An Intermediate Representation for the Exploitation of Instruction Level Parallelism in Embedded Synthesis and VLIW Compilation Environments

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

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

2. kastner@ece.ucsb.edu
   Department of Electrical Engineering
   University of California, Santa Barbara

Abstract

This paper introduces the All-Pairs Common Slack Graph (APCSG), an intermediate representation of the instruction level parallelism that exists within a computation. The APCSG is intended for use in high level synthesis systems and compilers that target VLIW architectures. To exploit the benefits of the APCSG, we have developed the Parallel Template Generation Algorithm, a general purpose framework for compilation and synthesis optimizations. The efficacy of the framework is established by applying it to four problems at the forefront of compilation and synthesis research: (1) global resource allocation during CDFG synthesis; (2) instruction scheduling for VLIW architectures; (3) generation application-specific VLIW instruction set extensions; and (4) code compression for VLIW architectures.

I. INTRODUCTION

Both behavioral synthesis frameworks and compilers that target VLIW architectures must identify, extract, and exploit instruction level parallelism (ILP) to optimize the performance of an application. To aid in the development of such tools, we introduce a data structure, the \textit{All-Pairs Common Slack Graph (APCSG)}, which serves as an intermediate representation of the parallelism within an application. The APCSG contains edges between operations within an application that may be scheduled concurrently. A framework called the Parallel Template Generation Algorithm (PTGA) is applied to four problems in order to demonstrate the versatility of the APCSG. The problems are:
(1) Global resource allocation for behavioral synthesis
(2) Instruction scheduling for Very Long Instruction Word (VLIW) architectures
(3) The generation of application-specific VLIW instruction sets for extensible processors
(4) Code compression for VLIW architectures

FPGA technology has enabled reconfigurable computing, a new paradigm where hardware designs are programmed onto a chip with the same ease as software programs are currently loaded into memory prior to execution. System designers require tools that automatically translate embedded applications into well-optimized hardware. The extraction of ILP from within the application is paramount to accelerating the performance of such designs. The allocation of computational resources is one of the most important tasks during synthesis. The PTGA can be tuned and optimized in order to perform the allocation task.

The majority of processors designed for the embedded and signal processing markets are based on VLIW architectures. To performance of the application on a VLIW architecture is highly dependent on the compiler. One of the most important phases of compilation is scheduling, which determines the time step at which each operation in the program executes. The PTGA can easily be adapted to perform instruction scheduling for such compilers.

Another recent trend is the emergence of extensible processors for embedded markets. An extensible processor contains a Turing-complete RISC core coupled with a co-processor that performs user-specifiable application-specific functionality. The core is typically configurable; in other words the user may specify such parameters as cache size and the number and bitwidth of registers on chip prior to synthesis. Extending a RISC instruction set is a simpler than designing an application-specific instruction set processor (ASIP) from scratch. Automating the extension process requires an analysis of the parallelism that exists within an application in order to offer performance comparable to custom hardware designs. The PTGA is an ideal framework for the generation of application-specific VLIW instruction set extensions.

Hardware supported decompression reduces the cost of storing a program on-chip, and has been shown to reduce energy consumption and sometimes even increase performance due to reduced I-cache miss rates. A recent paper by Brisk [2004] has developed a compression technique in a framework that is in many ways quite similar to the PTGA. The PTGA can be used to perform instruction scheduling for VLIW...
architectures in such a manner that similar sets of operations are often grouped together and issued concurrently as a *fetch packet*. Increasing uniformity among fetch packets in the program increases the quality of compression achieved when statistical coding methods are used to reduce the number of bits used to encode the opcode sequence of the final program.

The paper is organized as follows. Section II introduces the APCSG and the PTGA. Section III discusses the application of the PTGA to the four problems mentioned earlier. Our experimental methodology is summarized and results are presented in Section IV. Related work is discussed in Section V. Section VI concludes the paper.

II. **THE ALL-PAIRS COMMON SLACK GRAPH AND THE PARALLEL TEMPLATE GENERATION ALGORITHM**

This Section presents the All Pairs Common Slack Graph (APCSG), an intermediate representation for the ILP occurring within an application, and the Parallel Template Generation Algorithm (PTGA), a framework that is applied in Section III to four problems at the forefront of research in compilation and synthesis. Sections A and B describe the construction of the APCSG from an established intermediate representation. Sections C and D describe the PTGA and its component operations in detail.

### A. The Control Data Flow Graph Intermediate Representation

An acyclic computation, or *basic block*, is represented as a *Data Flow Graph (DFG)*, from which an APCSG is built. A DFG is a *Directed Acyclic Graph (DAG)*. DFG vertices represent computational operations, and edges represent dependencies between operations. An application is generally not acyclic, and is represented by a *Control Data Flow Graph (CDFG)*. A CDFG contains a *Control Flow Graph (CFG)*, where vertices are basic blocks and edges are control dependencies between blocks.

Let \( G = (V, E) \) be a DFG. Each vertex \( v \in V \) is assigned an integer type, \( t(v) \), in accordance with \( v \)'s operation; a type, in this context is similar to an opcode in assembly. Edge \( e = (u, v) \in E \) has an integer type \( t(e) = (t(u), t(v)) \). Edge types can be mapped to the set of integers via Cantor's diagonalization proof. Since DFG edges are directed:

\[
(t(u), t(v)) = (t(v), t(u)) \iff t(u) = t(v)
\]  

(1)
The level assigned to vertex by a latency constrained scheduling heuristic $\chi$ is denoted $L_\chi(v)$. Here, we are concerned only with the As-Soon-As-Possible (ASAP) and the As-Late-As-Possible (ALAP) heuristics.

A source is a vertex that has no predecessors; a sink is a vertex that has no successors. Every DAG must have at least one source and at least one sink. For every source $s$ and non-source vertex $v$:

$$L_{\text{ASAP}}(s) = 1$$  \hspace{1cm} (2)

$$L_{\text{ASAP}}(v) = \max\{L_{\text{ASAP}}(u) + 1 \mid (u, v) \in E\}$$  \hspace{1cm} (3)

The Critical Path Length, denoted CP, is defined as follows:

$$CP = \max_{v \in V}\{L_{\text{ASAP}}(v)\}$$  \hspace{1cm} (4)

For every sink $t$ and non-sink vertex $u$:

$$L_{\text{ALAP}}(t) = CP$$  \hspace{1cm} (5)

$$L_{\text{ALAP}}(u) = \min\{L_{\text{ALAP}}(v) - 1 \mid (u, v) \in E\}$$  \hspace{1cm} (6)

The Slack of vertex $v$, denoted Slack($v$), is defined in terms of $L_{\text{ASAP}}(v)$ and $L_{\text{ALAP}}(v)$:

$$\text{Slack}(v) = L_{\text{ALAP}}(v) - L_{\text{ASAP}}(v) \geq 0$$  \hspace{1cm} (7)

A critical vertex $c$ satisfies Slack($c$) = 0. Any source-to-sink path from consisting of only critical vertices is a critical path. Every DFG must have at least one critical path. As an example, consider the DFG in Fig. 1 (a) with levels assigned by the ASAP and ALAP heuristics for each vertex in Fig. 1 (b). From these values, the slack for each vertex is computed using Eq. 7. This DFG contains five critical vertices: A, C, F, H, and I. Vertices B, D, E, and G all have slack equal to 1.

![DFG diagram](a)  \hspace{1cm} ![DFG diagram](b)  \hspace{1cm} ![DFG diagram](c)

Fig. 1. A DFG (a), level assignments for all vertices by the ASAP and ALAP heuristics (b), and the APCSG (c).
B. The All-Pairs Common Slack Graph

The common slack between two vertices $u$ and $v$, denoted $\text{CSlack}(u, v)$, is the number of levels at which $u$ and $v$ may be scheduled in parallel by a latency-constrained scheduling heuristic. The APCSG is a weighted, undirected graph that is derived from a DFG. An APCSG has a vertex corresponding to every vertex in the DFG, while APCSG edges represent pairs of operations that can be scheduled concurrently. Specifically, edge $e = (u, v)$ has weight $w(e) = \text{CSlack}(u, v)$. In summary, there is an edge for every pair of vertices that could be scheduled concurrently. Conceptually, each edge represents one unit of potential ILP between a pair of operations. The ILP is only realized if the operations are scheduled concurrently.

