Scalable Subgraph Mapping for Acyclic Computation Accelerators

Nathan Clark, Amir Hormati, Scott Mahlke
Advanced Computer Architecture Lab
University of Michigan - Ann Arbor, MI
{ntclark, hormati, mahlke}@umich.edu

Sami Yehia
ARM Ltd.
Cambridge, United Kingdom
sami.yehia@arm.com

ABSTRACT

Computer architects are constantly faced with the need to improve performance and increase the efficiency of computation in their designs. To this end, it is increasingly common to see acyclic computation accelerators appear in embedded processor designs. One major problem with adding accelerators to a design is that it is difficult to generate high-quality code utilizing them. Hand-written assembly code is typical, and if compiler support does exist, it is implemented using only greedy algorithms. In this work, we investigate more thorough techniques for compiling to processors with acyclic accelerators. Where as greedy solutions only explore one possible solution, the techniques presented in this paper explore the entire design space, when possible. Intelligent pruning methods are employed to ensure compilation is both tractable and scalable. Overall, our new compilation algorithms produce code that performs on average 10%, and up to 32% better than standard greedy methods. These algorithms also run in less than one second for more than 98% of basic blocks tested.

Categories and Subject Descriptors

D.3.4 [**Processors**]: [Code Generators]; C.3 [**Special-Purpose and Application-Based Systems**]: [Real-time and Embedded Systems]

General Terms

Algorithms, Experimentation, Performance

Keywords

Compilation, Embedded Processors

1. INTRODUCTION

Many portable devices must be capable of performing computationally demanding tasks, such as processing images, signals, video, or packet streams. However, current embedded processors are not capable of meeting the performance requirements within their tight power and cost constraints. Traditionally, application-specific integrated circuits (ASICs) are utilized to do the heavy lifting in system-on-chip designs, where critical portions of applications are mapped directly to hardware implementations. These ASICs are nonprogrammable accelerators that can achieve extremely efficient solutions through hardware customization. ASICs often yield orders of magnitude wins over programmable solutions in cost, performance, and energy.

Lack of programmability is the central drawback associated with ASICs. The need to freeze a hardware implementation is a major obstacle, as the software is often a moving target due to changing standards, bug fixes, and the desire to incorporate more features. A middle-ground solution is to employ smaller, but compilable hardware accelerators, referred to as *computation accelerators*, within the context of a programmable processor. Such ASIPs (application specific instruction processors) utilize computation accelerators that are tightly integrated into a processor pipeline. The computation accelerators are essentially small ASICs that can atomically execute portions of an application's dataflow graph, termed computation subgraphs. The processor is augmented with a set of new instructions or a dynamic mapping mechanism to invoke the computation accelerators. Computation accelerators offer several potential advantages, including reduced latency for subgraph execution, increased execution bandwidth, improved utilization of pipeline resources, and reduced burden on the register file for storing temporary values. And unlike ASIC solutions, computation accelerators do not sacrifice the post-programmability of the system.

Many computation accelerator designs have been proposed by researchers. The most widely used in industry is the multiply-accumulate, or MAC, unit. Many DSPs have specialized hardware for common computations in signal and image processing, such as dot product, sum of absolute differences, and compare-select. A number of generalized accelerator designs have also been proposed, such as 3-1 ALUs [22, 25], closed-loop ALUs [27], or ALU pipelines [5]. Larger accelerators can support bigger subgraphs and thus enhance the performance advantages. Examples that exemplify this approach include FPGA-style accelerators [14, 23, 26, 30], configurable compute accelerators [8], and programmable carry function units [31]. An alternative strategy is to synthesize specialized computation accelerators for a particular application [3, 4, 6, 9, 13, 17]. These approaches analyze an application to identify important computation subgraphs to implement in hardware. Hardware synthesis creates highly specialized accelerators to execute the selected subgraphs. Several commercial tool chains utilize this approach, including Tensilica Xtensa, ARC Architect, and ARM OptimoDE.

Compiler support to exploit computation accelerators has been an overlooked challenge, and comprises the focus of this paper. The compiler has two major tasks when targeting a computation accelerator. First, it must identify candidate subgraphs in the target application that are functionally executable on the accelerator. This is essentially a subgraph isomorphism problem. The second task is to select which candidate subgraphs to actually execute on the

CASES'06, October 23–25, 2006, Seoul, Korea.

computation accelerator. Candidates often overlap, thus the compiler must select a subset to maximize performance gain. This task is essentially a graph covering problem.

Most prior solutions employ a greedy compiler approach for both subgraph identification and selection [5, 16]. With this approach, a seed operation is selected and a subgraph compatible with the accelerator is grown by iteratively including connected operations. As with all greedy approaches, this approach can achieve sub-optimal solutions in both identification and selection. Further, disjoint subgraphs cannot be identified. However, for small accelerators, such as 3-1 ALUs, this approach is sufficient due to the simple nature of compatible subgraphs. The greedy approach breaks down for larger accelerators where correspondingly larger subgraphs must be identified. As a result, others have proposed using exact methods for subgraph isomorphism and covering [20, 21, 24]. These methods grow exponentially in subgraph size, region size (the unit of operations analyzed by the compiler), or both. As a result, exact methods can suffer from excessive compilation times for moderate to large applications and hence may not be practically deployable.

In this paper, we propose an approach for compiler subgraph mapping that combines exact methods with a set of intelligent pruning techniques. Pruning ensures the proposed algorithms are scalable in both application and accelerator size to provide practical compilation times. The approach has three distinct phases. First, potential subgraphs are identified using bounded enumeration. Subgraph isomorphism is then used to remove candidates that are not compatible with the computation acceleration. Finally, unate covering is used to select subgraphs that will be executed on the accelerator.

This paper makes the following three contributions:

- It collects and describes state of the art algorithms for accelerator compilation.
- It presents new algorithms for identifying and mapping subgraphs optimally with intelligent pruning mechanisms.
- It evaluates these new algorithms in terms of both performance and compilation time across a variety of accelerator designs, and compares the results to a traditional greedy approach.

2. PROBLEM STATEMENT AND RELATED WORK

Compiling an application to make use of computation accelerators boils down to two steps: *enumerating* portions of the application's dataflow graph (DFG) that can be executed on the accelerator, and *selecting* which portions to accelerate.

Enumeration consists of generating a set of subgraphs from a given DFG, and determining if they can run on an accelerator. Generating a set of subgraphs is difficult because the number of possible subgraphs grows exponentially with the size of the DFG. Determining if the subgraphs can run on an accelerator, i.e., determining if they perform the same computation, is essentially equivalence checking, which is NP-complete. The problem is further complicated if the accelerators perform a superset of the desired computation (e.g., an accelerator for dot-products could also accelerate multiply-accumulates in an application).

Selecting which subgraphs to accelerate is also difficult. Typically, the selection problem is formulated to push as much computation as possible onto the accelerators, while minimizing overlap between subgraphs. That is, given a set of enumerated subgraphs, find the group that covers the largest portion of the DFG while minimizing the number of nodes appearing in multiple subgraphs. This problem is also NP-complete and is quite similar to the well known technology mapping problem in VLSI design. Clearly, mapping applications to subgraphs is a challenging compilation problem.

To side step the problem, the vast majority of previous work relies on hand coding or greedy heuristics. Work on automated accelerator design typically does not discuss strategies for utilizing the accelerators with compilers. Work by Hu [16] is typical of the greedy solutions: a seed node is selected in the DFG and is grown along dataflow edges. The compiler then replaces that subgraph and repeats the process. Here, enumeration consists of finding a seed and growing it, while selection is implicit (any subgraph enumerated is automatically selected). Other previous work [27] performs more thorough enumeration, but still uses greedy selection.

More thorough, traditional code generation methods for tackling subgraph mapping use a tree covering approach [1]. In this approach, all computation subgraphs potentially supported by the accelerator must be constructed a priori. During compilation, the DFG is split into several trees. The trees are then covered by the computation subgraphs using an algorithm that minimizes the number of computation subgraphs used. The purpose behind splitting the DFG into trees first is that there are linear time algorithms to optimally cover trees, making the process very quick.

