Basic-Block Graphs: Living Dinosaurs?

Jens Knoop[1]*, Dirk Koschützki[1], and Bernhard Steffen[2]

[1] Universität Passau, D-94030 Passau, Germany
e-mail: {knoop,koschuetzki}@fmi.uni-passau.de
[2] Universität Dortmund, D-44221 Dortmund, Germany
e-mail: steffen@ls5.cs.uni-dortmund.de

Abstract. Since decades, *basic-block* (*BB*) graphs have been the state-of-the-art means for representing programs in advanced industrial compiler environments. The usual justification for introducing the intermediate BB-structures in the program representation is *performance*: analyses on BB-graphs are generally assumed to outperform their counterparts on *single-instruction* (*SI*) graphs, which, undoubtedly, are conceptually much simpler, easier to implement, and more straightforward to verify. In this article, we discuss the difference between the two program representations and show by means of runtime measurements that, according to the new computer generations, performance is no longer on the side of the more complex BB-graphs. In fact, it turns out that no sensible reason for the BB-structure remains. Rather, we will demonstrate that *edge-labeled* SI-graphs, which model statements in their edges instead of in their nodes as classical flow graphs do, are most adequate, both for the theoretical reasoning about and for the implementation of analysis and optimization algorithms. We are convinced that this perception has far-reaching consequences for the design of compiler systems.

1 Motivation

In program analysis and optimization it is common to work on so-called *flow graphs*, whose edge structure makes the control flow of the underlying program explicit. Most widely used are *node-labeled basic-block* (*BB*) graphs, whose nodes represent maximal sequences of straight-line code. This most prominent representation can be modified according to (1) its granularity in order to arrive at *single-instruction* (*SI*) graphs and/or to (2) its way of instruction modelling: *edge-labeled* graphs model instructions/basic blocks by edges rather than nodes.

In this article we investigate these four variants of program representation both from a theoretical and practical point of view. It turns out that the most prominent representation in practice is no longer adequate in times where already the main storage of home computers easily accommodates SI-graphs for huge procedures. Moreover, we show that *edge-labeled* graphs simplify both the theoretical reasoning about analysis and optimization as well as their implementation in comparison to their *node-labeled* counterparts. This is mainly due to

* The work of the author was funded in part by the Leibniz Programme of the German Research Council (DFG) under grant Ol 98/1-1.

the fact that edge-labeled graphs allow us to use nodes as the natural place for storing analysis results and information within the graph structure, whereas node-labeled graphs require separate means and operational overhead. The advantage of edge-labeled graphs is even more drastic when looking at programs with a parallel operator [18].

Our investigation is complemented by runtime measurements demonstrating that the "classical" reason for BB-structuring, i.e., opening analysis and optimization to realistic programs, did not survive the radical changes at the hardware front: as SI-graphs fit into main memory now, BB-graphs can at the most gain some performance for programs with basic blocks of comparatively large average size. In fact, in everyday's life, we never experienced any situation, where BB-graphs were superior to SI-graphs.

Moreover, the BB-structuring is limited in its application scenario (cf. Section 2.2). Thus, there are strong reasons to considering edge-labeled SI-graphs as the most adequate uniform representation for compiler optimization. In particular, it is fair to state that BB-graphs outlived their time, and that they can be considered *living dinosaurs.*

Structure of the Article. In Section 2 we critically re-investigate the properties usually attributed to BB-graphs. This leads directly to the central thesis of this article stating the superiority of edge-labeled SI-graphs for analysis and optimization. Subsequently, we present our preliminaries including a taxonomy of flow-graph variants in Section 3. Central are then Sections 4 and 5. In Section 4 we give theoretical evidence for the superiority of edge-labeled SI-graphs taking three different view-points. In Section 5 we complement this by runtime measurements demonstrating that BB-graphs do not compensate for their conceptual complexity in practice. Together, this confirms our thesis of Section 2 both theoretically and practically. Section 6, finally, contains our conclusions.

2 Basic Blocks: "Folk Knowledge"

2.1 Benefits

The central benefits commonly claimed are summarized by two keywords:

- *Performance:* ... because less nodes take part in costly fixed-point iterations.
- *Compactness:* ... thus, larger programs fit into the main memory.

Both points do not reflect the situation of the late nineties: state-of-the-art fixpoint algorithms can easily deal with graphs of more than 10^5 nodes in real time, a size which will hardly be exceeded by procedural SI-graphs (of course, this also depends on the fact that these graphs fully fit into the main memory of modern computers).

2.2 Short-comings

In contrast, BB-graphs are infected with a number of inquestionable shortcomings:

- *Higher conceptual complexity:* ... basic blocks introduce undesired *hierarchy* complicating both theoretical reasonings as well as implementations (cf. Section 4.1).
- *Demand for pre- and postprocesses:* ... usually required for managing the subtleties of hierarchy (e.g, *dead code elimination*, *constant propagation*, ...), or "tricky" formulations mandatory for by-passing them (e.g., *partial redundancy elimination*) (cf. Section 4.2).
- *Limited generality:* ... certain practically relevant analyses and optimizations are hard or even impossible to be expressed on the basic-block level (e.g., *faint variable analysis and elimination*) (cf. Section 4.3).

Higher conceptual complexity: Basic blocks structure a graph *hierarchically*. As a consequence, analysis and optimization problems must be designed, reasoned about, and implemented on two different levels, the *basic-block* and the *instruction* level; the latter in order to push the data-flow information computed globally for basic blocks to their constituting instructions. This two-level approach is particularly cumbersome, whenever the local analyses for several global analyses are performed in a single traversal, a situation which is common in practice for performance reasons. Maintaining a consistent view on basic blocks becomes then often a nontrivial task due to intricate interdependencies of different analyses and transformations based thereof (cf. [26]). This is a pity, particularly, because a single level, even more, the intellectually less sophisticated and less challenging instruction level would suffice. An observation, which previously was made by other researchers as well (cf. [26]). Here, however, we investigate its consequences in more detail and complement them with runtime measurements showing that the higher conceptual complexity of BB-graphs does not pay-off in practice in terms of performance.

