qui diminue la consommation énergétique (Zhuang *et al.*, 2003). De façon semblable, l'analyse de la durée de vie des données permet le partage d'un même emplacement entre des données qui ne coexistent pas temporellement.

A plus large échelle, la compression des données (Zhang *et al.*, 2002), bien que coûtant de l'énergie et du temps à l'exécution, permet également de diminuer la taille de la mémoire active et facilite donc la mise au repos de bancs mémoire non utilisés. Bien entendu, le surcoût de la (dé)compression doit être limité et mis en balance avec le gain de place et d'énergie. Cette technique est particulièrement appropriée pour des données à longue durée de vie mais accédées peu fréquemment.

3.2. *Mémoires scratch-pad*

Les caches aident à améliorer la vitesse des programmes. Cependant, ils ne sont pas très adaptés aux systèmes embarqués : ils augmentent la taille du système et sa consommation énergétique (avec la zone cache et sa logique de gestion) et sont peu prédictibles, ce qui est gênant pour les systèmes temps réel. Une alternative intéressante est la *scratch-pad memory*, petite zone mémoire rapide (SRAM) gérée directement par le compilateur, et non pas par le matériel. Ses avantages sont nombreux : taille plus faible de 34% par rapport à un cache, consommation énergétique moindre de 40%, bornes temps réel plus précises (Banakar *et al.*, 2002). Le compilateur cherche donc à gérer cette zone en y plaçant les données les plus fréquemment accédées.

Diverses approches existent pour gérer les *scratch-pads*. Les approches statiques (Avissar *et al.*, 2002), dont la gestion (choix de placement) est faite à la compilation, offrent de bonnes performances et de bonnes caractéristiques temps réel.

Les approches dynamiques prennent en compte des situations plus complexes (par exemple deux boucles successives sur des tableaux trop gros pour être placés en même temps en *scratch-pad*, mais qui peuvent l'être successivement). En revanche, elles sont plus difficiles à prendre en compte dans le cadre du temps réel. Elles ont aussi un surcoût (temps et énergie) non négligeable à l'exécution, qui vient notamment des coûts de transfert des données entre *scratch-pad* et mémoire principale décidés par le compilateur. Ce dernier effectue ses choix en fonction du coût des transferts et de la fréquence future d'utilisation des données, que ne connait pas un cache matériel. Le compilateur peut ainsi plus efficacement placer les bonnes données en mémoire rapide, et peut aussi réutiliser immédiatement la mémoire rapide occupée par une donnée qui ne sera plus utilisée. Pour éviter les surcoûts dûs aux transferts de données entre mémoire principale et *scratch-pad*, il est intéressant que le compilateur alloue directement en SRAM certaines parties du tas. Les gains en énergie rapportés dans (Dominguez *et al.*, 2005) sont d'environ 40% par rapport à un placement en mémoire.

3.3. *Fenêtres de registres*

Une technique d'optimisation classique pour améliorer les performances est la fenêtre de registres. Elle permet d'allouer plus de registres virtuels qu'il n'y a de registres réels, les registres virtuels étant répartis en plusieurs ensembles dont un seul est actif à un instant donné.

Ceci permet de réduire sensiblement le débordement en mémoire centrale par manque de registres (*register spill*), au prix d'un léger surcoût de maintenance et de changement des fenêtres de registres. Travailler en registres plutôt qu'en mémoire améliore notablement la vitesse, en évitant des transferts. Pour la même raison, ceci a également des effets positifs en terme d'énergie. Les résultats de (Ravindran *et al.*, 2005) font apparaître une économie d'énergie atteignant 25%.

4. Modes et ressources

Les optimisations qui jouent sur les modes de marche des diverses ressources (processeur, mémoire, disque...) sont parmi les plus répandues et les plus efficaces en terme d'économies d'énergie.

4.1. *Gestion dynamique du voltage et de la fréquence (DVS/DFS)*

Le DVS (et son pendant le DFS) offre des opportunités significatives en terme d'optimisation d'énergie et de puissance, même dans les programmes assez fortement optimisés pour la vitesse, et cela avec peu ou pas de perte de performances, puisque puissance et énergie sont proportionnels à $V^2 f$. Ceci explique que cette technique soit très courante et bien étudiée (Xie *et al.*, 2004).

Le DVS est souvent lié à l'ordonnancement, qui permet d'en tirer tout le parti, ou au moins à la fourniture d'information de profil d'applications, pour indiquer au compilateur des phases de l'exécution où la fréquence (donc la tension) peut être diminuée. Bien entendu, les contraintes temps réel souples, en tolérant une légère diminution de vitesse, permettent des gains en énergie bien plus importants que les contraintes temps réel dures.

Le DVS permet aussi d'économiser de l'énergie en tenant compte du parallélisme entre calculs et accès de données. Le compilateur peut en effet diminuer la fréquence du processeur plutôt que de laisser le programme en attente de données fournies par des accès mémoires lents.

