2005

# A Dictionary Construction Technique for Code Compression Systems with Echo Instructions

Philip Brisk
*University of California, Los Angeles*

Jamie Macbeth
*University of California, Los Angeles*, jmacbeth@smith.edu

Ani Nahapetian
*University of California, Los Angeles*

Majid Sarrafzadeh
*University of California, Los Angeles*

# A Dictionary Construction Technique for Code Compression Systems with Echo Instructions

Philip Brisk Jamie Macbeth Ani Nahapetian Majid Sarrafzadeh
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095

## Abstract

*Dictionary compression* mechanisms identify redundant sequences of instructions that occur in a program. The sequences are extracted and copied to a dictionary. Each sequence is then replaced with a codeword that acts as an index into the dictionary, thereby enabling decompression of the program at runtime. The problem of optimally organizing a dictionary consisting solely of redundant sequences in order to maximize compression has long been known to be NP-Complete [23]. This paper addresses the problem of dictionary construction when redundant code fragments are represented as *Data Flow Graphs* (*DFGs*) rather than linear sequences of instructions. Since there are generally multiple legal schedules for a given DFG *G*, a compiler must determine a schedule for *G* so that other DFGs that are subgraphs of *G* can reference some substring of *G*'s final code sequence. This reduces the size of the dictionary, and in turn, the size of the compressed program. Our experiments with 10 MediaBench [18] applications yielded reductions in dictionary size ranging from 21.14% to 29.76% compared to a naïve approach.

***Categories and Subject Descriptors*** C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.4 [**Processors**]: Compilers

***General Terms*** Algorithms, Performance, Design.

***Keywords*** (Dictionary) Compression, Scheduling, Echo Instructions

## 1. Introduction

Dictionary compression methods [19][21] identify and extract repeated sequences of instructions within a program. A single instance of each sequence is placed into a dictionary; all instances of the sequence in the program are replaced with a codeword that points into the dictionary. When a codeword is encountered during program execution, control is transferred to the dictionary and the sequence is executed; afterward, control is transferred to the instruction following the codeword in the original program.

Compiler support for dictionary compression has mostly focused on identifying redundant code sequences in a post-compilation pass, often at link time. Little attention has been paid to the construction and layout of code within the dictionary, despite the fact that it has long been known that the problem is NP-Complete [23]. It is well known that two code sequences, one of which is a contiguous subsequence of the other, can share the same dictionary entry. For example, the sequence ABC, if entered into a dictionary, includes the contiguous subsequences AB and BC.

This paper develops a compiler technique for laying out the code within the dictionary. A dynamic programming heuristic is presented to perform dictionary construction targeting embedded systems with *Echo Instructions* [10][17][3], an emerging dictionary compression technology that provides low-overhead decompression at runtime with minimal hardware cost. Echo instructions (or some variant thereof) are a likely candidate for inclusion in the next generation of embedded architectures due to their low cost and wide applicability.

In our compiler, code sequences are represented as DFGs rather than sequences of instructions. This eliminates the initial/default schedule of each basic block in the program as a factor that may affect the quality of compression. Scheduling constraints for large DFGs are established in order to maximize the number of smaller DFGs that can reference the dictionary entry for the larger one.

The paper is organized as follows. Section 2 discusses related work. Section 3 summarizes our technique for extracting identical code sequences along with extensions for dictionary organization. Section 4 details our dictionary construction algorithm. Section 5 presents experimental results. Section 6 addresses limitations and future work. Section 7 concludes the paper.

## 2. Related Work

The benefits associated with executing compressed programs have been well-documented over the past fifteen years. Compressed programs reduce the silicon requirements for storing a program in an on-chip ROM in an embedded system; another notable benefit is reduced power consumption [2]; finally, performance increases may result from improved I-cache utilization [4][16]. Here, we focus on the most influential techniques from a historical perspective, and those that are most relevant to our work.

### 2.1 Pre-Cache Decompression

Pre-cache decompression places the decompression circuitry on the cache refill path. When an I-cache miss occurs, a compressed cache block is fetched from main memory, decompressed, and then loaded into the cache. The seminal work in this field was the *Compressed Code RISC Processor* (*CCRP*) [24][20]; IBM commercialized this approach with CodePack [14][19], a CCRP-influenced decompressor for the PowerPC architecture.

### 2.2 Variable Bitwidth Instruction Formats

By the early 1990s, RISC had emerged as the dominant paradigm for general purpose computing. RISC processors typically had a fixed 32-bit instruction format. 16-bit instruction formats—which sacrificed the number of addressable registers, and the bitwidth of immediate operands—were introduced to reduce code size. The number of raw instructions typically increased by 15-20%; however, overall code size was reduced significantly more. Commercial examples include Arm/Thumb [1] and MIPS16 [15].

### 2.3 Procedural Abstraction

*Procedural abstraction* replaces redundant code sequences with procedure calls. This requires no hardware support, but the benefits are limited by procedure call overhead—parameter passing, saving/restoring registers, allocating and deallocating the stack frame, etc. To date, the most widely recognized approaches for procedural abstraction are linear substring matching [11] coupled with register renaming [6][8][9], and local reordering of instructions [8][17], parameterization [7], and predication [17]. More recently, this optimization has been performed prior to register allocation using isomorphism or some variant thereof [22][3].

De Sutter et al. [8] developed a set of abstraction techniques for programs written in C++, where redundancy may arise due to class inheritance and template instantiation. Basic blocks having a similar *fingerprint* are identified, and register renaming is applied to enhance the quality of procedural abstraction. De Sutter also reported an experiment with instruction rescheduling that yielded minimal effects on the quality of abstraction.

### 2.4 Dictionary Compression

The *Call Dictionary* (*CALD*) instruction [21] was an early hardware-supported implementation of dictionary compression. The dictionary is simply a list of instructions. Each sequence of instructions extracted by the compiler is inserted into the dictionary and replaced with a CALD instructions, which has the form CALD(Addr, N), where CALD is an opcode and Addr and N are fields. Addr is the address of the beginning of the code sequence as an offset from the beginning of the dictionary. To execute a CALD instruction, control is transferred to dictionary address ADDR, the next N instructions are executed, and then control is returned to the call point.

Assuming a fixed 32-bit ISA, further compression can be achieved if the size of the CALD instruction is reduced to less than 32 bits (e.g., 8 or 16). Single instruction sequences can be compressed using the same basic dictionary mechanism [19].

Echo instructions [9][17] are similar to CALD instructions. Rather than storing instruction sequences externally, one instance of each is left *inline* in the program; all other instances refer to the inlined sequence. Control is transferred to memory location PC – Addr rather than treating Addr as an offset from the start of the dictionary. Echo instructions allow for dictionary entries to reside anywhere in the program. Unlike CALD instructions, dictionary entries are not likely to reside next to one another in memory.

### 2.5 Parameterized Compression with DISE

*Dynamic Instruction Stream Editing* (*DISE*) is a recent architectural innovation that offers a parameterized model for compression [7]. Recall that CALD and echo instructions require that code sequences have identical register usage; DISE eliminates this requirement by providing a local set of registers. Each call to a dictionary sequence under DISE must provide an explicit mapping from machine registers to local registers; in other words, register names are passed as parameters.

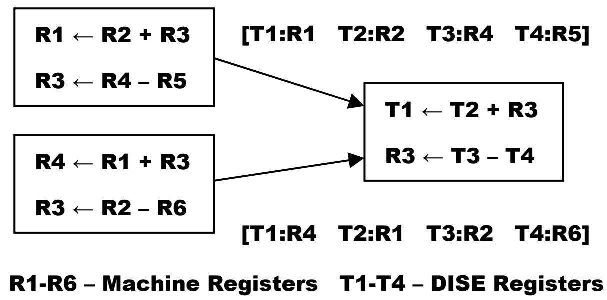


**Figure 1.**
**Illustration of parameterized compression**

Fig. 1 illustrates parameterized compression. R1...R6 are machine registers, and T1…T4 are local DISE registers. Two identical code sequences within a partial renaming of registers are shown on the left. On the right, an equivalent code sequence using local DISE registers is shown along with the mapping of machine registers onto DISE registers. The usage of R3 is identical in both sequences, so there is no need to replace it with a local DISE register. The advantage of parameterization is that the criteria for matching code sequences is less than for standard dictionary compression—identical register usage is no longer required; the disadvantage is that each call (to DISE) requires additional bits to specify the register mapping.

