CPR: A Configuration Profiling Tool

Srihari Cadambi*and Seth Copen Goldstein†

Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Abstract

In this paper we describe a **C**onfiguration **PR**ofiling tool (CPR) and show how it can be used to aid compiler designers, FPGA architects and in the construction of a macro-generator libraries. CPR uses subgraph matching to identify the parts of an application which are most important to achieve high performance. Using CPR as a guide we implemented a few macros for a macro-generator library, which yielded significant improvement in both the quality of configurations and speed of compilation.

1 Introduction

In this paper we describe a **C**onfiguration **PR**ofiling tool (CPR) and show how it can be used to aid compiler designers, FPGA architects and in the construction of a macro-generator library, an important component of an efficient portable compiler. CPR uses a restricted form of subgraph isomorphism to identify the parts of a configuration which are most important to achieve high performance and utilization.

CPR provides usage information about applications for compiler writers, FPGA architects, and power users. By analyzing the data flow graphs of different programs, CPR can indicate which kinds of operations are the most important to optimize. This information will guide compiler writers in choosing what optimizations to implement. For FPGA architects it indicates what kinds of logic and routing structures are most important. For power users who need to increase the speed of a configuration it can point out specific areas where hand optimization will be most beneficial.

The original motivation for CPR was to aid in the development of an Architecture Specific Macro (ASM) Library. An ASM library contains pre-placed and routed parameterized macros which can replace a group of nodes in a netlist or dataflow graph. The power of the ASM library is that it allows configuration to use specific features of a target FPGA without having to incorporate such features in the compiler proper. Instead, when patterns in the input graph match macros in the library, the patterns are replaced with a module generated from the macro. Effective use of an ASM library can improve utilization and efficiency of the implementation, while simultaneously reducing compilation time.

The impact of identifying and generating modules in configurations is analogous to the 90:10 rule in standard code profiling [10], which states that 90% of the execution time is spent in 10% of the code. In our case this is reflected in the fact that a configuration has few unique static patterns of nodes. Thus, if the patterns that occur most often are identified and hand-optimized, the overall application will be accelerated.

Identifying frequently occurring patterns in hardware designs assists in determining the proper focus of effort for FPGA architects and compiler designers. As a result, common and useful optimizations will be implemented first, and infrequently occurring optimizations will not slow the design cycle of a new FPGA architecture or compiler.

In addition, configuration profiling finds the drawbacks and inadequacies of the compiler optimization passes. In order to optimize compilation time and efficiency, some optimizations may be turned on or off

*cadambi@ece.cmu.edu
†seth@cs.cmu.edu

depending on a characterization of the subgraphs in the configuration. Obviously, turning off optimizations enhances the speed, while having more of them trades off speed for efficiency.

CPR employs an algorithm for finding all single sink DAGs in a dataflow graph. The graph cannot have control in it, but CPR may be applied to the dataflow parts of a control-flow graph.

The large number of generated patterns are pruned and sorted according to architecture-specific heuristics to emphasize the most important and frequently-occurring patterns. CPR is part of DIL, a data-flow intermediate compiler for pipelined-reconfigurable architectures being developed at Carnegie Mellon [2]. While the algorithm is described in the context of DIL and dataflow graphs, it is equally applicable to traditional CAD tools and netlists.

In the next section we describe the context within which CPR is based and present some example graphs. In Section 3 we describe our algorithm and its implementation. Section 4 takes a look at some of the important patterns generated by CPR and shows how these patterns have influenced our compiler and the ASM library. We go over some related work in Section 5 and conclude in Section 6.

2 An Architecture Specific Macro Library

One of the primary goals motivating CPR is to provide information to the designer of an Architecture Specific Macro library (ASM library). An ASM Library is a key enabler for retargetability of the DIL compiler. The ASM library contains macros which can generate pre-placed and routed modules for a particular FPGA architecture. Each macro is associated with a fragment of a data flow graph. For example, Figure 1 shows the graph for the add-with-carry-in macro. Associated with this graph is a macro generator which will generate a pre-placed and routed configuration for add-with-carry-in. Instead of including code in the compiler specific to a particular architecture, the compiler can use graph matching to take advantage of architecture specific features of the FPGA. When a macro in the library matches a portion of the data flow graph, the nodes

