Instruction Set Synthesis Using Operation Pattern Detection

Marnix Arnold and Henk Corporaal

Computer Architecture Laboratory
Department of Electrical Engineering
Delft University of Technology
{marnix, heco}@cardit.et.tudelft.nl

Keywords: pattern matching, co-design, design space exploration, instruction set synthesis.

Abstract

An important problem in the area of processor design for embedded systems is determining the proper instruction set architecture. Trade-offs have to be made between programmability and reusability of dedicated hardware for special functionality on the one hand, and a high performance dedicated instruction set on the other hand. This paper addresses the question of how to find specialized instruction set extensions for a set of applications.

We describe the application of a new pattern matching technique to the problem of the identification of recurring patterns of operations. By implementing frequently occurring operation patterns in hardware, and using this hardware as special function units, a fine-grained hardware/software partitioning can be found. The fine granularity, and the fact that patterns are taken from a number of different target applications rather than a single one, increase the opportunities for reuse of the special-purpose hardware. We illustrate our technique with experiments on a number of benchmarks from the DSP domain.

1 Introduction

Hardware/software co-design is often performed on a per-application basis, yielding systems that are highly application-specific. The approach taken is usually a coarse-grained one: entire functions of the application are mapped either in hardware or software [4]. Any hardware thus generated is only reusable by other applications if those include the same function. For application-specific systems this is obviously not a problem. For application-*domain*-specific, reprogrammable systems, however, we may want to increase the granularity of the hardware/software partition, to increase the chances of hardware reuse. It is with this in mind that we consider a partitioning approach that centers on groups of instructions rather than entire functions. In this paper, we present a new algorithm for the automatic, on-the-fly detection of groups (patterns) of instructions as they occur in the application(s) that we want to generate an execution engine for. Patterns that occur frequently among the target applications can then be considered for implementation in hardware.

Section 2 discusses work from the related areas of instruction set synthesis, technology mapping and code generation. An overview of the algorithm used for the detection of recurring operation patterns is given in section 3. Experiments and their results are described in section 4. We conclude this paper with section 5.

2 Related Work

Pattern matching techniques have been around for quite some time, originating from the string matching problem [1]. Keutzer [5] first applied pattern matching techniques to the problem of technology mapping, noting similarities with the code generation problem [2]. Recent work by Kukimoto [6] extended these techniques to allow for rooted-DAG-shaped subject graphs, as opposed to tree-shaped graphs. However, no matching techniques are available that deal with patterns that are not trees or single-output DAGs [8]. We would like to detect and exploit such patterns, though, so we were forced to come up with a new matching algorithm [3].

The work in this paper is somewhat related to the field of instruction set synthesis. Liem e.a. [7] use matching and covering techniques to identify recurring instances of patterns from a predefined library, rather than constructing this library automatically on-the-fly, as we will do. The search for new operation patterns as described in [9] is limited to chains (sequences) of operations, whereas we consider patterns of any shape.

3 Algorithm Overview

For the matching of pattern graphs that are not trees, conventional techniques are not suitable. For this reason, we have come up with a new matching algorithm, based on the principle of *incremental matching*. In this section we will give an overview of the matching algorithm and how it can be extended to automatically construct a library of recurring patterns. The covering algorithm we employ in section 4 will only be discussed briefly. A comprehensive description of the algorithms can be found in [3].

3.1 Incremental Matching

Given a subject graph G_{sub} and a library of pattern graphs $PatLib = \{G_{pat_i}\}$, as shown in figure 2, we iterate over all operation nodes in G_{sub} (nodes I, II and III), in no particular order. Each of the subject graph's operation nodes N_{sub} is matched against all the pattern graphs' operation nodes N_{pat}. A pair of operation nodes (N_{sub}, N_{pat}) match if they meet all of the following criteria:

- The opcodes of N_{sub} and N_{pat} are the same.
- For the pair of result operands (O_{sub}, O_{pat}) of N_{sub} and N_{pat}, respectively:
  - $NumUses(O_{sub}) \geq NumUses(O_{pat})$.
  - $NumUses(O_{sub}) = NumUses(O_{pat})$, unless O_{pat} is a pattern output.
- For each pair of corresponding inputs (O_{sub}, O_{pat}) of N_{sub} and N_{pat}, respectively[1]:
  - $NumUses(O_{sub}) \geq NumUses(O_{pat})$.
  - $NumUses(O_{sub}) = NumUses(O_{pat})$, unless O_{pat} is a pattern input or output (not an internal operand node).
- For each pair of corresponding inputs (O_{sub}, O_{pat}) of N_{sub} and N_{pat}, respectively, if O_{pat} is an immediate value (literal) rather than a register reference, then O_{sub} must be an immediate of the same value. Note that the converse does not have to hold, since it is always possible to map the value of an immediate (O_{sub}) into a register (O_{pat}).

[1]For commutative operations, more than one pair can be formed for each input. These pairs are checked separately.

where $NumUses(a)$ denotes the number of times operand node a is used as an input to an operation node.

Figure 1 illustrates how the matching criteria can keep the number of partial matches under control (we will show later why this is important). If we consider the pair of operation nodes (I, n_1) in figure 1a, we see that $NumUses(c) < NumUses(o_4)$, which violates the second criterion. Also, if we consider the pair (II, n_3), we see that for the operand pair (c, o_4) the fourth criterion is violated. Violations of the third and fifth criterion are illustrated in figure 1b, where $NumUses(d) > NumUses(o_3)$, but o_3 is an internal operand node.

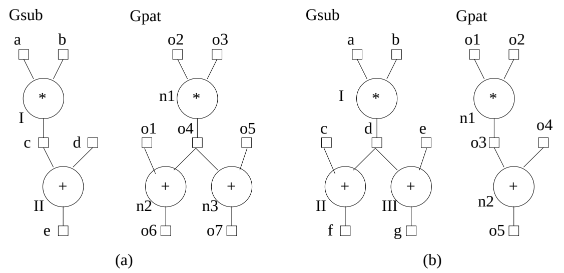

Figure 1: Examples of graphs failing the matching criteria.

In figure 3, the *partial matches* m_1 and m_2 constructed for pattern graph G_{pat4} are shown (there would, of course, also be partial matches for the other patterns). As can be seen, a match is really nothing more than two reference vectors that refer to subject operations and operands, respectively. The position a refence occupies in a reference vector indicates with which pattern operation (operand) it corresponds to.

Looking at the partial matches shown in figure 3, we notice that the matches m_1 and m_2 share the reference to subject operand node c. These two matches can be combined into a new match m_3, which is the union of the two. This match has no empty spaces in its reference vectors and is said to be *complete*, otherwise it is still a partial match. In the same manner, we can combine partial matches for other patterns into complete matches, regardless of the size and shape

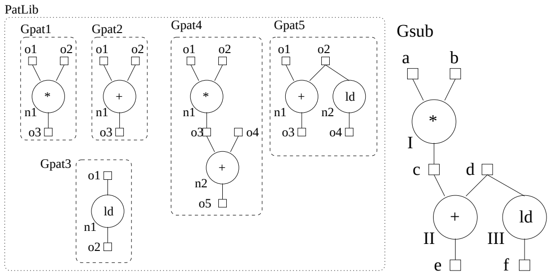

Figure 2: Example of pattern and subject graphs.

of those patterns. Indeed, pattern graph G_{pat5}, which is a multiple-output DAG, is matched in the exact same way as graph G_{pat4} from our example.

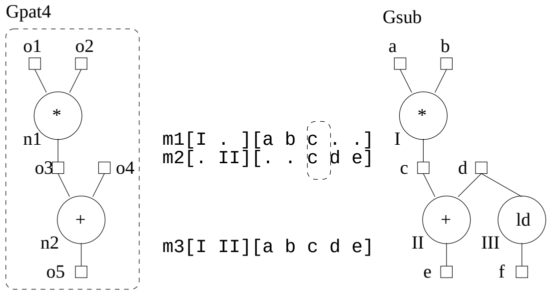

Figure 3: Partial matches m_1 and m_2 merge into the complete match m_3.