Through DISE, a parameterized implementation of echo instructions is possible. The layout optimization described in this paper could be used with either parameterized or standard echo instructions.

## 3. Dictionary Organization

This section presents an overview of our technique for organizing dictionaries for echo instructions. Code sequences are represented as DFGs rather than linear lists of operations. Data dependencies within each DFG impose a partial ordering on the operations within each sequence. The dictionary layout must schedule the operations within each DFG in an order that minimizes the size of the dictionary.

Fig. 2 shows two instruction sequences $I_1$ and $I_2$ represented as DFGs $G_1$ and $G_2$. $G_1$ has two legal schedules: {ABC, BAC}; whereas $G_2$ has only one: {AC}. Two dictionaries are shown: $D_1$ = BAC and $D_2$ = ACABC. $D_1$ is the preferable dictionary because it contains fewer instructions that $D_2$. The reason that $D_1$ is smaller is that AC is a substring of BAC. By selecting BAC as the schedule for $G_1$, $I_2$ can use the same dictionary entry as $I_1$.

### 3.1 Overview

Here, we place the analysis and optimization techniques described in this paper into the context of a larger compiler framework which we are developing. This framework will provide compile-time code compression, with specific optimizations targeting

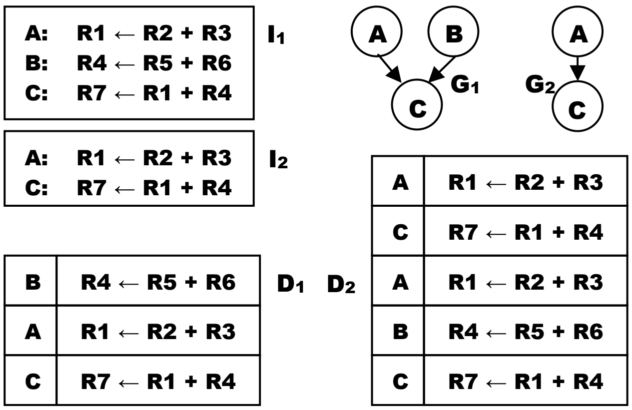


**Figure 2.**
**The relationship between DFG scheduling and dictionary size**

architectures with (parameterized) echo instructions. The first step in this framework is to identify and extract repeated non-overlapping computational patterns that occur in the compiler's intermediate representation. To accomplish this, we have adopted a technique pioneered by Kastner et al. [13] and Brisk et al. [3] to identify isomorphic subgraphs that occur within a set of DFGs representing the basic blocks of a program. These identical DFG fragments will later be replaced with echo instructions.

Register allocation tries to ensure that all instances of the same pattern occurring throughout the program have identical usage of registers. Enforcing this constraint ensures that a semantically equivalent program results when the patterns are replaced with echo instructions. The register allocation mechanism is beyond the scope of this paper. Between pattern identification and register allocation lies the topic of this paper – layout of the dictionary.

As illustrated by Fig. 2, the subgraph relation—which was not addressed previously [3]—allows patterns that are subgraphs of one another to reference the same code sequence. This information will be provided to the register allocator so that it can properly enforce code reuse constraints among patterns and sub-patterns. Henceforth, our compilation framework in the absence of dictionary layout optimization will be referred to as *naïve*.

### 3.2 Problem Statement

Let $G = \{G_1, G_2, ..., G_n\}$ be a set of uniquely identifiable patterns represented as DFGs. Let $D$ be the *dictionary* that we intend to construct. Specifically, $D = \{D_1, D_2, ..., D_m\}$, where $D_i$, $1 \le i \le m$, is called a *dictionary entry*. Specifically, $D_i = (G_i, G_i^*)$, where $G_i = (V_i, E_i) \in G$ and $G_i^* \subseteq G$, such that every DFG $G_j \in G_i^*$ can be scheduled to use the same code sequence as $G_i$. In other words, $D_i$ contains all of the DFGs whose code sequences will reference the same supersequence; for this to be legal, each DFG $G_j$ must be isomorphic to some convex subgraph of $G_i$. In the final program layout, $D_i$ will be represented by a linear sequence $S_i$ of instructions that is a legal schedule for a pattern $G_i$.

An ideal dictionary will satisfy the following two criteria:

$$\bigcup_{i=1}^{m} G_i^* = G \text{, and} \qquad (1)$$

$$\sum_{i=1}^{m} |V_i| \text{ is minimized} \qquad (2)$$

Constraint (1) ensures that all DFGs are included in at least one dictionary entry. Constraint (2) attempts to minimize the size of the dictionary. In practice, (2) is treated as an objective function describing the quality of the solution rather than a constraint that must be met in order to guarantee the legality of a solution.

### 3.3 Constructing the Subgraph Hierarchy

Referring back to Fig. 2, dictionary $D_2$ is chosen because $G_2$ is a subgraph of $G_1$ (denoted $G_2 \subset G_1$). Unfortunately, the general problem of determining whether one graph is a subgraph of another (the well-known *Subgraph Isomorphism Problem*) is NP-Complete [11]. Rather than solve this problem directly, we describe how to compute the relation in conjunction with pattern identification as described by Brisk [3].

Brisk's algorithm uses a technique called *edge contraction* to generate a set of DFGs that occur repeatedly throughout a program. Consider a DFG $G = (V, E)$ and an edge $(u, v) \in E$. $u$ and $v$ are both assigned integer labels which represent their opcodes (e.g. ADD, MUL, etc.). A label can be computed for each edge as well. If many independent edges exist with the same labels as $u$ and $v$, then all induced subgraphs $(\{u, v\}, \{(u, v)\})$ are isomorphic to one another, i.e. they could all be replaced with echo instructions. To model this action, the two vertices and edge are removed and replaced with a supernode (also called a *template*) that maintains the same connectivity to the rest of the graph as $u$ and $v$. As the algorithm iterates, larger patterns are formed by combining adjacent nodes into templates in this fashion; additionally, two templates can be merged into a larger template. The interested reader is encouraged to consult papers by Kastner et al. [13] and Brisk et al. [3] for details.

The sequence of operations that leads to the terminating set of templates can be tracked as the algorithm progresses. The two basic operations involved are adding a vertex to a template and merging two templates into one. We encapsulate this information into a data structure called the *Subgraph Hierarchy* (*SH*).

Fig. 3 illustrates the information maintained in the SH. In (a), a MUL node is merged with template $T_1$, which contains two ADD nodes; the resulting template is labeled $T_2$. Assume that we have not yet encountered a template isomorphic to $T_2$. Initially, the SH contains a vertex representing $T_1$, and the mapping from the instance of $T_1$ in the original DFG to the DAG in $T_1$*'s* vertex in the SH. A new SH node must be added for $T_2$. The mapping from the template in the DFG to the SH vertex is established via an isomorphism test performed during Brisk's algorithm.

The mapping from the instance of $T_1$ in the original DFG is no longer needed, since $T_2$ has replaced $T_1$; however, if another instance of pattern $T_1$ exists elsewhere in the program, then $T_1$ and $T_2$ may share the same code sequence in the dictionary. Therefore, it is useful to compute and maintain a mapping from the vertices of $T_1$, the sub-pattern, onto the vertices of $T_2$.