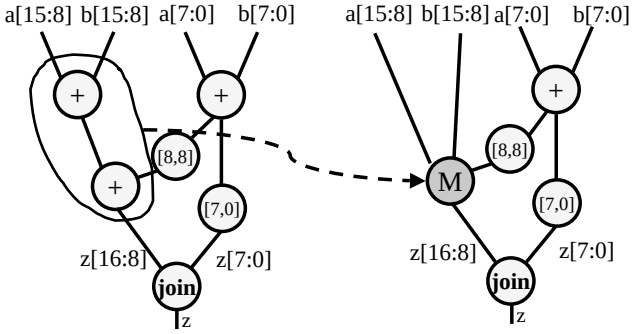

Figure 1: *An instance of the add-with-carry-in macro is identified in the left hand graph and replaced by the single node, M, on the right.*

in the data flow graph are replaced with the macro node. This macro node is handled specially in the place and route portion of the compiler to create the proper configuration.

CPR is used to determine the set of macros which should be included in the ASM library. CPR analyzes a program and yields patterns of varying sizes and their frequencies. By analyzing these patterns the designer of the ASM library can determine which ones should be included as macros in the ASM library. In the rest of this section we give two examples of how macros can increase the effective utilization of an FGPA. In the first example, we show how the macro add-with-carry-in can increase the utilization of configurations targeted to PipeRench, a pipelined reconfigurable FPGA being designed at CMU [3, 6]. In the second example, we show how the macro conditional-add can increase the utilization on the Xilinx 4000 series FPGAs. Before describing the macros we describe the compilation process.

2.1 The DIL Compiler

The DIL compiler compiles a high-level, architecture independent Dataflow Intermediate Language (DIL) and produces configuration for FPGAs. Currently DIL targets PipeRench, but is being extended to produce configurations for other FPGAs. DIL is intended to be used both by programmers and as an intermediate language for a high-level language compiler. Details of DIL and its compiler are described in [2]. Here we focus on the two passes most important to the ASM library: Operator Decomposition and Place-and-Route.

2.1.1 Operator Decomposition

Operator decomposition is a pass of the compiler with two goals. First, it is used to reduce complex operations, e.g., multiplication, to primitives, e.g., shifts and adds, which can be mapped directly to the functional resource on the FPGA. Second, it decomposes primitive operators of large bit-widths, e.g., addition of 128-bit numbers, which are too large or too slow to execute within a particular time bound.

The decomposition is aided by a technology-independent library which stores the decomposition rules written in DIL. DIL supports operations which can determine the size of the operands and operators. The decomposition phase is parameter-driven so that, for example, the library writer is shielded from the underlying timing details of the target FPGA. The fact that all the decompositions are written in DIL has both advantages and disadvantages. The main advantages are that the library is portable across different architectures and that it is easy to write. The disadvantage is that the library writer is unable to directly take advantage of the details of the underlying architecture. This results in potentially suboptimal usage of the fabric. As we will show, the ASM library compensates for the potential suboptimal results of the decomposition pass by grouping nodes together into macro nodes which can take advantage of the specific features of the architecture.

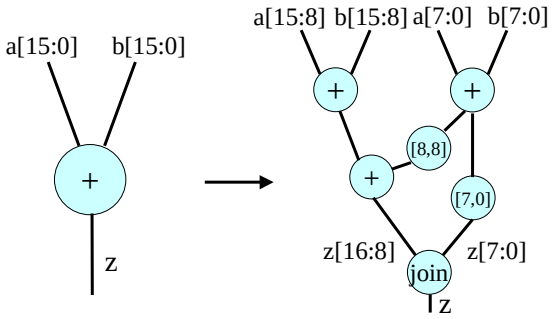

Figure 2: *Example of a decomposition of an 16-bit add into pipelined 8-bit adds with carry.*

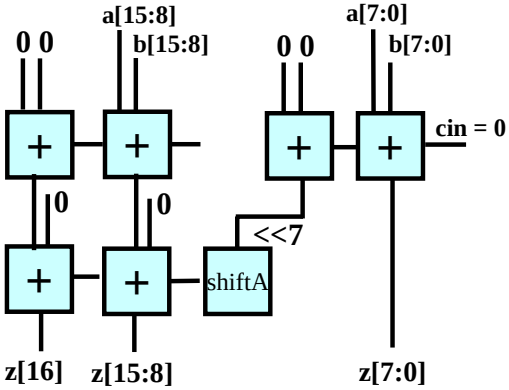

Figure 3: *Mapping the decomposed addition onto PipeRench without using the ASM library.*

2.1.2 Place and Route

The place and route algorithm in DIL [2] is a greedy, deterministic algorithm best suited for primitive operations that have a small number of inputs and outputs. The place and route happens in two main phases. The graph is first transformed so that it is guaranteed to be placeable and routable. This transformation involves reducing the complexity of routing operators and inserting *lazy noops* in the graph. Then, the nodes of the graph are actually placed and routed. Each node is tested to determine if it can be placed and routed. This feasibility test is based on the sources of the node that have already been placed. If the node is unplaceable, the lazy noops are turned into real noops, which improves routability, and the lazy noop is placed on the ready list.