The major problem with this method is that many DFGs and accelerators are not trees. It is shown in [20] that tree covering methods can yield suboptimal results, particularly in the presence of irregular computation commonly targeted by embedded systems. To overcome this, [20] proposes splitting all instructions into "register-transfer" primitives and recombining the primitives in an optimal manner using integer programming. Work by Liao [21] attacked the same problem and developed an optimal solution for DFG covering by augmenting a binate covering formulation. While both of these solutions are optimal, they also have worst case exponential runtime and do not report how long their algorithms take.

Another major problem with previously mentioned approaches is that they also require permissible accelerator subgraphs to be enumerated a priori. If an accelerator supports a wide range of computations, such as an ALU pipeline, this can cause an explosion in runtime.

Research in [24] describes a different way to look at the accelerator mapping problem. In this work, an application is initially decomposed into an algebraic polynomial expression that is functionally equivalent to the original application. Next, the polynomial is manipulated symbolically in an attempt to use accelerators as best as possible. For example, a polynomial could be expanded using function identities (e.g., adding 0 to a value) to better fit an accelerator. This enables the algorithm to utilize subgraphs where the accelerator performs a superset of the desired computation. As with previous solutions, though, this technique also has exponential worst-case runtime. Additionally, handling bitwise operations, such as XOR, is difficult using polynomials. Rearranging application to better fit a targeted accelerator, such as [24] proposes, is an interesting area of future work, though.

In this work, we present compilation techniques to exploit acyclic computation accelerators. These techniques produce higher quality code than greedy heuristics, do not require a priori enumeration of permissible accelerator subgraphs, and are scalable to large applications.

3. ACCELERATOR COMPILATION

In this section, we present two different approaches for compiling to acyclic accelerators. The first approach, *greedy enumeration*

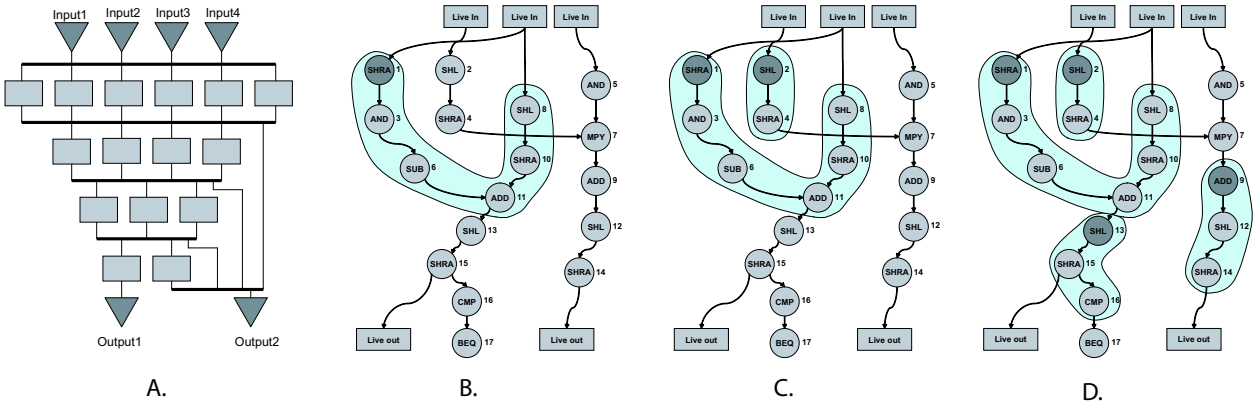

Figure 1: A. An acyclic accelerator from [8] targeted in examples. B. The first step in a greedy mapping algorithm on a basic block from g721encode. C. The second step and D. final step in the greedy mapping algorithm.

- *immediate selection*, is the most commonly used approach today. This method generates a set of subgraphs by greedily adding vertices to a seed vertex from the DFG. Once the subgraphs are grown, they are immediately replaced in the application, thus the name immediate selection. The second approach, *full enumeration - unate covering selection*, is our contribution. This approach generates all possible dataflow subgraphs subject to certain constraints of the targeted accelerator. The set of subgraphs is then pruned down using subgraph isomorphism, and finally unate covering selects the subgraphs to execute on the accelerator.

3.1 Greedy Enumeration - Immediate Selection

Greedy enumeration - immediate selection, or greedy algorithms for short, is the standard method used to target acyclic accelerators, e.g., in [5, 15]. The greedy algorithm consists of two phases: seed selection and subgraph growth. Using a basic block as input, the greedy algorithm selects an operation as a seed and tries to expand that seed by iterating over dataflow edges. After growing one seed as much as possible, the subgraph is replaced. Next, another seed is selected, and the same steps will be repeated. The algorithm finishes when no more seeds are available for growing.

The first step in the greedy algorithm, seed selection, can be performed in several different ways. For example, operations closer to the critical path can be chosen as seeds before less critical operations. Alternately, long latency operations can be selected before shorter operations. In our experiments, changing seed selection order made very little difference in the results of the greedy algorithm, because the targeted accelerator was relatively large in relation to the size of a typical basic block. The results presented in this paper selected seeds in topological order, according to dataflow edges.

After choosing a seed, a subgraph consisting only of that operation is formed. The algorithm then enters its second phase, subgraph growth, trying to expand this subgraph. Neighbors of the seed operation are temporarily added to the subgraph one at the time. If this temporary subgraph is executable on the accelerator, then the new node permanently becomes part of the subgraph. If the temporary subgraph is not executable, then the newly added node will be removed. When it is no longer possible to add neighbors to the subgraph, it is immediately replaced in the application, and a new seed is selected from operations not already appearing in a subgraph.

An example of the greedy algorithm is shown in Figure 1. Figure 1B is a DFG from the g721encode benchmark, used in examples throughout the paper. Figure 1A shows the acyclic accelerator targeted in all of the examples. This accelerator, similar to one proposed in [8], has 4 inputs, 2 outputs, and 15 function units (FUs) organized in 4 rows. The FUs in each row can communicate with FUs in subsequent rows, meaning computations with dependence heights of up to 4 are supported. These FUs support the complete set of addition, subtraction, and bitwise operators on two inputs.

Figure 1B highlights the first subgraph enumerated using the greedy method. Operation 1 is chosen as the first seed. The subgraph then greedily adds neighbor operations 3, 6, and 11. After adding operation 11, 13 cannot be added since that would create a subgraph with dependence height 5, which is not supported by the accelerator. Operation 11's neighbors, 10 and 8, can be added, resulting in the final subgraph shown in Figure 1B. The process then repeats with operation 2 as a seed node. This subgraph is grown along dataflow edges to include operation 4 in Figure 1C. Growth stops at operation 7, a multiply, which is not supported by the accelerator. The final greedy mapping of the DFG is shown in Figure 1D. Assuming a single issue processor where each operation and the accelerator take one cycle to execute, this mapping would yield a speedup of $\frac{17}{3+4} = 2.43$, since there are 17 original operations, 3 unaccelerated operations, and 4 accelerated subgraphs.

3.2 Full Enumeration - Unate Covering Selection

Greedy subgraph mappers have proven reasonably effective in many previous works. Certain combinations of greedy algorithms with more thorough strategies have also proven effective [27]. In this section, we describe the full enumeration - unate covering selection, or FEU, algorithm, which solves the mapping problem using exact formulations. This effectively avoids local minima that inherently cause greedy algorithms to fail. When the exact formulations are intractable, the FEU algorithm intelligently reduces the search space to workable levels.

There are three main phases to the FEU algorithm: Enumeration, Pruning, and Covering. Enumeration generates a set of all subgraphs within a DFG subject to input/output constraints of the targeted accelerator. Pruning takes the set of subgraphs and performs additional checks based on functionality and interconnect to determine if the subgraphs actually can be executed on the targeted accelerator. Once unusable subgraphs are pruned, unate covering is used to select the best set of subgraph instances to map onto the accelerator.

Individually, each of these steps either grows exponentially with the size of the input (enumeration) or is NP-Complete [11, 12] (pruning and covering). This has lead most researchers to opt for (typically) linear-time greedy solutions. In the remainder of this

section, we will demonstrate that with careful design, each of these problems can be made tractable for most practical cases in accelerator compilation. Additionally, we demonstrate in Section 4 that using more powerful algorithms yields noticeable performance improvements in code generated over the standard greedy approaches.

