HAL
open science

LoRA: a Package for Loop Optimal Register Allocation

Christine Eisenbeis, Sylvain Lelait

To cite this version:

Christine Eisenbeis, Sylvain Lelait. LoRA: a Package for Loop Optimal Register Allocation. [Research Report] RR-3709, INRIA. 1999. inria-00072959

HAL Id: inria-00072959
https://inria.hal.science/inria-00072959v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

LoRA: a Package for Loop Optimal Register Allocation

Christine Eisenbeis , Sylvain Lelait

N ° 3709

Juin 1999

THÈME 1

Rapport de recherche

ISSN 0249-6399

LoRA: a Package for Loop Optimal Register Allocation

Christine Eisenbeis* , Sylvain Lelait†

Thème 1 — Réseaux et systèmes
Projet A3

Rapport de recherche n ° 3709 — Juin 1999 — 23 pages

Abstract: Instruction-level code parallelization increases the register pressure and renders the register allocation phase crucial. In the case of software pipelined loops, unrolling has to be performed when variables are alive during more than one iteration resulting in code size increases. Loop unrolling also influences the register pressure. LoRA is a package that implements several algorithms for trading the register pressure against code size. In LoRA either the register pressure or the unrolling degree can be constrained. We explain the different strategies used in LoRA and show experimental results on a large benchmark of loops. Our experiments show that in concrete cases the unrolling degree can be kept reasonable although the worst case is exponential in the number of registers thought.

Key-words: register allocation, loop optimisation, cache management, DSP processor

(Résumé : tsvp)

This work was partially supported by the Esprit Project #5399 COMPARE and by a Lise-Meitner Stipendium from the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung).

* Christine.Eisenbeis@inria.fr
† Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria. E-mail: sylvain@complang.tuwien.ac.at

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

LoRA : un outil pour une allocation de registres optimale dans les boucles

Résumé : La parallélisation du code au niveau instruction accroît les besoins en registres et rend cruciale la phase d'allocation de registres. Dans le cas de boucles pipelinées logiciellement, le déroulage doit être appliqué lorsque des variables sont en vie pendant plus d'une itération, ce qui accroît la taille du code. La déroulage de boucles influence aussi les besoins en registres. LoRA est un outil qui contient plusieurs algorithmes pour mettre en balance les besoins en registres et la taille du code. Dans LoRA les besoins en registres ou le degré de déroulage peut être contraint. Nous expliquons différentes stratégies utilisées dans LoRA, et nous montrons des résultats expérimentaux sur un large choix de boucles. Nos expériences montrent que dans des cas réels le degré de déroulage peut être conservé raisonnable bien que le pire des cas soit exponentiel en terme du nombre de registres requis.

Mots-clé : allocation de registres, optimisation de boucles, gestion de cache, processeurs DSP

1 Introduction

In order to exploit instruction-level parallelism by compilers, numerous methods have been developed. For loop compilation, which is a major source of performance improvement, new scheduling techniques known as software pipelining have been proposed [10, 20, 14, 12]. Another technique which can improve code performances is loop unrolling. Apart of increasing the size of basic blocks that can then be more easily parallelized, loop unrolling also improves in most cases the register allocation by reducing the number of registers required [6, 7]. All these techniques are very efficient for general-purpose processors and can drastically improve code performance. However increasing the code size can have severe consequences on the overall performance. In the case of instruction cache this can cause unnecessary misses. This problem is exacerbated in embedded applications where the size of the program memory is constrained.

This paper presents our loop register allocation tool called LoRA that implements several algorithms for combining register allocation and loop unrolling for simple loops without branches.

Usually the register allocation is performed by coloring the interference graph G of some loop [3]. The chromatic number $\chi(G)$ of the interference graph is the least number of registers required for allocating the loop variables in registers when storing each instance of the same variable into the same register. Now we consider the interference graph G^u of the same loop unrolled u times and compute its chromatic number $\chi(G^u)$[1]. In most cases the variations of $\chi(G^u)$ as a function of u are chaotic [7]. But we know that we can always find an unrolling degree u such that the number of registers required is equal to a minimum R_{min}, given by the maximum number of registers simultaneously alive (also called $MaxLive$ in some work [12]). LoRA implements several heuristics for finding as small a u as possible under fixed register constraints (also when the number of available registers is equal to the lower bound R_{min}). LoRA also implements a heuristic for minimizing the number of registers used under a code size constraint. The minimum constraint on the code size is that the loop size is greater than every variable lifetime. All these heuristics are explained in the next section.

LoRA can be easily interfaced with any loop code optimizer. We present in the second section experiments conducted in the MOST environment [1] on a large set of loops from various benchmarks. These experiments exhibit the different behaviors of the heuristics implemented in LoRA. They also show that the theoretical upper bound on the minimum unrolling degree required for allocating with R registers — $e^{(1+O(1))\sqrt{R \ln R}}$ [15] — can be kept under control in general.

[1] LoRA does not compute the chromatic number of the interference graph of the loop – it is a NP-complete problem [9].

2 LoRA

Nine heuristics are implemented in LoRA. Four deal with the meeting graph introduced in [7], two deal with the work of [6], two are a combination of these works. The last one is a combination of [14] and [11]. Heuristics 1, 3, 6 and 8 try to find an allocation with R_{min} registers by unrolling the loop. Heuristics 2, 4, 7 and 9 try to find an allocation with R registers by also unrolling the loop if necessary. These heuristics find an unrolling degree under register constraints. In heuristic 5, the loop is unrolled following the bound of Lam [14], and the fat cover heuristic of Hendren et al [11]. is used to allocate the unrolled loop. This last heuristic limits first the code size and then tries to find an allocation with as few registers as possible.