Demand for pre- and postprocesses: Working on BB-graphs requires usually pre- and postprocesses on the analysis and optimization side. In fact, this holds for almost every optimizing program transformation, and is another source of additional conceptual complexity. Obvious, though comparatively simple examples are procedures for *dead code elimination* and *constant propagation*. After computing the required data-flow information for basic blocks, they must be inspected themselves by a postprocess in order to apply the transformation under consideration to the complete program. Sometimes pre- and postprocesses can be avoided. However, this usually relies on "tricky" formulations often injuring conceptual clarity and transparency of the transformation. A representative example is the *busy-code-motion* (*BCM*)-transformation of [16] for the elimination of partially redundant computations in a program. The *BCM*-transformation does not require a postprocess as e.g. dead code elimination. This, however, comes at the price of a more complicated reasoning about the correctness of the transformation as the *meet-over-all-paths* (*MOP*)-solutions of the data-flow properties involved do not directly fit to the *maximal-fixed-point* (*MFP*)-solutions computed because they apply to basic-block internal program points rather than to their "natural" entry and exit points.

Limited generality: The applicability of BB-graphs for practically relevant problems is limited. ***Faint code elimination*** (cf. [8, 10, 17]), a generalization of dead code elimination, is a typical representative of such a problem, which seems to be impossible to formulate on a basic-block level. The point is that the local properties of a basic block are not invariant under the global faintness analysis. This invariance, however, is the prerequisite for lifting an analysis from the instruction to the basic-block level, i.e., for hierarchically decomposing it into a global analysis on basic blocks followed by their local inspection.

2.3 What Remains?

Whereas modern computers easily deal with the size of even large SI-graphs, humans will hardly be able to comprehend small ones of a few hundred nodes only. Here a factor of 5 to 10 in size may well make a significant difference: it is easy to graphically deal with up to 60 or 80 nodes, but 500 nodes are definitely beyond a comfortable graphical treatment. Thus, the BB-structure can well be regarded as a means to extend the range of graphically manageable programs. However, the question remains whether they are most adequate for this. Here, the answer is no for two reasons:

- *Syntactic* reduction in terms of a macro or sub-routine concept, structuring the argument program, is much superior to an algorithmic BB-reduction, as this structuring allows an almost arbitrary reduction, while at the same time expressing some of the intention of the programmer. Thus the reduced programs "are meant" to be understandable.
- *Semantic* reduction according to a certain aspect of the program reduces the program by hiding all details irrelevant for the aspect currently under investigation. This reduction typically has an effect far beyond a BB-collapse, and it collapses program parts according to their properties rather than according to some comparatively trivial syntactic criterion. This allows to maintain understandability on the level of the collapsed program, which is by no means guaranteed by a BB-collapse.

Both syntactic and semantic reduction can easily be computed in real time in order to provide the user with the most adequate "view" of the program. Both reduction techniques have in fact been successfully applied in an industrial project, where they were one of the key "unique selling propositions" [24, 25]. In the remainder of this article we give theoretical and empirical evidence advocating our thesis that *edge-labeled SI*-graphs are the graph variant simultaneously fitting the needs of theoreticians and practitioners best.

3 Preliminaries: A Taxonomy of Flow Graphs

Programs are basically represented by *directed flow graphs* consisting of a set of *nodes* and *edges* together with a unique *start node* and *end node*, which are

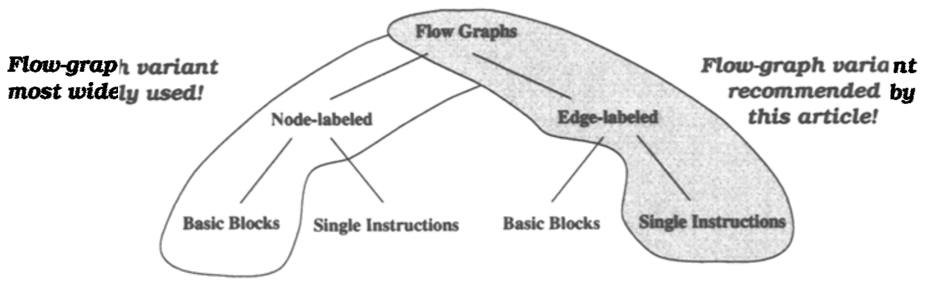


Fig. 1. A taxonomy of flow graphs.

assumed to have no incoming and outgoing edges, respectively. Flow graphs can either be *node-labeled* or *edge-labeled*; they can be *basic-block* (*BB*) graphs or *single-instruction* (*SI*) graphs. Together this leads to the taxonomy of flow graphs displayed in Figure 1. We recall that node-labeled BB-graphs are prevailing both in practice and in the literature on analysis and optimization. They can be considered the de-facto standard.[1] In contrast, we argue that from both a theoretical and practical point of view edge-labeled SI-graphs are the most appropriate flow-graph variant. Figure 2(a) illustrates the different flow-graph variants by means of small flow-graph fragments.