3.2.1 Full Enumeration

The first step of the FEU compilation algorithm, enumeration, generates a set of dataflow subgraphs that can potentially be run on a targeted accelerator. The primary reason for enumerating subgraphs and then later pruning them is that it is much faster than performing both steps at once. Very fast techniques for finding high-quality subgraphs for acceleration have been widely developed in the past few years, e.g., [3, 4, 9], and this strategy allows us to take advantage of them.

Tractable subgraph enumeration is clearly a difficult problem. In the most general sense, each operation in a DFG could either be included or excluded in a potential subgraph instance, yielding 2^N potential candidates. Because of space restrictions, the large body of previous work, and the relative complexity of proposed techniques, we will only describe how to efficiently enumerate subgraphs at a high level.

Dataflow subgraph enumeration can be thought of as a binary tree, where each level of the tree represents an operation (op for short), and each branch in the tree represents whether or not to include that op in a subgraph [3]. The leaves of the tree represent all possible subgraphs for a DFG. There are many keys to make full exploration of this tree tractable.

The most important technique is based on input/output restrictions of the accelerator. Using the DFG from Figure 1B as an example, if a targeted accelerator only supported 2 inputs, then any candidate subgraph including ops 1, 2, and 5 would be infeasible. Enumeration can be bounded for each branch of the tree that includes all of those ops. Likewise, bounding for outputs greatly reduces the search space. Care must be taken to avoid prematurely bounding the search space, though. For example, a subgraph with ops 6 and 10 would appear to have 2 outputs; however, if op 11 is included, then subgraph 6, 10, 11 only has 1 output, perhaps making it feasible.

Another important bounding technique is excluding candidates with values that leave and then reenter the subgraph. Using Figure 1B as an example again, this filter would bound the search space of any subgraph that included ops 1 and 6 but excluded op 3. Subgraph 1, 6 could not be run on an accelerator since the output of 1 is used to calculate an input to 6.

These techniques to bound growth of the search tree make subgraph enumeration practical for the vast majority of blocks within applications; there are some instances where additional steps are needed, though. In these instances, the DFG is heuristically partitioned into several sub-blocks, which are then enumerated. The implication of partitioning is that no candidate subgraphs can cross the boundary (i.e., a subgraph cannot have ops in multiple partitions). Edges are heuristically weighted to guide the partitioner so that is does not unnecessarily cut edges for important subgraphs. For example, if the targeted accelerator did not support multiplication, then all the edges to and from op 7 in Figure 1B would be given weight 0, since the ops on either side of the edges could never be in a feasible candidate. Edges bordering memory operations are also given weight 0 whenever the accelerator does not support memory accesses. All other edges are given weights based on characteristics such as whether or not they are on the critical path. Heuristic partitioning does introduce potential suboptimality into the FEU algorithm; however, previous work [9] demonstrated that this is an effective way to bound the enumeration space without unnecessarily removing useful subgraph candidates.

3.2.2 Pruning Through Subgraph Isomorphism

Pruning is the next step after enumeration generates potential subgraphs to execute on the accelerator. The purpose of pruning is to ensure that candidates can actually be executed on the accelerator. This takes into account functionality and connectivity issues that were ignored during enumeration. Pruning occurs after enumeration because these checks are either not possible to perform on partial candidates, or are too heavy weight to use during construction of the enumeration search tree.

The method employed to determine that subgraphs can execute on an accelerator is based on subgraph isomorphism. Loosely stated, subgraph isomorphism determines whether or not a subset of the nodes in a particular graph are equivalent to a separate graph. In this case, a graph representing the hardware structure is constructed, and we attempt to find a subset of hardware vertices that can create a computation equivalent to the subgraph created in enumeration. If we find such a subset, then the dataflow subgraph is capable of being executed on the accelerator.

There are several pros and cons to pruning based on subgraph isomorphism. One benefit is that, as with enumeration, a great deal of related work (e.g., [19, 28]) has looked at developing heuristics to efficiently solve subgraph isomorphism the problem. We leverage and improve upon these prior techniques in this work. An additional benefit is that previous work [29] has shown it is possible to automatically generate hardware subgraphs from a microarchitectural specification. This means that a compiler targeting accelerators could potentially be retargeted by simply feeding it a hardware description of the targeted accelerator(s).

The main weakness of isomorphism-based pruning is that it is not a true equivalence check. That is, the algorithm only checks that nodes used to represent computation form equivalent graphs, not that they are equivalent computations. For instance, if a DFG represented a multiplication by 10 as a left-shift by 3 bits, a left-shift by 1 bit, and an addition of those two results, then this would not match an accelerator with a multiplier. Additionally, isomorphism pruning will reject DFGs that could potentially map onto accelerators by reexpressing the computation using distributivity, associativity, or other operator properties. In order to recognize that multiple graphs perform the same computation, pruning would have to perform a full equivalence check, typically using BDDs [7] or their relatives (ADDs, BMDs, etc.). This is far more computationally demanding than isomorphism for accelerators of practical size, although an interesting avenue for future work.

The implications of this drawback are twofold. First, the compiler is at the mercy of the software writer to a certain extent. If the algorithm is described in software differently than it is represented in the hardware graph, then the compiler will be unable to accelerate it. Second, accelerator hardware structures that do not map directly to a single node in the DFG are difficult to utilize. For example, a lookup-table is capable of executing any number of consecutive bitwise operations from a dataflow graph. Because of this, there is no equivalent (finite) hardware graph that can represent this computational structure.

This drawback affects both full-enumeration-based and greedy-based compilation algorithms, and leaves room for improvement. However equivalence-based algorithms have proven intractable to this point.

Subgraph Isomorphism Algorithm: The algorithm used to determine isomorphism, Algorithm 1, is based on the backtracking search strategy described in [19], which was itself adapted from [28].

```
1  Input: S' = (V', E'), T = (W, F)
2  foreach v'_i ∈ V' do
3      foreach w_j ∈ W do
4          if v'_i is equivalent to w_j then
5              if dependence_height(v'_i) ≤ dependence_height(w_j) then
6                  M_i = M_i + w_j
               end
           end
       end
   end
7  Call AssignVertex(M, x, 1);

   Procedure AssignVertex(M, x, vertex)
8  if vertex > |S'| then
9      if Subgraph outputs map then
10         return ISOMORPHIC;
       end
11     else
12         return NOT ISOMORPHIC;
       end
   end
13 foreach m_i ∈ M_vertex do
14     edges_match = true;
15     for j = 1..vertex do
16         if e_{v'_j, v'_vertex} ∈ E' and e_{x(v'_j), m_i} ∉ F then
17             edges_match = false ;
           end
       end
18     if edges_match then
19         set x(v'_vertex) = m_i;
20         M' = M;
21         assignment_works = true;
22         for j = vertex + 1..|V'| do
23             M'_j = M'_j − m_i;
24             foreach m_k ∈ M'_j do
25                 if e_{v'_vertex, v'_j} ∈ E' and e_{x(v'_vertex), m_k} ∉ F then
26                     M'_j = M'_j − m_k;
                   end
27                 else
28                     if e_{v'_vertex, v'_j} ∈ E' and k < i then
29                         M'_j = M'_j − m_k;
                       end
                   end
               end
30             if |M'_j| == 0 then
31                 assignment_works = false;
               end
           end
32         if assignment_works then
33             result = call AssignVertex(M', x, vertex + 1);
34             if result == ISOMORPHIC then
35                 return ISOMORPHIC;
               end
           end
       end
   end
36 return NOT ISOMORPHIC;
```

Algorithm 1: Subgraph isomorphism algorithm

The basic idea is to recursively assign one vertex from S', the dataflow subgraph, to a corresponding vertex in T, the hardware graph, and check to ensure that the corresponding edges exist in both graphs whenever a new node is assigned. In order for this algorithm to be computationally feasible, a number of steps are taken to prune the search space.

