Compression of Compiler Intermediate Representations of Program Code

Philip Brisk\textsuperscript{1}, Ryan Kastner\textsuperscript{2}, Jamie Macbeth\textsuperscript{1}, Ani Nahapetian\textsuperscript{1}, and Majid Sarrafzadeh\textsuperscript{1}

\textsuperscript{1} Department of Computer Science  
University of California, Los Angeles  
Los Angeles, CA 90095  
{philip, macbeth, ani, majid}@cs.ucla.edu

\textsuperscript{2} Department of Electrical and Computer Engineering  
University of California, Santa Barbara  
Santa Barbara, CA 93106  
kastner@ece.ucsb.edu

A previous version of this paper appeared in SCOPES 2004 under the name Instruction Selection for Compilers that Target Architectures with Echo Instructions.

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

Additional Key Words and Phrases:  
(Code) Compression, (Sub)Graph Isomorphism, Independent Set
Abstract

Code compression reduces the size of a program to be stored on-chip in an embedded system. We introduce an algorithm that a compiler may use to compress its intermediate representation of a program. The algorithm relies on repeated calls to an exact isomorphism algorithm in order to identify redundant patterns that occur within the program, which is represented as a Control Data Flow Graph (CDFG). Repeated patterns are then extracted from the program, and each occurrence is replaced with a pointer to a single representative instance of the pattern. This algorithm can also be used to perform instruction selection for embedded architectures that feature specialized assembly instructions for dictionary compression. In experiments with 10 embedded applications written in C, the number of vertices and edges in the intermediate representation were reduced by as much as 51.90% and 69.17% respectively. An experiment using quadratic regression yielded strong empirical evidence that the runtime of the compression algorithm is quadratic in the size of the program that is compiled, under the assumption that no unrolled loops are present in the initial code that is compiled.
1. Introduction

In order to facilitate fast design turnaround times, there is considerable pressure on embedded system engineers to implement as much functionality as possible in software rather than customized hardware. In embedded systems where programs are stored in on-chip ROMs, a reduction in code size yields a reduction in ROM size, which in turn reduces the silicon cost of the design—and inevitably the cost paid by the consumer. The cost of storing a program on chip is comparable to the cost of microprocessor that executes the program. Therefore, minimizing code size is of the utmost importance for competitive embedded system vendors.

Due to the rigorous semantics of programming languages, compiler intermediate representations of programs exhibit a considerable amount of redundancy, which can be exploited to reduce code size. From an abstract perspective, an intermediate representation is a hierarchical collection of graphs that represent the computations performed by the program. Within this collection, redundancy takes the form of subgraphs that are isomorphic to one another.

This paper describes a compression algorithm that identifies and extracts isomorphic patterns occurring within a program represented as a Control Data Flow Graph (CDFG). The compiler maintains a single representative instance of each pattern that occurs throughout the program. Each pattern occurring within the intermediate representation is replaced by a single vertex that points to the representative instance. This effectively reduces the total number of vertices and edges in the program’s intermediate representation.

Historically, compression has been performed by a link-time optimizer that operates on pre-compiled assembly code. This work, in contrast, represents an effort to move the compression step to a point as early as possible during compilation. The compressed intermediate representation that results from applying the algorithm is in fact an abstract representation of redundancy. As compilation progresses, many patterns that were identical at earlier stages become specialized—i.e. they no longer remain identical—due to differences in register allocation/assignment and instruction scheduling within the two patterns.

By identifying redundant patterns prior to these optimization steps, future strategies can be developed that attempt to preserve redundancy within the application. Once the program is compiled, all patterns that have remained identical throughout all stages of compilation can be extracted from the program and placed into a dictionary for compression using one of the facilities listed above. The compression algorithm described in this paper is intended to be the first step in this framework. Future efforts will focus on the register allocation and instruction scheduling tasks.

We have identified two applications for this algorithm. The first application is to compress the intermediate representation to reduce storage costs. The second application is to aid the back-end optimization stages of compilers that target architectures that feature ISA facilities for code compression such as Echo instructions (Fraser [2002], Lau et al. [2003], Brisk et al. [2004]), Call Dictionary (CALD) instructions (Liao et al. [1999], Lefurgy et al. [1997]), and Dynamic Instruction Stream Editing (DISE) (Corliss et al. [2002]).
The technique presented in this paper managed to find considerable redundancy within 10 embedded benchmarks taken from the MediaBench (Lee et al. [1997]) application suite. The percentage reduction in the number of vertices in the intermediate representation ranged from 34.08% to 51.90% and the percentage reduction in edges ranged from 44.90% to 69.18%. Compilation time for these benchmarks ranged from 2.51 seconds to 5 minutes and 46 seconds. A larger benchmark yielded respective vertex and edge reductions of 58.76% and 79.89%, but required 50 minutes and 4 seconds to compile.

The paper is organized as follows. Section 2 discusses related work in the fields of code compression and abstraction. Section 3 introduces preliminary concepts required to understand the compression technique in Section 4. Experimental results are presented in Section 5. Section 6 concludes the paper.

2. Related Work

Here, we provide an overview of techniques that have been used to identify redundant structure in program source code. Many, but not all, of these techniques focus on code size minimization. Sections 2.1-2.5 summarize related work in five fields where redundancy identification is used.

2.1 Procedural Abstraction

Procedural abstraction is the process of identifying redundant code segments and replacing them with procedure calls. The two primary applications of abstraction are code size minimization and software maintenance of legacy code. Fraser et al. [1984] developed a technique based on substring matching that identified identical instruction sequences that occur throughout the program, and then replaced each sequence with a procedure call. This technique was appropriate for mid-1980s processor technology, where almost all computation was routed through a single general-purpose register; however, as RISC processors with 32+ general purpose registers emerged, strict substring matching was rendered ineffective. Since then, substring matching has been augmented with additional techniques such as parameterization (Zastre [1995]), predication (Cheung et al. [2003]) register renaming (Cooper and McIntosh [1999], Debray et al. [2000], De Sutter et al. [2002]), and instruction rescheduling (Lau et al. [2003]). De Sutter et al. [2002] used specialized versions of the above techniques to detect redundancy arising in C++ programs due to programming techniques such as inheritance and polymorphism.

All of the above techniques attempted to minimize code size. Several techniques for procedural abstraction for software maintenance have been proposed in a series of papers by Komondoor and Horwitz [2000] [2001] [2003]. Their technique employs program slicing and uses a data structure called the program dependence graph to alleviate the reliance on linear string matching. Unlike the compaction techniques described above, their approach is integrated into the early stages of compilation—prior to register allocation and instruction scheduling. In recent years, Runeson [2000], Chen et al. [2003] and Brisk
et al. [2004] have proposed similar methods that identify redundancy to reduce code size prior to register allocation and/or scheduling. This paper summarizes Brisk’s technique in detail.

2.2 Dictionary Compression

Dictionary compression is a hardware (or software) supported approach to code size minimization that is similar in principle to procedural abstraction. Each repeated code fragment is collected into a dictionary, an external memory that contains all of the program fragments. In the actual program, each fragment is replaced with a Call Dictionary instruction CALD(addr, N) (Liao et al. [1999]), where addr is the dictionary address of the start of the sequence and N is the length of the sequence. To execute a CALD instruction, control is transferred to the dictionary address, the next N instructions are executed from the dictionary, and control is then transferred to the instruction following the CALD. Assuming a fixed 32-bit ISA, code size can be further reduced if CALD instructions are expressed with fewer bits (i.e. 8 or 16) (Lefurgy et al. [1997]).