Conventions: We denote BB-graphs and SI-graphs by quadruples $\mathbf{G} = (\mathbf{N}, \mathbf{E}, \mathbf{s}, \mathbf{e})$ and $G = (N, E, s, e)$, respectively. Basic blocks are usually denoted by β, and instructions by ι, both possibly indexed. $\mathit{lhs}(\iota)$ denotes the left-hand side variable of an instruction ι, and $\mathit{block}(\iota)$ the basic block containing ι. Moreover, $\mathit{start}(\beta)$ and $\mathit{end}(\beta)$ denote the first and the last instruction of β, respectively. For an SI-graph G, $\mathit{pred}_G(n) =_{df} \{ m \mid (m,n) \in E \}$ and $\mathit{succ}_G(n) =_{df} \{ m \mid (n,m) \in E \}$ denote the set of all immediate predecessors and successors of a node n. Usually, ε will be used as an identifier for edges. A *finite path* in G is a sequence $(n_1, \ldots, n_q)$ of nodes such that $(n_j, n_{j+1}) \in E$ for $j \in \{1, \ldots, q-1\}$. $\mathbf{P}_G[m,n]$ denotes the set of all finite paths from m to n, and $\mathbf{P}_G[m,n[$ the set of all finite paths from m to a predecessor of n. Finally, every node $n \in N$ is assumed to lie on a path from s to e. These notions are used analogously for BB-graphs.

4 Theory: Short-comings of BB-Graphs

In this section we give theoretical evidence for the short-comings of BB-graphs for analysis and optimization as summarized in Section 2.2. Each of the points mentioned, *higher computational complexity*, *demand for pre- and postprocesses*, and *limited generality*, is investigated in a separate subsection.

[1] See e.g. [1–7, 9, 10, 12, 16, 17, 19–22, 26]. One of the few exceptions is [18] considering edge-labeled SI-graphs.

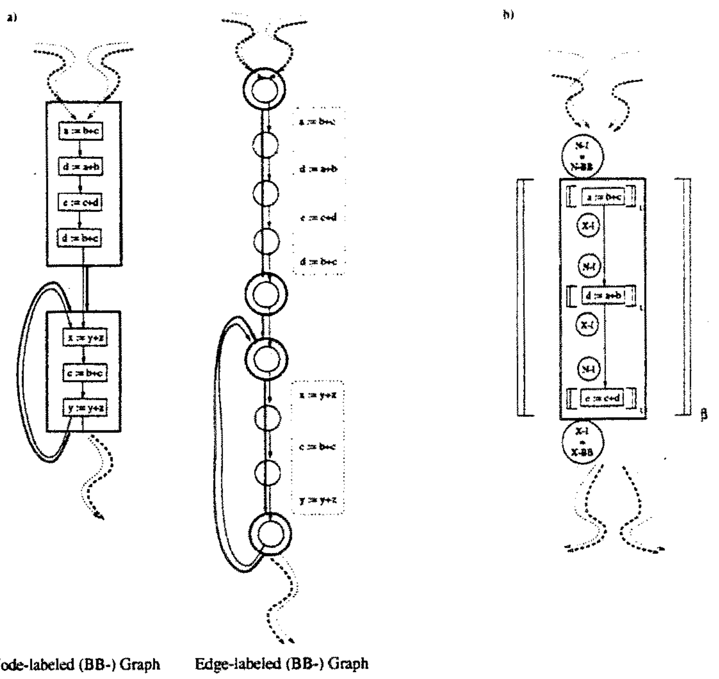


Fig. 2. a) Node-labeled vs. edge-labeled (BB- and SI-) flow graphs. b) eNtry- and eXit-points, and local semantic functions.

4.1 Higher Conceptual Complexity

BB-graphs are inherently hierarchical. They have a two-level structure. This enlarges the conceptual and technical complexity of specifying analysis problems, and of reasoning about them as well as their implementations. We demonstrate this on two different levels of abstraction. First, on the level of the abstract-interpretation framework underlying data-flow analysis (DFA) (cf. Section A). Second, on the *concrete* level of a typical and practically relevant DFA-problem, the availability of program terms (cf. Section B).

A) Correctness and Precision: *MOP*-Solution and *MFP*-Solution. Fundamental for reasoning about correctness and precision of a DFA are the *meet-over-all-paths* (*MOP*) solution and the *maximal-fixed-point* (*MFP*) solution in the sense of Kam and Ullman [11]. Intuitively, the *MOP*-solution defines the desired solution of a DFA-problem. It directly reflects the operational semantics of a program because it is the "meet (intersection)" of all (data-flow) informations contributed by some program path reaching a specific program point. Unfortunately, this solution does in general not induce an effective computation procedure. Fortunately, this holds in practice for the *MFP*-solution of a DFA-problem. It is defined as the greatest solution of an equation system expressing consistency of an annotation of the program with (data-flow) informations, which, under certain side-constraints, can effectively be computed by an iterative fixpoint procedure. For each analysis, however, the *MFP*-solution must be proved precise or at least correct with respect to the *MOP*-solution. Though this reduces in practice to checking the premises of the well-known Coincidence and Safety Theorems of Kildall [13], and Kam and Ullman [11], the technical complexity of the definitions of the *MOP*- and *MFP*-solution, and hence of applying these theorems, varies significantly for the flow-graph variants of Figure 1. We demonstrate this by contrasting the definitions of the *MOP*- and *MFP*-solution for edge-labeled SI-graphs, leading to the most elegant and concise versions, with their counterparts for node-labeled BB-graphs, leading to the most complex ones. It is common to all variants that the specification of a DFA-problem consists basically of a local semantic functional $[\![\ ]\!]$ describing the effect of the instructions (basic blocks) in terms of a function on a complete lattice $\mathcal{C}$ representing the data-flow informations of interest, and a start information $c_s \in \mathcal{C}$, which is assumed to be valid on calling the program under consideration. The effect of a program path is then defined as the effect of the sequential composition of its elements (i.e., nodes or edges).