Let $f_1$ be the mapping from the instance of $T_1$ in the DFG to the node representing $T_1$ in the SH; define $f_2$ similarly for $T_2$. Let $g_{12} : T_1 \rightarrow T_2$ be the mapping from vertices in the DFG representation

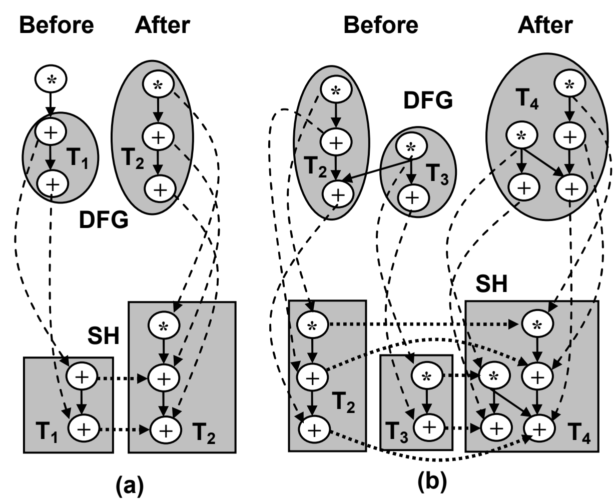


**Figure 3.**
**Illustration of subgraph hierarchy (SH) Construction via adding a node to a template (a) and merging two templates (b)**

of $T_1$ to a subset of those in $T_2$. We can establish $h_{12} : T_1 \rightarrow T_2$, the mapping between $T_1 \rightarrow T_2$ in the SH, as follows:

$$h_{12}(t) = f_2\left(g_{12}\left(f_1^{-1}(t)\right)\right),\ t \in T_1 \tag{3}$$

In Fig. 3 (a), $f_1$ and $f_2$ are represented by dashed arrows from templates in the DFG to the SH; $h_{12}$ is represented by the dotted arrows from $T_1$ to $T_2$ in the SH. The mapping $g_{12}$ represents the implicit temporal relationship between the vertices of $T_1$ and $T_2$ in the DFG; it is not explicitly drawn.

Fig. 3 (b) shows another example, where two templates are combined into a larger one. The fundamental principles are essentially the same as those underlying Fig. 3 (a).

In conclusion, the purpose of the SH is twofold:

(1) To maintain a single DFG to represent all instances of the same pattern that occur in the program (specifically, the mapping between each instance and the representative in the SH).

(2) To maintain the subgraph relation between patterns in the SH.

### 3.4 Scheduling Constraints and the Subgraph Relation

Let $G_i = (V_i, E_i)$ be a DFG. A *cut* $C_{i\rightarrow j,k} = (V_j, V_k)$, $V_k = V_i - V_j$, is a partition of the vertices of $V_i$ into two *cut sets*, $V_j$ and $V_k$. A *cut edge e* crosses $C_{i\rightarrow j,k}$ if one of its endpoints is in $V_j$ and the other is in $V_k$. For a directed graph, a set of cut edges is defined as:

$$E_{(V_j,V_k)} = \left\{ \left(v_j, v_k\right) \middle| v_j \in V_j, v_k \in V_k \right\} \tag{4}$$

$C_{i\rightarrow j,k}$ is defined to be a *convex cut* if

$$E_{(V_k,V_j)} = \phi \tag{5}$$

Next, suppose that we contract edge $e_i = (v_j, v_k)$, and let $G_i$ be the resulting pattern. W.L.O.G assume that both $v_j$, $v_k$ are templates represented by DFGs $G_j$ and $G_k$. Then, $G_i = (V_i, E_i)$, where

$$V_i = V_j \cup V_k \text{, and} \tag{6}$$

$$E_i = E_j \cup E_k \cup E_{(V_j,V_k)} \tag{7}$$

The set of cut edges is maintained externally by the templates $v_j$ and $v_k$; in our representation, the set of cut edges is a dynamic data structure accessible via either $v_j$ or $v_k$. Let $C_{i\rightarrow j,k} = (V_j, V_k)$ be a convex cut of $G_i$. Let $S_j$ and $S_k$ be any legal topological orderings (schedules) of $G_j$ and $G_k$ respectively. Then a schedule $S_{i\rightarrow j,k}$ can be constructed for $G_i$ by concatenating $S_j$ and $S_k$, denoted $S_{i\rightarrow j,k} \rightarrow S_jS_k$. This reduces the size of the dictionary for these three patterns from $2|V_i|$ to $|V_i|$.

In a DAG, *sources* and *sinks* are vertices of in- and out-degree 0 respectively. To enforce $S_{i\rightarrow j,k}$ as a partial ordering constraint on $G_i$, we introduce a set of *separating edges*, $E_{i\rightarrow j,k} = \{(t_j, s_k) \mid t_j \in sinks(G_j),\ s_k \in sources(G_k)\}$. Let $G_{i\rightarrow j,k} = (V_i, E_i \cup E_{i\rightarrow j,k})$ be called a *separated DFG*. Any topological sort of $G_{i\rightarrow j,k}$ ensures that all operations in $V_j$ are scheduled before those in $V_k$.

**Theorem 1.** Any schedule $S_{i\rightarrow j,k}$ of $G_{i\rightarrow j,k}$ corresponds to a schedule $S_i \rightarrow S_jS_k$ *of* $G_i$.

**Proof.** Let $S_{i\rightarrow j,k}$ be a legal schedule of $G_{i\rightarrow j,k}$, and assume to the contrary that $S_{i\rightarrow j,k}$ does not correspond to a schedule $S_i \rightarrow S_jS_k$ of $G_i$. Then there exist two vertices, $v_j \in V_j$, $v_k \in V_k$, such that $v_k$ is scheduled before $v_j$ in $S_{i\rightarrow j,k}$. Let $t \in V_j$ be a sink in $G_j$ such that there is a path from $v_j$ to $t$, and let $s \in V_k$ be a source in $G_k$ such that there is a path from $s$ to $v_k$. $E_{i\rightarrow j,k}$ must therefore contain edge $(t, s)$. Hence, there exists a path from $v_j$ to $v_k$ in $G_{i\rightarrow j,k}$. Therefore $v_j$ must be scheduled prior to $v_k$, a contradiction. □

### 3.5 A Grammar for Subgraph Hierarchies

In the preceding section, we adopted the nomenclature of grammars, $S_{i\rightarrow j,k} \rightarrow S_jS_k$, to represent scheduling constraints that arise in our application domain. Here, we adopt the notation to represent the subgraph relation. A *production* $G_i \rightarrow G_jG_k$ indicates that $G_j$ and $G_k$ are subgraphs of $G_i$ and that $C_{i\rightarrow j,k}$ is a convex cut of $G_i$. A set of productions $P$ is called a *Subgraph Hierarchy Grammar* (*SHG*). $P_i$ is defined to be the subset of $P$, where all grammars have $G_i$ as the left-hand-side. $P_i$ effectively represents all of the pairs of subgraphs that have been combined to form $G_i$ during the execution of Brisk's algorithm [3].

To construct the SHG, we apply one of four possible rules each time a new template is generated. Let $v_j$ and $v_k$ be vertices, and $T_j$ and $T_k$ be templates. Let $e_i$ be an edge that is contracted. Then productions are added to $P_i$ based on the following set of rules:

$$e_i = (v_j, v_k) \implies P_i \leftarrow P_i \cup \{G_i \rightarrow v_jv_k\} \tag{8}$$

$$e_i = (v_j, T_k) \implies P_i \leftarrow P_i \cup \{G_i \rightarrow v_jG_k\} \tag{9}$$

$$e_i = (T_j, v_k) \implies P_i \leftarrow P_i \cup \{G_i \rightarrow G_jv_k\} \tag{10}$$

$$e_i = (T_j, T_k) \implies P_i \leftarrow P_i \cup \{G_i \rightarrow G_jG_k\} \tag{11}$$

In the context of the SHG, $v_j$ and $v_k$ are *terminals* (the opcodes of assembly instructions) and $G_i$, $G_j$, and $G_k$ are *non-terminals*. There cannot be "*recursive*" productions of the form $S_{uv} \rightarrow \alpha S_{uv}\beta$ in $P$,s where $\alpha$ and $\beta$ represent any (possible empty) string of terminals and/or non-terminals. Given a production $G_i \rightarrow G_jG_k$, a legal schedule $S_i$ can be constructed by repeatedly substituting productions of the form $G_j \rightarrow \ldots$ and $G_k \rightarrow \ldots$ for $G_j$ and $G_k$; this is repeated until all nonterminals are replaced with terminals. This is called a *derivation*. One derivation $S_i$ for $G_i$ will eventually be selected as $G_i$'s dictionary entry. Determining an optimal schedule for every DFG in order to maximize pattern overlap is a complicated optimization problem, which is addressed in the sections that follow.