1. MTG_MINR Minimize the unrolling degree and ensure an allocation with R_{min} registers with the Meeting Graph heuristic.
2. MTG_MINU Minimize the unrolling degree and ensure an allocation with $R < R_{given}$ registers with the Meeting Graph heuristic.
3. EJL_MINR Minimize the unrolling degree and ensure an allocation with R_{min} registers with Eisenbeis' heuristic.
4. EJL_MINU Minimize the unrolling degree and ensure an allocation with $R < R_{given}$ registers with Eisenbeis' heuristic.
5. LAM_HENDREN Unroll the minimum and make the allocation trying to minimize the number of registers with Lam's bound for unrolling and the heuristic of Hendren et al. for the register allocation.
6. MTGEJL_MINR Minimize the unrolling degree and ensure an allocation with R_{min} registers with the Meeting Graph heuristic for the unrolling bound and the heuristic of Eisenbeis et al. for computing the unrolling degree.
7. MTGEJL_MINU Minimize the unrolling degree and ensure an allocation with $R < R_{given}$ registers with the Meeting Graph heuristic for the unrolling bound and the heuristic of Eisenbeis et al. for computing the unrolling degree.
8. DUAL_MINR Minimize the unrolling degree and ensure an allocation with R_{min} registers with the heuristic of Dual Chords.
9. DUAL_MINU Minimize the unrolling degree and ensure an allocation with $R < R_{given}$ registers with the heuristic of Dual Chords.

Furthermore, LoRA can handle different types of registers. The user just has to specify for each variable to what kind of register file it should be allocated. In such a case, LoRA performs the register allocation by unrolling the loop u times with $u = lcm(u_1, \ldots u_n)$, where u_i is the unrolling degree computed for the ith register set.

We first explain the input and output of the procedure which is the interface of LoRA, and then we detail the heuristics that are used.

2.1 Input/Output

The input of the procedure which is the interface of LoRA has been designed to be used as simple as possible to use. It requires only :

- an integer matrix which must contain the number of cycles of the loop[2], the number of variables and for each variable: its start cycle, its end cycle and the register file to which it belongs. The cycles can be given modulo II or not.
- an array giving the total number of registers in each register file
- an array giving the number of registers available in each register file (for heuristics 2, 4, 7 and 9)
- an array giving the number of cycles a variable is really used during its lifetime. It is used for spilling purposes.

Spilling information is available using a variant of heuristic 1 that computes a critical cycle in the meeting graph. LoRA returns a list of variables to spill in order to enable the register allocation. It can give good results as it respects the requirements given in [18].

As output LoRA returns the integer matrix which is modified as follows. First if the loop is unrolled, the matrix is extended to contain all the variables of the new loop. LoRA gives the final unrolling degree of the loop and the final number of variables, and for each variables it returns its start and end cycles and the register to which it has been assigned. The start and end cycles are modulo II this time. The number of registers effectively used for each kind of register file is also given in the first array. R_{min} of each register file is given in the second array.

2.2 The Meeting Graph Heuristic

The meeting graph [7] is used in heuristics 1 and 2 among others. It is a more accurate graph than the usual interference graph, as it has information on the number of cycles of each variable lifetime and on the succession of the lifetimes all along the loop. It allows to compute an unrolling degree which enables an allocation with R_{min} registers of the loop. A meeting graph can have several connected components of weights $r_1, \ldots, r_n$ (if there is only one connected component, its weight is R_{min}), this leads to the upper bound of unrolling $u_{max} = lcm(r_1, \ldots r_n)$ (R_{min} if there is only one connected component). Moreover a possible lower bound is computed by decomposing the graph into as many circuits as possible and then computing the *lcm* of their weights. The circuits are then used to compute the final allocation. This method can handle variables that are alive during several iterations. This heuristic always finds an allocation with an optimal number of registers.

The main drawback of this method is that some nodes may be introduced in order to build the graph, as it can only be used for interval families of constant width. This means

[2]Noted II for Initiation Interval in the sequel of this paper.

that there must always be the same number of live variables at any cycle. So the computing time can be dramatically high in some cases during the decomposition of the graph.

Example 1 Here is a small example, in Figure 1 an interval family representing variable lifetimes is drawn. From these lifetimes, the corresponding meeting graph is drawn. The interval family has $R_{min} = 4$, this means that one needs at least 4 registers (colors) to be allocated successfully. On each node, a weight equal to the number of cycles of the lifetime is added. The weight of the connected component is $\frac{20}{5} = 4$, as $II = 5$ and the sum of the nodes' weights is 20.

Moreover it has 4 chords, leading to 2 ways to decompose it. One way leads to an unrolling on $lcm(1,3) = 3$ iterations and the second to an unrolling on $lcm(2,2) = 2$ iterations. Here $u_{max} = 4$, we have only one connected component. The way the decomposition is found relies on the search of the greatest number of chords that "do not intersect" inside the graph. This search is equivalent ot look for the maximum stable set in the circle graph induced by the chords.

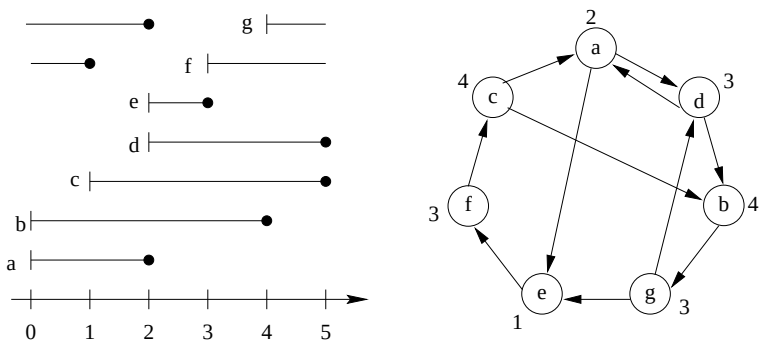

Figure 1: Example of meeting graph

2.3 The Heuristic of Eisenbeis et al.

This method [6] is used in heuristics 3 and 4. If we try to allocate a loop straightforward, we will have problems with variables which are alive across iterations. This heuristic first allocates the lifetimes straightforwardly and then computes the permutation needed on this allocation to get a valid one. The unrolling degree of the loop is the order of this permutation. It can handle loops in which some variable lifetimes last more than one iteration and, as with the previous method, the register allocation is always optimal.