The Echo Instruction (Fraser [2002], Lau et al. [2003]) is similar to the CALD instruction, but with one major exception. Rather than moving all instances of each code sequence to a dictionary, one instance of each sequence is left inline in the program. All other instances are replaced with an instruction ECHO(addr, N), such that the desired instruction sequence (of length N) begins at memory location PC – addr. Other than this one distinction, Echo and CALD are similar. The algorithm described in this paper was first developed by Brisk et al. [2004] to target next-generation architectures that feature echo instructions.

Dynamic Instruction Stream Editing (DISE) decompression (Corliss et al. [2002]) offers a parameterized implementation of the CALD instruction. DISE allows for instruction sequences that are identical within a renaming of registers to share a common code fragment. The compiler explicitly places instructions that enable register renaming within DISE into the program. When recognized at runtime, registers are renamed so that code fragments become identical when executed.

The analysis presented in this paper was originally targeted for architectures featuring echo instructions (Brisk et al. [2004]). Practically speaking, the technique could be used in any compiler that targets an architecture that features dictionary compression.

2.3 Pre-Cache Decompression

Statistical compression mechanisms for text such as Huffman encoding (Huffman [1952]) exploit the fact that the distribution of characters within larger data segments is usually non-uniform. For example, in the English language, vowels appear quite frequently, whereas letters such as ‘j’, ‘k’, ‘q’, ‘x’ and ‘z’ occur much less frequently. The same argument can be applied to assembly instructions as well.
Consequently, frequently occurring characters are encoded with shorter codewords than infrequently occurring characters.

The *Compressed Code RISC Processor (CCRP)* (Wolfe and Chanin [1992], Kozuch and Wolfe [1994], Lekatsas and Wolf [1998]) was one of the first hardware-support compression mechanisms to be proposed. Cache blocks are compressed using a statistical encoding method and are stored in memory. When a cache miss occurs, each cache block is decompressed by a custom hardware placed on the cache refill path. This approach was later integrated into IBM’s *CodePack* decompressor for the PowerPC (Kemp et al. [1998], Lefurgy et al. [1999]). Similar systems that perform software decompression were developed by Kirovski et. al. [1997] and Lefurgy et al. [2000]. Debray and Evans [2002] developed a software-based decompression technique that used runtime profile information to compress the least frequently executed portions of the program.

A well-known result from information theory is that for every dictionary compression method, there exists a statistical compression method offers equal or better compression (Bell [1990]). Although our technique has been specialized to perform dictionary compression, the basic concepts and ideas could easily be applied to improve the quality of statistical compression. Each code fragment is mapped to a unique character. If a fragment exists in many program locations, then its character frequency will be large, and it will be encoded with a shorter codeword than less frequently occurring instructions. This is an avenue for future work, and is not addressed elsewhere in this paper.

### 2.4 Custom Instruction Set Specialization

Generating a customized instruction set for an application yields significant reductions in code size relative to other compression mechanisms. Such programs are typically executed through a software interpreter; the alternative is to design application-specific hardware, which may entail significant development and fabrication costs.

*Superoperators* (Proebsting [1995], Proebsting and Fraser [1997]) are virtual machine operations that are built from smaller operations. Compilers that use superoperators produce a stack-based bytecode representation of a program—the custom instruction set—along with an interpreter to execute it. Araujo et al. [1998] developed a similar approach that used hardware rather than software to perform decompression.

To generate superoperators, Ernst et al. [1997] use an analysis that consists of two phases, operand factorization, and opcode combination. To understand operand factorization, consider an assembly instruction of the form: \( \text{dst} \leftarrow \text{src}_1 \text{ op } \text{src}_2 \). This instruction could be assigned a single opcode, essentially fixing the source and destination operands. For example, if multiple destinations are possible, then \( * \leftarrow \text{src}_1 \text{ op } \text{src}_2 \) would be an appropriate specialization. This specialization requires the user to specify \( * \), the field of the destination register along with the opcode. In general, there are \( 2^N \) specializations for each instruction having \( N \) operands.
Lucco [2000] later extended this technique by separating opcodes and operands into streams which are compressed independently; this is combined with dictionary compression techniques as described in Section 2.2. Fraser [1999] used machine learning techniques using a set of training program to separate the program into streams to improve the overall quality of compression. Evans and Fraser [2001] also used machine learning to rewrite a grammar representation of a program to ensure a shorter derivation, thereby reducing code size.

Once an appropriate specialization has been selected for each instruction, the opcode combination phase merges adjacent instructions into superoperators. The algorithm presented in this paper could be used in place of an opcode combination phase. Opcode combination assumes a fixed ordering of instructions. Unlike opcode combination, our technique uses graph isomorphism to identify potential candidates, eliminating the default ordering.

2.5 Compression for Software Distribution

The program compression techniques in Sections 2.2-2.4 all require a mechanism to interpret the program on the fly. Because the goal is to minimize code size, the system does not have the option to decompress the entire program prior to execution. In software distribution systems over the Internet, in contrast, the purpose of compressing the program is to minimize transfer times and overall bandwidth. Once the transfer is completed, the receiving host decompresses the program and executes it. This allows for more aggressive compression techniques to be used that need not worry about whether the resulting program can be interpreted efficiently—or not. Examples of such compression systems include the slim binary format (Franz and Kistler [1997]) as well as a wire format developed by Ernst et al. [1997]. The algorithm presented in this paper could be integrated into a compression mechanism for such a system.

3. Preliminaries

In this section, we introduce preliminary concepts and definitions that are fundamental to this paper. Sections 3.1 and 3.2 discuss the independent set and graph isomorphism problems and the algorithms that we used to solve them. Section 3.3 discusses the Control Data Flow Graph (CDFG) intermediate representation which is used throughout this paper.

3.1 The Independent Set Problem

Let \( G = (V, E) \) be a graph. For subset \( V' \subseteq V \), let \( G' = (V', E') \) be the subgraph of \( G \) induced by \( V' \), where \( E' = \{(u, v) \mid u \in V' \land v \in V'\} \). \( V' \) is defined to be an independent set in \( G \) if \( E' \) is empty. The decision problem of determining whether \( G \) contains an independent set of cardinality at least \( K > 0 \) is NP-Complete (Garey and Johnson [1979]).
We need to solve the corresponding optimization problem: find the independent set of maximal cardinality in G. To accomplish this task, we use a simple iterative improvement algorithm developed as part of a larger graph coloring package by Kirovski and Potkonjak [1997]. Pseudocode for this algorithm is shown in Fig. 1. The algorithm takes two parameters: a graph $G = (V, E)$, and an integer $\text{limit}$, which controls the stopping condition, effectively allowing the user to tradeoff between runtime and solution quality. Lines 1 and 2 initialize empty independent sets $\text{Best}$ and $\text{S}$ to the empty set. Line 3 initializes an integer $\text{no\_improvement}$ to 0. $\text{Best}$ tracks the largest independent set observed thus far. $\text{S}$ is the current independent set at each step of the algorithm. $\text{no\_improvement}$ tracks the number of iterations of the algorithm that have passed since the last improvement to $\text{Best}$. The algorithm terminates when the condition $\text{no\_improvement} = \text{limit}$ is satisfied.

During each iteration of the algorithm, $\text{S}$ is perturbed using the functions $\text{randomized\_vertex\_exclusion}$ and $\text{randomized\_vertex\_inclusion}$. These functions randomly add and remove vertices from $\text{S}$, allowing the iterative improvement algorithm to randomly explore the search space. The algorithm in Fig. 1 was selected due to its simplicity and ease of implementation; of course, it could easily be replaced with any other heuristic or branch-and-bound algorithm if desired.

