DEPENDENCE GRAPHS AND COMPILER OPTIMIZATIONS*

D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract

Dependence graphs can be used as a vehicle for formulating and implementing compiler optimizations. This paper defines such graphs and discusses two kinds of transformations. The first are simple rewriting transformations that remove dependence arcs. The second are abstraction transformations that deal more globally with a dependence graph. These transformations have been implemented and applied to several different types of high-speed architectures.

1. Introduction

1.1 Background

This paper presents some compiler transformations that can be carried out on a dependence graph which represents a high-level language program. Some transformations are variations on well-known techniques and others are new. The goal of the transformations is to enhance the performance of programs; in other words, they are dependence graph optimization steps. All of the ideas we discuss are rooted in a working compiler/analyzer of FORTRAN programs for various architectures; the system is called PARAFRASE. This paper discusses theoretical as well as practical ideas. The practical ideas have been verified on a collection of about 1,000 programs (gathered from many sources) that we use as a test set.

For a number of years, we have been studying compilation techniques that exploit four kinds of architectural features. These are parallel processing [KuMC72], pipeline processing [KKLW80], multiprocessing [PaKL80], and virtual memory [AbKL79].

*This work was supported in part by the National Science Foundation under Grant Nos. US NSF MCS76-81686 and MCS80-01561.

Our software system consists of some 50 modules that can be used to transform an internal program representation; after each module it is possible to regenerate a source program. Thus, the modules can be interconnected in various ways to achieve desirable results (in fact, it is sometimes necessary to empirically determine the best ordering). Exploitation of each of the four architectural features requires a different module ordering, but good results for each kind of architecture can be obtained from the same set of modules.

We feel that our results have greatly benefited from using dependence graphs. These benefits include the ease of implementing, maintaining, and modifying the software. But dependence graphs are also a good vehicle for developing new algorithms for optimization.

Our work has been done in terms of FORTRAN programs, but we believe that the ideas can be extended to many other languages. In this spirit, the paper begins with the assumption that a good dependence graph has somehow been obtained from a program, and we discuss graph transformations. Basically, only two ideas are pursued; first, we give a collection of ways to remove dependence arcs, and second, we give ways of abstracting the graphs that lead to optimizations. The paper contains a number of definitions and theoretical results as well as some discussion of the practical implementation and use of the ideas.

1.2 Dependences

Any algorithm that is formalized and expressed in a language (programming or natural) contains some kind of dependences between the atomic operands and between the steps of the algorithm. Programmers generally pay little attention to the dependences in a "pure" algorithm or to any "artificial" dependences that they may introduce when expressing the algorithm in some language. Nevertheless, if a program is to be run on a machine with any kind of simultaneously operating subsystems, the dependences may be very important. In many cases, reducing the number of dependences leads to direct reductions in a program's running time.

Roughly speaking, there are four times at which dependences can be reduced: when a language is selected for implementing a program, when an

algorithm is expressed as a program in that language, when the program is compiled, and when it is executed. Most languages have an explitely stated dependence between consecutive statements in the control flow graph (e.g., PASCAL, ALGOL, FORTRAN, SNOBOL, etc.). A few languages [AcDe79], [ArGP78] are defined so that some types of dependence are disallowed and others are greatly reduced, however, making the programmer responsible for reducing dependences imposes a difficult task on the programmer. High-speed computer systems with multiple functional units or pipelines commonly employ a lookahead control unit that breaks dependences at execution time. Studies have shown that the average speedup due to lookahead hardware is about a factor of 2 [Kuck78]. We believe that compilers are most well suited to solving the problem of breaking program dependences.

In fact, much work has been done on this problem in the past. The renaming of variables and code motion are traditional optimization techniques which result in an improved dependence graph [AhUl73], [Grie71]. Transformations explicitly aimed at lookahead control units are also well known [AlCo72].

In this paper, we assume that a dependence graph exists and that it is sharp (i.e., arcs are included only when necessary). We will show in Section 3 that a number of graph transformations exist to remove arcs from a sharp dependence graph. In Section 4, we discuss node transformations that abstract the graph in useful ways. This results in a directed acyclic graph that is amenable to rather straightforward code generation.

2. Source Language and Dependence Graphs

2.1 Source Language

The transformations described in this paper will operate on programs written in a language consisting of three types of statements: assignment, *for* loops, and *while* loops. The last two are well-known compound statements. In this paper, *for* loops are restricted in such a way that the initial value and the increment of the index variable is always one, and the loop limit is always a constant or a variable. These are not serious restrictions, since any more general *for* loop can be automatically translated into this form [BCKT79].

Only two types of variables are allowed, scalars and arrays of arbitrary dimension. In this paper, variable element will stand for either a scalar variable or an array variable element. Assignment statements are of the form

<variable> = <expression>.

The expression can be any valid Boolean or arithmetic expression, or something of the form

if <Boolean-expression> *then* <expression$_1$>.
else <expression$_2$>.

The value of this last construct will be <expression$_1$> if <Boolean-expression> is true, and <expression$_2$> otherwise.

It is certainly theoretically possible to automatically translate FORTRAN- or ALGOL-like programs into equivalent *go to*-less programs. This has been discussed in [BoJa66], [AsMa75]. However, the programs resulting from the transformations described in these papers could be too complex for practical purposes. A more practical approach to the automatic improvement of program structures has been taken in [Bake78]. The resulting programs in this last case, however, could include *go to* statements.

The assumption we make in this paper is that if the source program is written in a FORTRAN- or ALGOL-like language, some sort of preprocessor will attempt to translate it into the above language but without going to the extreme of producing unduly complex programs. The translator may then act only on segments of the program produced by the preprocessor, leaving the rest of the program as it is. This is similar to the approach taken in the PARAFRASE system. Our experience indicates that in most programs a large percentage of the code can be translated into well-structured, *go to*-less code.

2.2 Dependence Graphs

The main tool we will use in the translation process is the dependence graph. This is a digraph whose nodes represent program components, and whose arcs are one of five different types. A dependence graph can be built at different levels of abstraction; we will discuss this later in the section. For now, we will assume that a graph node represents one of the three types of program components: assignment statements, *for* loop headers, and *while* loop headers. We will assign labels to all the program components. These labels will be of the form A_i for assignment statements, F_i for *for* loop headers, and W_i for *while* loop headers. An example of a program and its dependence graph is given in Fig. 2(a).

A graph arc represents one of the five possible relations between the program components. We now proceed to define these five relations, starting with the loop dependence relation.