This heuristic does not have high computing time like the meeting graph heuristic can have, but its disadvantage is that the upper bound of unrolling is much higher. As we have said before we have $u_{max} = e^{(1+O(1))\sqrt{R \ln R}}$.

2.4 A Combination of the Meeting Graph Heuristic and Eisenbeis et al.

The heuristics 6 and 7 were made to take into account the upper bound of unrolling of the meeting graph heuristic and take advantage of the speed of the heuristic of Eisenbeis. Our experimental results show that the upper bound is very effective.

It first computes an upper bound of unrolling based on the meeting graph. We have then $u_{max} = lcm(r_1, \ldots r_n)$. And it calls the heuristic of Eisenbeis et al., if the unrolling degree found is greater than u_{max}, then the interval family is unrolled u_{max} times and colored accordingly.

2.5 The Dual Chords Heuristic

These heuristics 8 and 9 are a variant of the meeting graph heuristic. Instead of looking for a global set of chords that allow to decompose the meeting graph, this method looks iteratively for pairs of chords that:

- leave the greatest number of chords in the graph, when it is decomposed according to them
- minimize the lcm, if several pairs satisfy the first condition

Hence a pair of chords is chosen at each iteration, the graph is decomposed to them. And we continue to iterate this process until no chord remains in all the circuits created this way. This heuristic is greedy, but gives also good results.

2.6 A Combination of Lam and Hendren et al.

We designed heuristic 5 by first using Lam's heuristic [14] which computes an unrolling degree to get a loop in which no variable is alive during more than one iteration, and then applying the register allocation for loops of Hendren et al. [11], which simply colors the interference graph induced by the unrolled loop. This register allocation heuristic can not handle lifetimes which last longer than one iteration. This heuristic is more concerned with limiting the code size rather than finding an optimal register allocation. Nevertheless it should be noted that the coloring heuristic of Hendren et al. is very efficient and frequently succeed in finding an allocation with R_{min} registers.

The unrolling degree computed is optimal in the sense that it is the smallest degree which enables the register allocation of the loop. It is equal to $\lceil \frac{l_{max}}{II} \rceil$, where l_{max} is the number of cycles of the longest lifetime of the loop.

3 Experimental Results

We used MOST, a test-bed developed at McGill University [1]. It implements several modulo scheduling algorithms. We present here the results obtained with the heuristic of Gasperoni

and Schwiegelshohn [10], noted Gasperoni, and with DESP [20]. The benchmarks used here are the Livermore loops, Linpack and Nas. In all the figures, MTG denotes the meeting graph heuristic, EJL the heuristic of Eisenbeis et al. and Lam the combination of the heuristics of Hendren et al. and Lam. In our tests, we always considered one register file, as we wanted to have comparison between algorithms and not simply test a particular one.

In this section we present results on the size of the final loop code when it is unrolled. Then we have also results on the unrolling degree itself, including a comparison between the heuristics working with the meeting graph, and finally some results on the number of registers used to allocate the loops.

3.1 Code Size Constraints

We simulated the cache instruction behavior when the scheduled loop is unrolled. We supposed the instruction cache was 4 Kb big. These results are still valid if the embedded processor does not have an instruction cache, they can then be applied to its embedded memory. Figure 2 shows the effect of the code size of the scheduled and unrolled loop on the instruction cache. It depicts the percentage of loops of several benchmarks which fit in the instruction cache as its size increases. Each figure is for a software pipelining method and several unrolling methods.

As Lam's heuristic always finds the least unrolling degree; it always fits the cache better than the two other heuristics. The code is more compact. Moreover, the heuristic of Eisenbeis et al. seems to fit the cache better than the meeting graph heuristic on average.

Thus we can see that it is possible to choose a way to fit the entire loop in the cache, even if the cache is rather small, by choosing a scheduling technique that does not lead to a too much higher register pressure.

3.2 Amount of Loop Unrolling

These experiments show the amount of unrolling used by each heuristic. Figure 3 shows the effect of several loop unrolling strategies combined with software pipelining techniques on the code size. Each figure depicts the unrolling degree computed by two methods for the same benchmark scheduled with the same method, the heuristic of Gasperoni and Schwiegelshohn or DESP.

The heuristic of Eisenbeis et al. has less tight control on the unrolling degree than the meeting graph heuristic. This is particularly noticeable on the Nas examples, where the loop size explodes rather often with their heuristic. With Gasperoni, the code size can be up to 616 times the code size found with Lam's heuristic. On the other hand, the loop size is at worst only up to 11.33 times bigger than the optimal. The same phenomenon occurs with all scheduling techniques implemented in MOST.

Furthermore, the upper bound of unrolling of the meeting graph heuristic is directly related to the quality of the schedule, as said in Section 2.3. So the more the loop is compacted by the software pipelining algorithm, the higher the upper bound of unrolling.