The APCSG corresponding to the DFG in Fig. 1 (a) is shown in Fig. 1 (c). The common slack between every pair of vertices in the DFG is at most 1; therefore all APCSG edges have weight 1. For example, there is an APCSG edge $(A, B)$ because both $A$ and $B$ could be scheduled concurrently at level 1. If there is a path between from vertex $u$ to $v$ or $v$ to $u$, there should be no APCSG edge between regardless of their ASAP and ALAP levels. This is easy to verify for every pair of non-adjacent APCSG vertices in Fig. 1 (c).

To determine if there is a path between every pair of vertices, we compute the Transitive Closure of $G$, $G_{TC} = (V_{TC}, E_{TC})$; $V_{TC} = V$, and edge $e = (u, v) \in E_{TC}$ if there is a path from $u$ to $v$ in $G$. Since $G$ is a DAG, $E_{TC}$ cannot contain $(u, v)$ and $(v, u)$. The time complexity of computing $G_{TC}$ for a DAG is $O(VE)$ (Cormen et al. [2001]). Fig. 2 shows the transitive closure of the DFG in Fig. 1 (a).

Fig. 3 shows pseudocode for the computation of common slack between a pair of vertices $u$ and $v$. Line 3 tests to see whether or not there is a path from $u$ to $v$ or vice versa, which uses the transitive closure, $G_{TC}$. Using the transitive closure, the time complexity for computing common slack is $O(1)$.
CSlack( Vertex : u, Vertex : v, Trans_Closure : G_{TC} = (V_{TC}, E_{TC})) : Integer
1. If L_{ASAP}(v) < L_{ASAP}(u)
2. Swap u and v
3. If (u, v) \in E_{TC} or (v, u) \in E_{TC} or L_{ALAP}(u) < L_{ASAP}(v)
4. Then Return 0
5. Else If L_{ALAP}(u) < L_{ALAP}(v)
6. Then Return 1 + L_{ALAP}(u) - L_{ASAP}(v)
7. Else
8. Then Return Slack(v) + 1

Fig. 3. Computation of Common Slack

![Fig. 4. Illustration of Common Slack. (a), (b), and (c) refer to lines 3 (third condition) 5, and 7 of Fig. 3.](image)

Fig. 4 illustrates common slack, assuming there is no path between vertices u and v such that L_{ASAP}(u) \leq L_{ASAP}(v). Fig. 4 (a) shows the case where u and v have no common slack. Fig. 4 (b) and (c) show cases where u and v have positive common slack. Fig. 4 (a), (b), and (c) correspond to the conditions in lines 3 (last condition), 5, and 7 of the pseudocode in Fig. 3 respectively.

To illustrate the necessity of the transitive closure, observe that L_{ALAP}(D) = L_{ASAP}(G) = 3. Assuming there is no path between u and v, the last condition in line 3 and the condition in line 5 will evaluate to true. Line 6 then sets CSlack(D, G) to 1 since L_{ALAP}(D) = L_{ASAP}(G) = 3. The transitive closure prevents this from occurring.

Given a DFG G = (V, E), we construct an APCSG G* = (V*, E*, w), where E* is a set of undirected edges, and w is a function that assigns integer weights to edges in E*. Initially, let V* = V. Next, consider every pair of vertices u, v \in V. An undirected edge e = (u, v) is added to E* if CSlack(u, v) > 0, and w(e) = CSlack(u, v). In other words, the weight of each APCSG edge is equal to the common slack between the pair of vertices in the APCSG. If CSlack(u, v) = 0, then no edge adjoining u and v is added to the APCSG. Pseudocode is given in Fig. 5.
Build_APCSG( DFG : G = (V, E),
          Trans_Closure : G_{TC} = (V_{TC}, E_{TC}), ) : APCSG
1. Set of Vertices : V* ← V
2. Set of APCSG Edges : E* ← φ
3. Weight Function : w : E* → \{1, 2, ..., V\}
4. cslack : Integer
5. For every pair of vertices u, v ∈ V
6.   cslack ← CSliacck(u, v, G_{TC})
7. If cslack > 0
8.   APCSG Edge : e = (u, v)
9.   E* ← E* ∪ \{e\}
10. w(e) ← cslack
11. Return G* = (V*, E*, w)

Fig. 5. Algorithm to construct an APCSG from a DFG

We assign integer types to APCSG edges. Two orientations (u, v) and (v, u) exist for every undirected edge e, yielding two possible edge types, (t(u), t(v)) and (t(v), t(u)). For example, if t(u) = 1 and t(v) = 2, then it is not clear if t(e) = (1, 2) or t(e) = (2, 1). Edges e and e’ where t(e) = (1, 2) and t(e’) = (2, 1) should be assigned the same integer type. To eliminate this dichotomy, we assign edge types as follows:

\[
t(e) = (\min\{t(u), t(v)\}, \max\{t(u), t(v)\})
\]

(8)

C. Parallel Templates and Clustering

The APCSG represents the ILP within a DFG. An important applications that may use the APCSG is scheduling the DFG on a resources. First, we present establish that a clique within the APCSG corresponds to a set of operations that can be scheduled concurrently without violating precedence constraints. Then we introduce parallel templates, which represent the decision to schedule a subset of operations concurrently. Finally, we discuss clustering, a transformation that introduces a parallel template into a DFG, transitive closure, and APCSG.

A clique C is defined to be a complete subgraph of an undirected graph. In other words, if c₁, c₂ are vertices in C, then there exists an edge (c₁, c₂) in the graph. Cliques in the APCSG represent a subset of vertices that can be scheduled at the same level, as proven by the following theorem.

Theorem 1. Let G = (V, E) be a DFG and let G* = (V*, E*, w) be its APCSG. Let \( V' \subseteq V \) be a subset of vertices. Then the vertices in \( V' \) can all be scheduled at level L in G if and only if \( V' \) is a clique in \( G^* \).
Proof. Suppose the vertices in $V'$ can be scheduled at level $L$ in $G$. This can occur only if there are no paths between the vertices in $V'$; the first two conditions in line 3 of Fig. 3 evaluate to false. Since all vertices in $V'$ can be scheduled at level $L$, condition $L_{\text{ALAP}}(u) < L_{\text{ASAP}}(v)$ in line 3 will also be false for every pair of vertices. This ensures positive common slack between every pair of vertices in $V'$, so $V'$ is a clique in $G^*$.

Now, suppose that $V'$ is a clique in $G^*$ and assume to the contrary that there exists at least one vertex $v \in V'$ that cannot be scheduled at level $L$. We must consider two cases:

1. $L_{\text{ASAP}}(v) < L_{\text{ALAP}}(v) < L$
2. $L < L_{\text{ASAP}}(v) < L_{\text{ALAP}}(v)$

If case (1) occurs, consider vertex $w \in V' \ni L_{\text{ASAP}}(w) = L$. Then $L_{\text{ALAP}}(v) < L_{\text{ASAP}}(w)$, $\text{CSlag}(v, w) = 0$, and $(v, w) \not\in E^*$. If case (2) occurs, consider vertex $u \in V' \ni L_{\text{ALAP}}(u) = L$. Then $L_{\text{ALAP}}(u) < L_{\text{ASAP}}(v)$, $\text{CSlag}(u, v) = 0$, and $(v, w) \not\in E^*$. A contradiction arises due to the fact that $V'$ is a clique.

A legal schedule cannot be obtained for $G$ by partitioning $G^*$ into disjoint cliques, where each clique in $G^*$ corresponds to a subset of vertices that will be scheduled on the same level. As a counterexample, consider, DFG $G$ in Fig. 1 (a) and APCSG $G^*$ in Fig. 1 (c). Suppose that the clique partition contains $\{D, F\}$ and $\{E, G\}$. From Fig. 1 (a), $\{D, F\}$ and $\{E, G\}$ cannot both be assigned to the same level due to data dependencies. This demonstrates that a clique cover in $G^*$ is not always a legal schedule of $G$. This situation is rectified if scheduling decisions are made one clique at a time. To represent such a decision, the subset of vertices scheduled concurrently is collapsed into a single vertex called a Parallel Template. Clustering, introducing the parallel template to the DFG, transitive closure, and APCSG, is discussed next.

Suppose, for example, suppose that after analyzing the DFG and APCSG shown in Fig. 1, we decide to cluster vertices $E, F,$ and $G$ together. Let $T$ be the parallel template representing $E, F,$ and $G$. To ensure that all precedence constraints are satisfied, every incoming edge to $E, F,$ and $G$ is replaced with an incoming edge to $T$; similarly, every outgoing edge from $E, F,$ and $G$ is replaced with an outgoing edge to $T$. For example, the incoming edges to $T$, based on Fig. 1 (a), would be $(A, T), (B, T), (C, T),$ and $(D, T)$; the outgoing edges would be $(T, H)$ and $(T, I)$. 
Fig. 6. The DFG in Fig. 1 (a) after clustering E, F, and G (a), updated transitive closure (b), and updated APCSG (c).