Definition A program component C (either an assignment statement or a loop header) is said to be loop dependent on a loop header L (either a *for* loop header or a *while* loop header) if C is embedded in the loop statement whose header is L, or if $C = L$. In the text we will denote this dependence as $L\ \delta^L\ C$, and in the dependence graph it will be represented by arcs of the form shown in Fig. 1(a). ■

In Fig. 2, the reader can find examples of the loop dependence relation. We now proceed to discuss the next three types of relations.

Definition Assume a component C and n different loop headers $L_1, \ldots, L_n$, such that

(1) $L_i\ \delta^L\ C$ $\quad i = 1, \ldots, n$

(2) $L_i\ \delta^L\ L_{i+1}$ $\quad i = 1, \ldots, n-1$

(3) C is not loop dependent on any other loop header.[1]

An instance $C(k_1, \ldots, k_n)$ of a component C is defined as the component C when for $1 \leq i \leq n$ the loop whose header is L_i is executing its k_i-th iteration. Notice that when L_i is a *for* loop header, k_i will be the value of its index variable. This is because we have restricted the initial value and the increment in *for* loops to one. When all $L_1, \ldots, L_n$ are *for* loop headers, $(k_1, \ldots, k_n)$ is called an index set. ■

Component instances have two sets of variable elements associated with them: a set of inputs and a set of outputs. The set of inputs are those variable elements fetched by the component instance, and the set of outputs are those variable elements modified by it. When the component is a *while* loop header, the set of outputs is empty, and the set of inputs is given by the Boolean expression in the header. When the component is a *for* loop header, say F, the instance $F(\ldots, 1)$ (i.e., F at the first iteration of the loop of which it is a header) has the index variable as output, and the loop limit as input if it is a variable. The instances $F(\ldots, k)$, $k > 1$, also have the index variable as input. For an assignment statement the set of inputs is determined by the expression, and the set of outputs by the variable on the left-hand side. Notice that the set of outputs of a program component has always only one element.

Definition Consider two, not necessarily distinct components C_r and C_s and one instance of each, $C_r(\bar{i})$ and $C_s(\bar{j})$, such that $C_r(\bar{i})$ is executed before $C_s(\bar{j})$ in the proper serial execution of the program. We say that

(a) $C_s(\bar{j})$ is output dependent on $C_r(\bar{i})$, denoted $C_r(\bar{i})\ \delta^O\ C_s(\bar{j})$ iff they have the same output variable element.

(b) $C_s(\bar{j})$ is antidependent on $C_r(\bar{i})$, denoted $C_r(\bar{i})\ \delta^A\ C_s(\bar{j})$, iff the output variable element of $C_s(\bar{j})$ is an input variable element of $C_r(\bar{i})$.

(c) $C_s(\bar{j})$ is flow dependent on $C_r(\bar{i})$, denoted $C_r(\bar{i})\ \delta\ C_s(\bar{j})$ iff the output variable element of $C_r(\bar{i})$ is an input variable element of $C_s(\bar{j})$, and there is no other instance $C_t(\bar{k})$ executed after $C_r(\bar{i})$ but before $C_s(\bar{j})$ such that $C_r(\bar{i})\ \delta^O\ C_t(\bar{k})$. (Intuitively, the value computed by $C_r(\bar{i})$ is actually used by $C_s(\bar{j})$.)

A program component C_r is said to be output, anti, or flow dependent on component C_s iff there exist $\bar{i}$ and $\bar{j}$ such that $C_r(\bar{i})$ is, respectively, output, anti, or flow dependent on $C_s(\bar{j})$. The arcs representing the previous three relations are given in Figs. 1(b)-(d). ■

The presence of array variables poses particular problems in the computation of the previous three relations. For example, assume two assignment statements A_1 and A_2 such that an array variable V appears on the left-hand side of A_1 and on the right-hand side of A_2. To determine whether $A_1\ \delta\ A_2$, we need to determine whether for some instances $A_1(\bar{i})$ and $A_2(\bar{j})$, with $A_1(\bar{i})$ executed before $A_2(\bar{j})$, the element of V modified by $A_1(\bar{i})$ is the same as the element of V fetched by $A_2(\bar{j})$. If the subscript of V is a constant in both statements, this is a trivial task. The other extreme is when it is not possible to make such a determination at compile time because the subscript of V is a function of the program input. In this case, we have to be conservative and assume that the flow dependence relation holds in order to guarantee the correctness of our transformations. An intermediate case is when the subscript of V in $A_1(\bar{i})$ is a (possibly multidimensional) function $\bar{f}(\bar{i})$, and $A_2(\bar{j})$ a function $\bar{g}(\bar{j})$. U. Banerjee [Bane76], [Bane79]has developed efficient algorithms to determine whether $\bar{f}(\bar{i}) = \bar{g}(\bar{j})$ for some $\bar{i}$ and $\bar{j}$ when both $\bar{f}$ and $\bar{g}$ are polynomials (which is often the case). We do not know of any efficient algorithm to make such a determination when any of $\bar{f}$ or $\bar{g}$ is a more general nonlinear function. In such cases, we are again conservative and assume that the flow dependence relation holds.

Another problem is caused by the fact that some instructions may be executed conditionally; in this case, as before, we have to be conservative and assume the dependence when in doubt.

The fifth relation between components is that of input dependence.

Definition A component C_1 is said to be input dependent on another component C_2, denoted $C_2\ \delta^I\ C_1$, iff the same variable name appears as input to both C_1 and C_2. Notice that the δ^I relation is symmetric. Input dependence is represented by arcs like the one shown in Fig. 1(e). ■

For some of the transformations described later, we will need more information than that provided by the dependence graph as described. This additional information will be conveyed by the internal flow graph which describes the internal structure of each component. The internal flow graph of a *while* header will be the syntax tree of the Boolean expression in the header. For an assignment statement, the internal flow graph will be a tree with the left-hand side variable as root and the syntax tree of expression as the only subtree connected to it. For the arcs in the internal

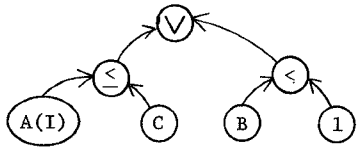
[1] Notice that if C is a loop header, then $C = L_n$.

flow graph, we will use the same type of arc used to represent flow dependence (Fig. 1(b)) since the concepts represented in both cases are the same. In *while* headers and assignment statements, this arc points towards the root.

