Extending hammocks for parallelism detection*

Michele Tartara, Stefano Crespi Reghizzi†, Simone Campanoni‡

Extended abstract

Introduction and basic definitions Finding the independent parts of a Control Flow Graph (CFG) is important for exposing program parallelism. Two types of instruction dependences constrain the order of execution, data- and control-dependences (CD). Abstracting from data-dependences, it is useful to study the idealized parallelism allowed by program control structure. CD's are assumed as the dependence relation of the trace (i.e., partially commutative) language defined by the DFA naturally associated to the CFG. Two CFG's are control(C)-equivalent if they have identical CD's and define the same trace language. The family of such C-equivalent CFG's is a model for programs obtained from each other by commuting C-independent instructions. Since single instructions are a too fine-grained unit for efficient parallelization on modern computer architectures, we characterize larger program parts, called Movable Control Components (MCC) that can be executed in different order preserving C-equivalence. MCC is a new notion derived from the established concept of single-entry single-successor subgraph, also known as a hammock, by imposing that no cycle can span across different MCC's. A recursive decomposition of a program into sets of mutually independent MCC's is introduced. It is proven that all and only the C-equivalent CFG's are obtained by local transformations that commute two successive MCC's that are part of a linear sequence of such components. This study provides part of the theoretical base needed for on-going work on automatic program parallelization.

We need to define some standard concepts from program and compiler theory.

In a Control Flow Graph (CFG) [5] a node v predominates (resp. postdominates) node w, written $v \dashv w$ (resp. $v \vdash w$), if every path from `START` to w contains v (resp., if every path from w to `END` contains v). Both relations are tree-orders.
The immediate postdominator $ipostdom(S)$ of a set of nodes S is the least node i, in the $\vdash$ order, such that $i \in \bigcap_{n \in S} postdom(n)$, the latter function denoting the set of postdominators of n.

An edge $n \to h$ is a back-edge if $h \dashv n$.

It is customary to assume that a node has at most two successors; nodes with two successors are conditional instructions.

For a node h of a graph we consider the subgraph, denoted by $SCC(h)$, which is the minimal Strongly Connected Component (SCC) containing h (excluding $SCC(h) = \{h\}$ unless it is the only SCC of h); for a SCC S, the exit set, $exit(S)$, includes the nodes in S having at least one successor out of S.

The following definitions of Binary and Ternary Control Dependence relations (BCD, TCD) from [2] are equivalent to the standard ones [1, 4].

A node y is BCD-dependent on a node x iff x is a conditional instruction and is the only predecessor of u and v; there exists a path $x, u, \ldots,$ `END` that does not contain y; every path $x, v, \ldots,$ `END` contains y.

y is TCD-dependent on x via v if y is BCD-dependent on x, $v \in succ(x)$, and v predominates y. We name C-dependence any one of BCD and TCD.

A hammock [4] H is a subgraph (V_H, E_H) of G with a distinguished node h in V_H called entry node and a distinguished node s in $V \setminus V_H$ called successor such that all edges from $(V \setminus V_H)$ to V_H

*Partially supported by MIUR 2007TJNZRE_002

†Politecnico di Milano

‡Harvard University

go to h and all edges from V_H to $(V \setminus V_H)$ go to s.
A node $e \in V_H$ with $s \in succ(e)$ is an exit from H.

Nodes in V_H and in $(V \setminus V_H)$ are resp. called internal and external to H. Clearly, all internal nodes are predominated by the entry point and immediately postdominated by the successor of H.

A hammock does not qualify as a program region suitable for parallelization, because a cycle may span across several hammocks, thus preventing their execution in a different order or in parallel. Refining hammock definition with ideas from [2], we introduce a more suitable notion.

Movable Control Component(MCC).
A MCC $M = M(h, s)$ is a hammock (V_M, E_M) with entry point h (called head) and successor s, such that no predecessor of h is predominated by h.