In order to both reduce the complexity and increase the portability of the place and route, the DIL compiler does not include any information about specialized routing resources. For example, on PipeRench there are one-bit wires which can be routed between processing elements. A processing element (PE) is essentially a group of identically configured 3-LUTs, some registers, carry chain logic, and zero-detect logic. These one-bit wires can be used to construct carry chains and control logic, and are not part of the more abstract architecture the main compiler targets.

The primary impact on the ASM library is that the algorithm works best when there are few inputs, i.e., three or fewer, to each node. For, as the number of inputs to a node grows, the choices of where to place it are more limited. Additionally, the ASM is the only way in which specialized wires, e.g., carry chains, can be directly utilized.

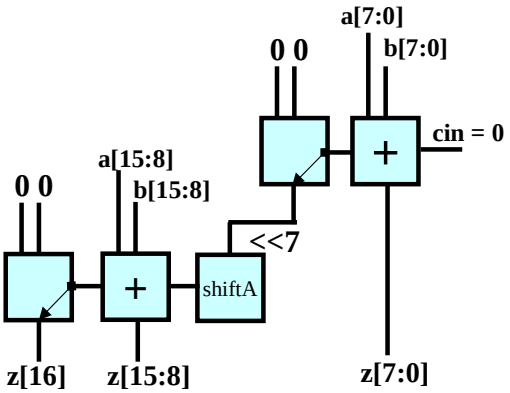

Figure 4: *Using the ASM library to map the decomposed addition.*

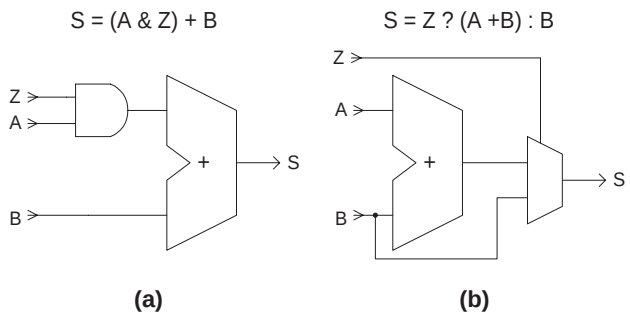

Figure 5: *Two logical structures for implementing conditional addition. The structure in (a) cannot easily be implemented in a single row of CLBs using the fast carry logic. The transformed structure in (b) can be implemented in a single row of CLBs.*

2.2 Add-With-Carryin

In general, to increase the clock speed of an FPGA design the user may want to constrain the total width of any operation which uses the carry chain. For example, on PipeRench, to sustain a 100Mhz clock the maximum width of an addition is 16 bits. To add numbers that are wider than 16 bits requires breaking up the addition and pipelining it. The DIL compiler will decompose large additions into graphs such as the one shown in Figure 2. Since DIL does not allow the expression of architectural details, the carry signals between pipelined additions are treated as regular data values. Without the ASM library, these data values would be routed on the regular interconnect and would require 7 PEs as shown in Figure 3. However, as show in in Figure 4, using the special interconnect we can reduce this to 5 PEs.

2.3 Conditional-Add

A common structure in many data paths is the conditional addition, illustrated in Figure 5(a). The conditional addition is the basic building block used to construct both serial and array multipliers. This structure cannot be implemented in a straightforward way on a single row of Xilinx 4000 series CLBs using the fast carry logic. The problem is that the inputs to the carry logic must be primary inputs to the CLB, and this is impossible due to the AND gate.

This structure can be implemented on a single row of CLBs using the carry logic if it is transformed to the equivalent structure shown in Figure 5(b), as described in [7]. In this implementation, the carry inputs are primary inputs, and the multiplexor logic can be subsumed into the lookup table that implements the XOR function for the addition.

Surprisingly, the commercial synthesis and physical design tools that we have used are incapable of employing this optimization, and generate CLB mappings that are 100% larger than necessary. A configuration profiling tool such as CPR would detect the importance of this structure in almost any arithmetic datapath. Once aware of this structure, an FPGA CAD tool could directly map it to the optimal solution. An FPGA architect could use a tool like CPR and modify the CLB design to implement this structure in a more straightforward manner.