Algorithm 1 takes the two graphs $S' = (V', E')$ and $T = (W, F)$ as input. In this formulation, V' represents operations in the subgraph, E' dataflow edges in the subgraph, W FUs in the accelerator, and F wires connecting those FUs. Initially, a group of sets, M, are calculated such that M_i contains all vertices in W that are of the same computation type as v_i. Essentially, this step creates a set of candidate nodes in T that each node in S' can be mapped to. For example, if v_1 was an ADD node, M_1 would contain all hardware nodes with addition capabilities in W. This process corresponds to lines 2 - 6 of Algorithm 1. This information is passed to the procedure $AssignVertex$, along with the mapping function, $x()$, and the vertex number to be mapped.

The $AssignVertex()$ procedure iterates over the set of possible nodes (line 13 in Algorithm 1) testing that every edge in E' has a corresponding edge in F for the nodes that have already been mapped (lines 15 - 17). Assuming that the edges match, $x()$ is updated and the sets of potential matches, M, is updated to reflect the new information. This pruning of the search space is critical to avoiding an exponential explosion of runtime.

Two techniques are used to remove nodes from M after a node assignment. The first, lines 25 - 26, looks at all vertices in V' not yet assigned and checks to see if there is an edge in E' connected to the node just assigned, v'_{vertex}. If such an edge exists, any nodes in M that do not have a corresponding edge in F connected with $x(v'_{vertex})$ can be removed from the search space. The second pruning technique (lines 28 - 29) leverages the fact that we are dealing with directed acyclic graphs. When creating S' and T, we impose the restriction that the vertices must be topologically sorted within the sets V' and W. That is to say, $\forall$ vertices i, j such that $i > j, e_{i,j} \notin E$. In other words, there are no edges from vertices with higher order numbers to vertices with lower order numbers. This restriction allows us to remove any vertex from M that has a lower order number than the currently assigned order number, since no such backward edge can exist in F. If at any point during pruning, the size of the candidate set falls to zero (line 30), then it is no longer necessary to examine this part of the search tree. These simple pruning techniques turn an intractable problem into one that is solved much faster than instruction scheduling in our compiler infrastructure.

After pruning the search space, $AssignVertex$ is recursively called to assign the next vertex using the reduced search space, M'. This is continued until all nodes in S' map to corresponding nodes in T though the function $x()$, or it is proven that no such mapping exists. Once a mapping is found, it is still necessary to ensure that the subgraph outputs map onto the targeted accelerator (line 9). This is done using the Dijkstra's algorithm to find the shortest path between nodes producing the outputs and output ports. If this final check passes, then the subgraph can indeed execute on the targeted accelerator.

Improvements Over Previous Work: There are three main algorithmic improvements over previous proposed subgraph isomorphism algorithms. First, as previously mentioned, vertex numbers are assigned topologically to ensure that if an edge exists, then the source number is less than the destination number. This dramatically reduces the sets of potential candidates, M, shown in lines 28 and 29 of Algorithm 1. Topological sorting of vertices has been previously proposed for a different style of isomorphism algorithms [18], but only to generate an initial solution, not to prune the search space.

A second improvement prunes the candidate sets by using dependence height of the candidates (line 5 of Algorithm 1). Dependence height refers to the maximum sized chain of operations that must precede a particular operation in a graph. For example, in Figure 1B, node 10 has a dependence height of 1 since 8 must precede it, and node 11 has a dependence height of 3 since the chain 1-3-6 must precede it. When creating a set of candidates for node 11 in the representative hardware graph, we know that skipping any nodes with dependence height less than 3 will not affect the solution. This optimization also relies on the acyclic nature of

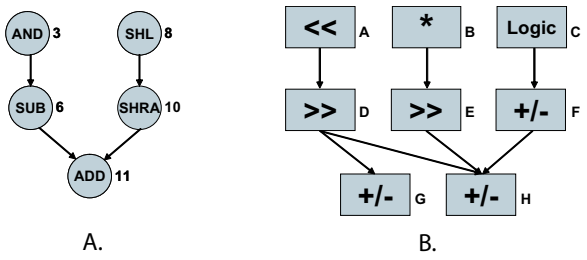

Figure 2: A. Subgraph from Figure 1A to be tested for subgraph isomorphism, B. hardware accelerator being targeted

the graphs that we are matching, and has a dramatic impact on the overall algorithm runtime[1].

The last optimization developed relates to the order in which nodes are assigned. Note that in $AssignVertex$, pruning of M occurs when edges do not match up in the current assign, $x()$. Thus, it is important to make these comparisons as high in the search tree as possible. This is accomplished by assigning vertices in order determined by a depth first search (not shown in Algorithm 1). Unlike the previous two optimizations, this technique is applicable for any style graph, not just directed-acyclic graphs. These three optimizations contribute to make subgraph isomorphism a tractable way to determine whether a dataflow subgraph can execute on a hardware accelerator.

Subgraph Isomorphism Example: Algorithm 1 is complicated and we will hopefully clarify it through the example in Figure 2. Here, the dataflow subgraph in Figure 2A (from Figure 1B) is checked for subgraph isomorphism on the accelerator graph in Figure 2B. First, a set of candidates in Figure 2B is constructed for each vertex in Figure 2A. This corresponds to M in the algorithm. Examining vertex 3, we see that only hardware vertex C can execute logic operations, so $M_3 = \{C\}$. Likewise, $M_6 = \{F, G, H\}$, since any of those hardware vertices could execute the subtraction. The candidate set of vertex 11, $M_{11} = \{G, H\}$ demonstrates the dependence height pruning; F can not be in the solution space because there is only one hardware vertex preceding it. The remaining two sets, $M_8 = \{A\}$ and $M_{10} = \{D, E\}$, are as would be expected.

After the candidate sets are computed, a depth first search is performed (irrelevant of edge directions) to determine the order in which to assign vertices. In this example, the assignment order will be 3, 6, 11, 10, and 8, although this ordering is irrelevant for correctness. $AssignVertex()$ is then called for node 3. The algorithm iterates over the set of candidates, M_3, and updates M for neighbor vertices. In this case, since vertex 6 neighbors vertex 3, M_6 can remove candidates G and H from its set, since neither of those vertices are neighbors of C. Next, $AssignVertex()$ is recursively called to map vertex 6. The algorithm maps vertex 6 to F, since that is the only possibility in M_6. Lines 15-17 of Algorithm 1 check to make sure that since there is an edge from vertices 3 to 6, that there is also an edge from C to F. Vertex G is removed from M_{11}, since there is no edge from F to G, and again $AssignVertex()$ is called for vertex 11. Vertex 11 is mapped to H, and 10 is mapped to E similarly to the previous two nodes. However, once 10 is mapped to E, then the candidate set M_8 becomes empty, since there is no edge from A to E. This bounds the recursion of $AssignVertex()$ which then tries another assignment for vertex 10, D. Using this mapping, vertex 8 can be assigned to A, which will complete the mapping, and prove that there is a subgraph of Figure 2B that is isomorphic to Figure 2A.

[1]Although some of the improvements over previous isomorphism algorithms rely on the acyclic nature of the subgraphs targeted, they can easily be extended to cyclic graphs by treating the backward edges in cyclic subgraphs separately from forward edges.

```
1  Input: boolean matrix M, where M_{i,j} = true if op i is in subgraph j
2  Output: A vector x, x_i ∈ {0,1}^n, where Mx = (1,1,1,...,1)^T and
       Σ_{i=1}^n x_i is minimized

3  Sort columns of M in order of decreasing size
4  Call Cover(1, true, M, x);
5  Call Cover(1, false, M, x);

   Procedure Cover(subgraph, add_subgraph, M, x)
6  if add_subgraph then
7      if (Mx && (M_{1,subgraph}, M_{2,subgraph}, ...M_{m,subgraph})^T) ≠
       (0,0,0...0)^T then
           // Subgraph overlaps with the partial solution.
8          return;
       end
9      x_{subgraph} = 1;
10     if Mx == (1,1,1,...1)^T then
11         if Σ_{i=1}^n x_i < fewest_subgraphs then
12             fewest_subgraphs = Σ_{i=1}^n x_i;
13             best_solution = x;
           end
           // Found a complete cover.
14         return;
       end
   end
15 if subgraph + 1 > n then
       // Did not find a complete cover after examining all subgraphs.
16     return;
   end
17 if Σ_{i=1}^n x_i + (m − Σ_{i=1}^n (Mx)_i) / (Σ_{i=1}^m M_{i,subgraph}) ≥ fewest_subgraphs then
       // The current solution cannot possibly be the best.
18     return;
   end
19 Call Cover(subgraph + 1, true, M, x);
20 Call Cover(subgraph + 1, false, M, x);
```

Algorithm 2: Unate covering selection algorithm

3.2.3 *Selection Using Unate Covering*

Now that we have a set of subgraphs that *can* execute on the accelerator, it is necessary to select which ones *to* execute on the accelerator. In standard greedy solutions, this step is implicit within enumeration: each enumerated subgraph is automatically selected. However, greedy selection can also be performed in conjunction with full enumeration algorithms, e.g., in [27]. Greedy selection algorithms, typically map the largest subgraph onto the application, remove all overlapping subgraphs from the consideration, and then repeat this process until no more candidates remain. The problem with this technique is that it will provide suboptimal results whenever the largest subgraph is not part of the best solution.