### 3.6 Scheduling Constraints and Compatibility

Any subset of productions $P_i' \subseteq P_i$ is defined to be *compatible* if a schedule $S_i'$ exists, such that for each production $G_i \rightarrow G_j'G_k' \in P_i'$, there exist schedules $S_j'$ and $S_k'$ of $G_j'$ and $G_k'$ respectively such that $S_i' \rightarrow S_j'S_k'$; otherwise, $P_i'$ is *incompatible*.

As an example, consider Fig. 4. A DFG $G_1$ is shown in the upper left, along with four subgraphs $G_2$, $G_3$, $G_4$, and $G_5$. $G_1$ can be formed by combining either $G_2$ and $G_3$ or $G_4$ and $G_5$. If $G_2$ and $G_3$ are combined, the resulting schedule is $S_1 \rightarrow S_2S_3$; likewise, combining $G_4$ and $G_5$ yields schedule $S_1 \rightarrow S_4S_5$. Ideally, we would like to construct schedule $S_1$ having $S_2$, $S_3$, $S_4$, and $S_5$ as substrings; however, this is impossible in this example.

The sets of separating edges are $E_{1\rightarrow 2,3} = \{(B, D), (C, D)\}$ and $E_{1\rightarrow 4,5} = \{(F, C)\}$ for cuts $(V_2, V_3)$ and $(V_4, V_5)$ respectively. The separated DFGs, $G_{1\rightarrow 2,3}$ and $G_{1\rightarrow 4,5}$ formed by adding $E_{1\rightarrow 2,3}$ and $E_{1\rightarrow 4,5}$ respectively to $G_1$ are shown on the bottom of Fig. 4, with non-redundant cut edges shown in bold. These graphs are both DAGs, so Theorem 1 ensures that any legal schedule of either satisfies the respective pattern overlap constraints.

The graph $G_{1\rightarrow(2,3),(4,5)}$ formed by adding both $E_{1\rightarrow 2,3}$ and $E_{1\rightarrow 4,5}$ to $G_1$ is also shown in Fig. 4. $G_{1\rightarrow(2,3),(4,5)}$ contains a cycle; therefore, no legal schedule $S_{1\rightarrow(2,3),(4,5)}$ can be constructed for $G_{1\rightarrow(2,3),(4,5)}$. Therefore, we provably cannot construct $S_1$ having $S_2$, $S_3$, $S_4$, and $S_5$ as substrings.

For the general case, let $G_i$ be a DFG whose subgraphs are under consideration, and Let $P_i$ be defined as above. Specifically,

$$P_i = \{p_{i_m} \mid 1 \le m \le n\}, \text{ where} \tag{12}$$

$$p_{i_m} = G_i \rightarrow G_{j_m}G_{k_m} \tag{13}$$

Associated with each production $p_{i_m}$ is a cut $C_{i\rightarrow j_m,k_m}$, where:

$$C_{i\rightarrow j_m,k_m} = (V_{j_m}, V_{k_m}) \tag{14}$$

Let $E_{i\rightarrow *}$ be the *Aggregate Set of Separating Edges* for $G_i$, defined as follows:

$$E_i^* = \bigcup_{m=1}^{n} E_{i_m \rightarrow j_m,k_m} \tag{15}$$

Define an *Aggregate Separating Graph (ASG)*, $G_i^*$, as follows:

$$G_i^* = (V_i, E_i \cup E_i^*) \tag{16}$$

The ASG simultaneously represents the scheduling constraints required to satisfy each production in $P_i$. Theorem 2 establishes the relationship between the ASG and the compatibility of $P_i$.

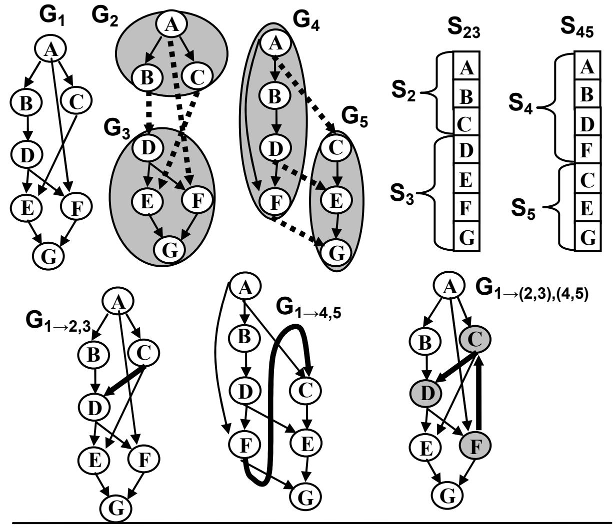


**Figure 4.**
**Illustrating the concept of incompatibility**

**Theorem 2.** The following four statements are equivalent.

1. $P_i$ is a compatible set of non-redundant productions.
2. $G_i^*$ is acyclic.
3. The cuts $C_{i\rightarrow j_m,k_m}$ of $P_i$ can be ordered such that:
$$V_{i_1} \subset V_{i_2} \subset \ldots \subset V_{i_n} \tag{17}$$
4. The cuts $C_{i\rightarrow j_m,k_m}$ of $P_i$ can be ordered such that
$$V_{j_1} \supset V_{j_2} \supset \ldots \supset V_{j_n} \tag{18}$$

**Proof. 1→2.** Let $P_i$ be a compatible set of productions. Assume to the contrary that $G_i^*$ contains a cycle $C = \langle v_1, v_2, \ldots, v_x, v_1 \rangle$. Consider production $p_{i_m}$ corresponding to $C_{i\rightarrow j_m,k_m}$, a convex cut, as defined in Eqs. (13) and (14). To satisfy this cut, there must exist two vertices $v_j \in V_{j_m}$ and $v_k \in V_{k_m}$ such that edge $e_i = (v_j, v_k)$ is included in $C$. To satisfy cycle $C$, there must exist vertices $v'_j \in V_{j_m}$ and $v'_k \in V_{k_m}$ such that $e_i' = (v_k', v_j')$ is an edge in C. Observe that $e_i \in E(V_{j_m}, V_{k_m})$, and $e'_i \in E(V_{k_m}, V_{j_m})$. $\left|E(V_{k_m}, V_{j_m})\right| > 0$ implies that $C_{i\rightarrow j_m,k_m}$ is not a convex cut, contradicting the assumption that $P_i$ is compatible. This is illustrated in Fig. 5 (a).

**2→3.** Assume that $G_i^*$ is acyclic. Assume to the contrary that productions $p_{i_x}$ and $p_{i_y}$ correspond to two cuts $C_{i\rightarrow j_x,k_x}$ and $C_{i\rightarrow j_y,k_y}$ such that $V_{j_x} \not\subset V_{j_y}$, $V_{j_y} \not\subset V_{j_x}$, and $V_{j_x} \ne V_{j_y}$.