Fig. 7. Three examples illustrating the clustering process between two vertices (a), one vertex and one parallel template (b), and two parallel templates (c). This is easily generalized to sets of more than two vertices.

Fig. 6 (a) shows the DFG from Fig. 1 (a) after clustering E, F, and G; parallel template T subsumes E, F, and G. The original DFG edges incident on E, F and G (shown as dashed lines) have been removed from the DFG, but must be maintained to preserve the original data dependencies. New edges have been introduced that are incident on T(bold lines). T stores a list of the vertices that it subsumes. Clustering may introduce indirect dependencies between otherwise independent vertices. For example, there is a path from D to node H in Fig. 6 (a), whereas there is no path between them in Fig. 1 (a). Clustering may also reduce common slack between pairs of vertices; therefore APCSG edge weights must be updated too. If clustering reduces the common slack between a pair of vertices to zero, then the APCSG edge between them is removed. Fig 6 (b) shows the transitive closure of the DFG in Fig. 6 (a); Fig. 6 (c) shows the APCSG.

Next, we address how to cluster parallel templates with one another and with other DFG vertices. Based on a discussion by Kastner et al. [2002], we decided against a hierarchical representation of templates. Instead, clustering two parallel templates merges them into a single template. The three cases for clustering (two DFG vertices, one DFG vertex and one parallel template, and two parallel templates) are illustrated in Fig. 7 (a), (b), and (c) respectively.
All vertices (including templates) store lists of their predecessors and successor, which may be accessed via the fields T.predecessors and T.successors, for example; The corresponding edges may be accessed via T.incoming_edges and T.outgoing_edges. We assume that adding vertex u to T.predecessors automatically adds edge (u, T) to T.incoming_edges. Parallel templates maintain a list of internal vertices.

Fig. 8 gives pseudocode for clustering for a DFG. The parameters are a DFG G = (V, E) and a subset of vertices S to be clustered. A parallel template T is initialized in lines 1-4. A set of edges, E_S, the set of edges incident on vertices in S, is initialized to the empty set in line 5. Since all vertices in S will be removed from the DFG, all edges in E_S must also be removed. The loop spanning lines 6-13 traverses every vertex s in S, and updates T and E_S; V and E are updated in lines 14-15 respectively.

Fig. 9 gives pseudocode for clustering the transitive closure once the DFG has been clustered. The first step is to apply the clustering procedure (Fig. 8) to the transitive closure as well; this alone, however, is not enough. As an example, suppose that we clustered vertices E, F, and G in the transitive graph in Fig. 2. The resulting graph would contain all edges in Fig. 6 (b), except for edge (D, H). This occurs because the transitive closure in Fig. 2 contains edges (D, G), (E, H), and (F, H). After clustering E, F, and H, these edges are replaced with (D, T), (E, T) and (T, H). Due to clustering, a new path from D to H has been introduced to the DFG. Consequently, we must compare every pair of vertices u and v (other than T) in the transitive graph, and check to see if the graph contains a path <u, T, v> or <v, T, u>. If so, an edge (u, v) or (v, u) must be added to the transitive closure; otherwise, no additional edge between u and v is necessary. If an edge between u and v already exists, then there is no need to check for a path between them.

Finally, we must apply clustering to the APCSG following the DFG and transitive closure. Clustering cannot increase the slack of any vertex or increase the common slack between a pair of vertices. In fact, clustering may reduce the slack of individual vertices within the DFG, which in turn may reduce the common slack between other vertices in the DFG. Additionally, data dependencies may be introduced between vertices that were once adjacent in the APCSG. This requires the reduction of certain weights in the APCSG, and the removal of others. Pseudocode for clustering an APCSG is shown in Fig. 10.

The input parameters are a DFG G, its transitive closure G_{TC}, APCSG G*, and a subset of vertices S that has already been clustered in the DFG and transitive closure. The transitive closure is required for the computation of common slack in line 11.
Cluster_DFG( DFG : G = (V, E), Set of Vertices : S ⊆ V )
1. Parallel Template : T
2. T.predecessors ← T.incoming_edges ← φ
3. T.successors ← T.outgoing_edges ← φ
4. T.internal_verts ← φ
5. Set of Edges : E_S ← φ
6. For each vertex s ∈ S
7. If s is a template
8. T.internal_verts ← T.internal_verts ∪ s.internal_verts
9. Else
10. T.internal_verts ← T.internal_verts ∪ {s}
11. T.predecessors ← T.predecessors ∪ s.predecessors
12. T.successors ← T.successors ∪ s.successors
13. E_S ← E_S ∪ (s.incoming_edges ∪ s.outgoing_edges)
14. V ← (V – S) ∪ {T}
15. E ← (E – E_S) ∪ T.incoming_edges ∪ T.outgoing_edges

Fig. 8. Clustering Algorithm for DFGs

Cluster_TC( Trans_Closure : G_TC = (V_TC, E_TC), Set of Vertices : S ⊆ V )
1. Cluster_DFG( G_TC, S )
2. Let T ∈ V_TC be the template introduced by clustering.
3. For every pair of vertices u, v ∈ V_TC – {T}
4. If ∃ (u, T), (T, v) ∈ E_TC
5. E_TC ← E_TC ∪ {(u, v)}
6. Else if ∃ (v, T), (T, u) ∈ E_TC
7. E_TC ← E_TC ∪ {(v, u)}

Fig. 9. Clustering Algorithm for the transitive closure

Cluster_APCSG( DFG : G = (V, E), Trans_Closure : G_TC = (V_TC, E_TC), APCS : G* = (V*, E*, w), Set of Vertices : S ⊆ V )
1. Let T ∈ V be the template introduced by clustering.
2. Set of Edges : E_new ← φ
3. For every vertex s ∈ S
4. For every vertex v adjacent to s
5. E* ← E* – {(s, v)}
6. If v ∉ S
7. E_new ← E_new ∪ {(T, v)}
8. E* ← E* ∪ E_new
9. Recompute L_{ASAP}(v), L_{ALAP}(v), and Slack(v) for every vertex v ∈ G
10. For every APCS edge e = (u, v) ∈ E*
11. Integer : cslack ← CSlack(u, v, G_TC)
12. If cslack = 0
13. E* ← E* – (e)
14. Else If cslack < w(e)
15. w(e) ← cslack

Fig. 10. Clustering Algorithm for the APCS
The loop in lines 3-7 introduces parallel template T into G*, removes all edges between vertices in S (line 5), and replaces every edge \((s, v)\), \(s \in S, v \in V - S\), with \((T, v)\) (lines 6-8). Slack is recomputed for every vertex in G (line 9). The loop in lines 10-15 examines every APCSG edge \(e = (u, v)\). If clustering reduces CSlack(u, v) to 0, then e is removed from the APCSG (lines 12-13). If CSlack(u, v) < w(e), then w(e) must be set to CSlack(u, v) (lines 14-15). Clustering terminates once every APCSG edge is processed.

We must also assign types to parallel templates. In previous work (Kastner et al. [2002]), templates are assigned the same type if and only if their internal subgraphs are isomorphic. Although isomorphism has never been proven NP-Complete (Garey and Johnson [1979]), all known solutions in the general case possess exponential worst-case running time (Ullman [1976], Schmidt and Druffel [1976], McKay [1978], Ebeling and Zajicek [1983], Ebeling [1988], Foggia et al. [2001]). Polynomial-time solutions are known for certain classes of graphs, e.g. trees (Aho et al. [1974]), planar graphs (Hopcroft and Wong [1974]), and graphs of bounded valence (Luks [1983]). The DFGs represented by parallel templates contain no edges; two templates are thus isomorphic if and only if they contain the same number vertices of each type. This isomorphism test can be performed in linear time. DFG vertices may be viewed as degenerate cases of parallel templates. These details have been omitted from Figs. 8-10 to save space and enhance clarity.

**D. The Parallel Template Generation Algorithm**

Here, we introduce the Parallel Template Generation Algorithm (PTGA), which uses the APCSG to identify and extract parallelism from within a DFG. The PTGA is a multi-purpose framework for behavioral synthesis and VLIW compilation. These applications are discussed in Section III.