Example The *while* header

while (A(I) $\leq$ C) $\vee$ (B < 1)

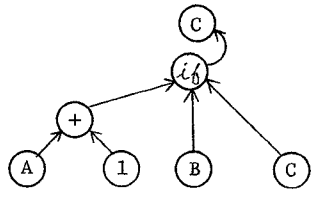
has the following internal flow graph.



The assignment statement

C ← *if* B *then* A + 1
else C

has the following internal flow graph

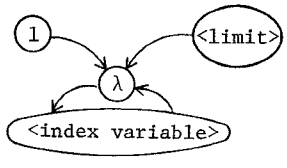


■

The internal flow graph of a *for* header of the form

for <index-variable> = 1 *to* <limit>

is as follows



where the λ operator performs all functions of the *for* header, like assigning one to the index variable the first time it is executed, and adding one to the index variable and comparing the result with the limit on subsequent executions.

The nodes in the internal flow graph representing variables or constants are called atoms. Because of this, a dependence graph including the internal flow graph is said to be represented at the atomic level. In such a graph, the data dependence arcs (anti, output and flow) will emanate and arrive at the atoms that cause these dependences. In Fig. 3, we show part of the dependence graph at the atomic level for the program in Fig. 2(a).

Later in the paper we are going to treat some compound statements as a single unit. For this purpose, we will name a compound statement with the label of the statement header. This means that such a label will have two functions; however, in the text the specific meaning of the label will always be clear from the context. In the graphic representation, a node representing a whole compound statement will be represented by two concentric circles; such nodes will be called compound nodes. The concepts of instance, dependence, sets of inputs, and sets of outputs can be very easily extended to deal with compound statements. In this paper, however, we will rely on the intuition of the reader and will not define such concepts.

A final comment. Since dependence graphs may become quite complex when all arcs are drawn, we will represent only those arcs of interest in the examples discussed in the rest of the paper.

3. Arc Transformations

In this section, we present some source program transformations that will modify the dependence graph by either removing arcs or breaking cycles. These transformations are renaming, expansion, node splitting, and forward substitution.

3.1 Renaming

Sometimes scalar or structured variables are used for different purposes at different points in a program. This is done sometimes to increase the readability of the program and often to decrease memory requirements. This approach is adequate for sequential programming. However, the use of the same memory location for different purposes could impose unnecessary sequentiality constraints on parallel programs. The renaming transformation will assign different names to different uses of the same variable, and as a consequence some output dependence arcs and some antidependence arcs will be removed from the dependence graph of the program.

Example 3.1 The program shown in Fig. 2(a) uses the variable A in three statements inside the *for* loop; this introduces a large number of arcs in the dependence graph (Fig. 2(b)). The variable A can be replaced by two variables, $A^{(1)}$ and $A^{(2)}$, as shown in Fig. 2(b). This eliminates several output dependence and antidependence arcs. ■

We now present an algorithm for renaming scalar variables. A powerful algorithm for renaming structured variables is an open problem.

Renaming Algorithm for Scalar Variables

Assuming a program, P, and a scalar variable, say A, in P.

[1] Build G, the dependence graph of P at the atomic level.

[2] Consider G', the subgraph of G consisting of the intercomponent flow dependence arcs only (i.e., we drop all other arcs including the flow dependence arcs in the internal flow graph). Find the connected components of G' where A appears. Assume there are k such components C_1, C_2, ..., C_k.

[3] Introduce k different variable names $A^{(1)}$, $A^{(2)}$, ..., $A^{(k)}$, none of them used in P. For $1 \leq i \leq k$, replace the occurrences of A in C_i by $A^{(i)}$. ■

Example 3.2 Part of the dependence graph G' for the program in Fig. 2(a) is shown in Fig. 4. Since there are two connected components involving A, we introduce two new variables, $A^{(1)}$ and $A^{(2)}$ to obtain the program in Fig. 2(b). ■

The concept of scalar renaming has been known for a number of years [AhU173].

3.2 Expansion

Expansion is a transformation that is not as well known as renaming (though it is implemented in the compilers for both the Burroughs BSP and the CRAY-1), but is of prime importance in compiling for parallel machines. The object of expansion is to take a variable that was used inside a *for* loop and to change it into a higher dimensional array (or other suitable data aggregate). Like renaming, this process reduces the number of arcs in the dependence graph. In this case, this is achieved by giving each iteration of the *for* loop its own set of locations.

Example 3.3 The dependence graph in Fig. 2(b) includes many output dependence and antidependence arcs because of the scalar variables $A^{(1)}$, $A^{(2)}$, and Y. If these variables are expanded into arrays by the algorithm described below, we obtain the program in Fig. 2(c) whose dependence graph is much simpler. ■

We now describe an algorithm for the expansion of scalar variables. To this end, we will need three definitions.

Definition A component or compound statement C is said to be directly *for* loop dependent on a *for* loop header F, denoted $F\ \hat{\delta}^L\ C$, iff

(1) $F\ \delta^L\ C$, and

(2) there is no other *for* loop header F' such that $F\ \delta^L\ F'\ \delta^L\ C$. ■

Definition A sequence of *for* loop headers $F_0, F_1, \ldots, F_m$ is said to form a chain iff $F_i\ \hat{\delta}^L\ F_{i+1}$ $0 \leq i \leq m - 1$. ■

In other words, a sequence of *for* loop headers forms a chain when their respective loops are nested in the order indicated in the sequence, and there is no other *for* loop in the nesting.

Definition Given a component or compound statement C, a *for* loop header F such that $F\ \hat{\delta}^L\ C$, and a scalar variable V, we say that C forwards V to the next iteration of F iff

(1) V is an output variable of C;

(2) there is no statement $\mathcal{D}$, with $F\ \delta^L\ \mathcal{D}$, which appears after C in the text of the loop such that V is an output variable of $\mathcal{D}$;

(3) there is no *while* loop header W such that $F\ \delta^L\ W\ \delta^L\ C$; and

(4) in the execution of the body of F, V could be fetched before it is modified. ■

Intuitively, C forwards V to the next iteration of a *for* loop F if the only value given to V by C at any iteration of F is still the value of V when the next iteration of F starts, and that value could be used in that iteration.

Scalar Expansion Algorithm

Consider an output variable, V, of a *for* loop F_0 with dependence graph G. In the algorithm, we will assume that $V(I_1, I_2, \ldots, I_m)$ and $V(I_1, I_2, \ldots, I_m, 0, \ldots, 0)$, $m \geq 0$, represent the same memory location (if m = 0 we have V and V(0, ..., 0).)