A note on the complexity of the matching algorithm: each of the subject graph's operation nodes is matched against each of the pattern graphs' operation nodes. New, partial matches are created each time an (N_{sub}, N_{pat}) pair matches. Subsequently, the newly created partial match is checked against *all* available partial matches (refering to the same pattern) on the current N_{sub}'s input (one or two) and output (zero or one) operands. Whenever the new match can be successfuly merged with another match, a new, more complete match is created (c.f, match m_3 in figure 3). We then have to check if we did not already construct this match in some other way (by merging a different pair of partial matches), by checking its equivalence against, again, all matches on the current N_{sub}'s inputs and outputs. If it is not equivalent to any existing match, we can add the new match to the graph.

The complexity of the matching algorithm can be formulated as:

$$\mathcal{O}(N_s \times N_p \times (1 + P_{mt} \times M \times (1 + P_{mg} \times M))) \quad (1)$$

where:

- N_s is the number of operation nodes in the subject graph.
- N_p is the number of operation nodes in all pattern graphs.
- P_{mt} is the chance of a match between the current subject and pattern operation nodes.
- P_{mg} is the chance of a successful merge of the newly created partial match and the current existing match.
- M is the number of matches present on the operands of the current subject graph operation node.

In the worst case, the complexity of the algorithm is quadratically dependent on M. In order to minimize the computational effort that has to be spent in the inner loop, as well as the memory required to store all the partial matches, it is important that the number of partial matches that are created is kept to a minimum[2]. It is for this reason that we use the rather elaborate set of matching conditions stated earlier in this section.

3.2 Library Construction

In the previous section, we saw that our new matching algorithm can handle patterns of any size and shape, whereas conventional matching algorithms only operate on tree-shaped patterns. Another drawback of using a conventional matching algorithm is that it requires the pattern library to be defined beforehand. This implies that we need some advance knowledge of which patterns to expect, but such knowledge may not be available. We will show that this problem can be overcome by extending the incremental matching algorithm to include automatic library construction capabilities.

As before, we start with a subject graph G_{sub} and a library of pattern graphs $PatLib = \{G_{pat_i}\}$, as shown in figure 4. The difference with the previous example is that the initial pattern library now only contains a minimal set of patterns: just the ones with a single operation node. When we start processing the subject graph's operation nodes, we will initially only construct matches for the single-operation pattern graphs. Whenever we finish constructing partial matches on an operation node, we can now look for opportunities to create new pattern graphs.

[2]Note that P_{mg}, the chance of a merge of two matches, decreases if there is an increase in the number of 'unmergeable' matches due to a large number of pattern graphs.

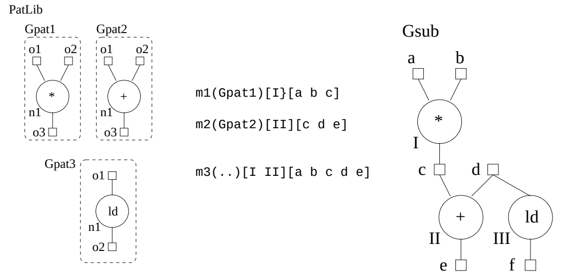

Figure 4: Pattern construction by combining matches.

In the example of figure 4, after we finish processing operation nodes I and II (again in an arbitrary order), we see that two matches m_1 and m_2 both contain a reference to operand node c. Note that now the matches refer to different patterns, which was not the case in the example of figure 3. By combining the matches m_1 and m_2 into a new match m_3, we have a recipe for constructing a new pattern! By copying all the nodes (operations and operands) referenced in m_3 into a new pattern and adding it to the library, we will be able to log all future occurrences of this pattern.