Fig. 11 shows pseudocode for the PTGA. The input parameter is a DFG G. The first two steps build the transitive closure, \(G_{TC}\) (line 1) and the APCSG \(G^*\) (line 2). The remainder of the algorithm is a loop in lines 3-8. Line 4 examines the APCSG and determines which subset(s) of vertices should be replaced with parallel templates; this is carried out by the function Select_Verts_to_Cluster. Each group of vertices is placed into a set, which are collected in a set of sets \(S^*\). In lines 5-8, a parallel template T replaces each set \(S \in S^*\) in G, \(G_{TC}\), and \(G^*\).

The implementation of Select_Verts_to_Cluster (line 4) and Stop_Conditions_Met (line 3) are domain-specific. Details about how we have implemented these algorithms are discussed in Section III.
Parallel_Template_Generation_Algorithm( DFG : G = (V, E) )
1. Trans_Closure : GTC ← Build_Transitive_Closure( G )
2. APCSG : G* ← Build_APCSG( G, GTC )
3. While !Stop_Conditions_Met( G, GTC, G* )
4. Set of Sets of Vertices : S* ← Select_Verts_to_Cluster( G* )
5. For each set of vertices S ∈ S*
6. Cluster_DFG( G, S )
7. Cluster_TC( GTC, S )
8. Cluster_APCSG( G, GTC, G*, S )

Fig. 11. The Parallel Template Generation Algorithm

III. APPLICATIONS

Here, we discuss four different applications that use the PTGA to perform optimization. Section III A discusses resource vectors, which represent the set of computational resources allocated for behavioral synthesis and the datapath exposed to the compiler by a VLIW architecture. Sections III B-E discuss our four application domains: resource allocation for behavioral synthesis (B), instruction scheduling for VLIW architectures (C), the generation of application-specific VLIW instruction set extensions for extensible processors (D); and a code compression mechanism for VLIW architectures.

A. Resource Vectors

We introduce a data structure called a Resource Vector (RV), which is used in the applications of the PTGA discussed in the following four sections. Suppose that the compiler supports N operations (e.g. addition, multiplication, shift, etc.); in other words, for each DFG vertex v, 1 ≤ t(v) ≤ N must hold. An RV V[1..N] contains N nonnegative integer values. V[t(v)] is the number of instances of resources of type t(v).

In synthesis, an RV represents the set of computational resources allocated by the compiler; in most synthesis frameworks, allocation of computational resources occurs prior to scheduling, binding, and allocation of interconnect and storage resources. For VLIW architectures, an RV represents the set of resources exposed to the compiler for scheduling.

An RV is also associated with each parallel template in the PTGA. In this context, the RV is the minimal set of resources that support concurrent execution of all operations in the template. When two parallel templates are merged (e.g. Fig. 7 (c)), the RV of the resulting template is the pairwise sum of the elements of the RVs of the original two templates. Vector addition is defined as follows:
\[ RV'' = RV + RV' = RV'[i] + RV''[i], \quad 1 \leq i \leq N \] (9)

We introduce the Dominance Relation denoted \( \succ \). Dominance is a binary relation between resource vectors \( RV_1 \) and \( RV_2 \), defined as follows:

\[ RV_1 \succ RV_2 \iff RV_1[i] \geq RV_2[i], \quad 1 \leq i \leq N \] (10)

A Global Resource Vector (GRV) is to the complete set of resources allocated during behavioral synthesis, or alternatively, the set of resources in a VLIW datapath exposed to a compiler. A Local Resource Vector (LRV), denoted \( LRV_T \), is associated with each parallel template \( T \). All of the operations in \( T \) can execute concurrently if and only if \( GRV \succ LRV_T \) for every template \( T \) in a DFG. For allocation, the GRV is computed directly from the LRVs associated with each template. Suppose that \( T^* = \{T_1, T_2, \ldots, T_m\} \) is the set of uniquely identifiable templates. There may be multiple instances of each template occurring within a DFG. The GRV, representing the total number of resources allocated, is computed as follows:

\[ GRV[i] = \max_{T \in T^*} \{LRV_T[i]\}, \quad 1 \leq i \leq N \] (11)

This construction ensures that the GRV dominates every LRV.

A maximum allowable area constraint, \( A_{\max} \), may be provided as an input to the system. An area estimate \( A(i) \) is associated with each resource of type \( i \). The area \( A(V) \) required to synthesize an RV \( V \) is:

\[ A(V) = \sum_{i=1}^{r} V[i] \times A(i) \] (12)

The goal of allocation problems is to construct a GRV such that \( A(\text{GRV}) \leq A_{\max} \).

In practice, a resource of type \( i \) will have length \( L(i) \) and width \( W(i) \) such that \( A(i) = L(i) \times W(i) \). The actual area of synthesizing a set of resources depends on the results of a floorplanning tool. Floorplanning, however, is NP-Complete; consequently, Eq. 12 is an approximation for the area of the GRV. Eq. 12 also fails to incorporate the area of storage elements (namely registers) and interconnect resources.

Data memories are not well-specified by RVs. If multiple concurrent memory accesses are required, we could either instantiate multiple memories on chip or use multi-port memories. The former approach would introduce coherency problems and overhead due to memory-to-memory data transfers. The latter approach does not lend itself well to the RV representation because Eq. 12 does not characterize memory area when the number of read and/or write ports increases.
To avoid these complications, we assume there is only a single data memory with fixed numbers of read and write ports, \( R \) and \( W \) respectively. \( \text{ld} \) and \( \text{st} \) are the types associated with load and store operations respectively. Any LRV \( \text{LVT}[\text{ld}] \) must satisfy \( \text{LVT}[\text{ld}] \leq R \) and \( \text{LVT}[\text{st}] \leq W \). For any GRV, we fix the values of \( \text{GRV}[\text{ld}] \) and \( \text{GRV}[\text{st}] \) to \( R \) and \( W \) respectively. For VLIW architectures, the existence of an instruction memory is implicit since it is not exposed to compiler.

### B. Resource Allocation for Behavioral Synthesis

We apply the PTGA to computational resource allocation in a behavioral synthesis framework by constructing a GRV of representing the set of allocated resources. Each parallel template is represented by an LRV; a GRV is constructed as described in the previous section. An initial allocation containing one instance of each necessary resource is provided at the start. For example, if a DFG contains addition, subtraction, and multiplication operations, then the initial allocation will contain 1 adder, 1 subtractor, and 1 multiplier. If the initial allocation violates Eq. 12, then the design is infeasible.

An LRV is associated with each APCSG edge \( e = (u, v) \), where \( u \) and \( v \) are DFG vertices or parallel templates; specifically, \( \text{LRV}_e = \text{LRV}_u + \text{LRV}_v \). For example, if \( u \) is an addition operation and \( v \) is a multiplication operation, then \( \text{LRV}_e \) is trivially dominated by the initial GRV. All APCSG edges \( e \), such that \( \text{GRV} \succ \text{LRV}_e \), are removed from the APCSG because \( e \) will not affect allocation. If \( \Lambda(\text{LRV}_e) > \Lambda_{\text{max}} \), then \( e \) is removed from the APCSG because the resulting GRV would be infeasible.

Let \( t(e) \) be the type of APCSG edge \( e \). Let \( N(t(e)) \) be the number of edges of type \( e \) occurring in the APCSG. At each step in the parallel template generation algorithm, \( N(t(e)) \) is computed for every APCSG edge type. Let \( t_{\text{max}} \) be the edge type that occurs with the greatest frequency. Select_Verts_to_Cluster is implemented to select a subset of APCSG edges of type \( t_{\text{max}} \) for clustering. To ensure that each vertex is placed into at most one parallel template, an independent set of edges must be selected for clustering. Not all independent sets are legal, however, because they may cause cycles to be introduced to the DFG. Figure Fig. 12 provides an example that illustrates this phenomenon. To rectify this situation, we select and cluster APCSG edges one-at-a-time, interleaved with updating the DFG, transitive closure, and APCSG. This is repeated until no more edges of type \( t_{\text{max}} \) remain in the APCSG. A priority function determines which APCSG edge is selected at each step during the PTGA.
In general, we desire a maximal independent set; however, this is not always feasible since clustering may reduce common slack between vertices, even eliminating some APCSG edges. We selected a priority function that attempts to quantify the net reduction in slack in the DFG. Let \( e = (u, v) \) be an edge under consideration. The priority of edge \( e \) for this application, \( P(e) \), is defined as follows:

\[
P(e) = |\text{Slack}(u) - w(e)|^2 + |\text{Slack}(v) - w(e)|^2
\]

At each step, the edge that minimizes \( P(e) \) is selected for clustering. Conceptually, \( P(e) \) estimates the reduction in slack when \( u \) and \( v \) are replaced by a parallel template.

Allocation terminates when all the APCSG contains no more edges, indicating that no more ILP has been extracted.