Instead of a greedy heuristic, we propose solving the selection problem by converting it to a unate covering. Informally speaking, unate covering problems operate on a Boolean matrix, M, where the rows represent vertices in a DFG, and the columns represent subgraphs; if the value of $M_{i,j}$ is true, this means that operation i occurs in subgraph j. Traditionally, the goal of unate covering is to find a set of columns (or subgraphs) with minimal cost, such that each operation is covered at least once. In this formulation, the cost of a subgraph could be a variety of things, such as the number of cycles needed to execute on a particular accelerator or the power consumed by a subgraph. As with using subgraph-isomorphism for the pruning algorithm, unate covering was chosen for selection because there is much prior work [10, 12] that can be leveraged to make this problem tractable.

Before discussing the details of our unate covering algorithm, Algorithm 2, it is important to point out one difference between this and standard unate covering formulations. Traditionally, unate cov-

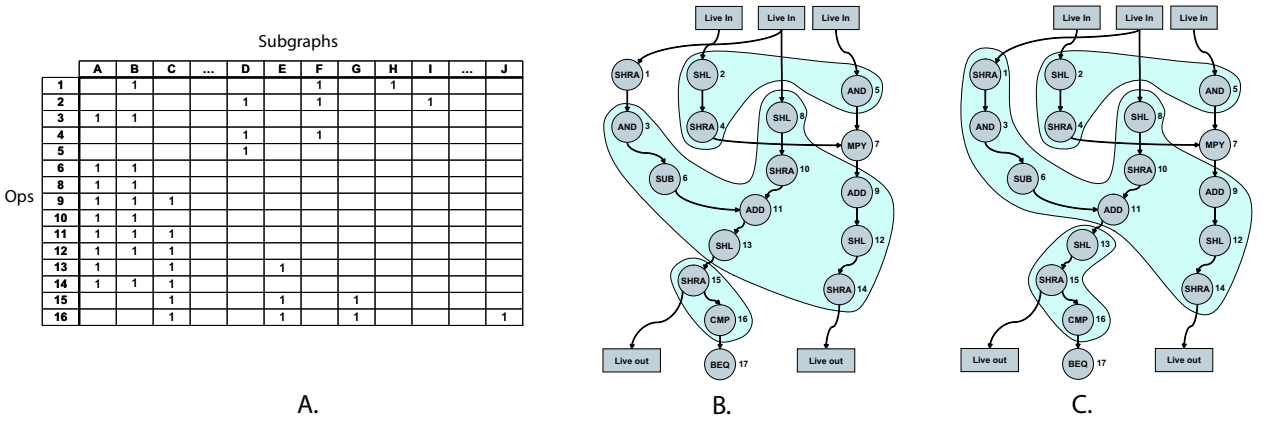

Figure 3: A. Example unate covering problem used to map subgraphs from the basic block in Figure 1. B. The mapping solution with full-enumeration and greedy selection. C. Mapping solution with full-enumeration and unate covering selection.

ering allows an operation to appear in multiple subgraphs in the final code. However, we have made the decision to disallow this possibility[2]. Allowing an operation to appear in multiple subgraphs essentially replicates the computation and will unnecessarily increase power consumption. The downside is that disallowing overlapping subgraphs can hurt application performance in multi-issue processors, and actually makes the covering search space much larger. Performance loss can occur because the first operation in a subgraph has to wait for all subgraph inputs to be ready before being executed. The covering search space becomes larger, because many techniques to prune the space, such as row and column dominance, no longer work if overlap is not allowed. Despite the changes resulting in a large search space, the runtimes of our unate covering formulation are quite reasonable for practical inputs, and the resulting code will be more suitable for embedded systems.

Unate Covering Algorithm: The algorithm used to perform unate covering based selection is shown in Algorithm 2. As previously mentioned, input to the algorithm is a m by n Boolean matrix, where rows correspond to operations and columns to subgraphs. The output of this algorithm (line 2) is a vector, x, where $x_5 = 1$ means that subgraph 5 is in the optimal cover. The constraint $Mx = (1, 1, 1, ..., 1)^T$ ensures that each operation is covered by exactly one subgraph. Note that the standard unate covering constraint, which allows overlap, is $Mx \geq (1, 1, 1, ..., 1)^T$. To ensure that a solution is feasible, each individual node is inserted into M as a subgraph which covers only one operation. Once M is constructed, the columns are sorted in decreasing order, and a standard branch-and-bound algorithm, $Cover()$, is called.

Inside the function $Cover()$, one subgraph is considered for addition to the current cover, x. Line 7 in Algorithm 2 tests to see if there is any overlap between the current cover and the candidate subgraph. The Mx matrix multiplication creates a column vector of the current set of ops that are covered, and $M_{i,subgraph}$ is the set of ops covered by $subgraph$. Assuming there is no overlap, line 9 adds $subgraph$ to the current cover, and then the cover is tested to see if all ops are covered (line 10). If a complete solution exists, the total number of subgraphs is calculated, and if it is the fewest yet seen, then this cover is recorded as being the best. Note that if there were multiple accelerators in the targeted processor, the notion of what constitutes the "best" solution (line 11), could easily be expanded to include column weights based on which accelerator a subgraph used.

If the $Cover()$ function does not have a complete solution, then two checks are performed to prune the search space before recursing down the search tree (lines 15 - 18). The first check, lines 15 and 16, simply bounds the search tree when it runs out of subgraphs to examine: essentially when it hits leaves of the tree. The second check bounds when the current solution cannot possibly be better than the best known solution, by computing a lower bound on the partial cover, x. The first portion of line 17, $\sum_{i=1}^{n} x_i$, calculates how many subgraphs are in the current cover. The second portion of the equation calculates the number of ops that still need to be covered and divides by the number of ops covered by the current subgraph. Since the subgraphs are sorted by size, and they are always added in order of decreasing size, the second portion of the equation gives a lower bound on the number of additional subgraphs that must be added to complete a cover. The check in line 17 is the primary catalyst that makes this unate covering algorithm practical for subgraph selection.

Improvement Over Previous Work: As with the isomorphism algorithm, there are several techniques that make this unate covering algorithm faster than previous solutions. The first of these is sorting the subgraphs in order of decreasing size (line 3 of Algorithm 2). While this does not directly prune the search tree, it does enable other pruning techniques, such as the check in line 17. Another technique is to always branch toward adding a subgraph first (lines 4 and 19). Since the subgraphs are sorted by size, and the subgraphs are considered in consecutive order, always adding ensures the first complete cover will be exactly the same as the greedy solution. The greedy solution provides an excellent bound to quickly prune bad portions of the search tree. Additionally, by reaching the greedy solution first, if the algorithm runs for an unusually long time, it can always be stopped at without fear of a solution worse than greedy.

Unate Covering Example: Figure 3A shows an example of the boolean matrix, M, used in Algorithm 2. This matrix shows several subgraphs that were enumerated from the basic block from g721encode, shown in Figure 1B (many subgraphs were omitted for space and clarity reasons). The subgraphs correspond to an accelerator which has 4 inputs, 2 outputs, and can support any computation with a dependence chain of 4 or less, also pictured in Figure 1. Notice how the subgraphs are sorted from largest at the left

[2]Technically, this restriction turns the unate covering into a binate covering, which is fundamentally more difficult than unate covering. However, this formulation has two characteristics missing from generalized binate covering, which makes this formulation easier to solve: a solution is guaranteed to exist, and adding subgraphs to the cover will never make the solution infeasible. The general formulation allowing operations to appear in multiple subgraphs is a true unate cover.