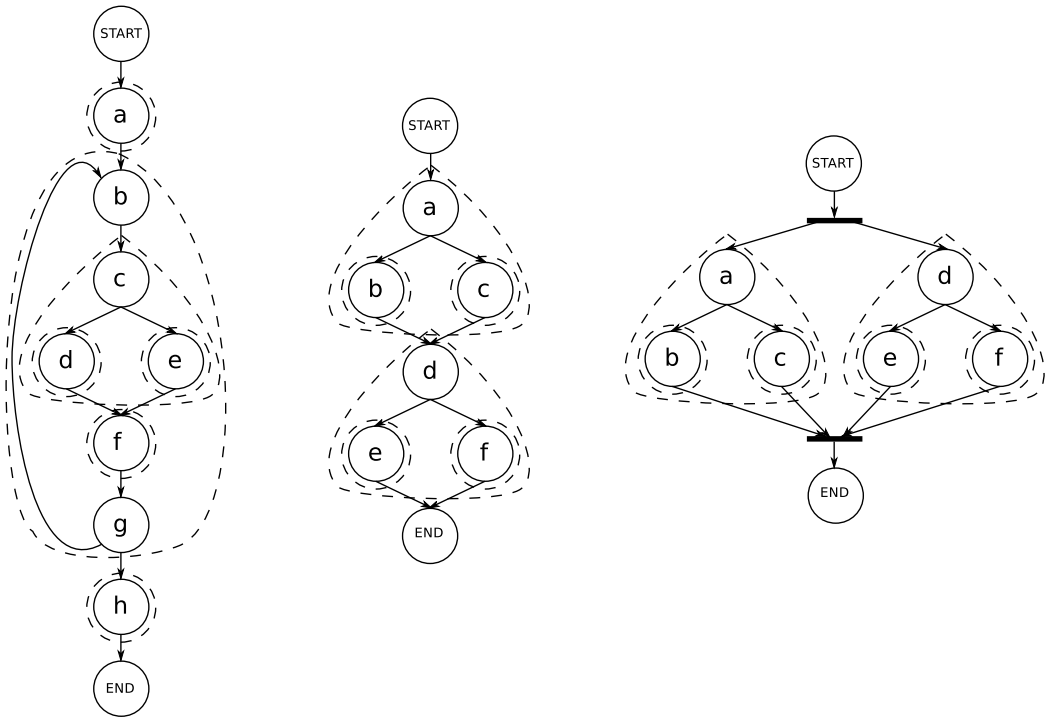

Figure 1: (a) MCCs of a CFG containing as hammocks the sets $\{a\}, \{abcdefg\}, \{abcdefgh\}, \{b\}, \{bcde\}, \{bcdef\}, \{bcdefg\}, \{bcdefgh\}, \{cde\}$, $\{cdef\}, \{d\}, \{e\}, \{f\}, \{h\}$; (b) another CFG with its MCCs; and, (c) the same CFG after parallelization.

Fig. 1(a) illustrates the difference between hammocks and MCCs.
The following properties of MCCs are straightforward:
Let $M(h, s)$ be a MCC contained in G.

An internal node C-depends on h iff h is a conditional instruction. (1)

No external node C-depends on an internal node. (2)

Any two MCCs are either disjoint or contained one in the other. (3)

No exact partition of a MCC into smaller MCCs exists. (4)

If h is the end-point of a back-edge, every internal node C-depends on every node in $exit(SCC(h))$. (5)

Any simple cycle that traverses a back-edge ending in h is wholly contained into M. (6)

An internal node has at least one BCD that no external node has. (7)

Therefore, a MCC is identified by its head h, and the same letter, written $\widetilde{h}$, will denote the MCC, and $\widetilde{V}$ the set of MCCs of a CFG. Prop. (4) states that a MCC is a minimal region with the defining properties. Prop. (6) makes the important difference w.r.t. hammocks. Using hammocks to detect parallel code is not practical, since different parts of a loop may belong to different hammocks and any attempt to parallelize them needs complex and inefficient handling of synchronization between different threads. On the other hand, it will be argued that a MCC can be moved, if needed, in different positions inside the CFG, and executed in parallel with some other MCCs, just taking care of not breaking data dependencies.

Relationships between MCCs We extend the predecessor and successor relations to MCCs, i.e., to $(V \cup \widetilde{V}) \times (V \cup \widetilde{V})$, marking with tilde the relations over $\widetilde{V} \times \widetilde{V}$, as follows:
$pred(\widetilde{a}) = pred(a)$
$succ(\widetilde{a}) = s$ where $\widetilde{a} = M(h, s)$
$\widetilde{pred}(\widetilde{a}) = \{\widetilde{b} \mid \exists x \in pred(a) \wedge x \in \widetilde{b} \wedge x \notin \widetilde{a} \wedge \widetilde{a} \notin \widetilde{b}\}$
$$\widetilde{succ}(\widetilde{a}) = \begin{cases} \widetilde{s} \quad if \quad \widetilde{a} = M(h, s) \\ \qquad \text{and } s \text{ is the head of a component} \\ \emptyset \qquad \text{otherwise} \end{cases}$$
Notice that both $\widetilde{pred}$ and $\widetilde{succ}$ are partial functions.

Next we consider MCCs that precede each other in a linear subgraph. A linear sequence (LS) of MCCs is a sequence $\widetilde{h}_1, \ldots, \widetilde{h}_n$, such that, for $1 \leq i \leq n-1$, $\{\widetilde{h}_{i+1}\} = \widetilde{succ}(\widetilde{h}_i)$ and $|\widetilde{pred}(\widetilde{h}_1)| \neq 1$ and $|\widetilde{succ}(\widetilde{h}_n)| \neq 1$.

A LS is in some sense analogous to a basic block [1], but the elements are, of course, more complex than single instructions.