It is possible that a pattern constructed in this way is already in the library. This can be checked by comparing the new match (m_3) to all complete matches on the current operand node (c). If a match is found that references the exact same operand nodes as m_3, the pattern it refers to must be equivalent to the newly constructed one. Adding the new pattern to the library would only add redundancy and more matches, which we want to avoid; we therefore discard the new pattern and its match m_3.

In principle, it is possible to detect patterns of arbitrary size. However, the number of possible patterns in the subject graph explodes when we allow large patterns. In addition, the chances of a pattern recurring in a graph drop as its size goes up, so there is a limit to the usefulness of larger patterns. Still, the number of patterns that can be found if the pattern size is only two or three operation nodes is considerable. Furthermore, a large number of patterns causes a large number of partial matches, each of which have to be considered in the inner loop of the matching algorithm (section 3.1). It is easy for the matching algorithm to become bogged down in partial matches that may never become complete. For this reason, we decided to keep track of only a limited number of patterns. A pattern buffer of 200 patterns is kept, in which the patterns are sorted according to how often they have been found in the subject graph up to that point. Each time the buffer becomes full, the least frequently occurring half of the patterns is discarded. That way, newly constructed patterns have some time to climb into the top 50% before the next purge. A drawback of this method is, that any recurrence of a discarded pattern is later seen as the occurrence of a new pattern.

3.3 Covering

When the matching algorithm finishes, all occurrences of library patterns in the subject graph have been detected and marked. We now want to find a *cover*, a subset of the total set of detected matches, such that:

- All operation nodes in the subject graph are included in exactly one match of the cover.
- A certain characteristic of the cover is optimized. This can be the number of matches in the cover, or the longest path's length, or a combination of both.

Note that it is not strictly necessary for each operation node to be included in exactly one match. It may be profitable to include overlapping matches in the cover, duplicating operation nodes, in order to reduce the length of the graph's critical path [6], at the expense of an increase in the number of matches in the resulting cover (meaning, in this case, chip area). However, since we aim at reducing the operation count of our graph (the number of matches in our cover), we thought it prudent not to allow node duplication.

The heuristic we use to select a cover operates on the premise that reducing the number of operations in all paths of the graph will also reduce the total operation count. In order to minimize the number of operations in a path, we apply an algorithm that is similar to dynamic programming [5], but differs in that it handles reconvergent paths in the graph.

Our algorithm starts by determining, for each subject graph operand node, the matches for which the operand maps to an output of the corresponding pattern graph. These matches are called the *output matches* of that operand. The output matches are then sorted ascendingly, on each operand node, according to the number of matches in the longest path leading up to it. After all operand nodes have been processed this way, we can determine which of the subject graph's output operands has the longest path leading up to it. We implement (select for the cover) the best match on that operand, and invalidate all matches that overlap with it (i.e, refer to operation or operand nodes that are also referenced in this match). We then recursively do the same for all paths leading up to the selected match, by order of path length. Reconvergence in the paths is handled by terminating the recursion whenever we come across an operand node which has been covered already (for which a match has been selected already). After all paths leading up to the current subject graph output operand have been covered this way, we can handle the other subject graphs outputs in the same fashion.

4 Experiments

We execute our method for detecting recurring patterns of operations on a number of well-known benchmarks from the DSP domain. In this paper, we limit the size of the patterns to two operation nodes, even though the pattern detection and matching algorithms can handle patterns of arbitrary size. An overview of the benchmarks, with their dynamic operation counts, is given in table 1.

Name	Description	#ops
bspline	FIR Filter	6149
compress	Compression (dct 2d)	163513
dft	Discrete FFT	6666
edge	Edge detection	268717
expand	Decompression (idct 2d)	151083
feowf	5th Order Elliptic Wave	13067
fir	35 pt. Lowpass FIR	30459
flatten	Level histogram of image	33960
iir	IIR highpass filter	10794
pse	Sehwa's FIR filter	6917
smooth	Convolution w. 3x3 kernel	83365

Table 1: DSP benchmarks.