Let $u$ and $v$ be vertices defined such that $u \in V_{j_x} \cap V_{k_y}$ and $v \in V_{j_y} \cap V_{k_x}$. To satisfy $C_{i\rightarrow j_x,k_x}$, there must exist a sink

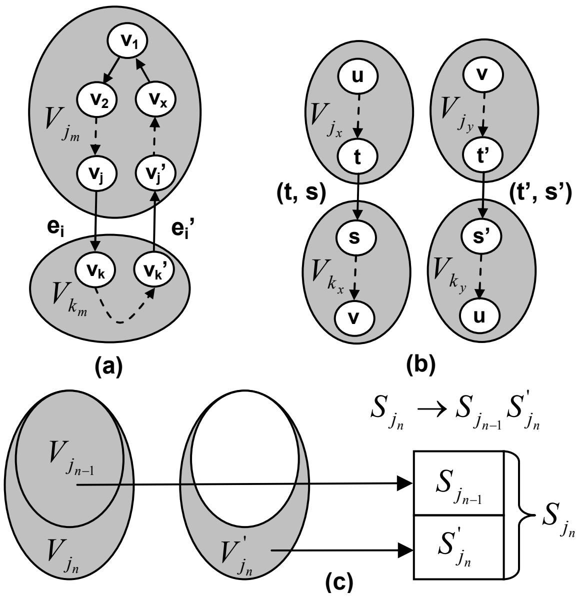


**Figure 5.**
**Illustrating the main concepts of the first 3 steps in the proof of Theorem 2**

$t \in V_{j_x}$ such that there is a path from *u* to *t*. Similarly, there must be a source $s \in V_{k_x}$ such that there is a path from *s* to *v*. Finally, observe that edge $(t,s) \in E_{i \to j_x k_x}$. Therefore there is a path from *u* to *v* in $G_i^*$. By an analogous argument for cut $C_{i \to j_y, k_y}$, a path from v to u can also be established, which contradicts the assumption that $G_i^*$ is acyclic. This is illustrated in Fig. 5 (b).

**3→1.** Define an order on the cuts of *G* such that (17) is satisfied. We prove this statement using induction on $n = |P_i|$. For the basis, suppose $n = 1$. Then $P_i$ contains a single production, $p_i = G_i \to G_j G_k$, which corresponds to a convex cut $(V_j, V_k)$. Consequently, $S_{i\to j,k} \to S_j S_k$ is a legal schedule of $G_i$ by Theorem 1. Without loss of generality, assume that there exist a set of cuts, ordered such that $V_{j_1} \subset V_{j_2} \subset \ldots \subset V_{j_n}$ for $n < |P_i|$. For the induction step, let $n = |P_i|$. By the induction hypothesis, Eq. (17) and (18) are satisfied for $P_i - \{p_{i_n}\}$, which is compatible.

Now, consider cutsets, $V_{j_{n-1}} \subset V_{j_n}$, and $V_{j_n}^{'} = V_{j_n} - V_{j_{n-1}}$. Let $S_{j_n}^{'}$ be a schedule for the subgraph of $G_i$ induced by $V_{j_n}^{'}$. Therefore, we can construct a schedule $S_{j_n} \to S_{j_{n-1}} S_{j_n}^{'}$, such that $S_{i \to j_n k_n} \to S_{j_n} S_{k_n}$ is a legal schedule for $G_i$.

By the induction hypothesis, $S_{j_{n-1}}$ is a legal schedule that includes $S_{j_1}, \ldots, S_{j_{n-1}}$ as sub-schedules. This is illustrated in Fig. 5 (c).

**3↔4.** Assume that (17) holds but (18) does not. Then there must exist productions corresponding to cuts $(V_{j_x}, V_{k_x})$ and $(V_{j_y}, V_{k_y})$ such that $V_{j_x} \subset V_{j_y}$ and $V_{k_x} \subset V_{k_y}$. This leads to the contradiction (19); the converse yields contradiction (20):

$$|V_i| = |V_{j_x}| + |V_{k_x}| < |V_{j_y}| + |V_{k_y}| = |V_i| \tag{19}$$

$$|V_i| = |V_{j_x}| + |V_{k_x}| > |V_{j_y}| + |V_{k_y}| = |V_i| \tag{20}$$

The case where $V_{j_x} = V_{j_y}$ and $V_{k_x} = V_{k_y}$ is redundant. □

Theorem 2 establishes two criteria which we may use to determine whether a set of productions is compatible. In practice, we can not assume that an entire set of productions $P_1$ will be compatible; instead, we focus on the problem of finding an optimal compatible subset of $P_1$.

### 3.7 A Production Compatibility Graph

In this section, we introduce a data structure called a *Production Compatibility Graph* (*PCG*), a DAG that represents compatibility among the productions in $P_1$. Our construction of the PCG is based on Criterion 3 from Theorem 2.

Let $P_i$ be defined as in Eqs. (12) and (13). Each production $p_{i_m} \in P_i$ describes the scheduling constraints including the subgraph relation between $G_i$, $G_{j_m}$, and $G_{k_m}$. A PCG for $G_i$ is denoted $G_i^{PCG} = (V_i^{PCG}, E_i^{PCG})$, where each vertex $v_{j_m} \in V_i^{PCG}$ corresponds to schedule $S_{i \to j_m, k_m} \to S_{j_m} S_{k_m}$ for $G_i$. An edge $e = (v_{j_x}, v_{j_y}) \in E_i^{PCG}$ if and only if the following criteria are satisfied:

$$V_{j_x} \subset V_{j_y} \tag{21}$$

$$\neg \exists V_{j_z} \ni V_{j_x} \subset V_{j_z} \subset V_{j_y} \tag{22}$$

Criterion (22) establishes that the subset relation is not trasitive—that there is no other subset of $V_{j_y}$ that is also a superset of $V_{j_x}$.

**Lemma 1.** $G_i^{PCG}$ is acyclic.

**Proof.** Let $c = (v_{j_x}, \ldots, v_{j_y}, v_{j_x})$ be a cycle in $G_i^{PCG}$. If $|c| = 2$, then there is an edge $(v_{j_x}, v_{j_x})$ which is trivially redundant and unnecessary. If $|c| > 2$, then there exists $v_{j_y} \neq v_{j_x}$ such that:

$$V_{j_x} \subset \ldots \subset V_{j_y} \subset \ldots \subset V_{j_x}, \tag{23}$$

which is a contradiction. □

**Lemma 2.** $p = (v_{j_x}, v_{j_{x+1}}, \ldots, v_{j_y})$ is a path in $G_i^{PCG}$ if and only if $V_{j_x}, V_{j_{x+1}}, \ldots, V_{j_y}$ are compatible.

**Proof.** Follows immediately from (21) and (22) taken in conjunction with Criterion 3 in Theorem 2. □

Recall that a compatible subset of productions in $P$ corresponds to a set of code sequences that can be embedded within schedule $S_i$ of $G_i$. By Lemma 2, any path in $G_i^{PCG}$ is a compatible subset. Lemma 3 and Theorem 3 help us establish which compatible subset of P should be selected to optimize code size reduction.

**Lemma 3.** The code size reduction attributable to every vertex $v_{j_m} \in V_i^{PCG}$ is $|V_i|$.

**Proof.** $v_{j_m}$ corresponds to cutset $C_{i \to j_m, k_m} = (V_{j_m}, V_{k_m})$. $S_{j_m}$ and $S_{k_m}$ are schedules of $G_{j_m}$ and $G_{k_m}$ respectively. By selecting schedule $S_{i \to j_m k_m} = S_{j_m} S_{k_m}$, we eliminate the need to store dictionary entries for $G_{j_m}$ and $G_{k_m}$. The corresponding reduction in dictionary size is given by $|V_i| = |V_{j_m}| + |V_{k_m}|$. □

Theorem 3 summarizes this result.

**Theorem 3.** The compatible subset $P_{max}$ of $P_1$ that maximizes code size reduction corresponds to the path $p_{max}$ of maximum length in $G_i^{PCG}$.

**Proof.** Follows immediately from Lemmas 1, 2, and 3. □

Fig. 6 illustrates the construction of the PCG, for a DFG $G_1$. Five pairs of convex cuts, $(V_i, V_{i+1})$ are shown for $i = 2, 4, 6, 8, 10$. A PCG, $G_1^{PCG}$, is shown. Each vertex $v_i^{PCG}$ in $G_1^{PCG}$ corresponds to cut $(V_i, V_{i+1})$. $G_1^{PCG}$ contains no transitive edges as a consequence of criteria (21) and (22).