3.2 The Graph Isomorphism Problem

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. $G_1$ and $G_2$ are isomorphic if $|V_1| = |V_2|$, $|E_1| = |E_2|$, and there exists one-to-one and onto function $f : V_1 \to V_2$, such that $(u, v) \in E_1$ if and only if $(f(u), f(v)) \in E_2$. The problem of determining whether two graphs are isomorphic to one another or not has never been formally proven NP-Hard (Garey and Johnson [1979]); nonetheless, all known algorithms that solve the problem exactly possess an exponential worst-case running time (Ullman [1976], Schmidt and Druffel [1976], McKay [1978], Ebeling and Zajicek [1983], Ebeling [1988], Cordella et al. [2004]). The related problem of subgraph isomorphism is NP-Complete (Garey and Johnson [1979]).
It should also be noted that polynomial-time isomorphism algorithms are known for certain classes of graphs such as trees (Aho et al. [1974]), planar graphs (Hopcroft and Wong [1974]), and graphs of bounded valence (Luks [1982]). For our implementation, we selected the publicly available VF2 isomorphism algorithm from the University of Naples (Cordella et al. [2004]) based on a performance comparison between several of the algorithms listed above (Foggia et al. [2001]).

3.3 The Control Flow Graph (CDFG) Intermediate Representation

In a target-independent compiler intermediate representation, a typical operation will be a quadruple of the form \( DST \leftarrow SRC_1 \ op \ SRC_2 \), which specifies two source registers, a destination register, and an integer opcode. Without knowledge of the target, the compiler will assume an infinite supply of registers called temporaries (or virtual registers). A quadruple is effectively a target-independent abstraction of an assembly language instruction for RISC architectures. A basic block is defined to be a linear list of quadruples with no branches or branch targets interleaved.

A Control Flow Graph (CFG) is a directed graph where each vertex represents a basic block, and each edge \( e = (b_i, b_j) \) represents a control transfer from \( b_i \) to \( b_j \). If \( e \) corresponds to a conditional branch, it may be labeled true or false depending on whether the condition causes the branch to be taken or not.

Each basic block may be decomposed into a directed acyclic graph (DAG) called a Data Flow Graph (DFG). A DFG eliminates the linear ordering imposed by the basic block. Instead, a DFG is a partial ordering of operations, where precedence constraints arise due to data dependencies inherent in the program that is compiled. An example of a code fragment represented as a basic block and a DFG is shown in Fig. 2. The pseudocode in Fig. 2 assumes an infinite supply of temporary registers, denoted \( tr_i \). The example in Fig. 2 will be used throughout the paper to illustrate the steps of the compression algorithm.

\[
\begin{align*}
\text{Live-in:} & \quad \{tr_1, tr_2, ..., tr_6\} \\
tr_7 & \leftarrow tr_1 + tr_2 \\
tr_8 & \leftarrow tr_4 + tr_5 \\
tr_9 & \leftarrow tr_3 + tr_8 \\
tr_{10} & \leftarrow tr_6 + tr_8 \\
tr_{11} & \leftarrow tr_7 + tr_8 \\
tr_{12} & \leftarrow tr_7 + tr_9 \\
tr_{13} & \leftarrow tr_9 + tr_{10} \\
tr_{14} & \leftarrow tr_{11} + tr_{12} \\
tr_{15} & \leftarrow tr_{15} \times tr_{13}
\end{align*}
\]

Figure 2.
A basic block (a) and corresponding DFG (b)
A Control Data Flow Graph (CDFG) is a CFG where basic blocks are represented as DFGs instead of lists. A CFG/CDFG represents the body of a single procedure in the program. A complete application will be represented as a set of CDFGs, one for each procedure body. The redundancy identification technique presented in this paper does not consider control flow. Therefore, it suffices to represent the entire application as a set of DFGs.

Let \( G = (V, E) \) be a DFG. Each vertex \( v \in V \) represents a computation performed by quadruple in the basic block, \( B \). An integer type, \( t(v) \), represents the opcode of \( v \)'s quadruple; i.e., \( t(v) \) specifies the computation performed by \( v \)—e.g., an addition, multiplication, load or store. A DFG edge \( e = (u, v) \) indicates that there is a direct data dependency between \( u \) and \( v \) in \( B \); i.e., operation \( u \) writes a value to some temporary register \( tr \); operation \( v \) later reads the value from \( tr \). Each edge \( e \) has an integer type \( t(e) = (t(u), t(v)) \). The edge types can be put into correspondence with the set of nonnegative integers via Cantor’s diagonalization argument.

An important detail in our DFG representation is non-commutative operators. For example, the computations \( A - B \) and \( B - A \) are isomorphic to one another—as shown in Fig. 3 (a).—despite the fact that these two computations are not the same. To remedy this, we label the left and right inputs with values \( L \) and \( R \) respectively, as shown in Fig. 3 (b); equivalently, we could introduce separate unary vertices labeled \( L \) and \( R \), as shown in Fig. 3 (c). This can be generalized to support non-commutative \( n \)-ary operators where input edges are labeled \( 1, 2, \ldots, n \).

4. Compression Algorithm

Here, we describe the main steps of the compression algorithm in detail. The component stages of the algorithm are presented in sections 4.1-4.4; an example is shown in Section 4.5; pseudocode is shown in Section 4.6; implementation details regarding the representation of the compressed program and the decompression step are described in Section 4.7.
4.1 Patterns and Classification

A pattern is defined to be any convex induced subgraph of $G$. For $V' \subseteq V$, and let $G' = (V', E')$ be the subgraph of $G$ induced by $V'$. $V'$ is a convex subgraph of $G$ if and only if $G$ does not contain a path $<v_1, \ldots, v_k>$ of length at least 3 where $v_1, v_k \in V'$, $v_i \in V - V'$, $2 \leq i \leq k-1$. Fig. 4 gives examples of convex and non-convex subgraphs.

We store the set of patterns generated during the subgraph enumeration phase in a hash table. For pattern $p$, a hash function $h(p)$ is computed over some combination of invariant properties of $p$. An invariant property is any numeric quantity that must be equal in order for two patterns $p$ and $p'$ to be isomorphic to one another. Invariant properties include the number of vertices and edges in the pattern, length of the longest path in the pattern, and the frequency distribution of vertex and edges by types.

Classification refers to the process of testing a newly generated pattern $p$ for isomorphism against a set of patterns maintained in a database. $p$ must be assigned an integer type $t(p)$, analogous in spirit to the types assigned DFG vertices. When $p$ is generated, it is tested for isomorphism against a database of patterns observed thus far. If a pattern $p'$ is found to be isomorphic to $p$, then $t(p) = t(p')$; otherwise, $t(p)$ is set to the smallest integer not already assigned to a pattern. $p$ is then copied, and the clone is inserted into the database.

4.2 Templates and Clustering

For the well-known task of instruction selection, the database would be limited to patterns that explicitly represent assembly instructions of the target architecture—the instruction selection problem. To
translate the intermediate representation into an assembly program, the compiler must cover \( G \) with a set of non-overlapping patterns from the database—an NP-hard optimization problem.

Kastner et al. [2002] added an extra degree of freedom to this problem whereby the compiler is allowed to add new patterns to the database. Given a graph \( G = (V, E) \), the compiler may add any of the \( 2^{|V|} \) induced subgraphs of \( G \) as long as the subgraph is convex. No additional constraints are necessary to identify redundant computations.