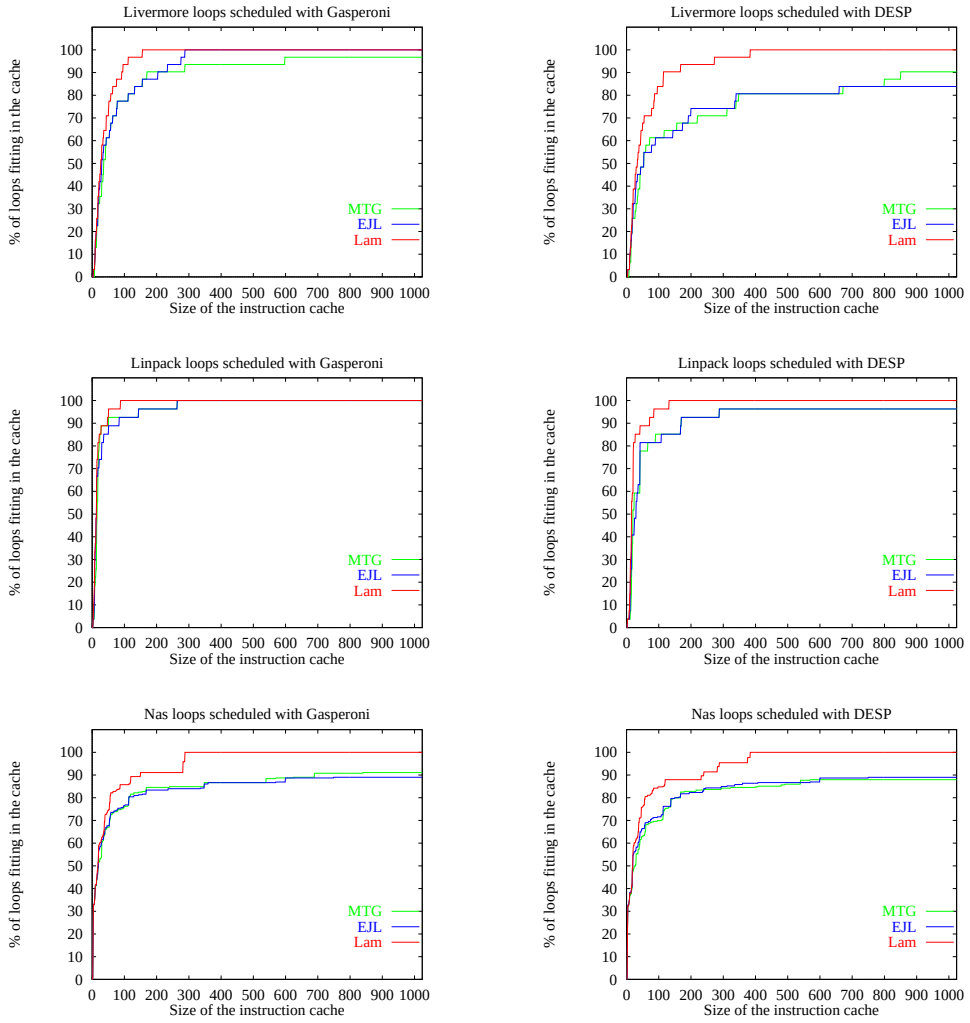

Figure 2: Memory occupation of the unrolled loops

DESP usually performs better than the heuristic of Gasperoni and Schwiegelshohn, but the unrolling degree needed to allocate the loop with R_{min} registers is then much higher.

From these experiments, we can see that control of the unrolling degree is possible even if we have strong constraints on the register requirements.

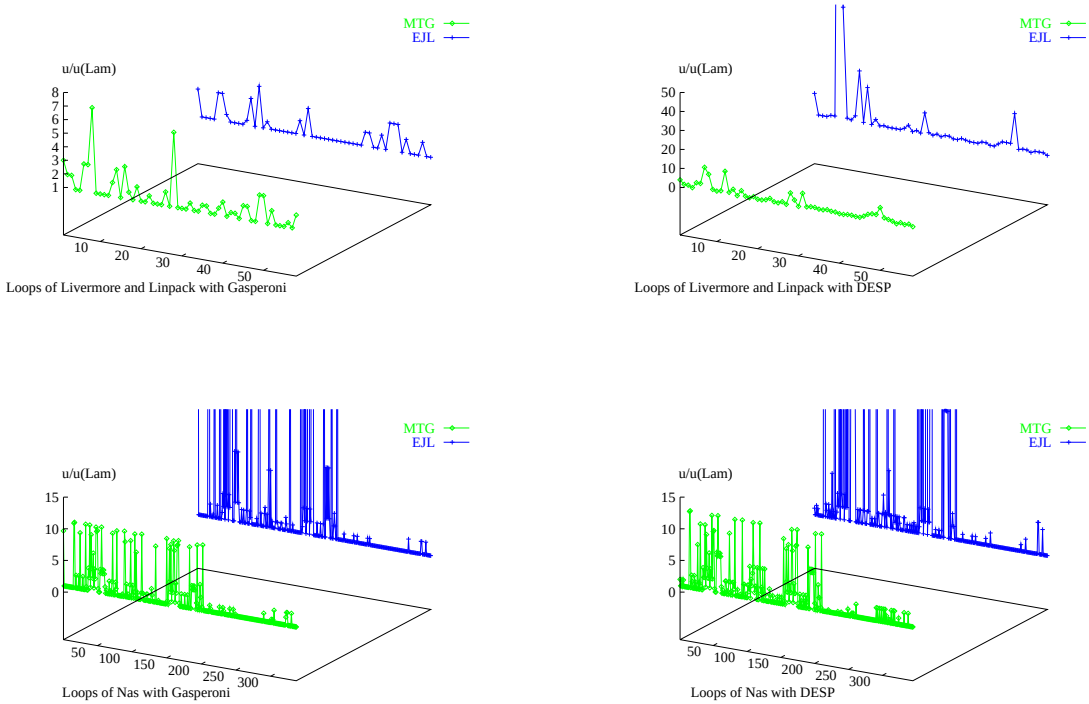

Figure 3: Unrolling degree of the loops

3.3 Register Allocation

When a loop is software pipelined, we can not use usual register allocation algorithms because of self-interferences in the usual interference graph [3, 11]. One way to deal with this problem is to perform register renaming [4], but this leads to bigger code than the original. The solution we take is to perform loop unrolling.