[1] For all components C in G, execute step [2]. Then go to step [5].

[2] Let $F_0, F_1, \ldots, F_m$ form a chain of *for* loop headers, such that

$$F_m\ \hat{\delta}^L\ C.$$

Let $I_0, I_1, \ldots, I_m$ be the index variables of these headers.

If V is an input variable of C, execute step [3].

If V is an output variable of C, execute step [4].

[3] Let $n \leq m$ be the largest number such that there exist a component $\mathcal{D}$ with $L_n\ \delta^L\ \mathcal{D}$, and V is an output variable of $\mathcal{D}$.

Replace all occurrences of V on the right-hand side of C by

(1) $V(I_0\text{-}1, \ldots, I_{n-1}\text{-}1, I_n)$ if there is a component $\mathcal{D}$ forwarding V to the next iteration of L_n such that $\mathcal{D}\ \delta\ C$.

(2) $V(I_0\text{-}1, \ldots, I_{n-1}\text{-}1, I_n\text{-}1)$ otherwise.

[4] Replace the occurrence of V on the left-hand side of C by

(1) $V(I_0\text{-}1, \ldots, I_{m-1}\text{-}1, I_m)$ if C forwards V to the next iteration of L_m.

(2) $V(I_0\text{-}1, \ldots, I_{m-1}\text{-}1, I_m\text{-}1)$ otherwise.

[5] For all loops (*for* or *while* loops) L_m inside L_0 execute step [6]. Then go to step [7].

[6] Let $F_0, F_1, \ldots, F_{m-1}$ form a chain of *for* loop headers with $F_{m-1}\ \hat{\delta}^L\ L_m$. If V is an output variable of L_m, insert the assignment statement

$$V(I_0\text{-}1, \ldots, I_{m-2}\text{-}1, \alpha) = V(I_0\text{-}1, \ldots, I_{m-1}\text{-}1, \beta)$$

immediately after the end statement of L_m.

Here β = $\begin{cases} 0 \text{ if } L_m \text{ is a } \textit{while} \text{ loop} \\ \max(u\ell_m, 0) \text{ if } L_m \text{ is a } \textit{for} \text{ loop with upper limit } u\ell_m \end{cases}$

and

$$\alpha = \begin{cases} I_{m-1} & \text{If } L_m \text{ forwards V to the next iteration of } F_{m-1} \\ I_{m-1}-1 & \text{otherwise} \end{cases}$$

[7] Immediately after the *end for* of F_0, insert the statement V = V($u\ell_0$) where $u\ell_0$ is the upper limit of F_0. ■

As was the case for renaming, a good expansion algorithm for array variables is an open problem.

3.3 Node Splitting

The node splitting transformation attempts to break cycles in the dependence graph by repositioning antidependence arcs. This is achieved through the introduction of new assignment statements.

Example 3.4 The dependence graph in Fig. 2(c) includes a cycle which can be broken if the arc representing A_4 δ^A A_5 is repositioned. To do this, we split A_4 into two assignment statements A_4' and A_4'' as shown in Fig. 2(d). ■

Node Splitting Algorithm

Consider a dependence graph G at the component level with the loop dependence arcs removed, and a cycle C in G.

[1] If the cycle C disappears when G is represented at the atomic level, then C includes an antidependence arc, say A, and the algorithm can proceed to step [2]. If C does not disappear, stop.

[2] If the arc A emanates from an atom a in component C, then introduce a new assignment statement of the form T ← a, and replace all occurrences of a in C by T, where T is a variable not appearing anywhere in the original program.

[3] Apply the expansion transformation to T. ■

Example 3.5 Part of the dependence graph at the atomic level for the program in Fig. 2(c) is shown in Fig. 5. Notice that the cycle in the graph of Fig. 2(c) disappears in Fig. 5. To remove the cycle, we introduce the statement T = X(I+1) and replace A_4 by $A^{(2)}$(I) = T + X(I-1). After T is expanded, we will obtain the program in Fig. 2(d). ■

The expansion and node splitting transformations, as discussed above, change the source program by introducing arrays and assignment statements. The goals of these two transformations can also be achieved by architectural means. Consider, for example, the DO ALL instruction of the Burroughs FMP multiprocessor [LuBa80]. Variables in the body of DO ALL are defined as local when they belong to neither the set of inputs nor the set of outputs of the DO ALL. A copy of all the local variables is created in the local memory of each processor before a DO ALL starts execution. This has the same effect as expansion. Also, all variables in the set of inputs of the DO ALL are fetched before the DO ALL starts execution. This produces the same effect as node splitting.

3.4 Forward Substitution

The forward substitution transformation eliminates flow dependence arcs from component level dependence graphs by substituting the right-hand side expression of an assignment statement, , into the right-hand sides of other assignment statements. The main use of this transformation is that it could be applied before tree-height reduction [Kuck78], enhancing the result of this last transformation.

Example 3.6 Consider the program segment in Fig. 6(a). Assume that only the variable F is used outside the segment. After applying forward substitution, we obtain the program segment in Fig. 7(a). The atomic dependence graph, if we assume that expressions are evaluated from left to right, is shown in Fig. 7(b). However, if we do not assume any evaluation order and apply tree-height reduction, we obtain the dependence graph in Fig. 7(c), which is much better than the graph in Fig. 6(b) from the parallel processing point of view. ■

Assume a set of consecutive assignment statements A_1, A_2, ..., A_n in a program P. To forward substitute a given A_i whose left-hand side is a scalar variable, S, we proceed as follows.

Scalar Forward Substitution Algorithm

[1] Apply renaming to all scalars on the right-hand side of A_i.

[2] For all A_j, j > i, such that

(1) A_i δ A_j

(2) there is no A_k, i < k < j such that A_i δ^A A_k (since scalar variables have been renamed, this antidependence is always caused by an array variable), replace the expression on the right-hand side of A_i by all occurrences of S in A_j.

[3] Apply dead code elimination [Grie71]. ■

4. Dependence Graph Abstraction

Graph abstraction is a process by which a set of nodes and their internal arcs are merged into a single compound node. Any arcs incident to (or from) the set are made incident to (or from) the compound node. Graph abstraction has been used in many areas of computer science. In particular, it has been used to organize optimization in several ways. For example, an interval is a graph

abstraction [Cock70] used in data flow analysis. Graph abstraction has also been used to control the scope of optimization as in the SIMPL optimizer of [ZeBa74], which optimizes structured blocks from the inside out. We use graph abstraction in yet a different way. Graph abstraction can be used to isolate sets of statements that can be translated into high quality machine code only when taken as an ensemble. Two examples of this type of graph abstraction will be presented.