Each benchmark is trace simulated, and the pattern detection experiments are performed on the dataflow graph of the execution trace. The detected patterns for all benchmarks are then put into a unified pattern library. This library is then used to cover the execution trace of each benchmark, to get a feeling for how much each pattern would help reduce the operation count of that benchmark. These coverings can be seen as the best results the covering algorithm can attain with an unbounded pattern library (after all, all patterns ever detected in any of the benchmarks are there).

The reason we use an execution trace rather than the static object code for our experiments is that a trace effectively masks control flow, making pattern matches visible that reach across basic block or even loop boundaries. If we cover the trace with the patterns we found, then the matches that cross control flow boundaries would seem to indicate that code motion (speculative execution, loop unrolling, etc.) could be beneficial.

Constructing a unified library After covering, we calculate how often each pattern was used for each benchmark. Note that it is misleading to just count the matches for that pattern, as it is unknown how many of those (likely to be mutually exclusive) matches will be chosen for the cover. The patterns can now be sorted according to how much they contributed to the unbounded-library covering (i.e, as a percentage of the total number of matches that were chosen for the cover). Since we are most interested in patterns that contribute to the operation count reduction of the entire application domain, rather than just one application, we sort the patterns by the average of their contributions to the per-benchmark coverings. The reason we average the contributions of the patterns (a percentage) rather than the absolute share in the coverings (a number of pattern instantiations) is that we do not want to skew our results towards the larger benchmarks. This yields a unified, sorted pattern library in which all benchmarks are represented equally, independent of the size of their respective data sets.

Unbounded library covering analysis In figure 5, we see the (cumulative) contribution to the individual benchmarks' coverings of pattern libraries that consist of the top-x patterns of the unified library. As can be seen, some benchmarks get a better-than-average contribution (bspline, pse) and some get a worse-than-average contribution (foewf). This can be interpreted as follows: the more an application's contribution graph pulls towards the upper-left corner of the figure, the more 'average' or representative for the application domain the application is. As a consequence of this, figure 5 can also be used to judge how well the applications fit together on the same instruction set, something which is difficult to determine by inspection of the benchmarks' source code alone. A final remark on figure 5: it can be seen that from the top-80 patterns onward, there is no additional contribution to the covering of any of the benchmarks (the 100% mark has been reached). This implies that none of the patterns added to the library from that point onward are ever actually used in the coverings of any of the benchmarks.