Property 1 *Let $a \in \widetilde{h}_i, b \in \widetilde{h}_j$ where $\widetilde{h}_i, \widetilde{h}_j$ are in a LS. Then a and b have identical C-dependencies w.r.t. any node c that is external to $\widetilde{h}_i$ and $\widetilde{h}_j$.*

The analysis of a CFG into MCCs, as in Fig. 1, produces a tree, more precisely, a forest decomposition. At the outermost level the CFG is a LS of MCCs preceded by `START` and ended by `END`. Two MCCs are called siblings if they belong to the outermost LS or if they have the same MCC as father. Clearly, all MCCs in a LS are siblings.

Control equivalence and component movement Program transformations that just reschedule instructions (without duplication or modification) are precisely defined by the notion of control equivalence introduced in [2], to be next informally presented.

The BCD relation of a CFG G can be made reflexive and symmetric, thus obtaining a relation D that can be used as dependence relation in a partially commutative (trace) monoid [3]. Let I be the complement of D, i.e., the independence relation. A CFG G can be taken as the graph of a deterministic finite automaton (DFA) A, and the language recognized $L(A)$ is then a conservative approximation of all possible runs of the program. By applying standard concepts of trace theory, we state the relevant definition.

Two CFGs $G = (V, E), G' = (V, E')$ are C-equivalent, if, they have the same dependence relation D, and, for the corresponding DFA A, A', $L(A)$ is equal to $L(A')$ modulo the trace equivalence relation induced by I.

This definition leaves unanswered the practical question: how to transform a CFG into a C-equivalent one. We are going to prove that all and only the C-equivalent graphs can be obtained by displacing the movable components.

MCCs are termed "movable" because two of them, under rather straightforward conditions, can be interchanged without affecting program semantics from the point of view of C-dependencies. On the other hand, a node that is not the head of a MCC (for example a cycle exit) is never movable.

Property: Let CFG G' be obtained from G by interchanging two MCCs in the same LS. Then G' is C-equivalent to G.

Proof: immediately from Property 1

The following recursive definition captures the idea that two MCCs are equivalent if they only differ in the linear arrangement of some sub-MCCs that are part of the same LS.

Consider two MCCs h and h'. Consider the LS-CFGs H and H' obtained by substituting each LS in $\widetilde{h}$ and $\widetilde{h}'$ respectively with a single node (termed LS-node). $\widetilde{h}$ and $\widetilde{h}'$ are equivalent if the following conditions are met: H and H' are isomorphic; moreover, corresponding LSs in $\widetilde{h}$ and $\widetilde{h}'$ are identical modulo a permutation of pairwise equivalent MCCs; and corresponding nodes of H and H' that are not MCC heads, are identical instructions.

Implicitly the definition introduces a program transformation consisting of permuting the MCCs that belong to a LS.

The main result follows.

Theorem 1 *Two CFGs G and G' are control-equivalent iff their outermost linear sequences are permutations of pairwise equivalent MCCs.*

The theorem is proven in the full paper by demonstrating that permutations of linear sequences are control-equivalent and by recursively proving that pairwise equivalent MCCs are control-equivalent. The reverse implication is proven by showing that pairwise equivalent MCCs preserve control dependencies and define the same trace language. The same holds for different permutation of LSs.

Related work and conclusion Research on static program analysis and compilation has introduced control dependencies, and identified hammocks as interesting components, especially for reverse engineering and program restructuring [6]. Program transformation by instruction rescheduling is a vast research area, based, of course, on the study of dependencies, but we are not aware of any paper using trace theory apart from [2].

The main contribution of this paper is to formalize all the programs compatible with the same control dependences in terms of permutability of well-defined hammock types. Concerning the future, theoretical developments are needed for studying other transformations, including code duplication and loop restructuring. We have produced tools, based on the MCC decomposition theory, that extract MCCs from real programs, and we plan to use the result, together with data-dependence analysis, for evaluating the speed-up obtainable by executing independent MCCs in parallel.

References

[1] Andrew W. Appel and Jens Palsberg. Modern compiler implementation in Java. Cambridge University Press, second edition, 2002.

[2] Simone Campanoni and Stefano Crespi-Reghizzi. Traces of control-flow graphs. In Volker Diekert and Dirk Nowotka, editors, Developments in Language Theory, volume 5583 of Lecture Notes in Computer Science, pages 156–169. Springer, 2009.

[3] Volker Diekert and Grzegorz Rozenberg, editors. The Book of Traces. World Scientific, Singapore, 1995.

[4] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst., 9(3):319–349, 1987.

[5] Keshav Pingali and Gianfranco Bilardi. Optimal control dependence computation and the Roman chariots problem. ACM Transactions on Programming Languages and Systems, 19(3):462–491, May 1997.

[6] Fubo Zhang and Erik H. D'Hollander. Using hammock graphs to structure programs. IEEE Trans. Softw. Eng., 30(4):231–245, 2004.