3 The Pattern Generation Algorithm

In this section we describe our pattern matching and ranking algorithm, the key component of CPR, and analyze its time complexity and memory requirements. The algorithm reports the frequency of occurrence of all possible single-sink DAGs (up to a given size) found in the input graph. The DAGs (also referred to as *patterns*) are then analyzed and ranked according to a heuristic which aims to present to the user of CPR the most important patterns. While the discussion in this section focuses on data flow graphs, the algorithm is equally applicable to traditional CAD tool graphs, e.g. netlists. It does

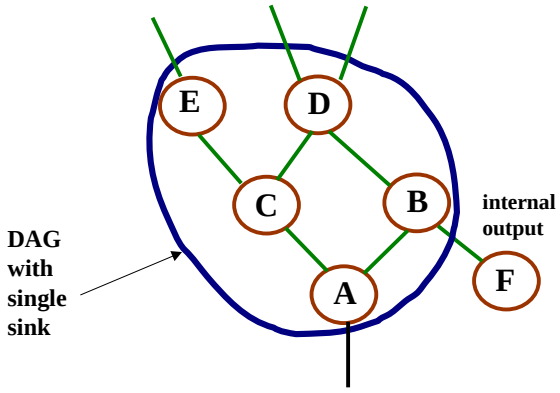

Figure 6: *An example single sink DAG, with an internal output.*

not report patterns with more than one sink, but such patterns by definition have multiple outputs and are seldom of interest in macro construction. Furthermore, our current compiler framework does not support operators with more that one output. It may be noted however, that if nodes in a single-sink DAG contain internal output wires, they will be reported. Figure 6 shows an example single-sink DAG and what constitutes an internal output.

3.1 The Algorithm

The algorithm central to CPR constructs all possible single-sink DAGs of size up to l in the graph and then compares these patterns to detect the frequencies of each pattern. The algorithm proceeds constructively, building up at each node N patterns of size l from previously constructed patterns of sizes up to $l-1$ as shown in Figure 7. Figure 8 pictorially depicts the algorithm and some data structures used. The algorithm is based on the following observation: Each pattern of size l ending at node N is composed of N with various DAGs ending at N, such that the total number of nodes including N is l. The only caveat to this is when there is reconvergent fan-in, which can be detected by taking the intersection of the nodes in the source patterns to node N. In this case, the pattern is ignored, since it is already accounted for as a pattern of size $l-1$ ending at node N.

Each constructed pattern is checked against a global hash table of patterns. If the new pattern is already in the table, the frequency count of that pattern is increased, otherwise the pattern is inserted

```
Inputs: Graph G, Size l
Outputs: Table T which contains patterns of all sizes
and annotations on the graph

genAllPatterns(Graph G, int size)
{
   for (int l = 1; l <= size; l++)
      foreach node n in G
         genPattern(n, l)
}

// genPattern for 2 input nodes.    3 input nodes
// proceed similarly but cover all permutations
// summing to "len" across all three inputs.
genPattern(Node n, int size)
{
 for (int a=0; a<size; a++) {
  foreach pattern left in n.left.patterns[a] {
   foreach pattern right in n.right.patterns[size-a-1] {
      if ((nodes in left ∩ nodes in right) ≠ ∅ skip
      Graph g = newgraph(n, left, right)
      add g to n.patterns[size];
      if g ∈ T increment frequency
      else insert g into T
   }
  }
 }
}
```

Figure 7: *Pattern construction algorithm.* `newgraph(n,l,r)` *creates a new graph with edges* $l \rightarrow n$ *and* $r \rightarrow n$. T *represents the global pattern hash table.*