## 4. Dictionary Construction via Dynamic Programming

In this section, we present a dynamic programming algorithm that constructs a dictionary from a set of DFGs representing patterns generated by applying Brisk's algorithm to the CDFG representation of a program. Pseudocode is shown in Fig. 7.

The input to the algorithm is a set of DFGs $G = \{G_1, G_2, \ldots, G_n\}$ and an SH, represented as a DAG $G^{SH} = (V^{SH}, E^{SH})$, where $V^{SH} = G$ and edge $e = (G_i, G_j) \in E^{SH}$ indicates that $G_j$ is an immediate subgraph of $G_i$. This is a slightly different representation of the SH than was described in Section 3.2. The primary difference is that here, we are interested in only the immediate subgraph relation. The simplest way to view this construction of $G^{SH}$ is to add an edge $(G_i, G_j)$ for each production $S_i \to \ldots S_j \ldots$ in the SHG.

Line 1 in Fig. 7 initializes an empty dictionary; the rest of the algorithm constructs the dictionary, which is returned in Line 14.

The loop spanning Lines 2-13 performs the actual dictionary construction. Line 3 removes all sources and sinks in the SH that correspond to patterns that do not exist in the SH. All such sinks were consumed by larger templates during Brisk's algorithm [3]. All such sources were considered, but not actually introduced. None of these patterns should be represented in the dictionary.

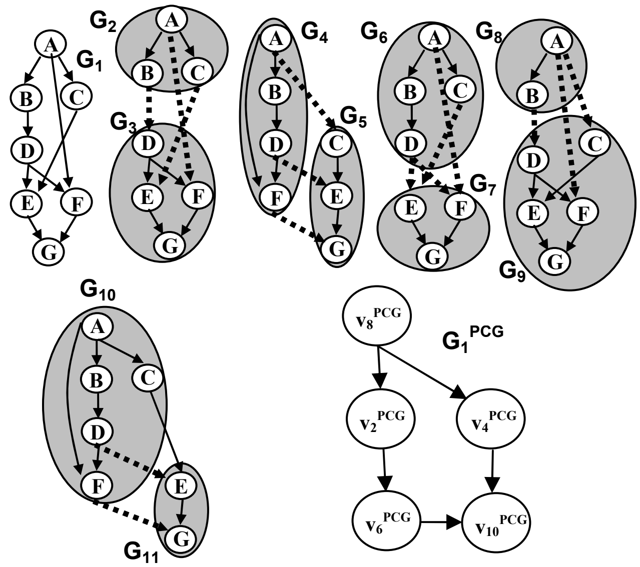


**Figure 6.**
**Illustrating construction of the PCG**

```
Construct_Echo_Dictionary(
    G = {G1, G2, …, Gn} : set of DFGs,
    GSH = (VSH, ESH) : subgraph hierarchy
) : Dictionary
1.  Dictionary : D ← initialize empty
        dictionary.
2.  While( |VSH| > 0 )
3.      Remove all unnecessary sources and
        sinks.
4.      Topologically sort GSH
5.      For each vertex vi ∈ VSH, taken in
        reverse topological order
6.          Let Gi be the DFG corresponding
            to vi.
7.          Propagate scheduling constraints
            from each compatible sub-pattern
            of Gi to Gi
8.          Compute gain(Gi)
9.      EndFor
10.     Let Gmax be the source in GSH such that
        gain(Gmax) is maximum.
11.     Recursively extract the compatible
        subgraph of GSH rooted at Gmax.
12.     Add the extracted subgraph to D
13. EndWhile
14. Return D
```

**Figure 7.**
**Dictionary construction heuristic**

The second step within the outer loop is to topologically sort $G^{SH}$ in Line 4. The inner loop spanning Lines 5-9 traverses $G^{SH}$ in reverse topological order—from sinks to sources. The outer loop terminates when all vertices have been removed from $G^{SH}$.

Let $gain(G_i)$ represent the benefit associated with creating a dictionary entry for $G_i$. This gain must account for all of the subgraphs $G_j$, $G_k$ of $G_i$ that will reference the eventual code sequence $S_i$. *gain*($G_i$) is computed in Line 8 of Fig. 7. A table of size *O(n)* stores *gain*($G_i$) for every DFG in *G*. By traversing $G^{SH}$ in reverse topological order, *gain*($G_j$) has already been computed for each successor (subgraph) $G_j$ of $G_i$. The table therefore uses dynamic programming to recursively compute *gain*($G_i$). The details of this computation are described in Section 4.1. The previous step in Line 7 is described in Section 4.2.

The computation of *gain*($G_i$) uses Theorem 3 to compute a compatible set of sub-patterns of $G_i$. Once this value is known for every pattern in $G^{SH}$, a source $G_{max}$ that maximizes *gain*($G_i$) is identified in Line 10. Line 11 removes all vertices from the PHG corresponding to patterns that will reference $G_{max}$'s dictionary entry. Line 12 creates the actual dictionary entry.

First, $G_{max}$ is removed from $G^{SH}$. Next, each compatible sub-pattern of $G_{max}$, $G_j$, is also removed from $G^{SH}$. Recursively, all compatible sub-patterns of $G_j$ are removed too, etc. Let $V^{SH}{}_{max}$ be the set of patterns in $G^{SH}$ that will reference $G_{max}$'s dictionary entry. The subgraph of $G^{SH}$ induced by $V^{SH}{}_{max}$ is removed from $G^{SH}$ and placed into the dictionary. Since all edges in the induced subgraph are compatible, a schedule for $G_{max}$ can be constructed that is compatible with all patterns in the induced subgraph, as discussed in Sections 3.6 and 3.7. The induced subgraph suffices as a dictionary entry for $G_{max}$ as well as all subsumed patterns.

### 4.1 Computing gain($G_i$)

For pattern $G_i$, let $b_i = 1$ if $G_i$ is one of the patterns occurring in the final program; otherwise, $b_i = 0$. Sources and sinks with $b_i = 0$ were removed in Line 3 of Fig. 7. Internal patterns are maintained to preserve the subgraph relation, which is transitive.

To compute *gain*($G_i$), we associate a gain with each production $G_i \rightarrow \ldots$, denoted *gain*($G_i \rightarrow \ldots$). There are 4 cases to consider:

$$gain(G_i \rightarrow v_j v_k) = 2 \cdot b_i \tag{24}$$

$$gain(G_i \rightarrow v_j G_k) = b_i \cdot |V_i| + gain(G_k) \tag{25}$$

$$gain(G_i \rightarrow G_j v_k) = b_i \cdot |V_i| + gain(G_j) \tag{26}$$

$$gain(G_i \rightarrow G_j G_k) = b_i \cdot |V_i| + gain(G_j) + gain(G_k) \tag{27}$$