A template (supernode) \( T \) is a vertex introduced to a DFG that represents the matching of some induced subgraph \( G' = (V', E') \) of \( G \) against a pattern \( p \) in the database. \( T \) literally subsumes \( G' \), which is physically removed from the topology of \( G \). Until the final stages of compression, \( T \) maintains \( G' \) internally and preserves the original internal-external connectivity across the cut \((V', V - V')\) in \( G \). First, \( E \) is partitioned into four disjoint sets:

1. The set of internal edges, \( E_{\text{internal}} = \{(u, v) \mid u \in V' \land v \in V'\} \)
2. The set of external edges, \( E_{\text{external}} = \{(u, v) \mid u \in V - V' \land v \in V - V'\} \)
3. The set of incoming edges, \( E_{\text{incoming}} = \{(u, v) \mid u \in V - V' \land v \in V'\} \)
4. The set of outgoing edges, \( E_{\text{outgoing}} = \{(u, v) \mid u \in V' \land v \in V - V'\} \)

All internal edges are subsumed by \( T \). The external edges are not incident on any vertices in \( V' \), so they remain untouched. The incoming and outgoing edges are removed from the topology of \( G \); however, \( T \) saves the incoming and outgoing edges internally, which are necessary in order to ensure the semantic correctness of the entire program. To represent the data dependencies in \( G \)—where \( T \) is the only exposed remnant of \( G' \)—the set of incoming/outgoing edges are replaced with edges incident on \( T \), defined as follows:

1. The set of incoming edges to \( T \), \( E(T)_{\text{incoming}} = \{(u, T) \mid \exists (u, v) \in E_{\text{incoming}}\} \)
2. The set of outgoing edges from \( T \), \( E(T)_{\text{outgoing}} = \{(T, v) \mid \exists (u, v) \in E_{\text{outgoing}}\} \)

The transformation to introduce the template then proceeds as follows:

\[
V \leftarrow (V - V') \cup \{T\} \\
E \leftarrow E_{\text{external}} \cup E(T)_{\text{incoming}} \cup E(T)_{\text{outgoing}}
\]

An example is shown in Fig. 5 where the subgraph induced by the subset of vertices \( \{B, E, F, H\} \) has been removed from \( G \) and replaced with a template \( T \). The different sets of edges are labeled as external, new edges \( E(T)_{\text{incoming}} \) and \( E(T)_{\text{outgoing}} \), or edges removed \( E_{\text{internal}}, E_{\text{incoming}}, \) and \( E_{\text{outgoing}} \). Edges incident on template vertices are drawn in bold; edges that cross template boundaries \( (E_{\text{incoming}} \) and \( E_{\text{outgoing}}) \) are drawn using dashed lines. Edges that cross no boundaries \( (E_{\text{internal}} \) and \( E_{\text{external}}) \) are drawn normally. This process of replacing an induced subgraph with a template is called clustering.
Multiple templates may be introduced to the same DFG, however, templates may not overlap; moreover, hierarchy is not allowed. In other words, one template cannot subsume another template. We do, however, allow for adjacent templates to merge with one another, as discussed in the next section. First, we must consider adjacency.

Consider a template $T$, and an adjacent vertex $v$ connected by edge $(T, v)$. Let $u$ be a vertex in the subgraph subsumed by $T$ such that $e = (u, v)$ was an edge in $G$ prior to clustering, i.e. $e \in E_{outgoing}$. Now, suppose that $v$ is later subsumed by a template $T'$. Then both $T$ and $T'$ will maintain a pointer to $e$. From the perspective of $T$, $(u, v)$ will be an outgoing edge, while from the perspective of $T'$, $e$ will be incoming. Additionally, an edge $(T, T')$ must be introduced because there is a data dependency between $u$ in $T$ and $v$ in $T'$. Fig. 6 illustrates the transformation.
4.3 Template Enumeration Along DFG Edges

Here, we provide an overview of the mechanism by which new templates are generated and introduced to a DFG, in preparation for the compression algorithm in Section 4.6. Templates are enumerated via DFG edges, as illustrated in Fig. 7. There are four possible cases for each edge depending on whether or not each vertex is a template.

Initially, no templates exist in the DFG. Therefore, every edge \( e = (u, v) \) defines a subgraph \( G' = (V', E') \), where \( V' = \{u, v\} \) and \( E' = \{e\} \). In this case, \( G' \) is replaced with a single template \( T \), as illustrated in Fig. 7 (a). Once templates are introduced, three additional cases (Fig. 7 (b)-(d)) must also be considered. In Fig. 7 (b) and (c), a vertex is combined with a template; in Fig. 7 (d), two templates are merged. W.L.O.G., when two templates are merged, the induced subgraphs they cover are merged, including edges between the two subgraphs. For example, in Fig. 7 (d), edge \( (u, v) \), an outgoing edge of template \( T \) and an incoming edge of \( T' \), becomes an internal edge in template \( T'' \), which subsumes the induced subgraphs covered by \( T \) and \( T' \); edge \( (T, T') \) is discarded.
If we simply introduced templates along every DFG edge, the set of templates introduced to the DFG would overlap. This covering by templates, however, would be meaningless. Therefore, the enumeration procedure must be organized in a manner to prevent overlapping templates from occurring; moreover, it must also accomplish our primary goal—the identification of redundant structures among a collection of DFGs.

Before a new template can be introduced, it must be classified through an isomorphism test. This requires that the induced subgraph that will be replaced must be isolated, extracted, and tested for isomorphism against the database. Fig. 8 illustrates this process. After pattern extraction, a label $L$ is then assigned to the pattern. If $T$ is the name of the template that is introduced, then $t(T) = L$.

4.4 Maintaining the Acyclicity Property

The introduction of templates to a DFG alters its topology. In order for the resulting DFG to have meaningful semantics, the DFG must remain a DAG in the presence of templates. This is why we restrict all possible patterns to convex subgraphs. As an example, refer back to Fig. 4 (b) in Section 4.1; if new DFG edges are introduced as described in Section 4.2, then this DFG would contain a cycle.

Certain combinations of templates may also introduce cycles to a DFG; these must be avoided at all costs. As an example, consider Fig. 9; various incarnations of this pattern do occur in real-world codes. There are two possible pairs of non-overlapping templates that can arise from this pattern—${(A, C), (B, D)}$ and ${(A, D), (B, C)}$. Fig. 9 shows the former; the latter leads to a similar situation. The only way to rectify this situation is to remove at least one of the offending templates.

Each time a new template is introduced to a DFG, a depth- or breadth-first search is first used to check for a cycle. If a cycle is found, then the offending template is rejected and is not introduced to the graph; otherwise, clustering proceeds as normal.
4.5 Compression Example

The first step of the compression algorithm is to enumerate the set of templates that would result from each DFG edge if it were selected for contraction. As each new template is generated, it is classified, yielding its type. A frequency distribution of edge types is constructed by processing each DFG edge in sequence. This distribution counts the number of edges of each type that occur in the program.

To illustrate (1), edges $A$ and $B$ are incident on the same vertex in Fig. 10 (a); therefore, there is an edge $(A, B)$ in the conflict graph. To illustrate (2), consider edges $A$ and $J$ taken in conjunction with one another. If both of these edges are clustered, the resulting DFG will contain a cycle. Therefore we add an edge $(A, J)$ to the conflict graph.

The maximum independent sets of each conflict graph are shown in bold in Fig. 10 (c) and (d). We used the heuristic in Fig. 1 to compute maximum independent sets. The total number of non-overlapping edges of types $(+, +)$ and $(+, *)$ are now 4 and 1 respectively. Since edge $(+, +)$ occurs with the greatest frequency, edges $B$, $D$, $H$, and $J$ are selected for clustering. The resulting DFG is shown in Fig. 11 (a).

![Figure 9.](image)
The introduction of non-overlapping convex templates to a DFG can create a cycle.

![Figure 10.](image)
A DFG (a) with edge type frequency distribution (b). Interference graphs for each edge type with maximum independent sets shown in bold (c) (d). Type frequency distribution for non-overlapping templates (e).
The edge type frequency distribution is trivial to construct if the DFG has no template—the edge type is given based on the labels of the two vertices. If templates are present, however, then the subgraph that would result from edge contraction must be generated and classified.