A1) *MOP*- and *MFP*-Solution for Edge-labeled SI-Graphs. Considering an edge-labeled SI-graph G, the local semantic functional $[\![\ ]\!]_\iota : E \to (\mathcal{C} \to \mathcal{C})$ specifying a DFA-problem defines for every edge e of G a function on $\mathcal{C}$. The index ι indicates that the local semantic functions define the effect of instructions, not of basic blocks. The solution of the *MOP*-approach is then given by:

The *MOP*-Solution:

$$\forall c_s \in \mathcal{C}\ \forall n \in N.\ MOP_{([\![\ ]\!]_\iota, c_s)}(n) =_{df} \sqcap \{ [\![p]\!]_\iota(c_s) \mid p \in \mathbf{P}_G[s,n] \}$$

Note that this definition is the formal counterpart to the informal definition given above. Next, we define the corresponding *MFP*-solution.

The *MFP*-Solution: $\forall c_s \in \mathcal{C}\ \forall n \in N.\ MFP_{([\![\ ]\!]_\iota, c_s)}(n) =_{df} \mathtt{info}_{c_s}(n)$

where info_{c_s} denotes the greatest solution of the following equation system:

$$\mathbf{info}(n) = \begin{cases} c_s & \text{if } n = s \\ \sqcap \{ [\![(m,n)]\!]_\iota(\mathbf{info}(m)) \mid m \in pred_G(n) \} & \text{otherwise} \end{cases}$$

The Coincidence Theorem 1 is the handle for proving the *MFP*-solution of a DFA-problem precise with respect to its *MOP*-solution (cf. [11, 13]).[2]

Theorem 1 (Coincidence Theorem).
The MFP-solution and the MOP-solution coincide, i.e.,

$$\forall c_s \in \mathcal{C}\ \forall n \in N.\ MFP_{([\![\ ]\!]_\iota, c_s)}(n) = MOP_{([\![\ ]\!]_\iota, c_s)}(n)$$

if the local semantic functions $[\![e]\!]_\iota$, $e \in E$, *are all distributive.*

[2] If the local semantic functions are monotonic, the *MFP*-solution is a safe (correct) approximation of the *MOP*-solution, i.e.: $MFP_{([\![\ ]\!]_\iota, c_s)} \sqsubseteq MOP_{([\![\ ]\!]_\iota, c_s)}$ (Safety Theorem, [11]).

A2) *MOP*- and *MFP*-Solution for Node-labeled BB-Graphs. For comparison we now recall the definitions of the *MOP*- and *MFP*-solution for node-labeled BB-graphs. This requires a two-level approach. First, defining them on the basic-block level. Second, defining them on the instruction level. The two levels are illustrated in Figure 2(b), which in addition makes the usually implicit distinction between eNtry- and eXit-points for both instructions (N-I,X-I) and basic blocks (N-BB,X-BB) of a node-labeled graph explicit. Note that making the eNtry- and eXit-points explicit, the node-labeled BB-graph is simply a (complicated) coding of an edge-labeled (SI-) graph.

Level 1 – Basic-block Level. On this level we need a local semantic functional $[\![\ ]\!]_\beta : \mathbf{N} \to (\mathcal{C} \to \mathcal{C})$, which defines the effect of complete basic blocks, not just of single instructions. In practice, this requires a preanalysis of every basic block being accomplished by some preprocess. The definition of the *MOP*-solution is then as follows. Note that it defines for every node an eNtry- and an eXit-information, which is mandatory for node-labeled flow graphs. Unfortunately, this introduces an inhomogeneous notion of program point into the reasoning.

The *MOP*-Solution: (Basic-block Level)

$$\forall c_s \in \mathcal{C}\ \forall \mathbf{n} \in \mathbf{N}.\ \mathit{MOP}_{([\![\ ]\!]_\beta, c_s)}(\mathbf{n}) =_{df} (\mathit{N\text{-}MOP}_{([\![\ ]\!]_\beta, c_s)}(\mathbf{n}),\ \mathit{X\text{-}MOP}_{([\![\ ]\!]_\beta, c_s)}(\mathbf{n}))$$

$$\text{with}\quad \begin{array}{l} \mathit{N\text{-}MOP}_{([\![\ ]\!]_\beta, c_s)}(\mathbf{n}) =_{df} \bigsqcap \{\, [\![p]\!]_\beta(c_s) \mid p \in \mathbf{P_G}[\mathbf{s}, \mathbf{n}[\, \} \\ \mathit{X\text{-}MOP}_{([\![\ ]\!]_\beta, c_s)}(\mathbf{n}) =_{df} \bigsqcap \{\, [\![p]\!]_\beta(c_s) \mid p \in \mathbf{P_G}[\mathbf{s}, \mathbf{n}]\, \} \end{array}$$

Also the fixed-point counterpart of the *MOP*-solution considers for every node of $\mathbf{G}$ a pair of DFA-informations:

The *MFP*-Solution: (Basic-block Level)

$$\forall c_s \in \mathcal{C}\ \forall \mathbf{n} \in \mathbf{N}.\ \mathit{MFP}_{([\![\ ]\!]_\beta, c_s)}(\mathbf{n}) =_{df} (\mathit{N\text{-}MFP}_{([\![\ ]\!]_\beta, c_s)}(\mathbf{n}),\ \mathit{X\text{-}MFP}_{([\![\ ]\!]_\beta, c_s)}(\mathbf{n}))$$

$$\text{with}\ \ \mathit{N\text{-}MFP}_{([\![\ ]\!]_\beta, c_s)}(\mathbf{n}) =_{df} \mathsf{pre}^\beta_{c_s}(\mathbf{n})\ \ \text{and}\ \ \mathit{X\text{-}MFP}_{([\![\ ]\!]_\beta, c_s)}(\mathbf{n}) =_{df} \mathsf{post}^\beta_{c_s}(\mathbf{n})$$

where $\mathsf{pre}^\beta_{c_s}$ and $\mathsf{post}^\beta_{c_s}$ denote the greatest solutions of the equation system:

$$\begin{array}{lcl} \mathbf{pre}(\mathbf{n}) & = & \begin{cases} c_s & \text{if } \mathbf{n} = \mathbf{s} \\ \bigsqcap \{\, \mathbf{post}(\mathbf{m}) \mid \mathbf{m} \in \mathit{pred}_{\mathbf{G}}(\mathbf{n})\, \} & \text{otherwise} \end{cases} \\ \mathbf{post}(\mathbf{n}) & = & [\![\mathbf{n}]\!]_\beta(\mathbf{pre}(\mathbf{n})) \end{array}$$