4.1 Loop Distribution

Loop distribution abstracts dependence graphs by finding and merging each strongly connected component in the body of a loop along with the loop header node into a compound node. (A strongly connected component (SCC) is a maximal set of nodes such that there is a path between any pair of nodes in the set.) Similarly, each loop body node not in any SCC, an independent node (IN), is merged with the loop header node into another compound node. Fig. 8 shows how the node merging in loop distribution is performed.

The following algorithm describes loop distribution. The most time-consuming step in the algorithm is step 1, finding the SCCs. However, it takes only $O(n \log n)$ time on a loop containing n statements if a depth-first algorithm such as Tarjan's algorithm is used [AhHU74]. This compares favorably with the fast data flow analysis algorithms such as [GrWe76].

Loop Distribution Algorithm

Consider a *for* loop F_0 whose body consists of the statements (simple or compound) $S_1, \ldots, S_n$. To distribute F_0, we proceed as follows.

[1] Compute the dependence graph G for F_0 and $S_1, \ldots, S_n$.

[2] Delete F_0 and create a *for* loop header node F_{oj} for each SCC, and each IN in the dependence graph. Make each statement in an SCC or IN loop dependent on the loop header associated with the SCC or IN, and flow dependent if the statement refers to the loop index (i.e., a *for* loop is created for each SCC and each IN).

[3] Build a new dependence graph by creating a compound node for each *for* loop. ■

The loop distribution algorithm can implement several optimizations, depending on how the dependence graph is computed in step 1. We will give two specific examples.

4.1.1 Loop Distribution for Vector Processors

The first optimization uses loop distribution to generate vector operations from multistatement loops. This is achieved by constructing a dependence graph consisting of flow, anti, and output dependence arcs for the multistatement loop and performing the loop distribution algorithm. The dependence graph output from loop distribution in this case is called a partial order graph, and each node in this graph is called a π-block. (The term π-block stems from the fact that loop distribution partitions the nodes in the graph into equivalence classes.)

Two types of π-blocks are derived. π-blocks whose bodies are INs represent vector operations, the goal of the optimization. π-blocks, whose bodies are SCCs, are called recurrences. (As a rule of thumb, there is approximately one recurrence per loop in scientific source programs.) Although recurrences, nonvector operations, are not the most efficient operations on vector and array processors, we have found that relatively few recurrences are intractable. Most recurrences are SCCs connected by only flow dependences, primarily because loop distribution is applied after several optimizations which remove anti and output dependence arcs (Section 3). These recurrences are most often linear recurrences, such as the row sum of a matrix, which can be speeded up [Kuck78] but are still slower than vector operations on vector processors. ([BCKT79] is a recent description of results in this area.) Loop distribution applied to a linear recurrence in effect abstracts the SCC to a single node representing a call to a linear recurrence solver. Other types of SCC that occur frequently are: Boolean recurrences which can be substantially speeded up [BaGK80], and simple nonlinear recurrences [Park77].

The partial order graph constructed in step 3 of the Loop Distribution Algorithm is a directed acyclic graph. It can be used to schedule the vector operations and recurrences on a parallel processor. The longest chain in the partial order graph defines the minimum execution time for the original source program loop. The maximum width or anti-chain in the graph defines an upper bound on the number of processors that can be used in parallel computation.

Example 4.1 Loop distribution for vector processors produces the program in Fig. 2(e) when applied to the program in Fig. 2(d). All statements become vector operations except for statements A_4 and A_5 which constitute a linear recurrence. In the transformed program, the statements are topologically sorted by the partial order graph. ■

4.1.2 Loop Distribution for Memory Management

A second application of loop distribution is in memory management; we call this name clustering [AbKL79].

Example 4.2 [AbKL79] Consider the program in Fig. 9(a). If each array referenced in this program is on a distinct page, or distinct sets of pages, then the F_1 loop requires 9 data pages to execute efficiently (with a minimum of page faults). After this type of loop distribution is applied, the transformed program (Fig. 9(b)) requires only 5 data pages to execute efficiently. Loop distribution has improved the program's data locality. ■

The input to loop distribution for memory management is a dependence graph constructed for flow, anti, output, and input dependences. Loop

distribution in this case does not use SCC but name clusters defined next. Therefore, in the algorithm above SCC should be replaced by name cluster.

Definition The set of variables referenced in a set of statements is called a name set (NS) and is a function of some statement set (SS). We can also compute the set of statements referencing any variable in a name set.

Let SS_0 be any statement in a given loop. Call its name set NS_0, and find the statement set of NS_0; call it SS_1. Loop distribution iterates this sequence until a stable statement set is found; this set is called a name cluster. ■

One might assume that in an average loop, the logical flow of the loop would connect all its statements into one name cluster; however, we have found that by using loop distribution the data page requirements of programs can be reduced by a factor of 6 [AbKL79].

Loop distribution for memory management can be compared with global register assignment (GRA) algorithms. Sophisticated GRA algorithms such as [Beat74] generate roughly the SS of each variable referenced in the loop. ("Roughly" here implies that if at some point in the loop the variable is dead, then a new SS is started.) A register is allocated for each SS. Computing SSs requires a connected component computation. GRA algorithms in this class can reallocate a register several times within a loop, but clustering does not reallocate page frames because of the relatively higher cost of page swapping.

4.2 Loop Fusion

As a graph abstraction, loop fusion is used selectively to merge two compound *for* loop nodes. Thus, it is nearly the inverse of loop distribution. But where loop distribution is applied globally to a loop, loop fusion is applied selectively.

Loop Fusion Algorithm

Consider two *for* loops F_0, and F_1 with the following characteristics:

(1) Both F_0 and F_1 have the same loop limit.

(2) F_0 and F_1 are consecutive in the source program with F_0 appearing before F_1. (If they are not consecutive, then try to make them consecutive by moving the statements separating F_0 and F_1 before F_0 or after F_1 whenever possible.)

[1] Let $S_{0.1}$, ..., $S_{0.n}$ be the statements in the body of F_0, and $S_{1.1}$, ..., $S_{1.m}$ the statements in the body of F_1. Temporarily create a *for* loop containing $S_{0.1}$, ..., $S_{0.n}$, $S_{1.1}$, ..., $S_{1.m}$ (renaming the index variable occurrences if necessary), and compute its dependence graph G.