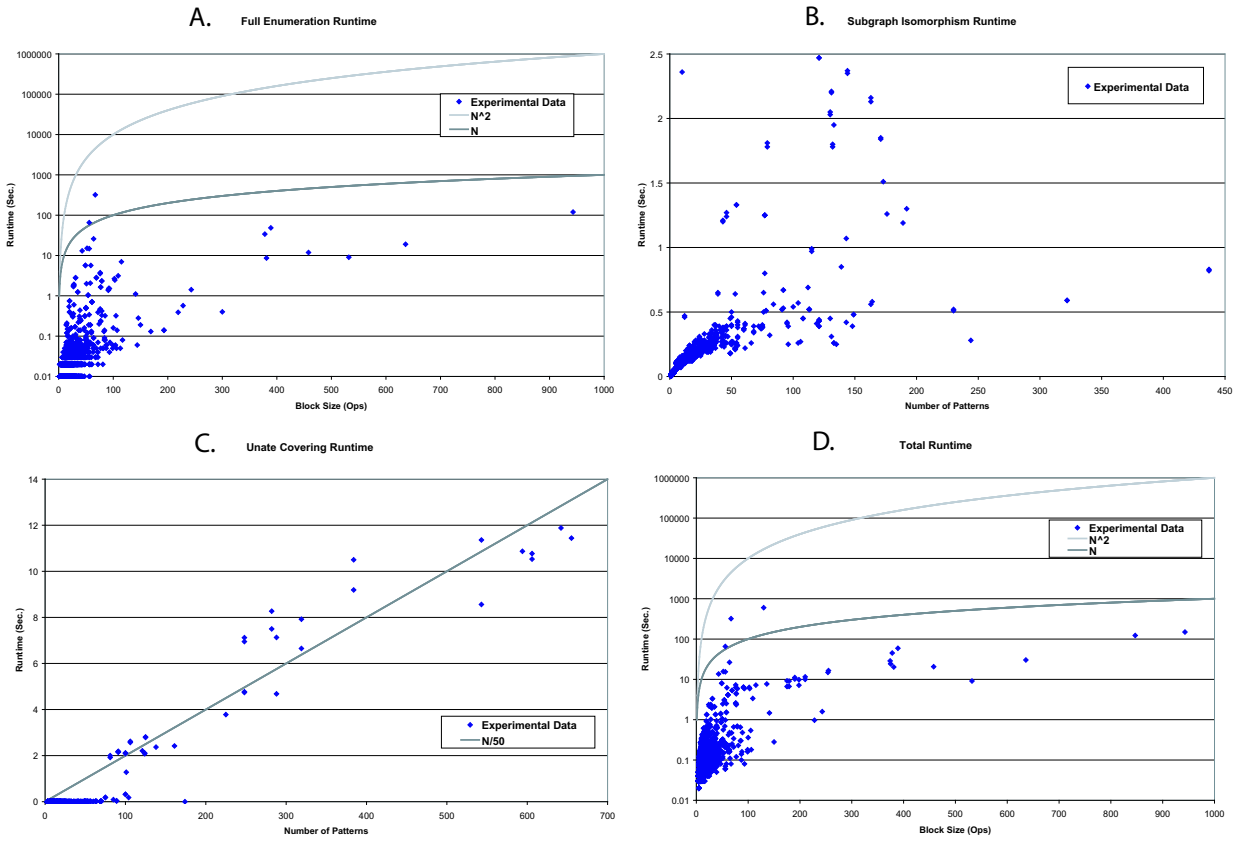

Figure 4: Compilation runtimes for various aspects of the proposed algorithms (each dot represents one basic block from an application). A. Full enumeration runtimes, B. Subgraph isomorphism runtime, C. Unate covering runtime, D. Total runtime

(covering 9 operations), to the smallest at the right (each operation node as a subgraph).

Algorithm 2 begins the $Cover()$ function by adding subgraph A, the largest subgraph, to its current cover, x. It will then recurse, and attempt to add B to x. The check at line 7 of Algorithm 2 will prevent this since the two subgraphs overlap, and this branch of the search space will be pruned. Eventually, by moving across the matrix in Figure 3, subgraphs D, then G, and then H will be added to A to create a complete cover, shown in Figure 3B. This is the full-enumeration / greedy-selection solution. Assuming a single-issue processor, and the accelerator and each operation in Figure 1B takes one cycle to execute, this solution will yield a speedup of $\frac{17}{3+3} = 2.83$ for this block. The first 3 in the denominator accounts for the right-shift, branch, and multiply that were not accelerated, and the second 3 is for each of the 3 subgraphs that will be run on the accelerator.

After the unate covering algorithm finds the greedy-selection solution, it will continue to explore the search tree and eventually discover the cover B, D, E, shown in Figure 3C. This solution uses fewer subgraphs, and will be recorded as the best solution on line 13 of Algorithm 2. The speedup for this solution is $\frac{17}{2+3} = 3.4$. This compares quite favorably with the speedup obtained using the greedy enumeration - immediate selection described in Section 3.1, which is only $\frac{17}{7} = 2.43$. Clearly, full-enumeration with unate-covering based selection can provide benefits beyond greedy heuristics.

3.2.4 *Algorithm Runtimes*

There are clearly performance benefits over the standard greedy algorithms if accelerators can be targeted using the NP-Complete formulations that we have proposed. The major concern is whether the proposed algorithms are tractable. Figure 4 demonstrates that they are.

Each point in these graphs represents the algorithm runtime of a basic block from 1 of 23 MediaBench and MiBench applications. The data was collected on a 3.06 GHz Pentium 4 machine with 1 GB of RAM. Applications were compiled to target an accelerator with 4 inputs, 2 outputs, and a maximum dependence height of 4 (similar to the accelerator proposed in [8]). Each algorithm was given a maximum time limit of 600 seconds per block, at which point the algorithm was terminated and reported the best solution seen up to that point. Note that *only one basic block out of 23 applications* reached the time limit for any of the proposed algorithms; that was during subgraph enumeration.

To summarize the results for subgraph enumeration, more than 99.8% of basic blocks were fully enumerated in less than 1 second, and more than 99.95% of the blocks were enumerated within 10 seconds. As mentioned previously, the worst case block timed out at 600 seconds. This could be prevented by more aggressively partitioning the block into smaller components. Overall, the enumeration algorithm runtime appeared to grow only *linearly* with the size of the basic block, which makes this algorithm quite scalable.

Runtimes for the subgraph isomorphism algorithm were also very reasonable. More than 99.7% of blocks had subgraph isomorphism checked for all their enumerated subgraphs in less than 1 second. The worst case runtime for any of the blocks was only 2.47 seconds.

As with subgraph enumeration, runtime for unate covering grew roughly linearly with the size of its input matrix, and the runtime

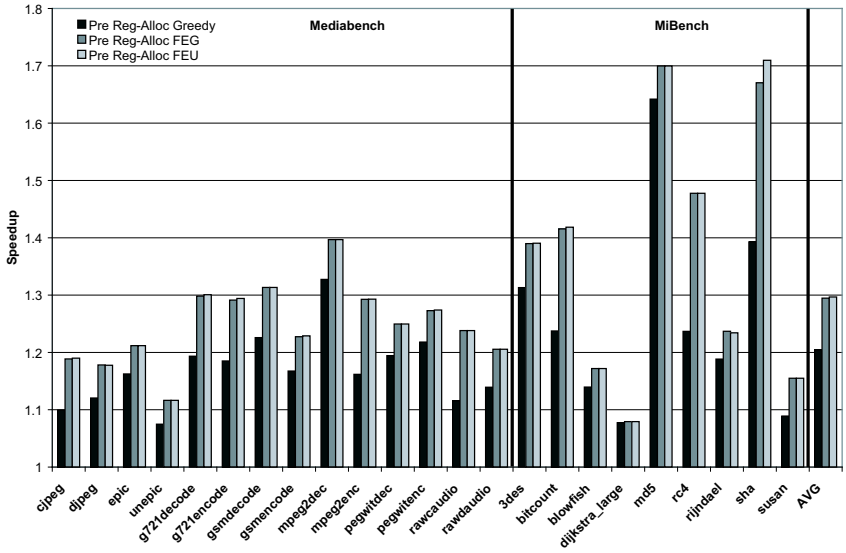

Figure 5: Comparison of subgraph mapping algorithms

was very fast in the common case. More than 99.1% of blocks ran unate covering selection in less than 1 second, while 99.8% finished in less than 10 seconds. The worst case runtime for any block was 60.25 seconds (this was the same block that timed out during enumeration).

