In the Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques – PACT'96.
October 20 – 23, 1996, Boston, Massachusetts, USA.

Using the Parallel Complexity of Programs to Improve Compaction

Marc Pouzet*
School of Computer Science
McGill University
3480 University Street
Montreal, Quebec H3A 2A7, Canada

Abstract

This paper focuses on the compaction of tail-recursive programs — arbitrary programs with conditionals, loops and function calls appearing in terminal position — for fine-grain parallel machines like VLIW or super-scalars. An iterative compaction method based on instruction move-up and fold/unfold transformations is studied. In order to speed-up compaction and produce shorter schedules, we propose to use some information about the parallel complexity of expressions. This information is used on the fly during the compaction process to decide whether to unfold a function or not.

1. Introduction

VLIW (Very Long Instruction Word) and super-scalar architectures are fine grain parallel machines executing many instructions per cycle. VLIW are controlled by a single instruction stream where each processor executes a dedicated field of a long instruction. In a super-scalar machine, the processor executes successive RISC-like instructions belonging to a small window by analyzing at runtime their dependencies.

Fine grain parallelization for these architectures, or *compaction*, statically recognizes and schedules groups of elementary operations that can be executed in parallel. Loops are compiled by *Software Pipelining.* This compaction can either by obtained in a *direct* way, extending classical scheduling techniques to programs containing control structures (for example [3]) or in an *iterative* way, by applying a sequence of local transformations to a control flow graph [2, 1, 5, 4]. Iterative methods are more powerful than direct methods because of their ability to exploit the parallelism of programs with arbitrary control structures [1].

This paper focuses on the compaction of general programs containing conditionals, loops and tail-recursive functions. For this purpose, we investigate iterative methods proposed by Aiken and Nicolau. These methods are based on the *Percolation scheduling* principle [6] combined with a controlled unrolling of loops and looking for a repeated pattern in the execution [2, 1]. These methods are the most powerful methods among iterative methods but they suffer from convergence problems: the time to obtain a repeated pattern and the size of it are unknown until the end.

This paper proposes a preliminary solution to these classical convergence problems by giving a fast, efficient and greedy loop unrolling strategy. The solution is expressed in the framework of a first-order recursive language, following the work started in [7] where a formal presentation of compaction, based on a set of fold/unfold transformations of the program, has be proposed. In this framework, compaction is applied to programs that may contain functions (recursive or not) and loop unrolling is replaced by function unfolding. We thus define an unfolding strategy to speed-up compaction. The strategy is based on the parallel complexity of expressions and the construction of an *abstract distance* of a recursive function. The abstract distance of a function measures the best possible overlapping between the body of the function and its argument in case of infinite resources. The abstract distance is then used *on the fly* during the compaction process to decide whether to unfold the function or not.

2. Convergence problems in iterative methods

Several iterative methods combining instruction moves and unrolling have been proposed. The compaction of loops can be achieved in a two-steps way, by first compacting some iterations and by moving down some operations to find a repeated pattern [2]. Because the two-steps algorithm only applies to simple

*e-mail: pouzet@cs.mcgill.ca. This work has been supported by an INRIA Fellowship.

[1]Finally, see [8] for an overview and an history of compaction.