We tested several techniques which all use loop unrolling [14, 6, 7]. Figure 4 shows the register requirements for several register allocation methods involving loop unrolling. It shows the optimal number of registers needed by the meeting graph heuristic and the number of registers needed with a combination of the heuristics of Hendren et al. and Lam. We

compared the heuristics with regard to the number of registers used to perform the register allocation. We did not put an upper bound on the number of registers of the processor, therefore no spill code is inserted in the loop.

The meeting graph heuristic always gives an allocation with an optimal number of registers with regard to a fixed schedule, whereas Lam's heuristic can sometimes have difficulties to keep the register pressure acceptable, as in the Nas benchmarks. It should be noticed that the heuristic of Hendren et al. used to find a register allocation after Lam's scheduling algorithm is not optimal, but achieves very good results in general. The problem encountered with Lam's heuristic can lead to important performance losses when spill code code insertion is needed, like in the Nas benchmarks. When the loops are scheduled with DESP this method can need up to 13 registers more than the optimal, this falls to 5 when it is scheduled with the heuristic of Gasperoni and Schwiegelshohn. The same phenomenon occurs with the Livermore loops and Linpack with 9 and 2 registers more respectively. Thus, as the register requirements are bigger than those of the meeting graph heuristic, the number of memory accesses will also be bigger and severely degrade code performances and also the final code size of the loop.

The code schedule found has the same impact on the register requirements as on the loop unrolling degree. Hence one more, the more the loop is compacted, the higher the register requirement.

3.4 Other Experiments

We made also experiments to test the quality of heuristics dealing with the meeting graph. We made experiments over 1924 cases.

Table 1 presents a comparison between the Dual Chords heuristic and the heuristic of Eisenbeis et al., and the heuristic of the meeting graph. Two parameters are considered: the unrolling degree, and the number of circuits building the decomposed meeting graph. Of course, the second one is only relevant for the meeting graph heuristic. It appears that the dual chords heuristic performs well in comparison to the 2 others.

Moreover the unrolling degree is not completely dependent of the number of circuits. This explains also why the Dual Chords heuristic performs better than the Meeting Graph heutistic. In fact, as it takes into account the *lcm* during the decomposition, it controls better the final *lcm*. The Meeting Graph heuristic should also do this, but until now no satisfactory solution has been found to do that.

Table 3 presents some results about execution times. It is clear that the Dual Chords heuristics is less time consuming than the Meeting Graph heuristic. The results represent for the first row 6.55% of the cases, for the second row 3.22% of the cases, then 2.34% of the cases, and finally 5.87% and 3.69% of the cases.

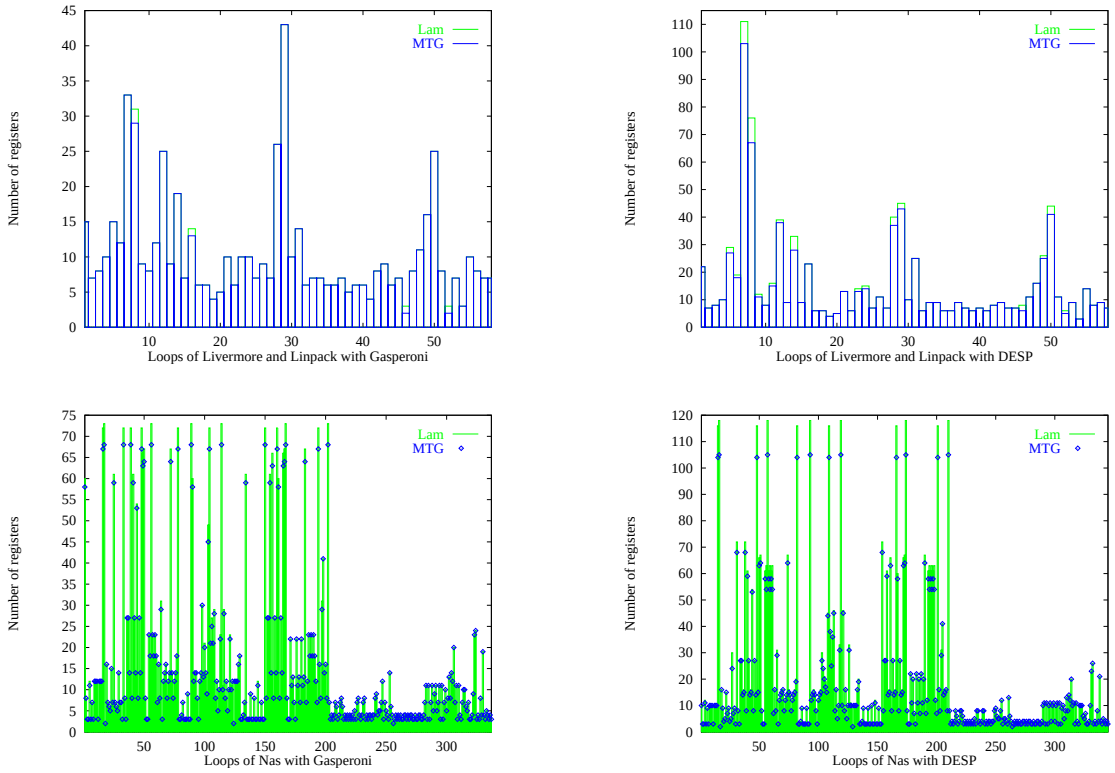

Figure 4: Register requirements of the loops

	Dual worse than	Dual as good as	Dual better than
Unrolling Degree			
Meeting Graph	4.78%	78.90%	16.32%
EJL	5.30%	81.86%	12.84%
Nb of circuits			
Meeting Graph	12.89%	83.16%	3.95%