For each production, $G_{i \rightarrow j_m, k_m} \rightarrow G_{j_m} G_{k_m}$, the quantity $gain(G_{i \rightarrow j_m, k_m} \rightarrow G_{j_m} G_{k_m})$ is assigned as a weight of the corresponding vertex $v_{j_m} \in V_i^{PCG}$. A locally optimal subset of productions for pattern $G_i$ can be constructed by finding the path of maximal weight in $G_i^{PCG}$. Let $P_{i,max}$ be the set of vertices in $G_i^{PCG}$ contained on the maximum weight path. Let *gain*($P_{i,max}$) be the sum of the gains of the productions associated with each vertex contained in $P_{i,max}$. Finally, let *gain*($G_i$) be the total gain associated with pattern $G_i$ and all of its sub-patterns. Then:

$$gain(G_i) = gain(P_{i,\max}) + b_i \cdot |V_i| \tag{28}$$

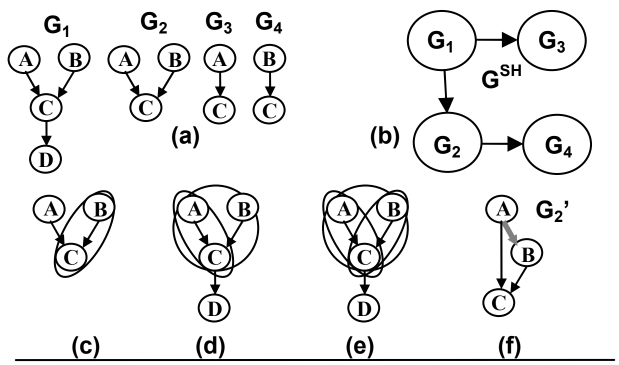


**Figure 8**
**Illustrating the propagation of scheduling constraints while traversing $G^{SH}$**

### 4.2 Propagating Scheduling Constraints

Consider DFGs $G_1$, $G_2$, $G_3$, and $G_4$ shown in Fig. 8 (a). $G^{SH}$ is shown in Fig. 8 (b). Observe that $G_3 \subset G_2$ despite the fact that there is no edge $(G_2, G_3)$ in $G^{SH}$. This simply indicates that no instance of $G_2$ in the program was formed by combining an instance of $G_3$ with a vertex labeled *B*. Assume $b_i = 1$, $1 \leq i \leq 4$.

Now, let us compute *gain*($G_i$), for $1 \leq i \leq 4$. Trivially, *gain*($G_3$) = *gain*($G_4$) = 2. $G_4$ is the only compatible sub-pattern of $G_2$, as illustrated in Fig. 8 (c). Consequently, *gain*($G_2$) = 5.

Now, let us process $G_1$, ignoring, for the moment, the fact that $G_4$ has been selected as a compatible sub-pattern of $G_2$. As illustrated by Fig. 8 (d), $G_2$ and $G_3$ are both compatible sub-patterns of $G_1$. Now, recall that $G_4$ is a sub-pattern of $G_2$; by transitivity, $G_4$ is also a sub-pattern of $G_1$. $G_4$, however, is not compatible with $G_3$, as illustrated by Fig. 8 (e). The dynamic programming algorithm has already selected $G_4$, as a compatible sub-pattern of $G_2$. Therefore, $G_2$ and $G_3$ are not compatible sub-patterns of $G_1$.

Selecting $G_4$ as a compatible sub-pattern of $G_2$ creates a scheduling constraint—the schedule of $G_2$ must place vertex A prior to the vertices in $G_4$. To represent this constraint, the separating edge *(A, B)* must be added to $G_2$. The resulting DFG, $G_2'$, is shown in Fig. 8 (f). $G_2'$ and $G_3$ are trivially incompatible with $G_1$. Since *gain*($G_2$) > *gain*($G_3$), $G_2$ is selected as the only compatible sub-pattern of $G_1$; consequently, *gain*($G_1$) = 9.

The separating edges corresponding to each compatible set of sub-patterns must be added to each DFG in the hierarchy as it is processed. Otherwise, scheduling decisions made during the early stages of dynamic programming will not percolate to the top of $G^{SH}$.

The dictionary entry for $G_1$ will cover 9 operations—4 from $G_1$, 3 from $G_2$, and 2 from $G_4$. A separate dictionary entry will be constructed for $G_3$. The final dictionary will contain 6 operations.

## 5. Experimental Results

We have integrated our dictionary construction algorithm into a compression framework [3] within the Machine SUIF compiler [25]. The first step of the back of the compiler is to instruction selection. Machine SUIF is a retargetable compiler that provides back end support for the Alpha, x86, and Itanium architectures.

Following the lead of Lau et al. [17] and Brisk et al. [3], we targeted a version of the Alpha architecture that has been modified to support echo instructions.

We used an isomorphic pattern generation algorithm described by Brisk [3] to identified recurring patterns within the compiler's intermediate representation. We modified the algorithm so that it built the PHG and performs the dictionary construction algorithm described in Sections 3 and 4 of this paper.

### 5.1 Benchmarks

Code compression is a topic that is primarily of interest to embedded system designers. With that in mind, we selected a set of 10 benchmarks from the MediaBench application suite [18]: Epic, G.721, GSM, JPEG, MPEG2 Decoder and Encoder, Pegwit, PGP, RSA (within PGP), and Rasta. Adpcm was not compiled because it is notably smaller than the others and exhibits considerable redundancy. Ghostscript and Mesa are larger than the others, and are thus less representative of embedded applications.

The source code files for each benchmark were linked using the *link_suif* pass. This required manual intervention to prevent namespace collisions. To reduce overall code size, we rolled the unrolled loops that occurred in several of the benchmarks.

### 5.2 Dictionary Construction Results

The majority of dictionary compression techniques (e.g., Lefurgy [19]) do not reduce the dictionary size using substring matching. They simply place one instance of each pattern in the dictionary—the naïve approach. Fig. 9 compares the sizes (in terms of operations) of the dictionaries constructed by the naïve and heuristic methods. The reductions in dictionary sizes ranged from 21.14% (JPEG) to 29.76% (Epic). Across all benchmarks, the number of operations in all dictionaries was reduced from 26629 to 20174, a reduction of 24.24%.

Table I lists three quantities for each benchmark: total compilation time, time spent during dictionary construction, and the percentage of time spent during dictionary construction; the third quantity can easily be derived from the first two. Because Brisk's algorithm [3] relies on repeated calls to an exact isomorphism algorithm, the time spent constructing the dictionary is small relative to the entire compilation process.

Compilation times ranged from 2.78 seconds (G.721) to 363 seconds (JPEG). The amount of time spent on dictionary construction ranged from 0.194 seconds (G.721) to 15.7 seconds (JPEG). As a percentage of compilation time, dictionary construction ranged from 2.38% (GSM) to 6.98% (G.721).

### 5.3 Discussion

Our intention was to compare the heuristic technique to a similar algorithm based on substring matching; however, to use substring matching, we must first schedule each DFG before constructing the dictionary. There may be many different schedules for each DFG—an exponential number per DFG in the worst case.

Suppose that we have $n$ DFGs, and there are $k$ possible schedules for each. Then the total number of possible schedules for all DFGs is $n^k$. As illustrated by Fig. 2, the quality of the results of substring matching depends on how the DFGs are scheduled relative to one another.

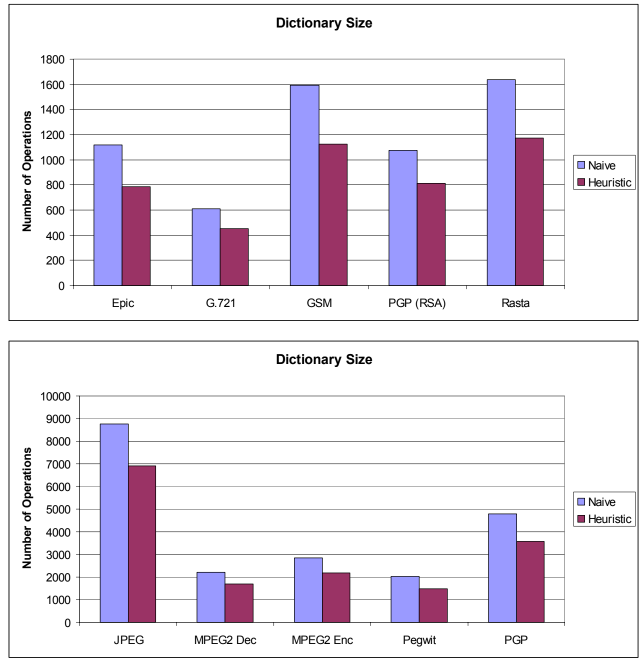


**Figure 9.**
**Dictionary construction results**

**Table I.**
**Compilation and dictionary construction time (seconds, %) for the 10 benchmarks**

| Benchmark | Total (sec) | Dictionary (sec) | (%) |
|---|---|---|---|
| Epic | 10.3 | 0.528 | 5.13 |
| G.721 | 2.78 | 0.194 | 6.98 |
| GSM | 34.3 | 0.816 | 2.38 |
| JPEG | 363 | 15.7 | 4.33 |
| MPEG2 Dec | 33.1 | 1.31 | 3.96 |
| MPEG2 Enc | 66.9 | 1.98 | 2.96 |
| Pegwit | 33.8 | 1.10 | 3.25 |
| PGP | 201 | 5.62 | 2.80 |
| PGP (RSA) | 9.36 | 0.516 | 5.51 |
| Rasta | 18.4 | 0.866 | 4.70 |

The problem of determining the best schedule for each DFG for dictionary compression must be solved prior to substring matching. The heuristic presented here solves this scheduling problem sub-optimally. The use of graph isomorphism during PHG construction eliminates the need for substring matching. Consequently, the two problems are identical.