As an example, consider the DFG shown in Fig. 10 (a), which contains 10 edges of types (+, +), and 2 of type (+, *), as shown in Fig. 10 (b). One cannot cluster all edges of the same type because the resulting set of templates would overlap. To determine a maximum independent set of non-overlapping edges, we compute conflict graphs for each edge type, as shown in Fig. 10 (c) and (d). The DFG edges labeled A…L in Fig. 10 (a) correspond to the respective conflict vertices in Fig. 10 (c) and (d).

A separate conflict graph is constructed for each edge type. A conflict edge \((X, Y)\), where \(X\) and \(Y\) correspond to edges of the same type in the DFG, is added if either:

1. \(X\) and \(Y\) share an incident vertex, or
2. Clustering both \(X\) and \(Y\) will cause a cycle to occur in the DFG

The next step of the algorithm is to repeat the preceding loop until stopping conditions are met. This time, the pattern enumeration step must combine patterns within templates. The set of patterns enumerated is shown in Fig. 11 (b), along with the type frequency distribution following the construction of the conflict graphs and computation of the independent sets. In this case, the most frequently occurring pattern occurs twice. Once this pattern is selected for clustering, the resulting DFG is shown in Fig. 12.

The process of enumeration, building the conflict graph, computing a maximum independent set, and clustering the respective vertices continues until the most frequently occurring pattern only occurs once. At this point, there is no additional redundancy in the intermediate representation, and the algorithm terminates. In Fig. 12, only three additional patterns can be generated, and all three occur exactly once. Therefore the algorithm terminates at this step.

Figure 11.
The DFG in Fig. 10 (a) after clustering 4 non-overlapping edges of type (+, +) (a). The three (legal) patterns, and the maximum number of non-overlapping instances (b).
4.6 Compression Algorithm

In this section, we present pseudocode for the algorithm described in the preceding section. The algorithm was described and an example was given in the context of a single DFG. The pseudocode, given in Fig. 13, extends the technique to a set of DFGs comprising an application that is being compiled. Specifically, the edge type frequency distributions are computed for the entire program, and separate conflict graphs and independent sets are computed for each edge type for each DFG; the result is the total number of non-overlapping instances of each pattern that occur throughout the entire application.

The algorithm begins by calling Label_Vertices_and_Edges(...). This function assigns integer labels to vertices such that two vertices are assigned equal labels if and only if their opcodes match, and all immediate operands—if any—have the same value. Edges are assigned labels to distinguish whether or not they are the left or right inputs to a commutative operator, as discussed in Section 3.3.

Second, the algorithm enumerates a set of patterns using the function Generate_Edge_Patterns(...). For each edge \(e = (u, v)\), the subgraph \(G_e = \{u, v\}, \{e\}\) is generated as a candidate pattern. Each candidate pattern is assigned a label as described in the previous section.

The next step is to identify the pattern that offers the greatest gain in terms of redundancy. Lines 5-17 of the algorithm accomplish this task. Given a pattern \(p\) and a DFG \(G\), the gain associated with \(p\), denoted gain\((p)\) is the number of subgraphs of \(G\) that can be covered by instances of \(p\) without overlap.

The function Compute_Conflict_Graph(...) creates the conflict graph, and the function Compute_MIS(…) computes its independent set using the algorithm in Fig. 1, which terminates after it undergoes a fixed number (Limit) of iterations without improving the size of the largest MIS. Limit is set to 500 for our experiments.

The cardinality of the MIS is the gain associated with pattern \(p\) for DFG \(G\). This is because each pattern instance combines two nodes and/or patterns into a single pattern, for a net reduction in code size of one. The best gain is computed by summing the gain of each pattern over all DFGs. The pattern with the largest net gain is the best pattern, \(p_{\text{best}}\). Clustering is performed by the function

![Figure 12. The DFG in Fig. 11 (a) after clustering the most frequently occurring template in Fig. 11 (b).](image-url)
Cluster Independent Patterns(...). Once all instances of a given pattern are clustered, we update the set of patterns and adjust their respective gains. Then, we can once again identify the best pattern, and decide whether or not to continue.

Now that an initial set of patterns has been generated, we must update the frequency count for all remaining patterns in G. The function Update_Patterns(...) performs this task. The most favorable pattern is then selected for clustering, and the algorithm repeats again. The algorithm terminates when the best gain is less-than-or-equal-to a user-specific parameter, Threshold; we set the value of Threshold to 1 for our experiments.

Algorithm:  Echo_Instr_Select(G*, Threshold, Limit)

Parameters:  G* := {Gi = (Vi, Ei)} : set of n DFGs
            Threshold, Limit : integer

Variables:  M : mapping from vertices, edges, and patterns to labels
             Pi : set of patterns
             Conflict(Gi, p) : conflict graph
             MIS(Gi, p) : independent set
             gain(p), best_gain : integer
             best_ptrn : pattern (DFG)

1.  For i = 1 to n
2.    Label_Vertices_and_Edges(M, Gi)
3.    Generate_Edge_Patterns(M, Gi)
4.  EndFor
5.  For each pattern p in M
6.    gain(p) := 0
7.    For i := 1 to n
8.        Pi := Generate_Overlapping_Patterns(Gi, p)
9.        If Pi is not empty
10.           Conflict(Gi, p) := Compute_Conflict_Graph(Pi)
11.           MIS(Gi, p) := Compute_MIS(G, Limit)
12.           gain(p) := gain(p) + |MIS(Gi, p)|
13.       EndIf
14.    EndFor
15.  EndFor
16.  best_gain := max{gain(p)}
17.  best_ptrn := p s.t. gain(p) = best_gain
18. While best_gain > Threshold
19.  For i := 1 to n
20.     Cluster_Indep_Patterns(M, Gi, MIS(Gi, best_ptrn))
21.     Update_Patterns(M, Gi, MIS(Gi, best_ptrn))
22.  EndFor
23.  best_gain := max{gain(p)}
24.  best_ptrn := p s.t. gain(p) = best_gain
25. EndWhile

---

Figure 13.

Pseudocode for the compression algorithm described in Section 4.5.
4.7 Compressing and Decompressing the Intermediate Representation

The algorithm presented in the preceding section identifies repeated isomorphic patterns that occur within a collection of DFGs. In this section, we describe the specific steps necessary to compress the program. Each template occurring in each DFG is replaced with a single vertex. The compressed program representation stores only one instance of each template.

Let $N$ be the number of instructions in the program, i.e. the number of vertices in the collection of DFGs. Suppose that template $T_i$ contains $|V_i|$ vertices, and occurs $n_i$ times throughout the program. Each instance of $T_i$ replaces $|V_i|$ vertices with a single vertex. $|V_i|$ vertices are also required to represent the single instance of $T_i$. If $m$ is the number of templates occurring in the program, then $\Delta N$, the reduction in vertices across all DFGs, is given by:

$$\Delta N = \sum_{i=1}^{m} ((|V_i| - 1)n_i - |V_i|) = \sum_{i=1}^{m} [|V_i| (n_i - 1) - n_i]$$

The intermediate representation must explicitly maintain all of the incoming and outgoing edges from each template. These edges are necessary to reconstruct the original DFG during decompression. Since the individual templates throughout the program no longer exist, these edges must be altered to refer to the one instance of every template that occurs in the program. Fig. 14 illustrates this concept.

Three DFGs, $G_1$, $G_2$, and $G_3$, are shown in Fig. 14 (a). Fig. 14 (b) shows them after the application of the pattern generation algorithm. Two uniquely identifiable patterns have been found: $T_1$ and $T_2$. $T_1$ contains 4 vertices and occurs 4 times; $T_2$ contains 2 vertices and occurs twice. Eleven incoming and outgoing edges incident on templates are labeled $e_1...e_{11}$ in Fig. 14 (b). The original DFG contains 22 vertices and 29 edges. The compressed representation contains a total of 14 vertices and 23 edges— reductions of 36% and 21% respectively.