In terms of total runtime for all three phases (full enumeration, isomorphism based pruning, and unate covering selection), more than 98% of blocks took less than 1 second to run. 99.5% of basic blocks took less than 10 seconds total. The worst case block out of the 23 applications took 11.03 minutes. If the worst case block proved too slow, the algorithms were designed so that the time-out could be reduced and still guarantee a solution no worse than greedy.

In the benchmarks examined, the majority of basic blocks were so small that the proposed algorithmic improvements made very little difference in the block's compilation time. However, the improvements were key in making runtimes for degenerate blocks tractable. Overall, the average benchmark compilation time was 43.8 minutes without the proposed algorithmic improvements. This figure includes 3 benchmarks that were stopped after not finishing in 6 hours. With the algorithmic improvements proposed in this work, average compilation time was more than an order-of-magnitude better at 3.6 minutes per benchmark (15.9 minutes worst case).

These results show that if you are compiling to target an acyclic accelerator statically, runtime is no reason to use a greedy heuristic.

4. EXPERIMENTS

In order to evaluate the proposed mapping algorithm, an experimental framework was built using the Trimaran research compiler and SimpleScalar ARM simulator. Trimaran was retargeted for the ARM instruction set and subgraphs to be accelerated were delineated in the binary. After compilation, the simulator recognized the subgraphs and modeled them as if an accelerator was present. SimpleScalar was configured to represent an ARM-926EJ [2], a popular embedded core, with accelerators that took one cycle to execute.

Twenty three benchmarks from MediaBench and MiBench were used to evaluate the proposed mapping algorithms. Omitted benchmarks were due to issues in the compiler infrastructure, not limitations of the subgraph mapping algorithm. We tested three different algorithms: greedy enumeration - immediate selection (as described in Section 3.1), full enumeration - unate covering selection, or FEU (described in 3.2), and a hybrid technique full enumeration - greedy selection, or FEG.

Algorithm Comparison: Figure 5 shows the speedups attained when using the three proposed algorithms to target the 4 input / 2 output accelerator shown in Figure 1A. The figure illustrates that the FEU algorithm consistently outperforms greedy on nearly every benchmark. On average, 10% more speedup was achieved by using the FEU algorithm instead of greedy heuristics. Sha showed the largest difference between greedy and FEU, a 32% improvement. The primary reason for this is that full enumeration identified a considerable number of disconnected subgraphs in the critical loop, which the greedy algorithm was not capable of finding. Dijkstra_large showed the least improvement when moving from greedy to FEU mapping. The important subgraphs in this benchmark only consist of 2 back-to-back instructions, thus the subgraphs are easy to identify regardless of enumeration algorithm. As would be expected, this shows that computation-bound applications with very large basic blocks benefit more from the FEU algorithm than applications with small basic blocks.

One surprising result illustrated in Figure 5 is that most applications did not benefit from unate covering selection (comparing FEG with FEU). On average, FEU performed only 1% better than FEG. The main reason for this is that the critical computation in most basic blocks was small enough that very few subgraphs were needed in the cover. If more subgraphs are used to cover the DFG (for example, when targeting a smaller accelerator), then greedy selection is more likely to get stuck in a local minima and perform worse. However, when targeting the large accelerator from Figure 1A, greedy selection is sufficient. In two instances, djpeg and rijndael, unate covering selection actually caused slight performance decreases. This is due to second-order effects, such as cache alignment, that are not modeled by the unate covering formulation.

Sensitivity to Targeted Accelerator: Figure 6 shows how much better FEU performs relative to greedy when varying the targeted accelerator. Bars greater than one imply FEU performed better than greedy and bars less than one imply greedy performed better than FEU. The rightmost bar for each benchmark represents the 4 input / 2 output accelerator used throughout this paper. The 3 input / 1 output accelerator consists of two back-to-back FUs, and is modeled

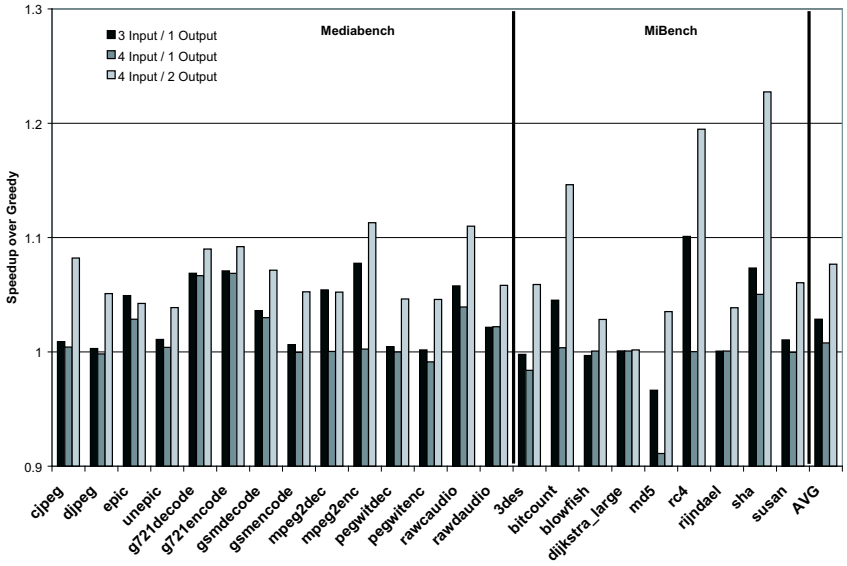

Figure 6: The speedup of Full-Enumeration/Unate Covering Selection over Greedy while varying the targeted accelerator

after the accelerator used in [16]. The 4 input / 1 output accelerator has computation capabilities between the others, with 7 FUs and maximum dependence height of 3.

There are several interesting trends illustrated by this figure. First note that FEU outperforms greedy much more on the 4/2 configuration than on the 4/1 or the 3/1. This is because accelerators with only one output preclude disconnected subgraphs from being executed. If no disconnected subgraphs are allowed, then greedy can potentially find the same subgraphs as full enumeration. This definitely helps narrow the gap between the two algorithms. In general, larger accelerators with multiple inputs and outputs place more importance on high quality subgraph enumeration.

A second important trend in Figure 6 is that FEU outperforms greedy more in 3/1 than in 4/1. The reason for this is that the small number of FUs in 3/1 (only 2) made the number of subgraphs selected in the final cover relatively high, compared with 4/1 which has 7 FUs. Since more subgraphs are needed, more emphasis is placed on the covering algorithm, and unate covering helped quite a bit. The 4/1 accelerator used relatively few subgraphs, that were all discoverable via greedy enumeration, therefore FEU provided little benefit beyond the greedy algorithm. This shows that more thorough strategies, used in FEU, are more important whenever the search space is very large.

A last trend to note in Figure 6 is that in certain benchmarks, such as md5, greedy actually performed better than FEU. This is due to the partitioning used during full enumeration. Recall that in order to make full enumeration tractable, some very large blocks sometimes have to be partitioned into smaller graphs. Occasionally, this partitioning precludes full enumeration from finding important subgraphs which can be discovered by greedy methods. This problem is pronounced in accelerators with only one output, since full enumeration cannot make up ground on greedy by using disconnected subgraphs. Figure 6 motivates future work to develop faster enumeration algorithms and better partitioners to alleviate the problem in md5.