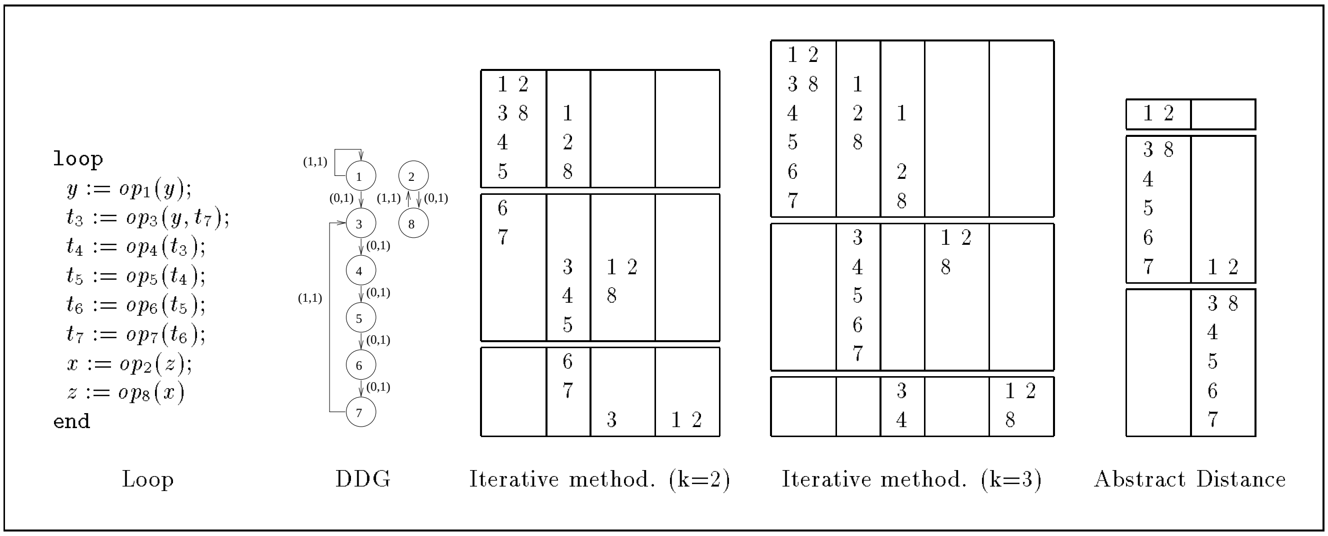

Figure 1: Convergence problems in iterative methods I

loops — loops without conditionals — and for a machine without resource constraints, a greedy version of the algorithm has been proposed [1]. At each step of the process, compaction moves-up instructions belonging to a *sliding window* of k successive iterations and continues with the rest of the program until a repeated pattern occurs. The figure 1 gives the result of the method for $k = 2$ and $k = 3$. In order to limit the code size explosion, it was proposed to try several values of k. Nonetheless, such an enumeration may have a high complexity and does not always produce the minimal code size, as it is shown in the example. This is due to the fact that a constant k is an arbitrary value that does not depend on the dependencies of the program. Here a shorter schedule is feasible (given on the right of figure 1): it is obtained by delaying the unrolling of the second iteration until cycle 6.

The method also tends to produce long fix point, even when there is a minimal k. With the loop of figure 2, the iterative method will produce a pattern containing three copies of one iteration. The successive execution times are $4, 6, 7, 10, 12, 13$... A shorter schedule is feasible (given on the right of figure 2). It is obtained by delaying the unrolling of the second and the third iterations. The execution times are $4, 6, 8, 10, 12, 14$... This schedule is strongly better: the size of the body is one over three the one of the first schedule and at worst, only one cycle is lost over a complete execution of the program.

Finally, note that all these convergence problems remain for programs with conditionals that may need different k for different paths. In that case, there is a useless code size explosion on the paths that need the minimal k.

The convergence problems of iterative methods appear because the process is *local*: instructions can be executed as soon as possible without considering the way they are used in the rest of the program. So in order to get a shorter schedule, it may be better not to start too early an iteration even if it contains some available instructions. Thus, a strategy needs to have some information about the use of an expression in a program before deciding whether to move it up or not.

2.1. Our approach

To keep the *iterative* characteristic of the compaction process — its simplicity and its ability to treat general programs and resources — we show that a good unfolding strategy depending on the dependencies of the program is feasible.

This solution will be expressed in the functional framework, thus following the work started in [7] where it has been shown that compaction can be described in a formal way as a set of program transformations. In this framework, compaction is applied to programs that may contain functions (recursive or not) and loop unrolling is replaced by function unfolding. At each time of the process, the compiler looks for the set of ready sub-terms — elementary computations that can be executed immediately in parallel —, moves them up and continues with the rest of the program. In such a framework, the software pipelining principle consists in unfolding the function F in $F(a)$ to allow some instructions from the argument a to be executed in parallel with F. Termination is obtained by looking for