Table 1: Comparisons for unrolling degree and decomposition

	Dual worse than MTG	Dual as good as MTG	Dual better than MTG
Unrolling Degree			
$n_{Dual} > n_{Mtg}$	13.71%	21.77%	64.52%
$n_{Dual} = n_{Mtg}$	2.38%	89.44%	8.19%
$n_{Dual} < n_{Mtg}$	26.32%	43.42%	30.26%
Nb of circuits			
$u_{Dual} < u_{Mtg}$	50.96%	41.72%	7.32%
$u_{Dual} = u_{Mtg}$	3.56%	94.27%	2.17%
$u_{Dual} > u_{Mtg}$	36.96%	41.30%	21.74%

Table 2: Comparative results between MTG and Dual

Execution Time	Dual is slower than MTG	Dual is faster than MTG
$\lvert t_{mtg} - t_{Dual}\rvert > 0.1s$	42.86%	57.16%
$\lvert t_{mtg} - t_{Dual}\rvert > 0.5s$	19.35%	80.65%
$\lvert t_{mtg} - t_{Dual}\rvert > 1s$	8.89%	91.11%
$\min(t_{mtg}, t_{Dual}) > 0.5s$	38.05%	61.95%
$\min(t_{mtg}, t_{Dual}) > 1s$	28.17%	71.83%

Table 3: Execution time comparison between MTG and Dual

4 Related Work

Register renaming [4] can ensure an optimal register allocation. But as it needs to insert copy operations, it degrades code compaction. Therefore another method, loop unrolling is more suitable for embedded processors. The loop body itself is bigger but no extra operations are executed in comparison with the original code. Loop unrolling is known as a very useful technique to improve code quality [16]. But methods used for DSP processors do not control the loop unrolling degree [13]. On the other hand, we elaborated a method to control it in

the general case [7]. This method allows us to compute an unrolling degree for which the register allocation is optimal. It also tries to lower this unrolling degree.

The code for embedded processors must be quick and compact [19], whereas techniques developed for exploitating instruction-level parallelism in general-purpose processors have less consideration for code compaction [2]. Some work on code compaction for DSP has been done by Leupers and Marwedel [17], they solve this problem by integer linear programming. They do not consider registers as the allocation is done before compaction but they take into account time constraints, it should be compared with works by Eichenberger and Davidson [5] and Eisenbeis and Sawaya [8], which control the register requirements during code scheduling but not the final code size, i.e. after loop unrolling.

5 Conclusion

LoRA is a tool for optimally allocate loops, either by controlling the register requirements, or the unrolling degree of the loop. LoRA can handle several register files, which is useful for embedded processors. It implements several register allocation heuristics, that all involve loop unrolling.

We used it to evaluate trade-offs between the code size of the final loop and the number of registers required to allocate the loop. From the experiments, it is clear that the compaction of the loop has a direct impact on the number of registers needed and the unrolling degree. But it is possible to control the code size while finding an optimal register allocation. It is the case with the first and third heuristics implemented in LoRA, even if the third heuristic has a less tight control than the first heuristic.

Acknowledgments

We thank Professor Gao from University of Delaware for providing us with the MOST platform. We also thank David Gregg, who helped us to improve the readability of this paper.

References

[1] Erik R. Altman. *Optimal Software Pipelining with Function Unit Register Constraints.* PhD thesis, McGill University, Montréal, Canada, October 1995.

[2] David F. Bacon, Susan L. graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):325–420, December 1994.

[3] Gregory J. Chaitin. Register Allocation and Spilling via Graph Coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[4] Ron Cytron and Jeanne Ferrante. What's in a Name? or the Value of Renaming for Parallelism Detection and Storage Allocation. In Sartaj K. Sahni, editor, *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, University Park, Pennsylvanie, August 1987. London: Penn State press.

[5] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. Optimum modulo schedules for minimum register requirements. In *Proceedings of the International Conference on Supercomputing*, pages 31–40, Barcelona, July 1995.

[6] Christine Eisenbeis, William Jalby, and Alain Lichnewsky. Compiler techniques for optimizing memory and register usage on the Cray-2. *International Journal on High Speed Computing*, 2(2), June 1990.

[7] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. The Meeting Graph: a New Model for Loop Cyclic Register Allocation. In Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT'95*, pages 264–267, Limassol, Cyprus, June 27–29 1995. ACM Press.

[8] Christine Eisenbeis and Antoine Sawaya. Optimal loop parallelization under register constraints. In Michael Gerndt, editor, *Proceedings of the sixth workshop on Compilers for Parallel Computers*, volume 21, pages pp 245–259, Aachen, Germany, December 11-13 1996.

[9] M.R. Garey, D.S. Johnson, G.L. Miller, and C.H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM J. Alg. Disc. Meth.*, 1(2):216–227, June 1980.

[10] F. Gasperoni and U. Schwiegelshohn. Efficient Algorithms for Cyclic Scheduling. Technical Report 571, New York University, July 1991.

[11] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. *The Journal of Programming Languages*, 1(3):155–185, September 1993.

[12] Richard A. Huff. Lifetime-Sensitive Modulo Scheduling. *SIGPLAN Notices*, 28(6):258–267, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[13] David J. Kolson, Alexandru Nicolau, Nikil Dutt, and Ken Kennedy. Optimal Register Assignment to Loops for Embedded Code Generation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 1(2):251–279, April 1996.

[14] Monica S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *SIGPLAN Notices*, 23(7):318–328, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[15] E. Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*, pages 222–229. Teubner, Leipzig, 1909.

[16] Rainer Leupers and Peter Marwedel. Optimierende Compiler für DSPs: was sit verfügbar ? In *DSP Deutschland '97*, Munich, October 1997.

[17] Rainer Leupers and Peter Marwedel. Time-constrained Code Compaction for DSPs. *IEEE Transactions on VLSI Systems*, 5(1), 1997.