Fig. 14 (c) shows the compressed intermediate representation. The compressed DFGs are showed on the left. Instances of $T_1$ and $T_2$ are shown on the right. The dashed edges shown in Fig. 14 (c) correspond to the incoming and outgoing edges in Fig. 14 (b).

To understand the transformation between Fig. 14 (b) and (c), consider two adjacent templates, $T_x$ and $T_y$, which subsume respective vertices $x$ and $y$. Suppose that there is an edge $(x, y)$, which is an outgoing edge with respect to $T_x$, and an incoming edge with respect to $T_y$. Due to $(x, y)$, an edge $e = (T_x, T_y)$ exists in the compressed program representation.

Let $T_x'$ and $T_y'$ be the single instances of templates of the same respective types as $T_x$ and $T_y$. Note that it is possible that $T_x'$ and $T_y'$ may be the same template if $T_x$ and $T_y$ have the same type. Let $x'$ and $y'$ be vertices in $T_x'$ and $T_y'$ corresponding to $x$ and $y$. Then $(x, y)$ is replaced by an edge $(x', y')$, which is maintained in a list associated with $e$. Upon decompression, $T_x$ and $T_y$ are replaced by the subgraphs they originally subsumed. These subgraphs are copied directly from $T_x'$ and $T_y'$. Each edge $(x', y') \in E$ is replaced with an edge $(x, y)$, where $x$ and $y$ now correspond to $x'$ and $y'$ when $T_x'$ and $T_y'$ are replicated.
Associating (x', y') with e ensures that (x, y) has the correct direction when reintroduced—a problem which arises when t(T') = t(T y'). This pertains specifically to edges e_1 and e_2 in Fig. 14 (b) and (c). Now W.L.O.G., consider a template T and vertex y such that (T, y) is an edge in a DFG G. Suppose that x is a vertex subsumed by T, and that (x, y) is an outgoing edge with respect to T. Suppose that T' is the single instance of the template having the same type as T in the compressed program.
representation, and let $x'$ be the vertex in $T'$ corresponding to $x$. Then $(x, y)$ is replaced with an edge $(x', y)$—literally an edge connecting $T'$ and $G$. To decompress the program, the single vertex representing $T$ is replaced with a copy of pattern $T'$, and edge $(x, y)$ is restored as described above. Ingoing edges with respect to $T$ are handled analogously. In Fig. 14 (c), $e_3$, $e_4$, $e_7$, and $e_8$ are handled in this manner.

5. Experimental Results

5.1 Implementation Details

We integrated the compression algorithm into the Machine SUIF retargetable compiler framework\(^1\). Target description libraries are provided for the Alpha, x86, Pentium Pro, and IA64 architectures. Instruction selection targeted the Alpha architecture in order to eliminate the numerous CVT\(^2\) operations that clutter the target-independent representation. This yielded a more compact representation of the resulting program. Following instruction selection, the intermediate representation was converted to a CDFG, which was then compressed. Our benchmarks are summarized in Section 5.2. Results of the experiment are presented and discussed in Sections 5.3-5.6.

5.2 Benchmarks

To test our compression algorithm, we selected 10 applications from the Mediabench benchmark suite (Lee et al. [1997]), which are summarized in Table 1. The individual source files within each benchmark were linked them together using a pass called *link_sui*. In several cases, this required manual intervention to prevent namespace collisions.

Several applications were written with loops manually unrolled. For reasons that will be discussed in Section 5.4, unrolled loops create significant runtime problems for the compression algorithm. Loop unrolling typically increases code size by replicating the bodies of critical loops. Because our goal is to compress the application, we decided to manually re-roll the loops, yielding a smaller code size. On the other hand, if profile-guided compression (Debray and Evans [2002]) is used to compress only the infrequently executed portions of the program, the unrolled loops could be left intact in the uncompressed portions of the program.

---

\(^1\) http://www.eecs.harvard.edu/hube/research/machsuif.html

\(^2\) type conversion
5.3 Compression Results

Table II shows the results of applying the compression algorithm to all of the benchmarks. The column labeled DFGs lists the number of non-empty DFGs created for each benchmark. The column labeled Patterns lists the total number of uniquely identifiable templates generated for each benchmark by the compression algorithm. The columns labeled Vertices and Edges list the total number of vertices (i.e. machine operations) and edges (dependencies) in each benchmark, before and after the compression algorithm. The columns labeled (%) show the percentage of vertices and edges eliminated by the compression algorithm. If $v$ and $v'$ represent the number of vertices in an application before and after compression respectively, then $\% = (v - v')/v$.

The percentage reduction in vertices ranges from 34.08% (G.721) to 51.90% (JPEG); the percentage reduction in edges ranges from 44.90% (GSM) to 69.18% (PGP) and 69.17% (JPEG). With the exception of benchmarks G.721 and GSM, all benchmarks yielded at least a 43% reduction in vertices and a 61% reduction in edges. The row labeled Total shows to the sum of the number of DFGs, vertices, and edges for the set of benchmarks. Altogether, the percentage reduction in vertices and edges was 45.82% and 64.48% respectively. The number of uniquely identifiable patterns for each benchmark ranged from 162 (G.721) to 2145 (JPEG). This value is not reported as a sum for the Total row because many patterns that occur across multiple application would be counted more than once.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epic</td>
<td>Experimental image compression that avoids floating-point computations.</td>
</tr>
<tr>
<td>GSM</td>
<td>European provisional standard for full-rate speech transcoding.</td>
</tr>
<tr>
<td>JPEG</td>
<td>Lossy compression algorithm for color/gray-scale images.</td>
</tr>
<tr>
<td>MPEG2 Decoder</td>
<td>Digital video decompression using an inverse discrete cosine transform.</td>
</tr>
<tr>
<td>MPEG2 Encoder</td>
<td>Digital video compression using a discrete cosine transform.</td>
</tr>
<tr>
<td>Pegwit</td>
<td>Public key encryption and authentication using elliptic curve cryptography.</td>
</tr>
<tr>
<td>PGP</td>
<td>One-way digital hash function used for computing digital signatures.</td>
</tr>
<tr>
<td>PGP (RSA)</td>
<td>Encryption and key management routines used by PGP.</td>
</tr>
<tr>
<td>Rasta</td>
<td>Speech recognition that handles additive noise and spectral distortion.</td>
</tr>
</tbody>
</table>

Table I. Summary of the Mediabench applications compiled.
**Table II.**
**Summary of the Compression Algorithm**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DFGs</th>
<th>Patterns</th>
<th>Vertices</th>
<th>Edges</th>
<th>Vertices (%)</th>
<th>Edges (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epic</td>
<td>1214</td>
<td>282</td>
<td>13104</td>
<td>7916</td>
<td>7361</td>
<td>2746</td>
</tr>
<tr>
<td>G.721</td>
<td>433</td>
<td>162</td>
<td>4225</td>
<td>2902</td>
<td>2785</td>
<td>1599</td>
</tr>
<tr>
<td>GSM</td>
<td>1564</td>
<td>358</td>
<td>15535</td>
<td>9858</td>
<td>9960</td>
<td>5405</td>
</tr>
<tr>
<td>JPEG</td>
<td>6934</td>
<td>2145</td>
<td>92158</td>
<td>59280</td>
<td>44330</td>
<td>18275</td>
</tr>
<tr>
<td>MPEG2 Decoder</td>
<td>2060</td>
<td>525</td>
<td>20163</td>
<td>12277</td>
<td>11125</td>
<td>4582</td>
</tr>
<tr>
<td>MPEG2 Encoder</td>
<td>2700</td>
<td>645</td>
<td>28764</td>
<td>17886</td>
<td>15431</td>
<td>5995</td>
</tr>
<tr>
<td>Pegwit</td>
<td>1541</td>
<td>430</td>
<td>19842</td>
<td>12957</td>
<td>10110</td>
<td>4200</td>
</tr>
<tr>
<td>PGP</td>
<td>7787</td>
<td>1186</td>
<td>76842</td>
<td>41786</td>
<td>42766</td>
<td>12877</td>
</tr>
<tr>
<td>PGP (RSA)</td>
<td>1064</td>
<td>288</td>
<td>12011</td>
<td>7319</td>
<td>6772</td>
<td>2807</td>
</tr>
<tr>
<td>Rasta</td>
<td>1344</td>
<td>441</td>
<td>16659</td>
<td>10143</td>
<td>9328</td>
<td>3684</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>26670</td>
<td>30577</td>
<td>188139</td>
<td>165655</td>
<td>16619</td>
<td>66819</td>
</tr>
</tbody>
</table>