Enfin, le DVS aide à maîtriser les pics de puissance en donnant plus de temps et une moindre fréquence aux tâches dont l'exécution induit une puissance trop élevée.

4.2. *Mise en hibernation dynamique des ressources*

La mise en hibernation dynamique des ressources matérielles (ou plus largement l'exploitation de leurs modes basse énergie) permet des gains en énergie considérables. Par exemple, (Hom *et al.*, 2001) montre que prédire les événements de *swapping* mémoire permet de n'activer la ressource pour le *swap* (dans leur cas une carte réseau sans fil) que lorsque c'est nécessaire, et amène une économie d'énergie de 20%.

La gestion directe par l'application compilée plutôt que par le système d'exploitation (OS) est beaucoup plus efficace. Un OS doit détecter *a posteriori* une phase peu active avant de passer en mode oisif, puis de passer encore plus tard dans un mode d'hibernation plus profonde. Le compilateur, lui, est capable de prédire le futur (inactif) d'une ressource et peut donc la mettre en hibernation profonde sans délai.

4.3. *Réordonnancement des accès pour améliorer la localité*

Le compilateur peut transformer un code source de façon à augmenter les périodes pendant lesquelles une ressource n'est pas utilisée, ce qui améliore encore l'économie d'énergie via les modes basse énergie.

Il est par exemple judicieux de grouper les accès disques, pour augmenter les périodes d'hibernation basse énergie. Ceci divise également le coût énergétique du disque par un plus grand nombre d'accès, d'où un coût par accès plus faible.

De même, l'amélioration de la localité des variables de types scalaires fournit, en plus d'une vitesse accrue, des opportunités supplémentaires pour mettre au repos des zones mémoires (*cf.* section 3) en regroupant les accès mémoire dans un même banc mémoire (-10% d'énergie selon (Graybill *et al.*, 2002, p. 208)).

L'entrelacement de tableaux en est un cas particulier. Il consiste à entrelacer dans la mémoire les éléments de plusieurs tableaux de façon à ce que les indices accédés en même temps se trouvent contigus dans une même zone mémoire. Cela permet de mettre au repos plus facilement les zones mémoires qui contiennent le reste des tableaux, d'où un gain en énergie supplémentaire dû au seul entrelacement de 54% sur un système sans cache et 15% avec cache, sur les tests de (Delaluz *et al.*, 2002).

La division de boucles (*loop fission*) consiste à diviser une boucle en plusieurs boucles plus simples consécutives, ce qui améliore la localité des caches et le parallélisme d'instructions. Du point de vue de l'énergie, il peut être rentable d'utiliser cette technique, notamment lorsque plusieurs tableaux sont utilisés dans la boucle initiale. Son découpage en plusieurs boucles travaillant chacune sur un seul tableau autorise en effet la mise en repos des zones mémoires contenant les autres tableaux. Selon (Graybill *et al.*, 2002, ch. 10), cette technique diminue sensiblement l'énergie de la boucle la plus coûteuse, tout en augmentant un peu celle des autres boucles .

4.4. *Optimisation des ressources inter-programmes*

Un résultat se mesure sur un système complet, pas seulement sur les points ou composants qu'on optimise. Il est donc judicieux de cibler des optimisations non pas pour un seul programme, comme le fait en général un compilateur, mais pour l'ensemble des programmes (tâches) présents sur un système, et cela d'autant plus s'ils utilisent les mêmes ressources.

Par exemple, (Hom *et al.*, 2005) propose une approche de compilation inter-programmes appliquée à la réduction de la consommation du disque dur. En utilisant un tampon mémoire pour le disque dont la taille dépend des différents programmes et en groupant leurs accès disque, la consommation diminue de 7 à 49% additionnels par rapport aux programmes optimisés isolément. L'intérêt en est donc considérable.

5. Conclusion et perspectives

Cette chronique a présenté un panorama rapide et non exhaustif de techniques de compilation pouvant servir à la génération de code à basse consommation énergétique.

De nombreuses solutions existent, à des niveaux extrêmement différents. Il est clair qu'aucune de ces solutions ne suffit à elle seule à résoudre le problème de la basse consommation en général. Il est donc nécessaire, en plus de l'amélioration des techniques existantes et du développement de nouvelles, d'étudier les interactions entre ces techniques basse consommation, ainsi qu'entre elles et les optimisations "classiques" qui touchent à la vitesse. Bien entendu, il est également crucial d'étudier les interactions avec le système matériel sous-jacent.

Une tendance future probable est l'implication plus importante des compilateurs dans le problème de la gestion de l'énergie, en interaction avec le matériel. En effet, la complexité peut être déportée dans le compilateur, où des algorithmes et analyses plus puissants mais coûteux peuvent être employés lors de la compilation.