[2] If G contains any arc from $S_{1.i}$ to $S_{0.j}$ for some $1 \leq i \leq n$, $1 \leq j \leq m$, then fusion is not possible. Otherwise, replace the loops F_0 and F_1 by a single loop containing the body of both loops ■

In the past, loop fusion has been applied globally to reduce the overhead of loop control [AlCo72], [Love77]. We will show that by applying loop fusion selectively with different criteria, different optimizations can be realized.

Loop Fusion for Virtual Memory Management

Above we saw that loop distribution applied to the proper dependence graph can reduce the data memory requirements of a source program. Loop fusion can be subsequently used to reduce unnecessary swapping. The criteria used to select pairs of loops for fusion in this case is that the NS (Name Set is defined above) of one loop is contained in the NS of the other. The following example illustrates loop fusion for virtual memory management.

Example 4.3 If the program in Fig. 9(b) is input to the loop fusion algorithm, the program will be transformed as shown in Fig. 10. In this case, the NS of each of the loops before loop fusion is:

Loop	Name Set (NS)
F_1	{A,B,C,G,H}
F_2	{D,E,F,X}
F_3	{D,E,F}

The NS for F_3 is contained in the NS for F_2, all conditions for fusion are satisfied, so F_2 and F_3 are fused. Page swapping has been reduced because once a page of D, E, and F are loaded, all operations using these pages are performed. ■

Loop Fusion for Vector Register Processors

Processors having vector registers such as the CRAY-1 present novel requirements for code generators. We will present a sequence of transformations which performs very well in this environment. (This sequence is contained in a vectorizer for pipelined machines described in [KKLW80].)

- Loop distribution for vector processors first isolates the recurrences from the vector operations.
- Loop fusion is applied to increase the NS of each loop until it is as large as the number of vector registers available.
- Loop blocking [AbKL79] transforms single *for* loops into doubly nested loops. The inner loop is set to the size of the vector registers. The outer loop increments in steps of the register size through the original loop range.
- Loop interchanging [Kuck80] attempts to avoid

recurrences on the inner loop and to reduce memory register traffic by interchanging loops when possible.

- Register assignment assigns vector registers globally. It also generates loads and stores at the entry and exit of each register allocation block.

We present this optimization sequence at this juncture because of the role played by loop fusion in particular. The initial loop distribution generates vector operations that are much larger than the register size. Loop blocking remedies that. However, without loop fusion the outer loop overhead would have to be paid for each vector operation. At the same time, fusing all of the outer loops together may over-allocate the available vector registers. Therefore, the loop fusion criterion used in this case is whether the NS of the fused loop will be larger than the number of available vector registers. Once fused loops with NS approximately as large as the available registers are created, register assignment need not be as complex. (We might label it a global few-to-few assignment strategy, using Day's terminology [DayW70]. The only other published method to optimize vector register assignment is found in [DuKu78], which describes a global assignment based on usage counts with no reallocation of vector registers.

5. Conclusion

The techniques of this paper have been implemented and used in compiling to improve the performance of ordinary programs for several architectures. The transformations of Section 3 remove dependence arcs and hence increase independence between nodes, while those of Section 4 abstract from a graph parts which are convenient for code generation. By removing cycles, they yield a DAG that also is convenient for scheduling.

It is important to realize that the resulting graph usually consists of many nodes that can be compiled directly into the machine languages of current high-performance machines. Array operations arise from independent nodes after loop distribution. The recurrences arising from strongly connected component cycles are often linear. Many machines have reduction instructions and some (e.g., BSP) have a more general linear recurrence solving instructions or functions. The remaining parts of a dependence graph must be executed using scalar instructions. Application of these ideas to machines that can execute many scalar operations at once is discussed in [PaKL80].

References

[AbKL79] W. Abu-Sufah, D. Kuck, and D. Lawrie, "Automatic Program Transformations for Virtual Memory Computers," Proc. of the 1979 Nat'l. Computer Conf., pp. 969-974, June 1979.

[AcDe79] W. B. Ackerman and J. B. Dennis, "Val--A Value-Oriented Algorithmic Language: Preliminary Reference Manual," Lab. for Computer Science (TR-218), MIT, Cambridge, MA, June 1979.

[AhHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, MA, 1974.

[AhU173] A. V. Aho and J. D. Ullman, The Theory of Parsing, Translation, and Compiling, Vol. 2: Compiling, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[AlCo72] F. E. Allen and J. Cocke, "A Catalogue of Optimizing Transformations," in Design and Optimization of Compilers (R. Rustin, Ed.), Prentice-Hall, Inc., NJ, pp. 1-30, 1972.

[ArGP78] Arvind, K. P. Gostelow, and W. Plouffe, "An Asynchronous Programming Language and Computing Machine," University of California at Irvine, CA, Dept. of Information and Computer Science Rpt. 114a, Dec. 1978.

[AsMa75] E. Ascroft and Z. Manna, "Translating Program Schemes to While-Schemas," SIAM J. on Computing, Vol. 4, No. 2, pp. 125-146, June 1975.

[BaGK80] U. Banerjee, D. Gajski, and D. J. Kuck, "Array Machine Control Units for Loops Containing IFs," Proc. of the 1980 Int'l. Conf. on Parallel Processing, Harbor Springs, MI, pp. 28-36, Aug. 1980.

[Bake77] B. S. Baker, "An Algorithm for Structuring Flow Graphs," J. of the ACM, Vol. 24, No. 1, pp. 98-120, Jan. 1977.

[Bane76] U. Banerjee, "Data Dependence in Ordinary Programs," M.S. thesis, Univ. of Ill. at Urbana-Champaign, Dept. of Comput. Sci. Rpt. No. 76-837, Nov. 1976.

[Bane79] U. Banerjee, "Speedup of Ordinary Programs," Ph.D. thesis, Univ. of Ill. at Urb.-Champ., Dept. of Comput. Sci. Rpt. No. 79-989, Oct. 1979.

[BCKT79] U. Banerjee, S. C. Chen, D. J. Kuck, and R. A. Towle, "Time and Parallel Processor Bounds for Fortran-Like Loops," IEEE Trans. on Computers, Vol. C-28, No. 9, pp. 660-670, Sept. 1979.

[Beat74] J. C. Beatty, "Register Assignment Algorithm for Generation of Highly Optimized Object Code," IBM J. of Res. and Dev., Vol. 18, No. 1, pp. 20-39, Jan. 1974.

[BoJa66] C. Bohm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," Comm. of the ACM, Vol. 9, No. 5, pp. 366-371, May 1966.