The 43-52% reduction in the number of vertices for the eight best benchmarks is comparable to the results reported for the BRISC program format (Ernst et al. [1997]), which achieved unprecedented compression at the time. The two formats are notably different, however—BRISC has been designed to be directly executable, whereas our intermediate representation is not. Additionally, BRISC encodes operands for each instruction, whereas our approach encodes DFG edges instead. In the future, we intend to use the compression algorithm as part of an optimizing back-end that targets dictionary compression technologies such as CALD instructions (Liao et al. [1999]), echo instructions (Fraser [2002], Lau et al. [2003]), and DISE (Corliss et al. [2003]), none of which offered compression ratios comparable to BRISC. The results in Table II suggest that this approach could be quite effective.

The compressed program representation maintains two sets of vertices. The first set consists of all vertices in the DFGs—namely templates, and all vertices in the original DFGs that were not subsumed by templates. The second set of vertices correspond to operations that arise in the single instance of each uniquely identifiable template that must be maintained by the compressed representation. We refer to these two sets as *DFG vertices* and *Template vertices* respectively.

Fig. 15 shows the distribution of DFG and template vertices in the compressed program representation for the 10 benchmarks. The number of DFG and template vertices is presented as percentages of all vertices in the uncompressed program. For DFG vertices, this percentage ranges from approximately 35% (GSM) to 57% (G.721); for template vertices, this percentage is less than 11% for all 10 benchmarks. In an earlier experiment, we had compiled GSM with loops unrolled. In this case, the percentage of template vertices rose to 33%; however, the compressed program size was significantly larger than the results in Table II. Motivated by this, we have developed techniques for compressing the set of representative instances of each template using the subgraph relation (Brisk et al. [2005]).
In the compressed program representation, edges can be classified into four types. **DFG edges** are those that occur within the set of DFGs following compression. **Template edges** are edges in the DFG representation of each instance of each uniquely identifiable template that occurs in the compressed program. **Template boundary edges** represent dependencies between templates and non-template vertices in the DFG. **Inter-template boundary edges** represent dependencies between two distinct template instances in the compressed program. As an example, refer to Fig. 14. The DFG edges are the unlabeled edges in $G_1$, $G_2$, and $G_3$. Template edges are the unlabeled edges in templates $T_1$ and $T_2$. Edges $e_3$, $e_4$, $e_7$, and $e_8$ are template boundary edges, and $e_1$, $e_2$, $e_5$, $e_6$, $e_9$, $e_{10}$, and $e_{11}$ are inter-template boundary edges.

Fig. 16 shows the distribution of these four classes of edges as percentages of the number of edges in the uncompressed program for the 10 benchmarks. For all benchmarks other than GSM, DFG edges accounted for approximately one-half of all edges in the compressed program. For GSM, the percentages of DFG and template edges are approximately equal; however, collectively they represent 71% of the edges in the compressed program. The relative contributions of template and template boundary edges vary from benchmark to benchmark. In all benchmarks, the contribution of inter-template boundary edges was at most 13.3% (JPEG); the contribution relative to the number of edges in the uncompressed program was 5% (G.721) or less.

**Figure 15.**

The percentage of operations classified as DFG and Template vertices in the compressed program representation.
3.0 Runtime Analysis

Table III shows the runtime of the algorithm on the 10 benchmarks. The experiments were performed on a 3.00GHz Intel Pentium 4 Processor running MEPIS Linux\(^3\). The processor contained 1 GB of memory, a 12K trace cache, 8k L1 data cache, and 512k L2 cache.

Three quantities are shown in Table III. The columns labeled *Total Time* list the runtime of the complete algorithm. The columns labeled *Isomorphism* list the amount of time spent performing isomorphism testing using VF2 (Cordella et al. [2004]), an exact algorithm which possesses an exponential worst-case time complexity. The columns labeled (%) list the percentage of the time spent on isomorphism testing.

---

\(^3\) http://www.mepis.org
The runtime of the compression algorithm ranged from 2.51 seconds (G.721) to 5 minutes and 46.2 seconds (JPEG). The smaller applications compiled faster than larger ones, which is to be expected. Isomorphism testing consumes between 16.01% (GSM) and 42.97% (PGP) of total execution time of all benchmarks. Since VF2 runs in exponential worst-case time, our primary concern was that isomorphism testing for large DFGs would dominate the overall runtime of the algorithm. For these small embedded benchmarks, our fears were allayed; however, this was only possible because of our decision to roll all of the loops we encountered.

Consider a loop $L$ whose body is a basic block, $B$. If $L$ is unrolled by a factor of $N$, then $B$ will be replicated and concatenated $K$ times. Let $B[1..N]$ represent the unrolled loop, where $B[i]$ is the $i$th copy of $B$. This causes significant runtime problems for the isomorphism algorithm. First of all, note that $B[i]$ is isomorphic to $B[j]$. In general, any sequence $B[i..i+k-1]$ containing $k$ contiguous copies of $B$ in the unrolled loop will be isomorphic to sequence $B[j..j+k-1]$. Now assume that $N$ is even. Then the largest possible identical isomorphic DFGs in $B[1..K]$ that could be generated by the isomorphism algorithm would be $B[1..N/2]$ and $B[N/2+1..N]$. As $N$ grows large, the cost of testing these two subgraphs for isomorphism against one another will increase accordingly; moreover, the number of isomorphism tests (all returning true) required to generate these two patterns increases with $N$ as well. Unrolled loops of significant size occur in several Mediabench applications, including PGP and GSM. The compilation time in these cases was in excess of several hours.

The purpose of compression is to reduce code size. Since unrolling loops increases code size to begin with, then it would be seemingly pointless to unroll the loops and then compress them. If a loop is unrolled to enhance the performance of an algorithm, then compression should only be applied to infrequently executed portions of the program, which was precisely the point of profile-guided compression (Debray and Evans [2002]). The purpose of this work is to compress an intermediate representation which can later be optimized for performance. Rolling the loops yielded both the smallest program size and the fastest compression time.

5.5 Scalability Analysis

Thus far, the compression algorithm has only been tested on small benchmarks whose compilation times range from seconds to minutes. In this section, we perform several experiments to attempt to determine whether this algorithm will scale for larger benchmarks. In addition to the benchmarks listed in Table I, we compiled Mesa, an OpenGL graphics library clone, also from Mediabench. The preprocessing stages for Mesa were considerably greater than for the other benchmarks; for example, the $\text{link\_sui}\bar{f}$ pass required 8-9 hours to complete following our manual intervention to prevent namespace collisions.