C. Scheduling for VLIW Architectures

Here, we describe an adaptation of the PTGA for compile-time scheduling of a DFG onto a VLIW processor. The set of explicitly parallel resources in the datapath is represented as a GRV. Each parallel template introduced by the PTGA represents a fetch packet, set of operations issued and executed concurrently. The GRV must dominate the LRV associated with each parallel template; otherwise, the PTGA risks issuing a fetch packet containing more operations than available resources. The goal of scheduling is to minimize the number of fetch packets that must be issued to execute the DFG. For example, the DFG in Fig. 1 (a) requires 9 fetch packets because no operations have been clustered; the DFG in Fig. 7 (a) requires 7 fetch packets; if vertices B, C, and D, were also clustered as a parallel template in Fig. 7 (a), then the number of fetch packets in the schedule would be reduced to 5.
The functional units in the VLIW datapath may have different latencies. For example, an addition/subtraction or logical operation may require a single clock cycle, while multiplication may require multiple cycles to execute. Consequently, the results for each operation in a fetch packet may not be available at the same time. We introduce a set of dummy vertices to represent the delay of each operation; each dummy consumes one unit of latency. If operation M requires k cycles, then M is replaced with a path \((M, D_1), (D_1, D_2), \ldots, (D_{k-2}, D_{k-1})\), where \(D_i\) is a dummy vertex. All successors of M become successors of \(D_{k-1}\). Since dummies are abstract NOPs, we remove all APCSG edges incident on dummies.

Select_Verts_to_Cluster selects a pair of vertices connected by an APCSG edge for clustering at each step. Edge types are not a concern because uniformity of fetch packets does not affect performance. A priority function selects which edge is selected at each step. Let \(\text{deg}(v)\) be the degree of vertex \(v\) in the APCSG, and let \(\text{deg}(uv) = \text{deg}(u) \times \text{deg}(v)\). The priority function, \(Q(e)\) for VLIW scheduling, is given by:

\[
Q(e) = \frac{1 + |\text{deg}(u) - \text{deg}(v)|^2}{\text{deg}(uv)} P(e)
\]

Ideally, we cluster pairs of vertices that minimally impact the remaining ILP in the DFG. Dense regions containing vertices of high degree are likely to yield wide-issue parallel templates. \(Q(e)\) adds an extra term to \(P(e)\) to favor vertices of high-degree in the APCSG and to favor pairs vertices having near-equal degrees. If we cluster a high-degree vertex with a low-degree vertex (unlikely), then we decrease the likelihood that additional operations can be added to the parallel template during future iterations.

The PTGA terminates when the APCSG contains no more edges. Either we have exhausted the ILP inherent in the DFG, or the DFG contains more ILP than the resource constraints of the target architecture.

D. Custom Instruction Generation for Extensible VLIW Processors

Custom instruction generation for extensible VLIW processors contains elements of both allocation and scheduling. The similarity to allocation arises because the compiler may add additional computational resources to a processor to accelerate performance; however, it is pointless to allocate additional resources unless there is a performance increase; consequently, effects of scheduling must also be considered. Our implementation of custom instruction generation for extensible VLIW architectures using the PTGA combines our implementations of allocation for behavioral synthesis with VLIW scheduling.
We modify the synthesis/ allocation framework to accomplish this task. First, we remove the requirement that all APCSG edges having LRVs dominated by the initial GRV be removed from the APCSG because these edges are necessary for scheduling. Otherwise, it would be impossible to schedule two operations of different types, in the same fetch packet. The second change is to use priority function $Q(e)$ rather than $P(e)$. Third, we incorporate the latencies of functional units in the DFG using dummy vertices. The motivation for these last two decisions should be obvious given the previous discussion. Finally, we place an integer limit on the size of each fetch packet. This limit depends on the underlying architectural constraints of the extensible core.

E. Code Compression for DSP and VLIW Processors

In a recent paper, Brisk et al. [2004] developed a compile-time code compression scheme that identified redundant isomorphic subgraphs occurring within a set of DFGs. They argued that the compiler could force identical usage of registers and scheduling of each subgraph, so that they would have identical instruction lists in the final program code. The identical code sequences could then be replaced with procedure calls or a dictionary compression mechanism. Brisk used an iterative approach to subgraph generation that has a similar structure to the PTGA; however, the templates generated did not have the parallel/VLIW structure of the templates described in this paper. Here, we demonstrate the applicability of the PTGA for a compiler optimization that attempts to increase uniformity among the opcodes of fetch packets throughout the program; this, in turn, increases the quality of statistical code compression. Since an opcode is associated with the isomorphic structure of each fetch packet, the system does not explicitly encode NOP operations issued to computational resources whose output is not needed by the current fetch packet.

As an example, consider the DFG in Fig. 13 (a) in conjunction with an abstract 2-issue datapath containing 2 general purpose ALUs that can implement all arithmetic and logical operations. This limits the maximum size of each fetch packet to 2 operations. Two possible organizations of fetch packets are shown in Fig. 13 (b) and (c). The solution in Fig. 13 (b) uses 4 instances of packet $<\text{ADD, MUL}>$. The solution in Fig. 13 (c) uses 2 instances of $<\text{ADD, MUL}>$ and one instance each of $<\text{ADD, ADD}>$ and $<\text{MUL, MUL}>$. 
Since one opcode is used in Fig. 13 (b), a single bit is necessary to encode the opcode; 4 bits will be
needed to encode the opcode sequence since there are 4 fetch packets. Since three opcodes are used in Fig.
13 (c), 2 bits are required to encode the opcodes; 8 bits are required for the opcode sequence. Statistical
encoding with variable length-codewords can reduce the number of bits in the opcode sequence. Huffman
encoding (Huffman [1952]) would assign the following opcodes for each fetch packet: \( <\text{ADD, MUL}> = 0, \)
\( <\text{ADD, ADD}> = 10, <\text{MUL, MUL}> = 11, \) reducing the number of bits in the opcode sequence to 6. bits
can be reduced from 8 to 6. Huffman encoding could not improve the result for Fig. 13 (b).

The PTGA can increase uniformity among fetch packets during scheduling for VLIW processors,
which improves the quality of statistical code compression. If an APCSG is built for Fig. 13 (a), the edges
corresponding to Fig. 13 (b) would all have the same type, whereas those corresponding to Fig. 13 (c)
would not. Therefore, the scheduler should make an extra effort to cluster as many APCSG edges of the
same type as possible as fetch packets are organized. Similar to custom instruction generation but unlike
general scheduling for VLIW architectures, the most frequently occurring APCSG edge type should be
selected by Select_Verts_to_Cluster; we use the same priority function \( Q(e) \) to determine an order in which
to cluster APCSG edges of the same type. The PTGA stops when the most frequently occurring APCSG
dge type occurs only once.

The mechanism described here is intended for stream-based compression mechanisms, whereby the
opcodes and operands of the set of instructions are separated into two streams that are compressed
separately. The PTGA enhances the quality of compression applied to the opcode stream; the operand
stream can be compressed using any existing technique. The experimental evaluation of this technique
focuses solely on the opcode stream.
IV. EXPERIMENTAL RESULTS

The APCSG and PTGA have been integrated into the Machine SUIF framework (Smith and Holloway [2002]), a retargetable open-source compiler back-end developed at Harvard University. Machine SUIF is built upon the SUIF2 compiler infrastructure (Hall et al. [1996]), which was developed primarily at Stanford. We implemented four different versions of the PTGA, one for each application described in the previous section. The implementations are quite similar and share a considerable amount of source code between them. In Section A we discuss the benchmarks that we used to perform the experiment. Sections B-E discuss our experimental results. Section B focuses on the synthesis application. Sections C, D, and E address VLIW scheduling, the generation of application-specific VLIW instruction set extensions, and code compression for VLIW processors respectively.

A. Benchmarks

The PTGA, as currently implemented, operates on a single DFG. We used the Machine SUIF compiler to generate a set of 8 benchmark DFGs which are used throughout this paper. All of these DFGs originated from benchmarks in the MediaBench application suite (Lee et al. [1997]). The 8 DFGs are summarized in Table I. The first four consist primarily of integer operations; the second four (originating from the Mesa benchmark) primarily employ floating-point operations.

An APCSG is generally a denser graph that a DFG, especially when the DFG exhibits significant ILP. The largest DFGs in MediaBench, which consist of unrolled loops, took too long to compile. We manually re-rolled these loops to reduce compilation time; the resulting DFGs were generally small and uninteresting. We only included one (gsm_encode), which proved to be larger and more challenging.