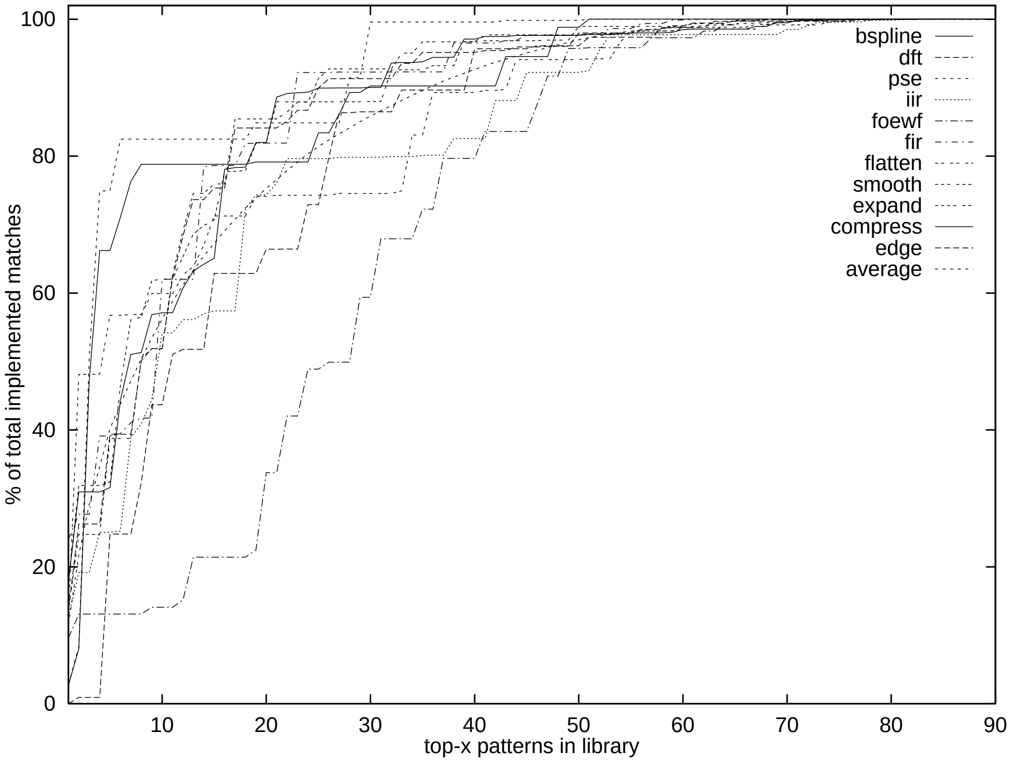

Figure 5: The contribution of patterns to the coverings.

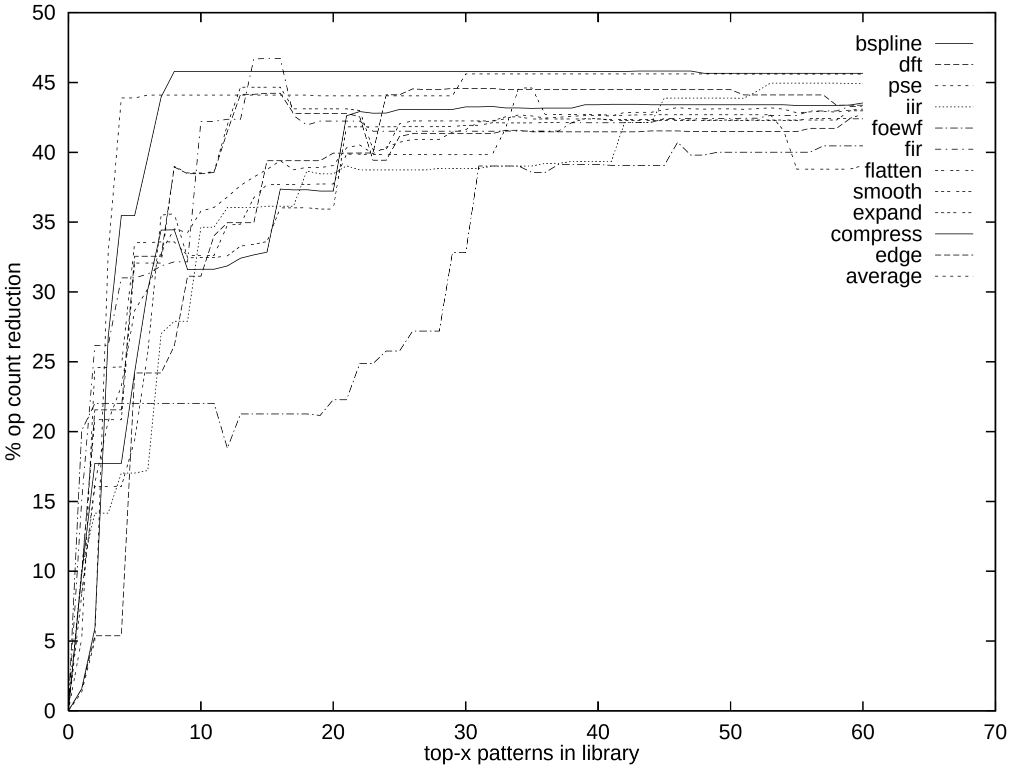

Figure 6: Operation count reduction for the incremental pattern libraries.