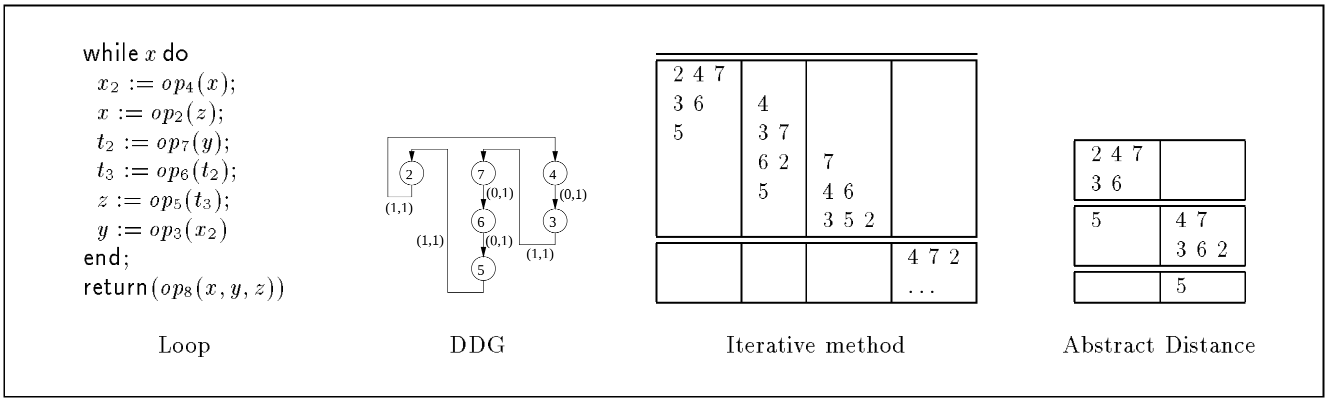

Figure 2: Convergence problems in iterative methods II

a repeated behavior — the presence of equivalent subexpressions — in the program. We thus define an unfolding strategy to speed-up compaction.

The idea of the strategy is the following. Instead of unfolding the function F *a priori* in $F(a)$, we propose to guide the unfolding by the possible gain. For that, we estimate the best overlapping of F with its argument for some executions, supposing we have unlimited resources. This best overlapping is given by computing the difference between the minimal execution time of $F(a)$ and the one of F, for n iterations of F. This sequence defines the *distances*. If a *distance* of F (from a) is zero, it means that there is an execution where the overlapping is complete. In this case, it may be useful to unfold F, otherwise, the compiler will produce a slower program. To be usable on the fly during compaction, we propose to express the distance of F in an abstract way. Thus, if F is a function of x, we define the *abstract distance* $d^n(x) = c^n_F(x) - c^n_F(0)$, that is the difference between the parallel complexity of $F(x)$ and the execution time of the body of F for n iterations of F. Its evaluation with integer parameters (corresponding to execution times of the argument of F) gives a sequence of integer distances. These values are then combined to decide whether to unfold F or not.

3 A complete example

Consider the while loop of figure 1. It can be represented as the function f given below. f is a recursive function with three arguments (the read-set of the loop).

$$
\begin{array}{l}
F = \mathsf{rec}_f(\lambda(x, y, z). \\
\qquad \mathsf{if}\ x \\
\qquad \mathsf{then}\ f(op_2(z), op_3(op_4(x)), op_5(op_6(op_7(y)))) \\
\qquad \mathsf{else}\ op_8(x, y, z))
\end{array}
$$

From f, we can compute the minimal execution times in an abstract way, replacing an expression by its cost. For example, the complexity of an expression $op_4(op_3(x), op_1(m))$ is simply $1 + \mathsf{max}(1 + x_1, 1 + m_1)$ if x_1 is the complexity of x and m_1 is the complexity of m. The expression c^n giving the parallel complexity of f for n iterations is the following:

$$
\begin{array}{rcl}
c^0(x_1, y_1, z_1) & = & 1 + \mathsf{max}(x_1, y_1, z_1) \\
c^n(x_1, y_1, z_1) & = & c^{n-1}(z_1 + 1, 2 + x_1, 3 + y_1)
\end{array}
$$

$c^1(x_1, y_1, z_1)$ is the minimal execution time of the program where the loop is executed once and the minimal execution time of the entry variables x, y and z are x_1, y_1 and z_1. These two definitions measure the size of the longest paths in the unfolded dependence graph of the loop. The distance can then be defined also in an abstract way.

$$
d^n(x_1, y_1, z_1) \quad = \quad c^n(x_1, y_1, z_1) - c^n(0, 0, 0)
$$

We shall illustrate now the compaction problem on F. Suppose that we have a machine reading this program in a left-to-right order and such that it executes all the successive and independent operations in parallel. Suppose now that we start the compaction with the application $F(x, y, z)$. In order to compact this term, F is first unfolded applying a rule $\mathsf{rec}_f(p) \rightarrow p[f \backslash \mathsf{rec}_f(p)]$ meaning that every occurrence of the recursive call is replaced by its definition. After a simplification, we get:

$$
\begin{array}{l}
\mathsf{if}\ \overline{x} \\
\mathsf{then}\ F(\overline{op_2(z)}, op_3(\overline{op_4(x)}), op_5(op_6(\overline{op_7(y)}))) \\
\mathsf{else}\ op_8(x; y; z)
\end{array}
$$

The principle of compaction is to move-up some of the elementary computations that can be executed immediately in parallel. We call these computations the *ready sub-terms*. They are overlined in the program. We move them up and get the following program:

$$
\begin{array}{l}
\mathsf{if}\ x \\
\mathsf{then}\ \ \mathsf{let}\ z = op_2(z) \\
\qquad\qquad x = op_4(x) \\
\qquad\qquad y = op_7(y) \\
\qquad\quad \mathsf{in}\ \underline{F(z, op_3(x), op_5(op_6(y)))} \\
\mathsf{else}\ op_8(x, y, z)
\end{array}
$$