Table I. Summary of Benchmark DFGs. 1-4 are integer benchmarks; 5-8 are floating-point

<table>
<thead>
<tr>
<th>DFG</th>
<th>Benchmark</th>
<th>File</th>
<th>Function</th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Rasta</td>
<td>fft.c</td>
<td>FR4TR</td>
<td>147</td>
<td>166</td>
</tr>
<tr>
<td>2</td>
<td>Pegwit</td>
<td>square.c</td>
<td>squareDecrypt</td>
<td>217</td>
<td>208</td>
</tr>
<tr>
<td>3</td>
<td>GSM</td>
<td>gsm_encode.c</td>
<td>gsm_encode</td>
<td>184</td>
<td>174</td>
</tr>
<tr>
<td>4</td>
<td>JPEG</td>
<td>jfdctfst.c</td>
<td>jpeg_fdct_ifast</td>
<td>142</td>
<td>145</td>
</tr>
<tr>
<td>5</td>
<td>Mesa</td>
<td>matrix.c</td>
<td>invert_matrix</td>
<td>299</td>
<td>295</td>
</tr>
<tr>
<td>6</td>
<td>Mesa</td>
<td>matrix.c</td>
<td>invert_matrix_general</td>
<td>276</td>
<td>273</td>
</tr>
<tr>
<td>7</td>
<td>Mesa</td>
<td>matrix.c</td>
<td>invert_matrix_general</td>
<td>494</td>
<td>498</td>
</tr>
<tr>
<td>8</td>
<td>Mesa</td>
<td>matrix.c</td>
<td>gl_rotation_matrix</td>
<td>123</td>
<td>116</td>
</tr>
</tbody>
</table>
Table II. Component Libraries

<table>
<thead>
<tr>
<th>Component</th>
<th>Latency</th>
<th>Area (CLBs)</th>
<th>Component</th>
<th>Latency</th>
<th>Area (Slices)</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT ADD</td>
<td>1</td>
<td>6</td>
<td>FP ADD</td>
<td>10</td>
<td>365</td>
</tr>
<tr>
<td>INT MUL</td>
<td>2</td>
<td>64</td>
<td>FP MUL</td>
<td>7</td>
<td>358</td>
</tr>
<tr>
<td>INT DIV</td>
<td>4</td>
<td>81</td>
<td>FP DIV</td>
<td>27</td>
<td>738</td>
</tr>
<tr>
<td>NOT</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHIFT</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B. Resource Allocation for Behavioral Synthesis

In our first set of experiments, we organized the PTGA to perform computational resource allocation and integrated it into an established behavioral synthesis framework (Bazargan et al. [1999]) targeting Xilinx Virtex FPGAs. The synthesis framework performs scheduling (Memik et al. [2001]), multiplexer and register insertion, and VHDL emission. Previously, the task of module allocation was performed by hand; by integrating the PTGA into Machine SUIF, this task is now automated. Following synthesis, a floorplan is generated, which provides the latency and area estimates reported in this paper.

We used two separate component libraries for integer and floating point operations. The integer library was provided with the synthesis framework, and has not been modified; the floating point library used latency and area estimates based on commercially-available components (QinetiQ [2002]). These libraries are summarized in Table II. Please note that the areas are reported in CLBs for the integer components and slices for floating-point components. Also, observe that the floating-point adder has a longer latency and slightly larger area than the multiplier, quite unlike the respective integer components.

Support for memory accesses is limited in the synthesis framework. All operations (memory or otherwise) are assumed to have a constant latency. The framework does not allow us to specify separate hit/miss times for memory, nor can we specify a hit ratio. Also, the memory is not included in the floorplan. We assumed single-cycle latency for memory accesses, which were always assumed to be hits. We also assumed the existence of a single memory that can support one access (read or write) per cycle.

Table III shows the results of our experiments for integer benchmarks. We converted DFGs 5-8 to an integer representation because we thought that they exhibited an interesting topology, making them worth studying. $A_{max}$ is defined to be the maximum area constraint allowable for the set of allocated computational units. We ran three experiments for each DFG, where $A_{max} = 250$, 500, and 1000.
Table III. Experimental results for integer benchmarks. Area values are reported in terms of CLBs.

<table>
<thead>
<tr>
<th>DFG</th>
<th>$A_{\text{max}} = 250$</th>
<th>$A_{\text{max}} = 500$</th>
<th>$A_{\text{max}} = 1000$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$A_{\text{Alloc}}$</td>
<td>$A_{\text{Floorplan}}$</td>
<td>Latency</td>
</tr>
<tr>
<td>1</td>
<td>206</td>
<td>780</td>
<td>48</td>
</tr>
<tr>
<td>2</td>
<td>248</td>
<td>784</td>
<td>61</td>
</tr>
<tr>
<td>3</td>
<td>248</td>
<td>756</td>
<td>58</td>
</tr>
<tr>
<td>4</td>
<td>248</td>
<td>580</td>
<td>35</td>
</tr>
<tr>
<td>5</td>
<td>245</td>
<td>960</td>
<td>90</td>
</tr>
<tr>
<td>6</td>
<td>248</td>
<td>992</td>
<td>98</td>
</tr>
<tr>
<td>7</td>
<td>245</td>
<td>1911</td>
<td>160</td>
</tr>
<tr>
<td>8</td>
<td>245</td>
<td>525</td>
<td>36</td>
</tr>
</tbody>
</table>

Table IV. Experimental results for floating-point benchmarks. Area values are reported in terms of slices.

<table>
<thead>
<tr>
<th>DFG</th>
<th>$A_{\text{max}} = 2500$</th>
<th>$A_{\text{max}} = 5000$</th>
<th>$A_{\text{max}} = 10000$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$A_{\text{Alloc}}$</td>
<td>$A_{\text{Floorplan}}$</td>
<td>Latency</td>
</tr>
<tr>
<td>5</td>
<td>2191</td>
<td>6303</td>
<td>844</td>
</tr>
<tr>
<td>6</td>
<td>2176</td>
<td>4368</td>
<td>457</td>
</tr>
<tr>
<td>7</td>
<td>2191</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>8</td>
<td>2184</td>
<td>4202</td>
<td>246</td>
</tr>
</tbody>
</table>

Table IV reports the results of a similar experiment for DFGs 5-8 using the floating-point library. Due to the large discrepancy between the areas of integer and floating point components, the values we selected for $A_{\text{max}}$ were 2,500, 5,000, and 10,000 slices respectively. The two experiments marked with X’s indicate that either the scheduler crashed ($A_{\text{max}} = 2,500$) or produced an erroneous result ($A_{\text{max}} = 10,000$); for these experiments.

In both Table III and IV, the values of $A_{\text{max}}$ do not correspond to the areas of actual FPGA devices; instead, we selected them based on their relative abilities to constrain the allocation for the DFGs that we studied. For each DFG, we reported three quantities: $A_{\text{Alloc}}$, the sum of the respective areas of each component (used during the allocation process); $A_{\text{Floorplan}}$, the area of the complete system reported after multiplexer and register insertion and floorplanning; and the latency of the resulting design. In all experiments, the $A_{\text{Floorplan}}$ proved to be much larger than $A_{\text{max}}$. To ensure feasibility, $A_{\text{max}}$ should be chosen to be a small fraction of the total area of the target device.

In many cases—especially for the integer benchmarks—diminishing returns were realized as $A_{\text{max}}$ increased. In the most extreme case, DFG 5 (Table III) yielded latencies of 69 for $A_{\text{max}} = 500$ and 1000. The additional resources allocated increased the ILP among operations that were not on the critical path of the design. Given a fixed value of $A_{\text{max}}$, it is thus quite easy to over-allocate computational resources.
For the integer benchmarks, memory proved to be a serious constraining factor; this was not the case for the floating-point benchmarks because of the significantly longer latencies of arithmetic operations. Intuitively, either increasing the number of on-chip memories or integrating multi-port memories into the synthesis tool would be necessary to increase performance.

Tables III and IV demonstrate that the PTGA can be used to perform computational resource allocation during synthesis; however, these tables do not compare the PTGA to other techniques or attempt to quantify its optimality (or lack thereof). We manually varied the allocations produced by the PTGA and re-synthesized the designs for several benchmarks. In all cases, the PTGA produced optimal or near-optimal results.