Covering with partial libraries Now that we have found a ranking of patterns in the unified library, we can check if there is a relation between the contribution of the patterns in the top-x libraries to the 'ideal' (unbounded library) covering, and the actual results of the covering algorithm for each of the top-x libraries. For this, it is necessary to re-cover each of the benchmarks using each of the top-x pattern libraries. The results of these coverings can be found in figure 6. It can be expected that if a library has a lower-than-average contribution (in figure 5), then the operation count reduction will also be below average. This is confirmed by the curve for the foewf benchmark. Similarly, benchmarks with a higher-than-average contribution should have a higher-than-average operation count reduction. The operation count reduction curves for the bspline and pse benchmarks confirm this.

Note that the curves in figure 6 do not increase monotonously, which is what we would expect if we give the covering algorithm an extra pattern to work with each time. These occasional dips are due to the fact that the covering algorithm is a heuristic method, which every once in a while gets confused and yields a sub-optimal result. Figure 7 illustrates how the addition of a pattern G_{pat3} actually worsens the cover: initially, as only patterns G_{pat1} and G_{pat2} are available, the covering algorithm will apply G_{pat2} twice, on operations $\{I, II\}$ and operations $\{III, IV\}$, arriving at a cover of size two. After pattern G_{pat2} becomes available, however, the covering algorithm will assume that reducing the number of matches in all paths will also reduce the total size of the cover. It therefore applies G_{pat3} to operations $\{II, IV\}$, shortening the path through operations II and IV to one match. This leaves operations I and III to be covered by G_{pat1}, resulting in a cover of three matches.

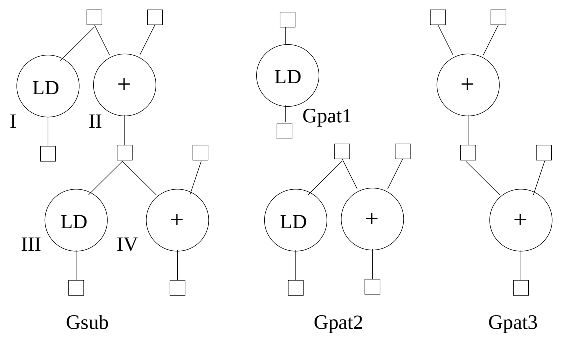

Figure 7: Adding patterns may worsen the cover.

Also note that 50% operation count reduction is a hard limit: since, for the purposes of this experiment, we only use patterns that are no larger than two operations, the best result we can theoretically get is obtained if all operation are replaced, in pairs, with two-operation patterns. In addition, it must be noted here that, since the covering algorithm operates on execution traces and hence ignores control flow, the operation count reduction figures must be seen as upper bounds for the operation count reduction that a compiler can achieve when performing code generation (when control flow is taken into account).

Analysis of the top 10 patterns The top ten patterns of the unified library are shown in figure 8. The most popular pattern (nr.1) is an integer add, followed by another integer add. In hardware, this can be implemented as an add, followed by an accumulate on the same unit, or as a 3-input, 2-output unit that can execute the pattern in a single processor cycle [10]. The second pattern is a conditional jump, where the condition is calculated by the greater-than node. Patterns 5 and 7 are array references, where the address calculation consists of a base-plus-offset calculation. Pattern 6 is the well-known multiply-add, pattern 8 the almost equally well-known add-shift. Note that pattern 9 performs the same calculation on both nodes! It looks as if the compiler missed an optimization opportunity, possibly hidden by control flow.

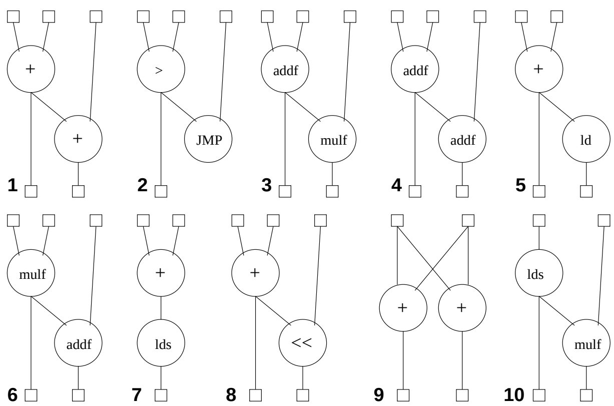

Figure 8: The top 10 patterns.