Level 2 – Instruction Level: On this level the information must be pushed into the basic blocks to the individual instructions. Like on the basic-block level we have to distinguish between eNtry- and eXit-informations. In addition to $[\![\ ]\!]_\beta$ this requires a second local semantic functional $[\![\ ]\!]_\iota : N \to (\mathcal{C} \to \mathcal{C})$, which defines the effect of instructions, not of basic blocks.

The *MOP*-Solution: (Instruction Level)

$$\forall c_s \in \mathcal{C}\ \forall n \in N.\ \mathit{MOP}_{([\![\ ]\!]_\iota, c_s)}(n) =_{df} (\mathit{N\text{-}MOP}_{([\![\ ]\!]_\iota, c_s)}(n),\ \mathit{X\text{-}MOP}_{([\![\ ]\!]_\iota, c_s)}(n))$$

with

$$N\text{-}MOP_{(\llbracket\,\rrbracket_\iota, c_s)}(n) =_{df} \begin{cases} N\text{-}MOP_{(\llbracket\,\rrbracket_\beta, c_s)}(block(n)) & \text{if } n = start(block(n)) \\ \llbracket p \rrbracket_\iota(N\text{-}MOP_{(\llbracket\,\rrbracket_\beta, c_s)}(block(n))) & \text{otherwise } (p \text{ prefix-path from } start(block(n)) \text{ up to } n) \end{cases}$$

$$X\text{-}MOP_{(\llbracket\,\rrbracket_\iota, c_s)}(n) =_{df} \llbracket p \rrbracket_\iota(N\text{-}MOP_{(\llbracket\,\rrbracket_\beta, c_s)}(block(n))) \quad (p \text{ prefix-path from } start(block(n)) \text{ up to and including } n)$$

Similarly, we get for the fixed-point counterpart.

The *MFP*-Solution: (Instruction Level)

$$\forall c_s \in C\ \forall n \in N.\ \ MFP_{(\llbracket\,\rrbracket_\iota, c_s)}(n) =_{df} (\,N\text{-}MFP_{(\llbracket\,\rrbracket_\iota, c_s)}(n),\ X\text{-}MFP_{(\llbracket\,\rrbracket_\iota, c_s)}(n)\,)$$

with $N\text{-}MFP_{(\llbracket\,\rrbracket_\iota, c_s)}(n) =_{df} \mathsf{pre}^\iota_{c_s}(n)$ and $X\text{-}MFP_{(\llbracket\,\rrbracket_\iota, c_s)}(n) =_{df} \mathsf{post}^\iota_{c_s}(n)$

where $\mathsf{pre}^\iota_{c_s}$ and $\mathsf{post}^\iota_{c_s}$ denote the greatest solutions of the equation system:

$$\begin{aligned} \mathbf{pre}(n) &= \begin{cases} \mathsf{pre}^\beta_{c_s}(block(n)) & \text{if } n = start(block(n)) \\ \mathbf{post}(m) & \text{otherwise } (m \text{ is } n\text{'s unique predecessor in } block(n)) \end{cases} \\ \mathbf{post}(n) &= \llbracket n \rrbracket_\iota(\mathbf{pre}(n)) \end{aligned}$$

Note that the greatest solution of the latter equation system can be computed quite efficiently by exploiting the fact that basic blocks represent straight-line code sequences. Hence, it suffices to visit the instructions of the basic blocks in their sequential order without having to visit instructions or basic blocks again. Note, however, that this requires a different implementation than for solving the first-level equation system. Moreover, inside a basic block the eNtry-information of an instruction coincides with the eXit-information of its unique predecessor. Thus, one of these informations can be dropped. This is another source of inhomogeneity between program points complicating theoretical investigations as well as implementations further.

The Coincidence and Safety Theorem can also be given for the 2-level setting of node-labeled BB-graphs. We omit this here for brevity. From the preceding presentation it should be obvious that the technical and notational details are much more complicated than for edge-labeled SI-graphs. We remark that this complexity is not restricted to theoretical investigations on correctness and precision. It directly carries over to the implementation side. One has to define and implement a DFA both on the basic-block as well as on the instruction level. This is illustrated in the following section. Note that the equation systems occurring in the following examples are specializations of the equation systems of paragraphs A1) and A2).

B) Availability of Terms: A Typical Application. In this section we demonstrate the impact of the choice of a specific flow-graph variant on the form of a DFA-specification considering a practically relevant analysis problem, the *availability* of terms, a representative of Hecht's famous taxonomy of DFA-problems

[9]. Intuitively, a term t is available at a program point, if it has been computed on every program path reaching this point without an intervening modification.