[Cock70] J. Cocke, "Global Subexpression Elimination," SIGPLAN Notices, Vol. 5, No. 7, pp. 20-24, 1970.

[DayW70] W. H. E. Day, "Compiler Assignment of Data Items to Registers," IBM Systs. J., Vol. 9, No. 4, pp. 281-317, 1970.

[DuKn78] D. D. Dunlop and J. C. Knight, "Register Allocation in the SL/1 Compiler," Proc. of the 1978 LASL Workshop on Vector & Parallel Processors, LA-7491-C, Los Alamos, NM, pp. 205-211, Sept. 1978.

[Grie71] D. Gries, Compiler Construction for Digital Computers, Wiley & Sons, NY, 1971.

[GrWe76] S. L. Graham and M. Wegman, "A Fast and Usually Linear Algorithm for Global Flow Analysis," J. of the ACM, Vol. 23, No. 1, pp. 172-202, 1976.

[KKLW80] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "Analysis and Transformation of Programs for Parallel Computation," to appear in Proc. of the Fourth Int'l. Computer Software & Applications Conf., Oct. 1980.

[Kuck78] D. J. Kuck, The Structure of Computers and Computations, Vol. I, John Wiley & Sons, Inc., NY, 1978.

[Kuck80] D. J. Kuck, Class Notes for C.S. 433, Univ. of Ill. at Urb.-Champ., Dept. of Comput. Sci., 1979.

[KuMC72] D. J. Kuck, Y. Muraoka, and S. C. Chen, "On the Number of Operations Simultaneously Executable in FORTRAN-Like Programs and Their Resulting Speed-Up," IEEE Trans. on Computers, Vol. C-21, No. 12, pp. 1293-1310, Dec. 1972.

[Love77] D. B. Loveman, "Program Improvement by Source-to-Source Transformation," J. of the ACM, Vol. 20, No. 1, pp. 121-145, Jan. 1977.

[LuBa80] S. F. Lundstrom and G. H. Barnes, "A Controllable MIMD Architecture," Proc. of the 1980 Int'l. Conf. on Parallel Processing, pp. 19-27, Aug. 1980.

[PaKL80] D. A. Padua, D. J. Kuck, and D. H. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," Special Issue on Parallel Processing, IEEE Trans. on Computers, Vol. C-29, No. 9, pp. 763-776, Sept. 1980.

[Park77] D. S. Parker, Jr., "Nonlinear Recurrences and Parallel Computation," in High Speed Computer and Algorithm Organization, pp. 317-320, Academic Press, Inc., 1977.

[ZeBa74] M. V. Zelkowitz and W. G. Bail, "Optimization of Structured Programs," Software Practice and Experience, Vol. 4, No. 1, pp. 51-57, 1974.

Fig. 1. Five types of dependence graph arcs

(a) loop dependence
(b) flow dependence
(c) output dependence
(d) antidependence
(e) input dependence

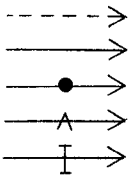
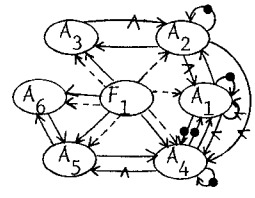


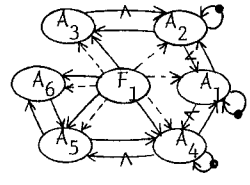
Fig. 2. Successive application of four transforms to a program

```
F1: for I = 1 to N
A1:   A = A + 1
A2:   Y = A ↑ 2
A3:   Z(I) = Y + V(I)
A4:   A = X(I+1) + X(I-1)
A5:   X(I) = W(I) + 1
A6:   W(I+1) = X(I) + 1
      end for
```



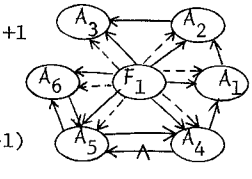
Original program 2(a)

```
F1: for I = 1 to N
A1:   A(1) = A(2) + 1
A2:   Y = A(1) ↑ 2
A3:   Z(I) = Y + V(I)
A4:   A(2) = X(I+1) + X(I-1)
A5:   X(I) = W(I) + 1
A6:   W(I+1) = X(I) + 1
      end for
```



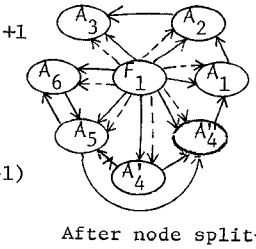
After renaming 2(b)

```
F1: for I = 1 to N
A1:   A(1)(I-1) = A(2)(I-1)+1
A2:   Y(I-1) = A(1)(I-1)↑2
A3:   Z(I) = Y(I-1) + V(I)
A4:   A(2)(I) = X(I+1)+X(I-1)
A5:   X(I) = W(I) + 1
A6:   W(I+1) = X(I) + 1
      end for
```



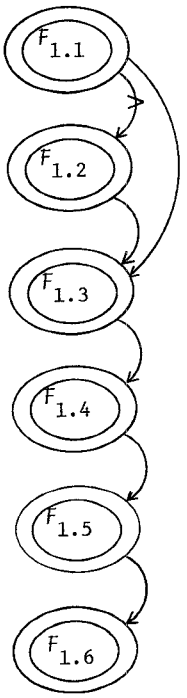
After expansion 2(c)

```
F1: for I = 1 to N
A1:   A(1)(I-1) = A(2)(I-1)+1
A2:   Y(I-1) = A(1)(I-1)↑2
A3:   Z(I) = Y(I-1) + V(I)
A'4:  T(I-1) = X(I+1)
A"4:  A(2)(I) = T(I-1)+X(I-1)
A5:   X(I) = W(I) + 1
A6:   W(I+1) = X(I) + 1
      end for
```



After node splitting 2(d)

```
F1.1:  for I = 1 to N
A'4 :    T(I-1) = X(I+1)
       end for
F1.2:  for I = 1 to N
A5  :    X(I) = W(I) + 1
A6  :    W(I+1) = X(I) + 1
       end for
F1.3:  for I = 1 to N
A"4 :    A(2)(I) = T(I-1)+X(I-1)
       end for
F1.4:  for I = 1 to N
A1  :    A(1)(I-1) = A(2)(I-1)+1
       end for
F1.5:  for I = 1 to N
A2  :    Y(I-1) = A(1)(I-1)+2
       end for
F1.6:  for I = 1 to N
A3  :    Z(I) = Y(I-1) + V(I)
       end for
```