It can be seen that the top 10 patterns are not proper trees, since most also export their intermediate values. However, with the exception of pattern 9, none of the patterns represent operations that execute in parallel. It is quite possible that this is an artifact of the covering heuristic, which favors chains of operations. It will be interesting to see whether this situation changes if we come up with a different covering strategy.

5 Conclusion and Future Work

We presented a technique for identifying common operation patterns across a range of applications, using a new pattern matching algorithm. This algorithm is innovative in that it can handle patterns of arbitrary shape, widening the scope of the search for operation patterns that can be implemented in hardware. Newly discovered patterns are added to the pattern library on-the-fly, resulting in a single pattern detection (finding new patterns) and matching (marking occurrences of patterns) pass.

Using the new technique, we found patterns of operations common to a set of benchmarks, which, when covering is applied, indicate that a substantial operation count reduction is possible (e.g, 20 patterns yield an average operation count reduction of 40%). Furthermore, we were able to incrementally construct a library of new operations (patterns) and analyze the influence of each new operation on the average operation count as well as on the operation count of each benchmark separately.

The covering heuristic, which was based on dynamic programming, still leaves something to be desired. The influence of the covering algorithm on the selection of the patterns of various shapes is not well understood at this point. Different algorithms may yield different top-x pattern libraries, which is something that needs to be investigated.

It has already been noted that our method operates on (dynamic) execution traces rather than (static) code. The absence of control flow in an execution trace can be seen as both an advantage and a disadvantage. Patterns invisible in static code can be detected dynamically, but exploiting these patterns during static code generation may prove difficult, requiring various code transformations, such as loop unrolling, speculative execution, etc. These transformations may cause an increase in code size, nullifying the effect of the patterns. In the future, we will concentrate on how to extend our current techniques to the problem of code generation.

References

[1] A. Aho and M. Corassick. Efficient string matching: An aid to bibliographic research. *Communications of the ACM*, 18(6):333–340, June 1975.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesly Series in Computer Science. Addison-Wesly Publishing Company, Reading, Massachusetts, 1985.

[3] Marnix Arnold and Henk Corporaal. Matching and covering with multiple-output patterns. Technical Report 1-68340-44(1999)-01, Delft University of Technology, http://cardit.et.tudelft.nl/MOVE/documents/Arnold99a.ps, 1999.

[4] Peter M. Athanas and Harvey S. Silverman. Processor reconfiguration through instruction set metamorphosis. *IEEE Computer*, (0018-9162/93/0300-0011):11–18, 1993.

[5] K. Keutzer. Dagon: Technology binding and local optimization by dag matching. In *DAC, Proceedings of the Design Automation Conference*, pages 617–623, May 1987.

[6] Yuji Kukimoto, Robert K. Brayton, and Prashant Sawkar. Delay-optimal technology mapping by dag covering. In *Proceedings of the Design Automation Conference*, 1998.

[7] Clifford Liem, Trevor May, and Pierre Paulin. Instruction-set matching and selection for dsp and asip code generation. In *Proceedings of EDAC-ETC-EUROASIC*, pages 31–37, 1994.

[8] Giovanni De Micheli. private communication, 1998.

[9] Frederick Onion, Alexandru Nicolau, and Nikil Dutt. Compiler Feedback in ASIP Design. Technical report, University of California, Irvine, September 1994.

[10] Stamatis Vassiliadis, James Phillips, and Bart Blaner. Interlock Collapsing ALU's. *IEEE Transactions on Computers*, 42:825–839, July 1993.