[18] Josep Llosa, Mateo Valero, and Eduard Ayguadé. Heuristics for Register-Constrained Software Pipelining. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 250–261, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[19] Peter Marwedel. *Code Generation for Embedded Processors*, chapter Code Generation for Embedded Processors: A Introduction. Kluwer, June 1995.

[20] Jian Wang, Christine Eisenbeis, Martin Jourdan, and Bogong Su. DEcomposed Software Pipelining: a New Perspective and a New Approach. *International Journal on Parallel Processing*, 22(3):357–379, 1994. Special Issue on Compilers and Architectures for Instruction Level Parallel Processing.

A An Example with LoRA

Here is a C function calling the procedure that interfaces LoRA.x

```
int Example ()
{
  int i,mtg_heu,II,nb_lifetimes,nb_of_classes_of_registers;
  int **Table,*spill,*Registres,*Nb;
  FILE *f;

  f = NULL;

  II = Compute_II_Elsewhere ();

  nb_lifetimes = Compute_Live_Ranges_Somewhere ();

  nb_of_classes_of_registers = Give_nb_of_register_classes ();

  Table = (int**) calloc (nb_lifetimes+1,sizeof(int*));
  for (i = 0;i <= nb_lifetimes;i++)
      Table [i] = (int*) calloc (4,sizeof (int));

  Table [0] [1] = nb_lifetimes;
  Table [0] [0] = II;

  for (i = 1;i <= nb_lifetimes;i++)
    {
      Table [i] [0] = Compute_Beginning_Cycle (i);
      Table [i] [1] = Compute_End_Cycle (i);
      Table [i] [2] = Register_Class (i);
    }

  Registres = (int*) calloc (nb_of_classes_of_registers+1,sizeof(int));
  for (i = 0;i <= nb_of_classes_of_registers;i++)
      Registres [i] = Total_Nb_of_Registers_of_Class (i);


  Nb = (int*) calloc (nb_of_classes_of_registers+1,sizeof(int));
  for (i = 0;i <= nb_of_classes_of_registers;i++)
      Nb [i] = Nb_of_Available_Registers_in_Class (i);
```

```
  mtg_heu = Heuristic_to_Test ();

  spill = NULL;

  Table = Blue_Oceans (f,Table,Registres,Nb,mtg_heu,spill,argv[1]);

  printf ("The loop must be unrolled %d times\n",Table [0] [3]);

  printf (" List of the lifetimes in the unrolled loop:\n");

  for (i = 1;i <= T [0] [1];i++)
     printf ("Lifetime %d begins at cycle %d and ends at cycle %d and \
             is assigned  to register %d of the class %d\n",
             i,T [i] [0], T [i] [1], T [i] [3], T [i] [2]);


}
```

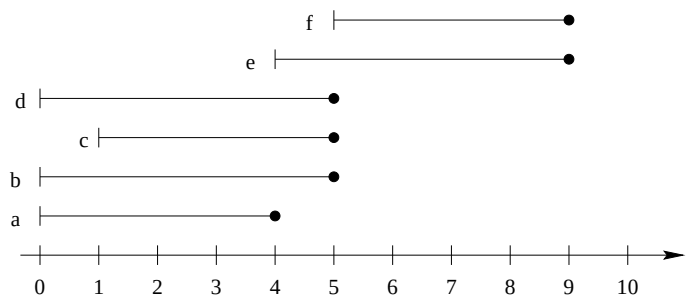

Figure 5: Lifetimes of the input loop

Here is now an example of execution. Let's suppose we have the lifetimes of figure 5 and $II = 4$ and we want to test heuristic EJL_MINR (number 3). The matrices given as parameter are then:

$$\mathrm{T} = \begin{pmatrix} 4 & 6 & 0 & 0 \\ 0 & 4 & 4 & 0 \\ 0 & 5 & 4 & 0 \\ 1 & 5 & 4 & 0 \\ 0 & 5 & 4 & 0 \\ 4 & 9 & 4 & 0 \\ 5 & 9 & 4 & 0 \end{pmatrix} \mathrm{Registres} = \begin{pmatrix} 2 \\ 8 \\ 60 \\ 58 \\ 0 \end{pmatrix} \mathrm{Nb} = \begin{pmatrix} 2 \\ 8 \\ 60 \\ 58 \\ 0 \end{pmatrix}$$

And here the matrices we get after the call to LoRA:

$$T = \begin{pmatrix} 4 & 12 & 1 & 2 \\ 1 & 5 & 4 & 1 \\ 1 & 6 & 4 & 5 \\ 2 & 6 & 4 & 6 \\ 1 & 6 & 4 & 7 \\ 1 & 6 & 4 & 8 \\ 2 & 6 & 4 & 9 \\ 5 & 9 & 4 & 1 \\ 5 & 2 & 4 & 2 \\ 6 & 2 & 4 & 6 \\ 5 & 2 & 4 & 3 \\ 5 & 2 & 4 & 4 \\ 6 & 2 & 4 & 9 \end{pmatrix} \quad \text{Registres} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 9 \\ 0 \end{pmatrix} \quad \text{Nb} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 9 \\ 0 \end{pmatrix}$$