De Sutter et al. [8] reported marginal results of an optimization strategy that attempted to minimize differences in instruction schedules in order to maximize the code size reductions obtained by a substring matching/register renaming technique. We observed that up to 40% of the patterns found by Brisk's algorithm across all benchmarks did not occur contiguously in the Machine SUIF CFG, where basic blocks are lists of instructions.

We attribute this discrepancy to two factors. First, De Sutter's optimization was performed at link-time, after a compiler scheduled the code using a deterministic heuristic. Secondly, De

Sutter's benchmarks were written in C++, where considerable redundancy occurred due to template instantiation and inheritance. The repeated code fragments, once again, are likely to be initialized to identical default schedules when the intermediate representation is first constructed. All Mediabench applications, in contrast, are written in C.

Finally, De Sutter replaced redundant code with procedure calls, whereas we are targeting a system with echo instructions. An echo instruction encodes the number of dictionary instructions to execute in one of its fields; a procedure terminates upon executing a return instruction. Under this model, a return instruction must terminate the substring. This return instruction would then preempt the superstring—incorrectly, unless the substring matches the terminating characters of the superstring.

## 6. Future Work

The dictionary construction method presented here is specific to echo instructions [9][17][3]. The technique could also be used for CALD instructions [21] and DISE decompression [7]. These two technologies offer opportunities for dictionary compression in excess of echo instructions. As an example, consider three code sequences AB, BC, and CD. A dictionary entry ABCD could be constructed that allows BC to span two the respective entries for AB and CD. Observe that BCis not a substring (or a subgraph in a DFG representation) of AB or BC. Since patterns AB and CD arunlikely to occur contiguously in the program, three separate entries would be needed for a dictionary using echo instructions.

## 7. Conclusion

We have developed a theoretical model for the problem of constructing a dictionary for a set of redundant code sequences represented as DFGs. This approach differs from post-compilation analyses that use linear substring matching to construct a dictionary. To solve this problem, we introduce an efficient dynamic programming heuristic that performed dictionary construction. Our experiments with 10 MediaBench applications yielded reductions in dictionary size ranging from 21.14% to 29.76% relative to naïve methods.

## Acknowledgements

The feedback provided by the anonymous reviewers enabled us to make significant improvements to the paper. Their time and effort are greatly appreciated.

## References


[1] Advanced RISC Machines Ltd. An Introduction to Thumb, Ver. 2.0. White Paper, March 1995.

[2] Benini, L., Macii, A., and Nannarelli, A. Cache-Code Compression for Energy Minimization in Embedded Processors. Int. Symp. Low Power Electronics and Design (ISLPED), 2001, 322-327.

[3] Brisk, P., Nahapetian, A., and Sarrafzadeh, M. Instruction Selection for Compilers that Target Architectures with Echo Instructions. Int. Workshop on Software and Compilers for Embedded Systems (SCOPES), 2004, 229-243.

[4] Chen, I., Bird, P., and Mudge, T. The Impact of Instruction Compression on I-cache Performance. Tech. Rpt. CSE-TR-330-97, EECS Dept., University of Michigan, 1997.

[5] Chen, W-K., Li, B., and Gupta, R. Code Compaction of Matching Single-Entry Multiple-Exit Regions. Static Analysis Symp., 2003, 401-417.

[6] Cooper, K. D., and McIntosh, N. Enhanced Code Compression for Embedded RISC Processors, Int. Conf. Programming Language Design and Implementation (PLDI), 1999, 139-149.

[7] Corliss, M. L., Lewis, E. C., and Roth, A. A DISE Implementation of Dynamic Code Decompression. Conf. Languages, Compilers, and Tools for Embedded Systems (LCTES), 2003, 232-243.

[8] De Sutter, B., De Bus, B., and De Bosschere, K. Sifting out the Mud:Low Level C++ Code Reuse. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2002, 275-291.

[9] Debray, S., Evans, W., Muth, R., and de Sutter, B. Compiler Techniques for Code Compaction. ACM Trans. Programming Languages and Systems (TOPLAS), 22(2):347-415, 2000.

[10] Fraser, C.W. An Instruction for Direct Interpretation of LZ77-compressed Programs. MSR-TR-2002-90, Microsoft Research, September, 2002.

[11] Fraser, C. W., Myers, E., and Wendt, A. Analyzing and Compressing Assembly Code. Int. Symp. Compiler Construction, 1984, 117-121.

[12] Garey, M. R., and Johnson, D. S. Computers and Intractability: A guide to the Theory of NP-Completeness, New York, W. H. Freeman and Co., 1979.

[13] Kastner R., Memik, S. O., Bozorgzadeh, E., and Sarrafzadeh, M. Instruction Generation for Hybrid-Reconfigurable Systems. ACM Trans. Design Automation of Embedded Systems (TODAES), 7(4):605-627, October, 2002.

[14] Kemp, T. M., Montoye, R. K., Harper, J. D., Palmer, J. D., and Auerbach, D. J. A decompression core for PowerPC. IBM Journ. Research and Development, 42(6):807-812, November, 1998.

[15] Kissel, K. D. MIPS16: High-density MIPS for the Embedded Market. White Paper. Silicon Graphics MIPS Group, 1997.

[16] Kunchithapadam, K., and Larus, J. R. Using Lightweight Procedures to Improve Instruction Cache Performance. Tech. Rept. 1390, CS Dept., University of Wisconsin-Madison, 1999.

[17] Lau, J., Schoenmakers, S., Sherwood, T., and Calder, B. Reducing Code Size with Echo Instructions. Int. Conf. Compilers, Architectures, and Synthesis for Embedded Systems (CASES), 2003, 84-94.

[18] Lee, C., Potkonjak, M., Mangione-Smith, W. H. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. Int. Symp. Microarchitecture (MICRO-30), 1997, 330-335.

[19] Lefurgy, C. R. Efficient Execution of Compressed Programs. Ph.D. Thesis, EECS Dept., University of Michigan, 2000.

[20] Lekatsas, H., and Wolf, W. Code Compression for Embedded Systems. Design Automation Conf. (DAC), 1998, p. 516-521.

[21] Liao, S., Devadas, S., and Keutzer, K. A Text-Compression-Based Method for Code Size Minimization in Embedded Systesms. ACM Trans. Design Automation of Electronic Systems (TODAES), 4(1):12-38, January, 1999.

[22] Runeson, J. Code Compression through Procedural Abstraction before Register Allocation. Master's Thesis. Computing Science Department, Uppsalla University, Sweden, March, 2000.

[23] Storer, J. NP-Completeness Results Concerning Data Compression. Tech. Rpt. 234, Dept. Electrical Engineering and Computer Science, Princeton University, 1977.

[24] Wolfe, A., and Chanin, A. Executing Compressed Programs on an Embedded RISC Architecture. Int. Symp. Microarchitecture (MICRO-25), 1992, 81-91.

[25] http://www.eecs.harvard.edu/hube/research/machsuif.html