Effect of Register Allocation: Figure 7 depicts the result of applying the FEU mapping algorithm before and after register allocation. This is an important result because many researchers have proposed subgraph mapping in virtual machines or as a part of binary-to-binary translation. The drawback of subgraph mapping after register allocation is that spill code essentially breaks dataflow edges by placing values in memory. This limits the size of computation subgraphs that can be identified for acceleration. On the other hand, register allocation does introduce some additional computation (e.g., stack adjustments) that could potentially be accelerated, which is not available when mapping before allocation.

On average, we found that performing subgraph mapping prior to allocation produced results with 8% more speedup than post-allocation mapping. In some benchmarks, like rawcaudio, the innermost loop was so small that there was no spill code, and so there was no difference in the results. In other benchmarks, such as 3des, the amount of spill code was so large that virtually none of the pre-allocation subgraphs were discoverable post-allocation. Only one benchmark, epic, performed better from post-allocation mapping. Figure 7 clearly shows that performing subgraph mapping pre-allocation in the compiler is much more effective than post compilation techniques, such as binary translation.

5. CONCLUSION

In this work, we addressed the inefficiencies of traditional compiler algorithms used to identify candidate subgraphs for execution on computation accelerators. Several new algorithms were developed to find better candidates for both small and large accelerators. These algorithms comprised enumerating subgraphs in a dataflow graph, using subgraph isomorphism to prune invalid subgraphs, and using unate covering to select which valid subgraphs to execute on the targeted accelerators. Simulation results demonstrate that our proposed algorithms achieve, on average 10%, and as much as 32% more speedup than traditional greedy solutions.

This work also quantified the effect of register allocation on subgraph identification. On average, performing subgraph mapping prior to register allocation results in 8% more speedup. This result implies that performing dynamic subgraph identification in hardware or a virtual machine would reduce the effectiveness of mapping algorithms.

6. ACKNOWLEDGMENTS

Much gratitude goes to the anonymous referees who provided excellent feedback on this work. This research was supported by ARM Limited, the National Science Foundation grant CCF-0347411, and equipment donated by Hewlett-Packard and Intel Corporation.

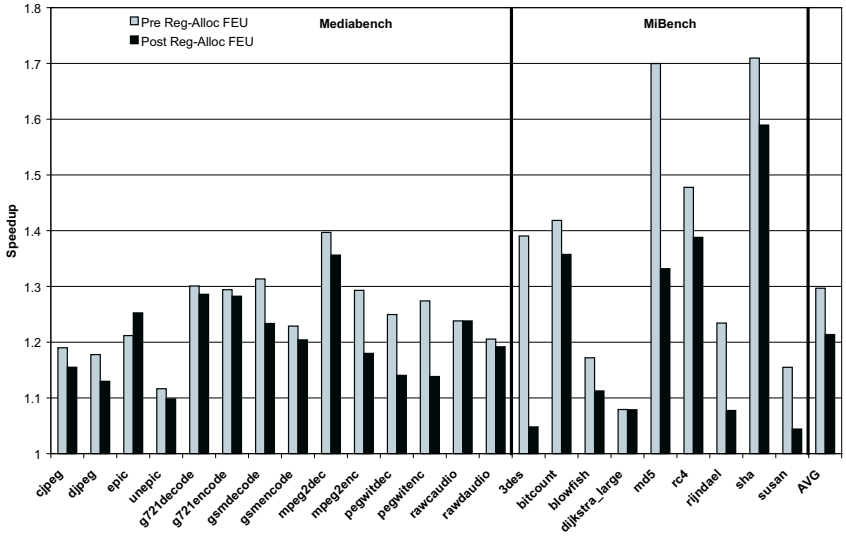

Figure 7: Comparison of mapping effectiveness before and after register allocation using the accelerator from Figure 1A

7. REFERENCES

[1] A. Aho, M. Ganapathi, and S. Tijang. Code generation using tree pattern matching and dynamic programming. *ACM TOPLAS*, 11(4):491–516, Oct. 1989.

[2] ARM Ltd. *ARM926EJ-S Technical Reference Manual*, Jan. 2004. http://www.arm.com/pdfs/DDI0198D_926_TRM.pdf.

[3] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proc. 40th DAC*, pages 256–261, June 2003.

[4] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi, and P. Ienne. ISEGEN: Generation of high-quality instruction set extensions by iterative improvement. In *Proc. 2005 DATE*, pages 1246–1251, 2005.

[5] A. Bracy, P. Prahlad, and A. Roth. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *Proc. 37th MICRO*, pages 18–29, Dec. 2004.

[6] P. Brisk et al. Instruction generation and regularity extraction for reconfigurable processors. In *Proc. 2002 CASES*, pages 262–269, 2002.

[7] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[8] N. Clark et al. An architecture framework for transparent instruction set customization in embedded processors. In *Proc. 32nd ISCA*, pages 272–283, June 2005.

[9] N. Clark, H. Zhong, and S. Mahlke. Automated custom instruction generation for domain-specific processor acceleration. *Trans. on Computers*, 54(10):1258–1270, 2005.

[10] R. Cordone, F. Ferrandi, D. Sciuto, and R. W. Calvo. An efficient heuristic approach to solve the unate covering problem. In *Proc. 2000 DATE*, pages 364–371, 2000.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[12] E. Goldberg, L. Carloni, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Negative thinking in branch-and-bound: the case of unate covering. *IEEE TCAD*, 19(3):281–294, Mar. 2000.

[13] D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *Proc. 2003 CASES*, pages 137–147, 2003.

[14] J. R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In *Proc. 5th FCCM*, pages 12–21, Apr. 1997.

[15] S. Hu, I. Kim, M. H. Lipasti, and J. E. Smith. An approach for implementing efficient superscalar cisc processors. In *Proc. 12th HPCA*, pages 213–226, 2006.

[16] S. Hu and J. E. Smith. Using dynamic binary translation to fuse dependent instructions. In *Proc. 2004 CGO*, pages 213–226, 2004.

[17] I. Huang and A. M. Despain. Synthesis of application specific instruction sets. *IEEE TCAD*, 14(6):663–675, June 1995.

[18] R. Kastner et al. Instruction generation for hybrid reconfigurable systems. *TODAES*, 7(4):605–627, Apr. 2002.

[19] E. Krissinel and K. Henrick. Common subgraph isomorphism detection by backtracking search. *Software: Practice and Experience*, 34(6):591–607, 2004.

[20] R. Leupers and P. Marwedel. Instruction selection for embedded DSPs with complex instructions. In *Proc. 1996 EuroDAC*, pages 200–205, Sept. 1996.

[21] S. Liao et al. Instruction selection using binate covering for code size optimization. In *Proc. 1995 ICCAD*, pages 393–399, 1995.

[22] N. Malik, R. Eickenmeyer, and S. Vassiliadis. Interlock collapsing ALU for increated instruction-level parallelism. In *Proc. 25th MICRO*, pages 149–157, Dec. 1992.

[23] K. V. Palem, S. Talla, and W.-F. Wong. Compiler Optimizations for Adaptive EPIC Processors. In *Proc. 2001 EMSOFT*, pages 257–273, 2001.

[24] A. Peymandoust et al. Automatic instruction set extension and utilization for embedded processors. In *14th ASAP*, pages 108–120, June 2003.

[25] J. Phillips and S. Vassiliadis. High-performance 3-1 interlock collapsing ALU's. *IEEE Trans. Comput.*, 43(3):257–268, 1994.

[26] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable function units. In *Proc. 27th MICRO*, pages 172–180, Dec. 1994.

[27] P. Sassone, D. S. Wills, and G. Loh. Static strands: safely collapsing dependence chains for increasing embedded power efficiency. In *Proc. 2005 LCTES*, pages 127–136, June 2005.

[28] J. R. Ullman. An algorithm for subgraph isomorphism. *JACM*, 23(1):31–42, 1976.

[29] S. J. Weber and K. Keutzer. Using minimal minterms to represent programmability. In *Proc. 2005 CODES/ISSS*, pages 63–68, 2005.

[30] Z. A. Ye et al. CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proc. 27th ISCA*, pages 225–235, 2000.

[31] S. Yehia et al. Exploring the design space of LUT-based transparent accelerators. In *Proc. 2005 CASES*, pages 11–21, Sept. 2005.