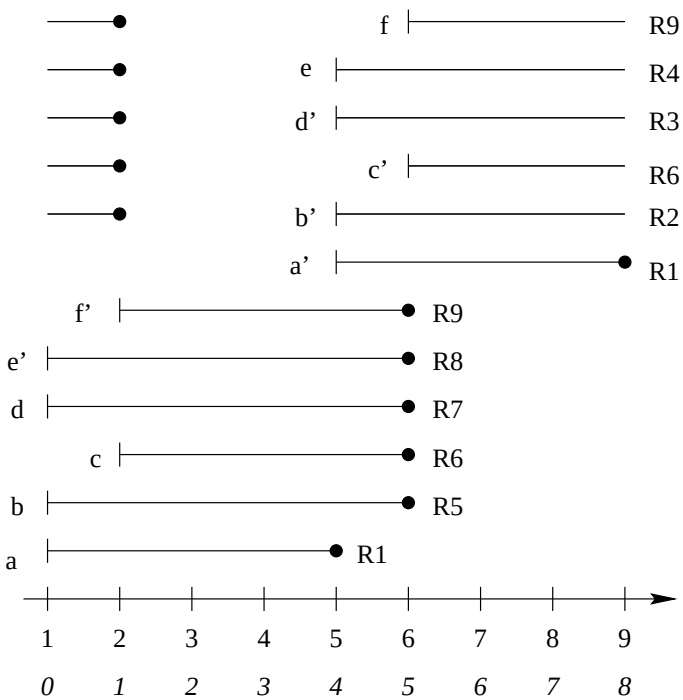

Figure 6: Lifetimes of the unrolled loop

In this example the loop was unrolled twice, T [0] [3] = 2, the number of intervals is 12, T [0] [1] = 12, and no spill is necessary as T [0] [2] = 1. We used Registres [4] = 9 registers and we needed at least Nb [4]= 9 registers.

One has to take care of the iteration to which the lifetimes of the unrolled loop belong. In our example, e and f were in the second iteration, so the lifetimes put during the cycles of the first iterations are in fact the copies e' and f' of the lifetimes of the original loop.

B Manual of LoRA

Procedure: `int** Blue_Oceans (f,T,Registres,Nb,heu,spill,nom_test)`
Parameters:

- `FILE *f` : File in which the results are written
- `int **T` : matrix which contains input data and results
  - input: T [0] [0] = II, T [0] [1] = number of intervals
  - output: T [0] [2] = 0 if spill is necessary, 1 otherwise, T [0] [3] = unrolling degree of the loop
  - For each interval i:
    * input/output: T [i] [0] = beginning of i
    * input/output: T [i] [1] = end of i
    * input/output: T [i] [2] = register type for i
    * output: T [i] [3] = color of i, -1 if this variable must be spilled (see parameter *spill code*).

  Beginning and end are not modulo II (from 0 til ...) **CAUTION**: after the call (so in the result matrix) the dates are +1 cycle and modulo II (from 1 til $II+1$)
- `int* Registres` : array containing data about registers
  - Registres [0] = number of register types
  - For each interval i:
    * input : Registres [i] = number of registers of type i
    * output: Registres [i] = number of registers of type i used, -1 if there are not enough registers
- `int *Nb` : array containing the number of usable registers
  - Nb [0] = number of register types
  - For each interval i
    * input : Nb [i] = number of usable registers of type i
    * output: Nb [i] = $MaxLive$ (R_{min}) for registers of type i
- `int heu` : heuristic to use
- `int **spill` : matrix for spill purpose

  For the moment it works like an array. For each interval i:
  - input: spill [i] [0] = penalty for variable i, i.e. number of cycles where it is effectively used.

If a variable i must be spilled, this is given with T [i] [3] = -1. In this case, no unrolling degree is computed, T [0] [3] = 0. If spill = NULL, it allocates with no constraints on the number of registers.

- `char *nom_test` : name of the example

Heuristics description:

1. - Minimization of the number of registers used and of the unrolling degree.
   Number of registers used = $MaxLive$, width of the interval family
   Unrolling degree = $lcm(r_1, ..., r_n)$ with r_i the weight of the circuits of the meeting graph decomposed with MTG heuristic 2 (can not be modified).

2. - Control of the unrolling degree for a given number of registers
   Number of registers of type $i \leq$ Nb [i], or $2r - 1$ if Nb [i] is too big.
   Unrolling degree = $lcm(r_1, ..., r_n)$, with r_i like above.

3. - Minimization of the number of registers and control of the unrolling degree, heuristic [6].
   Number of registers used = $MaxLive$, width of the interval family
   Unrolling degree = degree of the permutation on the intervals covering the origin.

4. - Control of the unrolling for a given number of registers
   Number of registers of type $i \leq$ Nb [i], or $2r - 1$ if Nb [i] is too big.
   Unrolling degree = degree of the permutation on the intervals covering the origin.

5. - Minimization of the unrolling degree using Lam's MVE and [11] register allocation heuristic.
   Number of registers = $MaxLive, MaxLive + 1$ in general...
   Unrolling degree = $\frac{\max(length(lifetimes))}{II}$

6. - Minimization of the number of registers used and of the unrolling degree using a mix of 1 and 3.
   Number of registers = $MaxLive$, width of the interval family
   Unrolling degree = $\min(u_{heuristic1}, u_{heuristic3})$

7. - Control of the unrolling degree for a given number of registers using a mix of 2 and 4
   Number of registers of type $i \leq$ Nb [i], or $2r - 1$ if Nb [i] is too big.
   Unrolling degree = $\min(u_{heuristic2}, u_{heuristic4})$

8. - Minimization of the unrolling degree using pairs of dual chords to decompose the meeting graph.
   Number of registers = $MaxLive$, width of the interval family
   Unrolling degree = $lcm(r_1, ..., r_n)$ with r_i the weight of the circuits of the decomposed meeting graph

9. - Control of the unrolling degree for a given number of registers (using dual chords to decompose the mtg)
   Number of registers of type $i \leq$ Nb $[i]$, or $2r - 1$ if Nb $[i]$ is too big.
   Unrolling degree $= lcm(r_1, ..., r_n)$, with r_i like above.

To install LoRA, go first to the directory LoRA/sxba and type `make` to install the library for bit arrays. Then change to LoRA and type `make` to install the library of LoRA itself. Then you just have to link your program with the library of your choice. When installing LoRA, 3 libraries are created in the directory LoRA/lib:

- `libmtg.a` is the standard library without message.
- `libmtg_debug.a` is compiled for debugging and gives detailed information about the computations.
- `libmtg_verbose.a` gives the different steps of the computation.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399