Table 1 recalls the specification of the availability problem for ***node-labeled BB-graphs***. Note that the two-level specification of the BB-approach requires a two-phase computation process. The first phase is concerned with basic blocks, the second phase with their individual instructions. Table 2 opposes this specification directly to its counterpart for ***edge-labeled SI-graphs***. As expected, it is much simpler as it does not require a two-level specification and dealing with entry- and exit-properties. The latter point makes it also simpler and thus superior to its counterpart for ***node-labeled SI-graphs***, which for comparison is displayed in Table 3. In fact, the specification for edge-labeled SI-graphs is the most concise and elegant one of the four variants (see [15] for the last one). A fact, which applies to other DFA-problems as well. In [15] this is demonstrated for the problems of ***very busy expressions*** and ***constant propagation***.

Availability for Node-labeled BB-Graphs:

Phase I: The Basic-block Level

Local Predicates: (associated with basic-block nodes)

- $\mathsf{BB\text{-}XCOMP}_\beta(t)$: β contains an instruction ι computing t, and neither ι nor any instruction of β following ι modifies an operand of t.
- $\mathsf{BB\text{-}TRANSP}_\beta(t)$: β contains no instruction modifying an operand of t.

The Equation System of Phase I:

$$\mathsf{BB\text{-}N\text{-}AVAIL}_\beta = \begin{cases} ff & \text{if } \beta = \mathbf{s} \\ \prod\limits_{\hat\beta \in pred(\beta)} \mathsf{BB\text{-}X\text{-}AVAIL}_{\hat\beta} & \text{otherwise} \end{cases}$$

$$\mathsf{BB\text{-}X\text{-}AVAIL}_\beta = \mathsf{BB\text{-}N\text{-}AVAIL}_\beta \cdot \mathsf{BB\text{-}TRANSP}_\beta + \mathsf{BB\text{-}XCOMP}_\beta$$

Phase II: The Instruction Level

Local Predicates: (associated with single-instruction nodes)

- $\mathsf{COMP}_\iota(t)$: ι computes t.
- $\mathsf{TRANSP}_\iota(t)$: ι does not modify an operand of t.
- $\mathsf{BB\text{-}N\text{-}AVAIL}^\star$, $\mathsf{BB\text{-}X\text{-}AVAIL}^\star$: greatest solution of the equation system of Phase I.

The Equation System of Phase II:

$$\mathsf{N\text{-}AVAIL}_\iota = \begin{cases} \mathsf{BB\text{-}N\text{-}AVAIL}^\star_{block(\iota)} & \text{if } \iota = start(block(\iota)) \\ \mathsf{X\text{-}AVAIL}_{pred(\iota)} & \text{otherwise (note that } |\,pred(\iota)\,| = 1) \end{cases}$$

$$\mathsf{X\text{-}AVAIL}_\iota = \begin{cases} \mathsf{BB\text{-}X\text{-}AVAIL}^\star_{block(\iota)} & \text{if } \iota = end(block(\iota)) \\ (\mathsf{N\text{-}AVAIL}_\iota + \mathsf{COMP}_\iota) \cdot \mathsf{TRANSP}_\iota & \text{otherwise} \end{cases}$$

Table 1. Node-labeled BB-graphs: Availability of term t.

Availability for Edge-labeled SI-Graphs:

Local Predicates: (associated with single-instruction edges)

- $\mathsf{COMP}_\varepsilon(t)$: instruction ι of edge ε computes t.
- $\mathsf{TRANSP}_\varepsilon(t)$: instruction ι of edge ε does not modify an operand of t.

The Equation System:

$$\mathsf{AVAIL}_n = \begin{cases} ff & \text{if } n = s \\ \prod\limits_{m \in pred(n)} (\mathsf{AVAIL}_m + \mathsf{COMP}_{(m,n)}) \cdot \mathsf{TRANSP}_{(m,n)} & \text{otherwise} \end{cases}$$

Table 2. Edge-labeled SI-graphs: Availability of term t.

Availability for Node-labeled SI-Graphs:

Local Predicates: (associated with nodes)

- $\mathsf{COMP}_\iota(t)$: ι computes t.
- $\mathsf{TRANSP}_\iota(t)$: ι does not modify an operand of t.

The Equation System:

$$\mathsf{N\text{-}AVAIL}_\iota = \begin{cases} ff & \text{if } \iota = s \\ \prod\limits_{\hat{\iota} \in pred(\iota)} \mathsf{X\text{-}AVAIL}_{\hat{\iota}} & \text{otherwise} \end{cases}$$

$$\mathsf{X\text{-}AVAIL}_\iota = (\mathsf{N\text{-}AVAIL}_\iota + \mathsf{COMP}_\iota) \cdot \mathsf{TRANSP}_\iota$$

Table 3. Node-labeled SI-graphs: Availability of term t.

4.2 Demand for Pre- and Postprocesses or "Tricky" Formulations

After focusing on analysis in the previous section, we now concentrate on optimization. Optimizations on BB-graphs demand typically for pre- and postprocesses in order to manage the technical subtleties caused by their hierarchical structure. We illustrate this by means of the busy-code-motion (*BCM*) transformation of [16] for *eliminating partially redundant expressions* in a program; the latter an optimization, which is implemented in many advanced industrial compiler systems like e.g. on the basis of [16] in the Sun SPARCompiler language systems.[3] In essence, the *BCM*-transformation places computations as early as possible in a program. This maximizes the potential of redundant code which can be eliminated by replacing the original computations of the program by references to temporaries initialized at the earliest possible program points.

[3] SPARCompiler is a registered trademark of SPARC International, Inc., and is licensed exclusively to Sun Microsystems, Inc.