After distribution
2(e)

Fig. 3. Partial dependence graph at the atomic level for the program in Fig. 2(a)

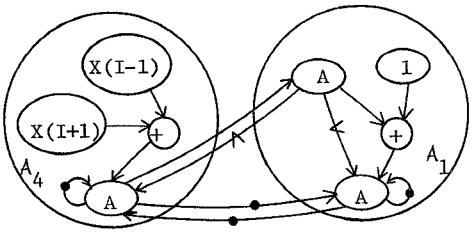


Fig. 4. Partial dependence graph at the atomic level for the program in Fig. 2(a)

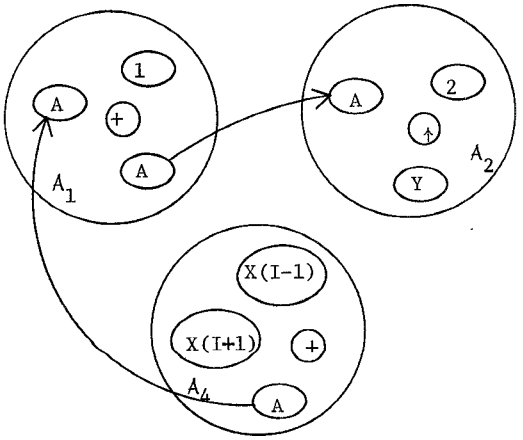


Fig. 5. Partial dependence graph at the atomic level for the program in Fig. 2(c)

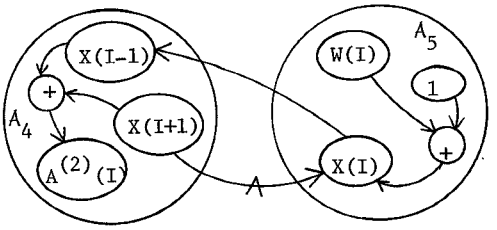
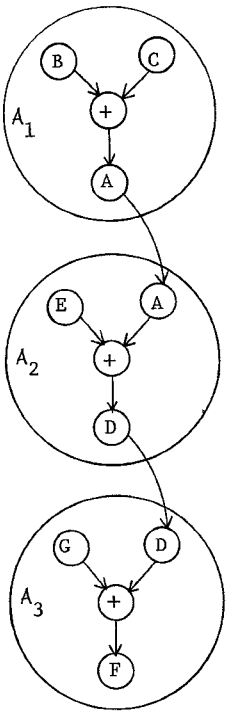


Fig. 6. Original program for Example 3.6

```
A1:  A = B + C
A2:  D = E + A
A3:  F = G + D
```

(a)

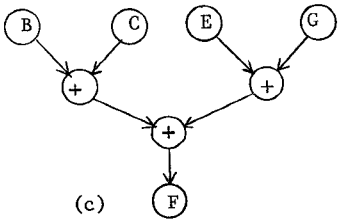


(b)

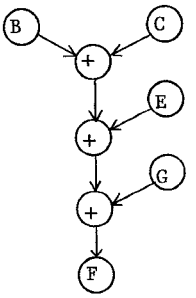
Fig. 7. Transformed program for Example 3.6 and two possible atomic dependence graphs

```
A'3:  F = B + C + E + G
```

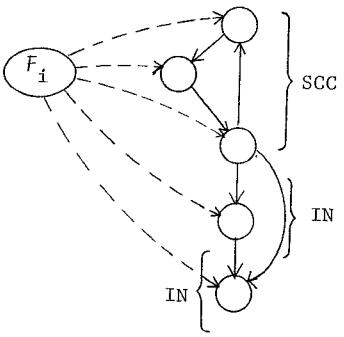
(a)



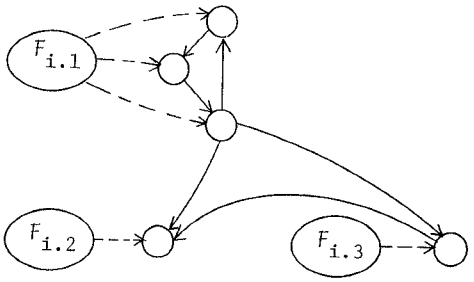
(c)



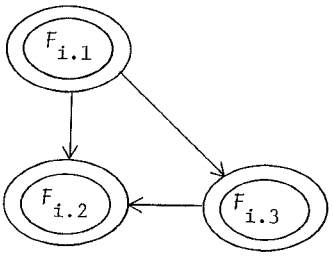
(b)



before abstraction
(a)



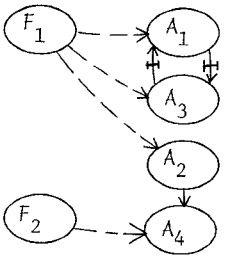
replicated loop header
(b)



resulting abstracted graph
(c)

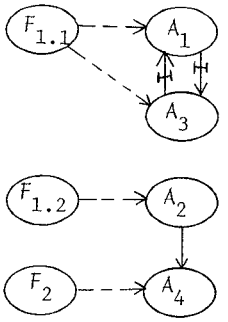
Fig. 8. Loop distribution as a graph abstraction

```
F1: for I = 1 to N
A1:   A(I) = B(I) + C(I)
A2:   D(I) = E(I)+F(I)+X(I)
A3:   G(I) = B(I) + H(I)
      end for
F2: for J = 1 to N
A4:   E(J) = D(J) * F(J)
      end for
```



Original program
(a)

```
F1.1: for I = 1 to N
A1  :   A(I) = B(I) + C(I)
A3  :   G(I) = B(I) + H(I)
      end for
F1.2: for I = 1 to N
A2  :   D(I)=E(I)+F(I)+X(I)
      end for
F2  : for J = 1 to N
A4  :   E(J) = D(J) * F(J)
      end for
```



Transformed program
(b)

Fig. 9. Loop distribution for memory management

```
F1: for I = 1 to N
A1:   A(I) = B(I) + C(I)
A3:   G(I) = B(I) + H(I)
      end for
F2: for I = 1 to N
A2:   D(I) = E(I) + F(I) + X(I)
A4:   E(I) = D(I) * F(I)
      end for
```

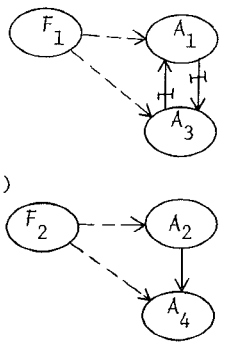


Fig. 10. Loop fusion for virtual memory management