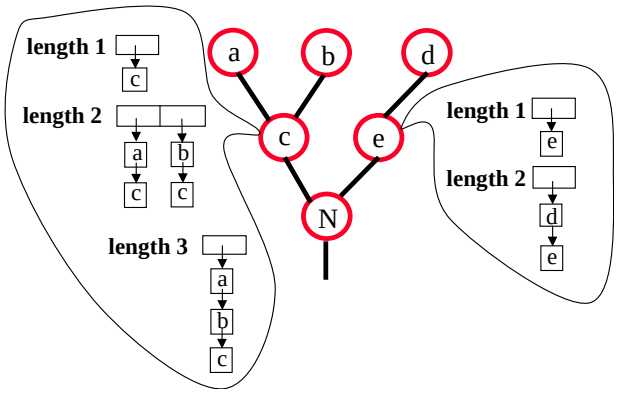

Figure 8: *The CPR pattern search algorithm and data structures. While searching for patterns of size 4 ending at node N, we look for patterns of sizes 0, 1, 2 and 3 ending at each of N's sources, each of which would have been computed earlier. Each entry in the array representing the patterns is a list of node pointers.*

into the table. Two DAGs have the same pattern if the nodes are of the same type and the wires connecting them are similar. We distinguish three kinds of wires, single bit wires, unsigned wires of any width, and signed wires of any width. The wires do not have to be identical widths since CPR aims at finding patterns that may be turned into macros. Each macro is parameterized so that it can generate modules for various wire widths.

3.2 Time Complexity

To construct all patterns of up to size l, each node in the entire graph, $G = (V, E)$, must be scanned. Assuming at most i inputs per node, there are at most $O(\begin{pmatrix} i^{(l-1)} \\ l \end{pmatrix}) = O(i^{(l-1)^l})$ trees ending at node N. Thus the worst case time to construct all the new DAGs of size l for all nodes in the graph, where $i \leq 3$ is $O(|V|3^{(l-1)^l})$. Thus, to find all patterns of size l in G, the algorithm takes time $\Sigma_{k=1}^{l} O(|V|3^{(k-1)^k})$.

In practice, most nodes in a graph have only 1 and 2 sources. Furthermore, graphs have a reasonable amount of reconvergence which reduces the time complexity substantially. However, it can still take many hours to look at all patterns of large sizes. To reduce the running time we allow users to limit the patterns created as discussed in Section 3.4.

3.3 Memory Requirements

In order to reduce memory requirements we do not actually create graphs for the new patterns. Instead, a pattern of size l ending at node N is represented by a list of the nodes that comprise that pattern. This list uniquely identifies the pattern. Furthermore, in the global hash table we are able to identify a pattern by a triple consisting of the sink node of the pattern, the size of pattern, and the position in the nodes list of patterns of that size.

3.4 Pattern search pruning heuristic

CPR may be used to find every possible single-sink DAG. However, the exponential search time limits the size of the largest practical pattern which can be generated. But it is possible to reasonably prune the pattern search space. The idea is to ignore those patterns which will not yield useful architectural specific macros or insight into the architecture or compilation process. The heuristic we use is based on the fact that patterns with a large number of inputs and outputs will never yield useful architectural specific macros.

Since patterns with lots of I/O are difficult to route, such patterns are ignored during a restricted search. This characteristic of patterns is applicable to a large number of architectures, since a large number of inputs and outputs make routing difficult. Also, outputs from internal nodes will require those nodes to be replicated if this pattern is made into a module, thereby incurring losses. This substantially reduces the time complexity. However, it might miss large patterns which have a lot of reconvergence.

Our general pruning heuristic was to ignore all patterns that satisfied three constraints: (nodes > N), (inputs > I) and (outputs > O), with N, I and O being variable parameters. Based on how many good patterns were retained[1], we assigned values of 7, 5 and 1 to N, I and O respectively. Figure 9 shows the number of patterns pruned away by this heuristic, a measure also indicative of the amount of time saved. It may be noticed that even when the size limit of the patterns was 9, about 70-80% of the patterns were eliminated using the heuristic.

In order to validate this pruning methodology, we manually examined the top 20 patterns of the unrestricted output and compared it to the top 20 patterns of the output using the heuristic. The top patterns were chosen by the ranking scheme described in Section 3.5. Table 1 shows this for different kernels. It may be seen that in all cases except the DCT, at least 90% of the good patterns are retained. However, if the heuristic is slightly modified for the DCT we see improvements, shown in Figure 10. Specifically, it may be seen that as the parameter I is varied, the percentage of good patterns retained increases from 60% for $I = 0$ to 85% for $I = 5$ (the current heuristic) and reaches 100% for $I = 7$. Alongside in the figure is shown the CPU time required to profile the dataflow graph of the DCT using the heuristic with the corresponding value of I. As I increases, the time goes up showing a sharp increase at $I = 7$. The time is directly indicative of the number of patterns

[1] the number of good patterns was determined by our ranking scheme, described in Section 3.5