As proved in [16] this leads to *computationally optimal* results, which cannot be improved any further by means of partial redundancy elimination. In essence, the computation of the *earliest* computation points for a computation requires the computation of the set of program points, where it is *available* (i.e., where it has been computed on every program path reaching the point without an intervening modification of any of its operands), and where it is *very busy* (i.e., where it will be computed on every program continuation without a preceding modification of any of its operands). The availability analysis has been considered in detail in Section 4.1; the very-busyness analysis is completely dual, and can be found in [15]. Of course, for BB-graphs the computation of availability and very busyness requires a two-level approach. In [16] this is avoided by computing a somehow "tricky" variant of availability and very busyness. The point of the modification, which is described in full detail in [15] and [16], is that the properties computed do not hold for the "natural" entry and exit point of a BB-node, but for a transformation-specific entry- and exit-*insertion* point inside the basic block itself, which depend on the computation pattern under consideration.

Though this avoids a postprocess, it makes reasoning about the correctness of the transformation more intricate as the *MFP*-solutions computed do not coincide with the "standard" *MOP*-solutions. The Coincidence Theorem (cf. Section 4.1) cannot directly be applied. In addition, the *BCM*-transformation still relies on a preprocess eliminating partial redundancies locally inside a basic block. It is worth noting that this is not specific for *BCM*, but applies to every PRE-algorithm working on BB-graphs (see [15] and [16] for details).

Considering SI-graphs instead of BB-graphs reduces the complexity of the specification of the *BCM*-transformation dramatically. In [15] this is demonstrated for *node-labeled SI-graphs*. A major reason for this decrease of both conceptual and technical complexity is that there is no longer a need for "tricky" variants of availability and very busyness nor for any pre- and postprocesses. The counterpart of the *BCM*-transformation for *edge-labeled SI-graphs* would even be simpler due to the homogeneity of program points. Moreover, the edge-label modeling additionally profits from the fact that the problem of *critical edges*, i.e., edges leading from nodes with more than one outgoing edge to nodes with more than one incoming edge (see [16] for details), does not arise here.

4.3 Limited Generality

The *faint variable analysis* (cf. [8, 10, 17]) is a striking example of a practically relevant problem where it is not at all obvious of how to express it on the basic-block level. Intuitively, a variable is *faint* if there is no program continuation on which it is used without a preceding modification, or if the left-hand side variable of the instruction it is used in, is faint as well. A simple example of a faint, though not dead variable, is the left-hand side occurrence of x in the statement $x := x + 1$ located inside a loop without any other occurrence of x elsewhere in the program. The specification of the faint variable analysis for both node-labeled and edge-labeled SI-graphs can be found in [15]. We conjecture that it is impossible to express this property adequately on the BB-level. The point here is that the basic-block properties of this problem are not "really" local, but depend on the globally computed information. Hence, a basic-block analysis must be interleaved with steps for updating basic-block informations. Conceptually, this is even more complicated than the pre- and postprocesses or the "tricky" formulation of the BCM-transformation of the previous section, and destroys the two-level approach of working on BB-graphs, i.e., iterating over the BB-structure first, and inspecting them locally second.

Besides faint variable analysis, there are many other practically relevant DFA-problems like *constant propagation* (cf. [11]) or the computation of *semantically equivalent program terms* (cf. [23]), which can quite naturally and easily be expressed on the instruction level, but not on the basic-block level. In [15] this is illustrated by means of constant propagation.

5 Practice: Empirical Evaluation

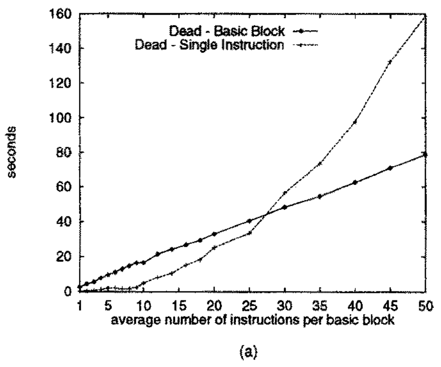
In this section we complement our conceptual investigation by empirical results. We will see that BB-graphs do by no means compensate performance-wise for their (artificial) conceptual complexity. We compared the runtimes for different DFA-problems for *edge-labeled* BB- and SI-graphs, for programs of different size, and varying average lengths of basic blocks. As expected, it turned out that (1) the average length of basic blocks and (2) the maximal chain length of the lattice of data-flow information are the key parameters for this comparison.

Figures 3(a) and (b) show a representative profile of these results. For the problem of computing *dead variables* (cf. [9]), Figure 3(a) shows that there is no pay-off for BB-graphs unless the average length of basic blocks becomes unrealistic large for "real world" programs. Figure 3(b), subsequently, illustrates the results of computing *available expressions* for a scenario, where the number of computation occurrences is small. In this practically frequent situation, the overhead for the basic-block handling is dramatically dominating.

The worst-case scenario for SI-graphs requires both large basic blocks, and large maximal chain lengths of the data-flow lattice, in order to force long iteration sequences. Both of these characteristics hardly arise in practice. E.g., Morel and Renvoise report that they never observed more than 3 iterations in their experiments [20], while Dhamdhere reports a number of 5 [3,4]. Typical DFA-problems requiring lattices with longer chains (than e.g. bitvector problems or constant propagation), like e.g. the computation of semantically equivalent terms (cf. [23]) are beyond the scope of a BB-modelling (cf. Section 4.3).