The successive operations op_2, op_4, op_7 and the condition are executed in parallel in the first step of the execution. Now, we have to compact the rest of the code, that is $F(z, op_3(x), op_5(op_6(y)))$. Because the first argument is a variable, it means that by unfolding F, some of the ready sub-terms of F will be able to be executed in parallel with some of the argument of F. We say that the recursion is ready to be unfolded. To know whether F has to be unfolded or not in the underlined expression, we compute the distance of F with its argument, that is the quantity $d^n(0,1,2) = c^n(0,1,2) - c^n(0,0,0)$. The successive values (over n) are $1, 0, 2, 1, 0, 2, 1, 0, 2, ...$ Now, using the distance information, at least two unfolding strategy can be adopted. If the unfolding is done when the minimal of the sequence is zero — there is one execution where the overlapping is complete — then no execution is sacrificed. Another policy can be adopted by considering the average of the distances. It means that the minimal unfolding is preferred. Let us take the second solution. Because, the average is bounded by $\lfloor 3/3 \rfloor \geq 1$, we can delay the unfolding of F. We get:

$$
F(z, op_3(x), op_5(op_6(y))) \rightarrow \begin{array}{l} \mathsf{let}\ x = op_3(x) \\ \qquad y = op_6(y) \\ \mathsf{in}\ \underline{F(z, x, op_5(y))} \end{array}
$$

Operations op_3 and op_6 are executed in parallel. Now, the underlined expression has to be compacted. To decide whether F has to be unfolded or not, we ask the same question: what are the possible values of $d^n(0,0,1)$ (over n)? The list is $0, 0, 1, 0, 0, 1, 0, 0, 1, ...$ Here, in 2 executions over 3, the overlapping is feasible (the average is bounded by $\lfloor 1/3 \rfloor < 1$). We thus have to unfold F. We get:

$$
\left(\begin{array}{l} \lambda(x, y, z).\ \mathsf{if}\ x \\ \qquad\qquad \mathsf{then}\ f(\ op_2(z), op_3(op_4(x)), \\ \qquad\qquad\qquad\quad op_5(op_6(op_7(y)))) \\ \qquad\qquad \mathsf{else}\ op_8(x, y, z) \end{array} \right) (z, x, op_5(y))
$$

Now, this application can be simplified by applying the simple renaming rule $(\lambda x.p)y \rightarrow p[x \backslash y]$ where every occurrences of x is replaced by y. We get:

$$
\left(\begin{array}{l} \lambda z_1 .\ \mathsf{if}\ \overline{z} \\ \qquad \mathsf{then}\ f(\ op_2(z_1), op_3(\overline{op_4(z)}), \\ \qquad\qquad\quad op_5(op_6(\overline{op_7(x)}))) \\ \qquad \mathsf{else}\ op_8(z, x, z_1) \end{array} \right) (\overline{op_5(y)})
$$

The ready sub-terms are overlined. We first move-up the test: the context is duplicated in both branches. We then move-up the others. For the false branch, we get the following term:

$$
\begin{array}{l}
\mathsf{if}\ z \\
\mathsf{then}\ \ \mathsf{let}\ y = op_5(y) \\
\qquad\qquad z = op_4(z) \\
\qquad\qquad x = op_7(x) \\
\qquad\quad \mathsf{in}\ \underline{(\lambda z_1 . F(op_2(z_1), op_3(z), op_5(op_6(x))))y} \\
\mathsf{else}\ (\lambda z_1 . op_8(z, x, z_1))(op_5(y))
\end{array}
$$

We simplify the underlined term. A new unfolding is useless since no argument is a variable. The compaction of the underlined expression produces:

$$
F(op_2(y), op_3(z), op_5(op_6(x))) \rightarrow \begin{array}{l} \mathsf{let}\ y = op_2(y) \\ \qquad z = op_3(z) \\ \qquad x = op_6(x) \\ \mathsf{in}\ \underline{F(y, z, op_5(x))} \end{array}
$$

This time, $F(y, z, op_5(x))$ equals, modulo a renaming, an already compacted term (an underlined term): the fix-point is reached. We can replace the term by a function call to a new function (*ff*). The final program is the following.