C. Scheduling for VLIW Processors

The bulk of our implementation effort focused on integrating the PTGA with the synthesis system described in the previous section. To target VLIW processors, we took a different approach. Rather than target a specific architecture VLIW architecture, we targeted an abstract one based on the SUIFv3 intermediate format. Because this is a prototype system designed to validate the feasibility of the PTGA, we scheduled individual DFGs rather than whole programs. We targeted 5 abstract architectures, with varying numbers functional units and memory ports that can be accessed in parallel. An architecture family $A_{mn}$ exposes $m$ read/write memory ports and $n$ general-purpose ALUs to the compiler for scheduling. For the this paper, we assume separate architectures for integer and floating point operations. Any real-world architecture (e.g. Itanium, as described in Hennessey and Patterson [2003]) must support a mix of integer and floating-point operations. We target five different abstract architectures, $A_{11}, A_{12}, A_{14}, A_{22},$ and $A_{24}$.

We report two quantities for each benchmark DFG: the first is the number of fetch packets issued by the compiler for each DFG, including the execution of NOPs when long-latency operations prevent the execution of operations that depend upon them. Second, we estimate the utilization of resources by the scheduler. Let $FP^* = \{FP_1, FP_2, \ldots, FP_k\}$ be the set of fetch packets DFG scheduled for architecture $A_{mn}$. Let $|FP_i|$ be the number of operations issued in fetch packet $FP_i$. Utilization, $\mu$, is computed as:

$$\mu = \frac{1}{k(m + n)} \sum_{i=1}^{k} |FP_i|$$

(15)
Fig. 14. The number of fetch packets issued for the PTGA-based VLIW scheduler for each of the 8 benchmark DFGs.

Fig. 15. Resource utilization for the PTGA-based VLIW scheduler for each of the 8 benchmark DFGs.

Fig. 14 shows the number of fetch packets issued for each of the five architectures, while Fig. 15 illustrates utilization. Increasing the issue-width tends to increase performance (i.e. fewer fetch packets are issued); however, the performance increase observed is subject to the law of diminishing returns as more and more functional units are added—in other words, utilization decreases as more functional units are added.
The most striking observation is Figs. 14 and 15 occurs with respect to DFG #5, a floating point benchmark. The number of fetch packets issued for A_22 is significantly fewer than for A_24, which is counter-intuitive. After analyzing the schedules, we observed that 47 NOPs were issued for A_24, while only 21 were issued for A_22. To understand why this occurred, consider the latencies of the floating point computational units in Table II. For each floating point operation, 7, 10, or 27 dummies are introduced. For DFG #5, the dummies dominated the rest of the graph. As a result, the organization of operations into fetch packets (i.e. the PTGA) is less important than the order in which fetch packets are issued.

We used a simple heuristic to construct the schedule for each DFG. At each step, all of the sources (vertices of degree 0) in the DFG are examined. All sources that are dummies are trivially removed from the graph at each step (since they are not executed). The first fetch packet that is found in the list is then issued (although the traversal continues to eliminate the maximum number of dummies). If no fetch packets are found during the traversal, then a NOP (to all computational units) is issued instead of a fetch packet. The essentially random approach of issuing the first fetch packet found (rather than making a more intelligent decision) caused the number of fetch packets to increase for A_24, not the PTGA. The investigation of algorithms for issuing fetch packets that have already been formed in order to minimize the number of NOPs in the schedule is beyond the scope of this paper. This situation only seems to arise when the code is dominated by high-latency (i.e. floating-point) operations. Also, in a real-world scenario, many of these NOPs may otherwise be masked by cache misses.

D. Custom Instruction Generation for Extensible VLIW Processors

Next, we apply the PTGA to the generation of custom instruction sets for extensible VLIW processors. To perform this task, we start with the basic A_{11} architectural model from the previous section. Assume that the core instruction set allows for as many as k operations to be issued in a single fetch packet, and that the user will allocate all computational units in excess of the core processor and single-port memory system. The PTGA simultaneously allocates additional computational resources to the design (within an area constraint) while organizing the DFG into a set of fetch packets with no more than k operations (assuming at most one memory operation per fetch-packet).
Without loss of generality a k-issue VLIW architecture will require k general-purpose ALUs to offer sufficient flexibility to the wide variety of programs that may execute upon it. As a simplification, assume that general-purpose integer ALU requires exactly one of each arithmetic and logical resource listed on the left of Fig. 1, at a cost of 171 CLBs. Consider a 3-issue VLIW (omitting the core), which will require 513 CLBs. Now, suppose that a user of an extensible processor knows that the application will issue at most 2 multiplication operations in parallel, at most 3 addition operations in parallel, and will not issue logical and/or shift operations. Then only 146 CLBs are needed for computational units, a savings of 367. Even if the user wishes to issue up to 6 multiplication and 7 addition operations respectively in parallel, then 426 < 513 CLBs are needed. In this section, we attempt to quantify the performance and economic gains of extensible VLIW processors relative to fixed instruction format VLIWs.

As our baseline, we take the A14 architecture from the previous section. Let E_k(A_{max}) define an extensible VLIW architecture that can support up to k operations issued simultaneously, where A_{max} is an area constraint (discussed here in terms of CLBs) taken from the discussion of resource allocation for synthesis. We target 3 different extensible VLIW processor models: E_6(250), E_8(500), and E_{10}(1000). Experiments are only performed for our integer benchmarks (DFGs 1-4).

Fig. 16 lists the number of fetch packets issued for each DFG by each extensible VLIW processor relative to the baseline A14 architecture. Fig. 17 shows the relative area costs of computational resources.

Fig. 16. The number of fetch packets issued for PTGA-based VLIW custom instruction set extensions relative to architecture A_{14} (see Fig. 14).
In all but two cases in Fig. 16 (DFG 2, E6(250) and E8(500)), the number of fetch packets was reduced relative to A14. In only one case (DFG 2, E6(250)) did the number of fetch packets issued actually increase; however, this design required less than 40% of the area as did A14 for computational resources. In Fig. 17, only one benchmark (DFG 4, E10(1000)) increased its area relative to A14; however, this design issued less than 80% of the number of fetch packets as did A14.

This experiment is fairly limited in scope and only focuses on four individual DFGs, not larger code corpuses. It should be noted, however, that for most programs, additional resources only need to be allocated in order to accelerate the performance of critical loops. Consequently, we believe that this experiment successfully demonstrates the applicability of the PTGA to the problem of custom instruction generation for extensible VLIW processors.

### E. Code Compression for VLIW Processors

The final application of the PTGA presented in this paper is for a code compression mechanism for embedded VLIW processors. By organizing as many identical fetch packets as possible, uniformity is increased throughout the program. This, in turn, increases the quality of statistical compression when applied to the program. The opcode compression technique (OC) has been implemented as described in [Fig. 17. The area cost of computational resources for PTGA-based VLIW custom instruction set generation relative to the resources required for the architecture A14.]
Section III. E. This approach is compared to the VLIW scheduling technique (VS). We experiment with two different encoding schemes. The first encoding scheme, which we refer to as logn-encoding (LE) replaces each opcode with a codeword of fixed length; if there are $k$ unique opcodes in a program, then exactly $\lceil \log k \rceil$ bits are used for each codeword. The second scheme is Huffman encoding (HE, Huffman [1952]). This yields a total of four experiments for each benchmark: VS/LE, VS/HE, OC/LE, and OC/HE. The results, which are normalized to the code size of VS/LE, are shown in Fig. 18. The architecture that we targeted was the $A_{24}$, which supports the concurrent issuance of up to 4 ALU operations and 2 memory accesses. The results of this experiment are summarized in Fig. 18.

The effectiveness of the OC compression technique varied from benchmark to benchmark, however, the overall trend appears to indicate its effectiveness, especially for the larger DFGs. Both of the OC implementations dramatically reduced the size of the opcode sequence relative to the VS implementations for DFGs 5, 7, and 8; its affectiveness appeared to be more subdued for the smaller benchmarks, DFGs 1-4. For DFG 6, the VS/HE, OC/LE, and OC/HE techniques were considerably more effective than VS/LE. For DFGs 5-7, OC/HE offered no measurable improvement over OC/LE. Altogether, the OC/HE technique performed the best of the four techniques shown. In several cases, however, VS/LE slightly outperformed OC/LE.

![Fig. 18. The results of 4 PTGA-based code compression heuristics for VLIW architectures.](image-url)
The quality of compression for techniques such as these is likely to improve if the scope of compression is enlarged. It would be quite reasonable to perform statistical compression over entire function, module, or program bodies.

V. RELATED WORK

Related work with respect to the four application domains discussed in Section III is summarized here. Section V A focuses on CDFG synthesis and summarizes techniques for hardware allocation. Section V B discusses scheduling techniques for VLIW and DSP architectures. Section V C summarizes recent work on extensible processor customization, emphasizing techniques for VLIW architectures. Section V D provides an overview of recent work on code compression for VLIW and DSP architectures.