Table IV presents a summary of the compression algorithm applied to Mesa. The percentage reduction of both vertices and edges is considerably larger than any of the 10 benchmarks in Table I. In general, larger programs will contain more redundancy.
The primary concern here is not the quality of compression but instead the runtime of the compression algorithm. Fig. 17 plots compilation time as a function of the number of operations in the program. We used least-squares linear and quadratic regression to approximate linear and quadratic relationships between the 11 data points.

The line resulting from the linear regression is expressed in slope-intercept form, $y = Mx + B$, where the $Y$-axis represents time (in seconds) and the $X$-axis represents program size (in terms of the number of DFG nodes). The regression yielded slope $M = 0.0118$ and $y$-intercept $B = -258.38$. The correlation coefficient, $R^2$, which gives the quality of the linear regression, was 0.9236. Unfortunately, the largest deviation from this regression line occurred in the three largest benchmarks. The linear correlation between program size and the runtime of the algorithm is marginal, at best.

Next, we applied quadratic regression to fit the set of data points to a second-degree polynomial. The resulting curve has the form $y = Ax^2 + Bx + C$, where $A = 0.00000005$, $B = -0.0015$, and $C = 30.074$. For this curve, $R^2 = 0.9996$, a near-perfect correlation. Moreover, the error occurring at the larger
benchmarks was much less than for linear regression. This is fairly strong empirical evidence of a quadratic correlation between program size and runtime for a set of embedded benchmarks; however, 11 data points is far too few to draw wide-reaching conclusions of this sort.

Obviously, code size and compilation time will not always correlate. Consider, for example, two programs of equal size. The first has been written with loops manually unrolled; the second contains no unrolled loops. For the reasons described earlier, the compilation time of the first will be significantly greater than that of the second. Most importantly, we have only compiled 11 benchmarks in this study, and these are by no means representative of all applications, embedded or otherwise.

Given that isomorphism testing runs in worst-case exponential time, a primary concern was that a significant proportion of time would be spent testing large DFGs for isomorphism. As it turns out, this was not the case. Fig. 18 shows the distribution of time spent performing isomorphism testing on DFGs of varying sizes for the Mesa, JPEG, and PGP benchmarks—our three largest. For all three benchmarks, at least 48% of all time spent on isomorphism testing was for DFGs of size 2; and at least 95% of this time was spent on DFGs from size 2-6. For the benchmarks we studied, individual isomorphism tests of large DFGs consumed an insignificant amount of time.

The largest DFGs tested for isomorphism in each of these three benchmarks contained 136 vertices (Mesa—0.001645 seconds), 82 vertices (JPEG—0.000516 seconds), and 73 vertices (PGP—0.000227 seconds). The runtime of these individual tests was negligible compared to the overall time spent on isomorphism testing.

![Cumulative Distribution of Time Spent on Isomorphism Testing](image)

**Figure 18.**

Cumulative frequency distribution of the time spent on isomorphism testing.
5.5 Isomorphism vs. Linear Matching Techniques

In this section, we attempt to numerically justify our decision to detect recurring computational patterns using DAG isomorphism rather than a linear matching technique. Note that string matching is impossible at intermediate stages of compilation. The reason is that for two instructions to be mapped to the same character, both opcode and all operands must be the same. Prior to register allocation, the compiler assumes that the number of available registers in the target architecture is infinite. Consequently, identical instructions sequences are unlikely to occur within the program.

A more appropriate linear matching technique is operand specialization (Fraser and Proebsting [1995], Proebsting [1995], Ernst et al. [1997]) and factorization (Araujo et al. [1998]). These techniques simply abstract away all register names; they do not determine whether two computations that have been “specialized” are actually identical. Moreover, these techniques do not employ scheduling optimizations that rearrange computations locally in order to enhance the quality of compression. These techniques do, however, construct repeated sequences of frequently occurring “factorized” instructions in the program.

Let us consider some template $T$ identified by the compression algorithm described in this paper. Let us assume that if the operations that comprise $T$ occur contiguously within $T$’s basic block, then the specialization/factorization technique will detect the redundancy. Isomorphism, on the other hand, can detect repeated patterns that do not occur contiguously within each basic block. The result of this experiment is summarized in Fig. 19.

Let $T = \{T_1, T_2, \ldots, T_n\}$ be the set of templates that occur throughout program $P$. Let $|T_i|$ be the number of operations (vertices) contained in template $T_i$. Let $T_{\text{Contig}} = \{T_i \in T \mid T_i \text{ occurs contiguously in } P\}$. We report two quantities for each benchmark, $U_{\text{Contig}}$ and $W_{\text{Contig}}$, the unweighted and weighted percentage of patterns that occur contiguously in $P$. These quantities are computed as follows:

![Figure 19. Cumulative frequency distribution of the time spent on isomorphism testing.](image-url)
\[
U_{\text{Contig}} = \frac{|T_{\text{Contig}}|}{|T|}
\]
(4)

\[
W_{\text{Contig}} = \frac{\sum_{T_i \in T_{\text{Contig}}} |T_i|}{\sum_{T_i \in T} |T_i|}
\]
(5)

\(W_{\text{Contig}}\) simply weighs the average by the number of operations in each template.

Fig. 20 shows \(U_{\text{Contig}}\) and \(W_{\text{Contig}}\) for our initial set of 10 benchmarks (excluding Mesa). For all benchmarks, \(U_{\text{Contig}} > W_{\text{Contig}}\), indicating that smaller patterns are more likely to occur contiguously than larger ones.

For all benchmarks, at least 60% of all patterns occur contiguously. Therefore, at most 40% of all patterns occurring in these benchmarks could be detected by isomorphism, but not linear matching. Of course, these patterns could be identified if rescheduling was used in conjunction with linear matching. Both scheduling and isomorphism, however, are classically hard problems.

This result contradicts an observation by De Sutter et al. [2002], who found that local rescheduling within basic blocks did not improve the quality of their compression. De Sutter’s compression step, however, was performed at link-time, after the initial compiler performed scheduling as an optimization step—most likely using a sub-optimal, deterministic heuristic. This may account for the uniformity among schedules of similar operation sequences. Secondly, De Sutter’s benchmarks were written in C++, and made considerable use of object-oriented programming facilities such as template instantiation and inheritance. These features are likely to yield redundant code sequences that would not occur in the Mediabench applications, which were written in C.

Conclusion

A compression algorithm applicable to a program represented as a CDFG has been presented. The compression algorithm repeatedly generates and classifies patterns observed in the CDFG using a technique called edge contraction. This approach is iterative, in that smaller patterns are first generated, and then combined into larger patterns. To accomplish this task, the algorithm must solve two classically hard problems—maximum independent set and graph isomorphism. To solve both of these problems, we leveraged existing software solutions that have been published elsewhere. The compression algorithm has been integrated into a well-established retargetable compiler framework.

The algorithm has been shown to effectively remove up to 51.90% of vertices and 69.17% of edges from the CDFG representation of a set of 10 small embedded benchmarks within minutes. When a
larger application was compiled, significantly greater compression was achieved at the expense of considerable runtime (approximately 50 minutes). Despite the fact that the algorithm runs in exponential worst-case time due to repeated calls to an exact isomorphism algorithm, a strong quadratic correlation between program size and runtime was observed for these applications.

In the future, we intend to use the compression algorithm presented here as part of a larger optimizing framework that targets architectures with features such as Echo/CALD instructions or DISE decompression. The compression framework will use the algorithm presented in this paper to identify redundant subgraphs in the program; register allocation and scheduling will take these subgraphs into account in order to preserve the redundancy as registers are assigned and a final schedule is built. This process will effectively translate isomorphic subgraphs prior to code generation into identical instruction sequences in the final assembly program. Once this is completed, substituting Echo or CALD instructions for identical code sequences is trivial.

REFERENCES