Pour cela, les concepteurs de compilateurs ont besoin de meilleurs modèles de performance / énergie / puissance ainsi que d'architectures plus prédictibles et facilitant les mesures physiques, afin de pouvoir analyser quantitativement l'impact de différentes optimisations du compilateur. Il est également souhaitable d'avoir des interfaces standardisées (support au niveau ISA) permettant aux applications compilées d'interagir avec le système pour gérer "directement" certaines ressources. Il serait ainsi possible de concevoir des "co-optimisations" où le compilateur commencerait et où le matériel finirait (éventuellement avec des informations fournies par le compilateur) les optimisations le plus judicieusement.

Une seconde tendance est que les opérations mémoire représentent une part de plus en plus dominante de la consommation énergétique : 70 à 90% en 2010 (ITRS, 2005). La gestion mémoire faible consommation devient donc un facteur crucial et devrait donner lieu à de nombreuses recherches. Dans cette optique, des solutions comme les *scratch-pad* semblent extrêmement prometteuses, ainsi qu'à plus large échelle les ramasse-miettes basse énergie (*energy-aware garbage collectors*).

Enfin, on peut noter que les processeurs VLIW offent un potentiel considérable pour le parallélisme au niveau des instructions, ce qui peut être très efficace en terme d'énergie, pourvu que le compilateur puisse trouver ce parallélisme lors de la compilation. Un gros travail est donc dévolu au compilateur avec ce type de processeurs. Il s'agit d'un domaine encore largement à explorer.

6. Bibliographie

Avissar O., Barua R., Stewart D., « An optimal memory allocation scheme for scratch-pad-based embedded systems », *Transaction. on Embedded Computing Systems.*, vol. 1, n° 1, p. 6-26, 2002.

Banakar R., Steinke S., Lee B.-S., Balakrishnan M., Marwedel P., « Scratchpad memory : design alternative for cache on-chip memory in embedded systems », *Proceedings of the tenth international symposium on Hardware/software codesign (CODES'02)*, ACM Press, New York, NY, USA, p. 73-78, 2002.

Delaluz V., Kandemir M., Vijaykrishnan N., Irwin M. J., Sivasubramaniam A., Kolcu I., « Compiler-Directed Array Interleaving for Reducing Energy in Multi-Bank Memories », *Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design (ASP-DAC'02)*, IEEE Computer Society, Washington, DC, USA, p. 288, 2002.

Dominguez A., Udayakumaran S., Barua R., « Heap Data Allocation to Scratch-Pad Memory in Embedded Systems. », *Journal of Embedded Computing (JEC)*, 2005.

Graybill R., Melhem R., *Power aware computing*, Kluwer Academic Publishers, Norwell, MA, USA, 2002.

Hom J., Kremer U., « Energy Management of Virtual Memory on Diskless Devices. », *Workshop on Compilers and Operating Systems for Low Power (COLP'01)*, Barcelone, Espagne, septembre, 2001.

Hom J., Kremer U., « Inter-program optimizations for conserving disk energy », *Proceedings of the 2005 international symposium on Low power electronics and design (ISLPED'05)*, ACM Press, New York, NY, USA, p. 335-338, 2005.

ITRS, « International Technology Roadmap for Semiconductors », 2005. http ://public.itrs.net.

Kandemir M., Vijaykrishnan N., Irwin M., Ye W., Demirkiran I., « Register relabeling : A post compilation technique for energy reduction. », *Workshop on Compilers and Operating Systems for Low Power (COLP'00)*, Philadelphie, PA, USA, octobre, 2000.

Ravindran R. A., Senger R. M., Marsman E. D., Dasika G. S., Guthaus M. R., Mahlke S. A., Brown R. B., « Partitioning Variables across Register Windows to Reduce Spill Code in a Low-Power Processor », *IEEE Transaction on Computers*, vol. 54, n° 8, p. 998-1012, 2005.

Woo S., Yoon J., Kim J., « Low-Power Instruction Encoding Techniques », *SOC Design Conference*, 2001.

Xie F., Martonosi M., Malik S., « Intraprogram dynamic voltage scaling : Bounding opportunities with analytic modeling », *ACM Transactions on Architure and Code Optimization (TACO)*, vol. 1, n° 3, p. 323-367, 2004.

Zhang Y., Gupta R., « Data Compression Transformations for Dynamically Allocated Data Structures », *Proceedings of the 11th International Conference on Compiler Construction (CC'02), Lecture Notes in Computer Science*, vol. 2304, Springer-Verlag, London, UK, p. 14-28, 2002.

Zhuang X., Lau C., Pande S., « Storage assignment optimizations through variable coalescence for embedded processors », *LCTES '03 : Proceedings of the 2003 ACM SIGPLAN conference on Language, Compiler, and Tool for Embedded Systems*, ACM Press, New York, NY, USA, p. 220-231, 2003.