6 Conclusions

For decades, BB-graphs are the state-of-the-art means for representing programs in analysis and optimization. They are considered a guarantor of high performance and broad applicability, which is believed to fairly balance the higher conceptual complexity they cause for theoretical reasoning and implementation. In this article we have systematically investigated the benefits and short-comings



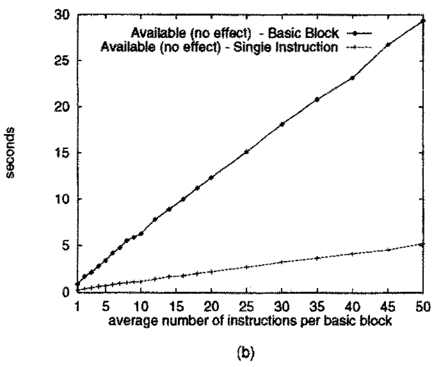


Fig. 3. Illustrating empirical results.

of the complete taxonomy of flow-graph variants. As a central result it has turned out that the severe short-comings of the currently most prominent representation is by no means compensated by its assumed benefit, namely performance. Empirical results show that the conceptually far superior SI-graphs are competetive in practice, often even superior. In fact, in everyday's life, we never experienced a situation, where the classical representation performed better. This strongly indicates that *edge-labeled SI-graphs* are the adequate representation for the considered application scenario. In fact, the experience with our DFA&OPT-generator (cf. [14]), which is based on edge-labeled SI-modeling, is extremely promising.

References

1. C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Prog. Lang. Syst.*, 17(2):181 - 196, 1995.
2. J. Cocke and K. Kennedy. An algorithm for reduction of operator strength. *Comm. ACM*, 20(11):850 - 856, 1977.
3. D. M. Dhamdhere. A fast algorithm for code movement optimization. *ACM SIGPLAN Not.*, 23(10):172 - 180, 1988.
4. D. M. Dhamdhere. A new algorithm for composite hoisting and strength reduction optimisation (+ Corrigendum). *Int. J. Comp. Math.*, 27:1 - 14 (+ 31 - 32), 1989.
5. D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyze large programs efficiently and informatively. In *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Impl. (PLDI'92)*, volume *27*,7 of *ACM SIGPLAN Not.*, pages 212 - 223, 1992.
6. V. M. Dhaneshwar and D. M. Dhamdhere. Strength reduction of large expressions. *J. Prog. Lang.*, 3(2):95 - 120, 1995.
7. A. Fong, J. B. Kam, and J. D. Ullman. Application of lattice algebra to loop optimization. In *Conf. Rec. 2nd Symp. Principles of Prog. Lang. (POPL'75)*, pages 1 - 9. ACM, NY, 1975.
8. R. Giegerich, U. Möncke, and R. Wilhelm. Invariance of approximative semantics with respect to program transformations. In *Proc. 3rd Conf. Europ. Co-operation in Informatics*, Informatik-Fachberichte 50, pages 1 - 10. Springer-V., 1981.

9. M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier, North-Holland, 1977.
10. S. Horwitz, A. Demers, and T. Teitelbaum. An efficient general iterative algorithm for data flow analysis. *Acta Informatica*, 24:679 - 694, 1987.
11. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309 - 317, 1977.
12. K. Kennedy. A survey of data flow analysis techniques. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 1, pages 5 - 54. Prentice Hall, Englewood Cliffs, New Jersey, 1981.
13. G. A. Kildall. A unified approach to global program optimization. In *Conf. Rec. 1st Symp. Principles of Prog. Lang.* (*POPL'73*), pages 194 - 206. ACM, NY, 1973.
14. M. Klein, J. Knoop, D. Koschützki, and B. Steffen. DFA&OPT-METAFrame: A tool kit for program analysis and optimization. In *Proc. 2nd Int. Workshop on Tools and Algorithms for Constr. and Analysis of Syst.* (*TACAS'96*), LNCS 1055, pages 422 - 426. Springer-V., 1996.
15. J. Knoop, D. Koschützki, and B. Steffen. Basic-block graphs: Living dinosaurs? Technical Report MIP-9715, Fak. f. Math. u. Inf., Univ. Passau, Germany, 1997.
16. J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Trans. Prog. Lang. Syst.*, 16(4):1117-1155, 1994.
17. J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proc. ACM SIGPLAN Conference Prog. Lang. Design and Impl.* (*PLDI'94*), volume *29*,6 of *ACM SIGPLAN Not.*, pages 147 - 158, 1994.
18. J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Bitvector analyses $\rightarrow$ No state explosion! In *Proc. 1st Int. Workshop on Tools and Algorithms for Constr. and Analysis of Syst.* (*TACAS'95*), LNCS 1019, pages 264 - 289. Springer-V., 1995.
19. T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. *Acta Informatica*, 27:121 - 163, 1990.
20. E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Comm. ACM*, 22(2):96 - 103, 1979.
21. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conf. Rec. 15th Symp. Principles of Prog. Lang.* (*POPL'88*), pages 2 - 27. ACM, NY, 1988.
22. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189 - 233. Prentice Hall, Englewood Cliffs, New Jersey, 1981.
23. B. Steffen, J. Knoop, and O. Rüthing. The value flow graph: A program representation for optimal program transformations. In *Proc. 3rd Europ. Symp. Programming* (*ESOP'90*), LNCS 432, pages 389 - 405. Springer-V., 1990.
24. B. Steffen, T. Margaria, A. Claßen, V. Braun, and M. Reitenspieß. An environment for the creation of intelligent network services. In *Annual Review Comm., Int. Eng. Consortium Chicago* (*USA*), *IEC,* (*invited contribution*), pages 919 - 935, 1996. Also invited contribution to the book *"Intelligent Networks: IN/AIN Technologies, Operations, Services, and Applications - A Comprehensive Report"*, IEC, 1996, pp. 287-300.
25. B. Steffen, T. Margaria, A. Claßen, V. Braun, M. Reitenspieß, and H. Wendler. Service creation: Formal verification and abstract views. In *Proc. 4th Int. Conf. Intelligent Networks* (*ICIN'96*), pages 96 - 101, 1996.
26. St. W. K. Tijan and J. L. Hennessy. Sharlit — A tool for building optimizers. In *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Impl.* (*PLDI'92*), volume *27*,7 of *ACM SIGPLAN Not.*, pages 82 - 93, 1992.