$$
\begin{array}{l}
\lambda\ (x, y, z). \\
\quad \mathsf{if}\ x \\
\quad \mathsf{then} \\
\quad \mathsf{let}\ \ z = op_2(z) \\
\qquad\quad x = op_4(x) \\
\qquad\quad y = op_7(y)\ \mathsf{in} \\
\quad \mathsf{let}\ \ x = op_3(x) \\
\qquad\quad y = op_6(y)\ \mathsf{in} \\
\quad (\mathsf{rec}\ _{ff}(\lambda(z, x, y).\ \mathsf{if}\ l \\
\qquad\qquad\qquad\qquad \mathsf{then}\ \ \mathsf{let}\ y = op_5(y) \\
\qquad\qquad\qquad\qquad\qquad\qquad z = op_4(z) \\
\qquad\qquad\qquad\qquad\qquad\qquad x = op_7(x)\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{let}\ y = op_2(y) \\
\qquad\qquad\qquad\qquad\qquad\qquad z = op_3(z) \\
\qquad\qquad\qquad\qquad\qquad\qquad x = op_6(x)\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad ff(y, z, x) \\
\qquad\qquad\qquad\qquad \mathsf{else}\ op_8(z, x, op_5(y))))(z, x, y) \\
\quad \mathsf{else}\ op_8(x, y, z)
\end{array}
$$

The false branch is left uncompacted for sake of clarity. The resulting program executes a prelude consisting of two successive instruction sets, then it loops in a steady state, i.e, a recursive function *ff*. The software pipelining of the recursion has been done: another unfolding does not produces a shorter schedule (using the same strategy).

4. Discussion

Thus, the unfolding strategy consists in computing a sequence of distances and to combine them to decide whether to unfold a function or not. These sequences of values can be combined in many different ways: by considering the minimum of the sequences, the unfolding is done when there is one execution path where the overlapping is complete; by considering the average of the distance, the unfolding is done when the overlapping is complete in the majority of executions. Other strategies are possible, for example, by taking into account the probability of some executions among others.

When conditional are present, the complexity expression will express the different possible execution times depending on the executed path and the different values are then combined.

The parallel complexity of an expression is a combination of the operators max and $+$. Because of their algebraic properties, the size of the expression giving the execution time of an expression is proportional to its number of free variables.

In the example, compaction has been done with the hypothesis that resources are not bounded. This constraint is not imposed by the algorithm since it is not necessary to move-up all the ready sub-terms of the program: real resources can be taken into account by restricting the set of ready sub-term that can be selected. In practice, the strategy will always verify that the maximal number of free variables — the number of living registers — and the number of functional units used at each step are compatible with the ones of the machine. Resources constraints can also be taken into account by considering the transformations as a preprocessing step and applying then a local scheduling on the program (or by simply using the internal scheduler of modern super-scalar processors).

Finally, using the algebraic properties of max and $+$ for the scheduling problem is not new (for example [9]) but, to our knowledge, existing methods only apply to programs without conditionals and for machines with unlimited resources.

5. Conclusions

We have presented some ideas to reduce convergence problems in a compaction system based on program transformations. In the functional framework, compaction can be described in a formal way as a set of fold/unfold transformations. This is not the only advantage of this framework. It allows to represent the parallel execution time and the distance information in an abstract way for a large class of programs with no hypothesis on the syntactical structure of the program. Moreover, this information can be simplified so the distance can be recomputed on the fly during compaction: the unfolding decision is fast.

References

[1] A. Aiken, A. Nicolau, and S. Novack. Resource-constrained software pipelining. *IEEE Transaction on Parallel and Distributed Systems*, 1995. To appear.

[2] Alexander Aiken and Alex Nicolau. Optimal loop parallelization. In *PLDI*, Atlanta, Georgia, June 22-24 1988.

[3] Monica S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *PLDI*, pages 318–328, Atlanta, Georgia, June 22-24 1988.

[4] S-M. Moon and K. Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *MICRO–25*, pages 55–71, 1992.

[5] Toshio Nakatani and Kemal Ebcioglu. Using a Lookahead Window in a Compaction-Based Parallelizing Compiler. In *MICRO–23*, pages 57–68, Orlando, Florida, November 27-28 1990.

[6] Alex Nicolau. Percolation scheduling: A parallel compilation technique. Technical report, Cornell University, 1985.

[7] Marc Pouzet. The Program Compaction Revisited: the Functional Framework. In *International Conference on Parallel Processing (EURO-PAR'95)*, LNCS 966, Stockholm, Sweden, August 29-31 1995.

[8] B.R. Rau and J.A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing*, 7:9–50, 1993.

[9] A. Zaky and P. Sadayappan. Optimal static scheduling of sequential loops on multiprocessors. In *International Conference of Parallel Processing*, volume III, pages 130–137, 1989.