A. Behavioral Synthesis

The task of generating hardware from a behavioral description has been at the forefront of CAD research for the last 15 years. The latest advances allow users to specify computations in high-level languages based on C/C++. For example, the DEFACTO system (Zeigler et al. [2003]) translated C code to VHDL for pipelined FPGAs using data flow analysis and parallel compilation techniques. Mittal et al. [2004] translated software binaries targeted for the Texas Instruments C6000 DSP architecture to register-transfer-level (RTL) VHDL or Verilog. First, the binaries were decompiled into a CDFG, allowing for optimizations such as scheduling and loop unrolling prior to synthesis. Kaplan et al. [2003] focused on minimizing data communication between basic blocks, each of which was synthesized as a separate hardware module. Memik et al. [2003] performed global resource sharing in a similar framework.

The PTGA focuses on the allocation of computational resources during high-level synthesis. Previous techniques for module allocation include clique partitioning (Tseng and Siewiorek [1983] [1984]), and latency constrained scheduling (Parker et al. [1986], Trickey [1987], Heijligers et al. [1995]). Devadas and Newton [1989] formulated allocation as a 2-dimensional placement problem of microinstructions in time and space, which is solved using simulated annealing. Gebotys [1992] developed an integer programming formulation that simultaneously performs scheduling and allocation. Gutberlet et al. [1992] used an
iterative improvement algorithm to find an optimal allocation of resources. After generating an initial allocation, new allocations are randomly generated by adding or removing components from the current allocation. Each allocation is evaluated by an area-delay product based on an estimated result of scheduling; the most favorable allocation is then selected.

The APCSG could be used to guide the allocation process in a similar framework. Rather than randomly generating new allocations, the APCSG can be analyzed to direct allocation decisions. This approach is would be different that the PTGA, which greedily allocates a single allocation. Modifying the PTGA to generate a set of allocations, all of which would be synthesized and then evaluated, is a possible direction for future work.

B. Instruction Selection for Embedded VLIW Architectures

The performance of code executing on VLIW architectures is highly dependent on the quality of the compiler that produced it. One of the most important optimizations that a compiler can perform for VLIW architectures is scheduling. Here we review the history of scheduling for VLIW architectures and describe where the APCSG and PTGA fit in. Much of the work on scheduling for VLIW processors focuses on compiler transformations to expose more ILP to the compiler. Typically, these transformations resulted in larger DFGs than the original code. Because the APCSG can be derived from any DFG, the PTGA can be applied to organize fetch packets once these initial transformations have been completed.

One of the earliest techniques was called trace scheduling (Fisher [1981]). A trace is a sequence of basic blocks that are executed consecutively when the program runs. Loop unrolling is typically used to generate traces (Hennessy and Patterson [2003]); inlining of function calls within traces can also expose additional ILP. After unrolling, the trace is represented as a large DAG, which has higher ILP than an unrolled loop body. A greedy scheduling algorithm optimizes the performance of the trace when it executes; however, if a trace is entered or exited at arbitrary entry points, bookkeeping code is required, which can become quite cumbersome. To address these issues, scheduling techniques have been developed for more general code regions such as superblocks (Hwu et al. [1993]) and treregions (Havanki et al. [1998]); these approaches both increase performance and reduce bookkeeping code relative to trace scheduling. Software pipelining (Aiken and Nicolau [1988], Lam [1988]) is a loop optimization that allows
operations in the k, k+1st, k+2nd, …, k+nth iteration to execute in parallel. Software pipelining also avoids the code bloat associated with loop unrolling/inlining and trace scheduling.

As mentioned earlier, the APCSG could be used for the actual scheduling step once each of these code transformations (e.g. trace or superblock formation) is performed. The key is to recognize that the APCSG represents combinations of operations that can be scheduled concurrently. Therefore, it should be used to guide the formation of fetch packets rather than the organization of the program code. Conceivably, the APCSG could be used to guide certain code transformations (such as trace formation, or how much to unroll a loop), however, we suspect that the overhead of repeatedly building the APCSG as new code regions are formed would make the technique unwieldy. An investigation of this idea is far beyond the scope of this paper.

C. Custom Instruction Generation for Extensible VLIW Processors

Extensible processors such as the Tensilica Xtensa (Gonzalez [2000]) have allowed compilers to add custom hardware to the design of a core RISC processor. The majority of these techniques enumerate subgraphs of the compiler’s intermediate representation, which are translated to hardware via behavioral synthesis. Sun et al. [2004] developed a candidate enumeration algorithm that supported independent (i.e. VLIW) instruction patterns. An algorithm developed by Brisk et al. [2002], which initially described the APCSG and PTGA, explicitly enumerated parallel custom instructions for reconfigurable processors.

The Program-In-Chip-Out (PICO) System (Aditya et al. [1999]) automated the design of application-specific VLIW processors. PICO employs uses clique partitioning to allocate computational units. The AutoTIE system (Goodwin and Petkov [2003]) also generates of application-specific VLIW instruction set extensions. The APCSG and PTGA are well-suited for frameworks such as PICO and AutoTIE.

D. Code Compression for VLIW Processors

Code compression for DSP and VLIW architectures has received a considerable amount of attention as of late. For example, Ishiura and Yamaguchi [1998] performed compression using field partitioning of the operations within VLIW instructions, while Nam et al. [1999] compressed instructions having identical
opcodes but slightly different sets of operands that occur throughout the program. Both of these techniques use statistical compression, and could therefore benefit from the approach discussed in Section III E.

Dictionary compression techniques for DSP architectures (Lefurgy and Mudge [1998], Liao et al. [1999], Centoducatte et al. [1999]) place all instructions that occur throughout the program into a dictionary. The program then consists of a set of indices into the dictionary that are smaller than a standard DSP instruction word. These approaches work best when identical instruction sequences occur repeatedly throughout the program. The PTGA could be coupled with the framework described by Brisk et al. [2004] to increase uniformity among VLIW code sequences. This is left open as possible future work.

The majority of VLIW instruction formats use a sequence of micro-operations, each corresponding to one functional unit in the architecture. Xie et al. [2001] developed a compression algorithm for flexible VLIW instruction formats that does not require encoding of NOPs for each computational unit that is not used into each instruction word. Compression is based on statistical Markov encoding and requires custom hardware for decoding during instruction fetch. Xie et al. [2002] used variable-to-fixed coding, which simplified the design of the decoder. An LZW-based VLIW code compression technique (Lin et al. [2004]) increases instruction throughput relative to variable-to-fixed encoding while enabling parallel decompression. Because these techniques employ statistical compression, the PTGA could be used to increase uniformity among the opcodes in a similar manner to that described here.

VI. CONCLUSION

To optimize performance, compilers and CAD tools must identify, extract, and exploit parallelism. This paper introduced the APCSG as an intermediate representation of the parallelism that occurs within an application. The conceptual advantage of the APCSG is that it represents parallelism within a computation, rather than the computation itself. APCSG edges represent the possibility—not the guarantee—that two operations may be scheduled concurrently on a machine that offers sufficient parallelism to accomplish this task. By explicitly representing parallelism, the APCSG offers a direct representation of potential performance optimizations. This makes the APCSG an ideal candidate for guiding branch-and-bound and iterative improvement algorithms for tasks such as design space exploration for embedded systems.
This paper also introduced the PTGA as an algorithmic framework for optimization using the APCSG. The efficacy of the PTGA has been established by applying the framework to four problems in CAD and compilation for VLIW processors: resource allocation for behavioral synthesis, instruction scheduling for VLIW processors, generation of application-specific VLIW custom instruction sets for extensible processors, and code compression for VLIW processors.

Several issues still remain unexplored with respect to the APCSG. Experience has dictated that it is a truly dense graph. This is especially true if the DFG from which it was constructed contains significant ILP. Additionally, the APCSG requires the transitive closure to be maintained (or recomputed at a high cost) in order to support the clustering operation. Despite its density, the APCSG must also be a malleable and dynamic data structure that can grow and/or shrink with ease. One possible approach would be to combine the representations of a DFG, its transitive closure, and APCSG into a single data structure to preserve space; on the other hand, this structure would be large, monolithic, and difficult to maintain and extend over time. For the results presented in this paper, we maintained three separate graphs and used hash tables to maintain correspondence between respective nodes in each graph. At present, we are more concerned with developing and perfecting new analysis and optimization techniques using the APCSG and PTGA than tuning the representation itself. Inevitably, though, large DFGs that result from loop unrolling and function inlining will necessitate the streamlining of the APCSG.

REFERENCES