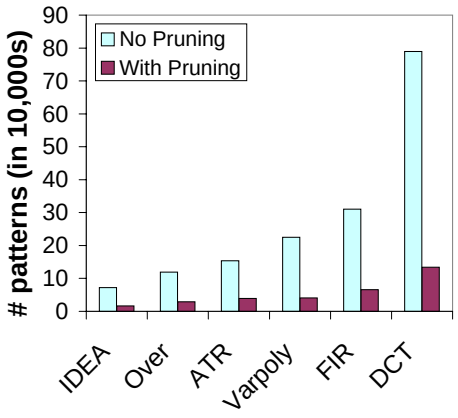

Figure 9: *The number of patterns pruned away by the pruning heuristic above. The pattern size limit here was 9.*

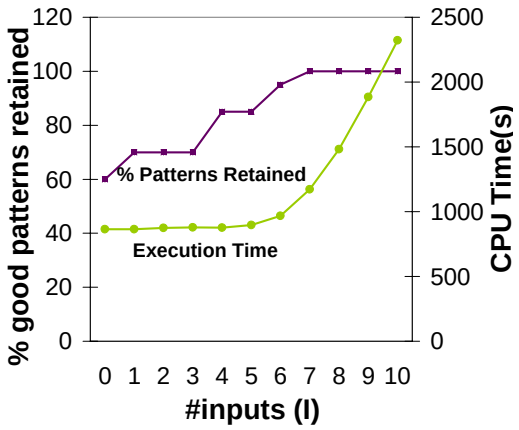

Figure 10: *The percentage of good patterns retained and the corresponding execution time when I is varied with $N = 7$ and $O = 1$.*

pruned away by the heuristic; that is, as I increases, more patterns with large IO will be admitted.

Kernel	good patterns retained
IDEA	95%
Over	95%
ATR	100%
Varpoly	100%
FIR	90%
DCT	85%

Table 1: *The percentage of good patterns retained when the pruning heuristic above was applied.*

3.5 Pattern Ranking

Once the patterns are identified, we use a second series of heuristics which attempt to rank the patterns according to their utility to the ASM library. Of course, when CPR is used as tool for architectural or compiler evaluation different rankings will be useful. The benefit of a macro comes from a confluence of several factors:

1. The synthesis time saved by grouping nodes together into a single preplaced macro.
2. The ease of placing and routing the macro-operation on the target architecture which is influenced by the number of inputs and outputs of the module. The routing difficulty increases as the number of I/O of a pattern increases.
3. The frequency of the pattern: the more often the pattern occurs, the more PEs may be saved.
4. The number of PEs saved by grouping the pattern into a single macro-operator as opposed to allocating resources for each operation separately.

We currently assign the rank, $r = l * f^{1.2}/(i_e * (o_e + o_i))^3$, where l is the size of the pattern, f the frequency, i_e and o_e the number of external input and output wires respectively, and o_i is the number of internal output wires. In addition, we also weight the pattern according to its utility on the target architecture. In the case of PipeRench, we weight patterns with 1-bit wires highly owing to its control interconnect described in [6]. Also, since PipeRench is LUT-based, we also assign more importance to patterns with logical operators. Other ranking schemes are currently being investigated.

4 Results

In this section we evaluate CPR. We begin by presenting a number of interesting patterns that CPR found in several kernels written in DIL. We then show the effect of incorporating some of these patterns into an ASM library.

4.1 Kernels employed

ATR implements the shapesum kernel of the Sandia algorithm for automatic target recogni-

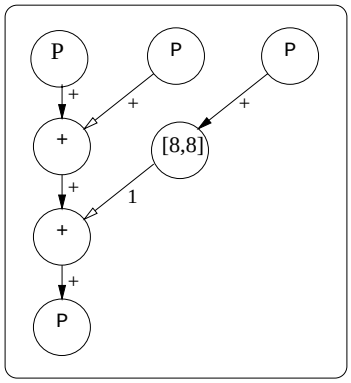

Figure 11: *An add with carryin that does not utilize the control interconnects.*

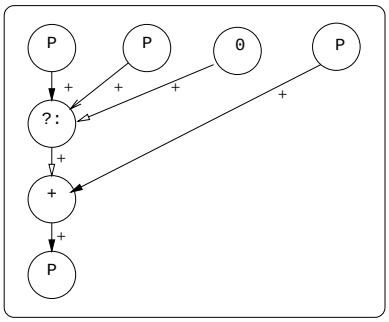

Figure 12: *A conditional add. This may be transformed as explained in Section 2.3 and mapped to a single PE on PipeRench*

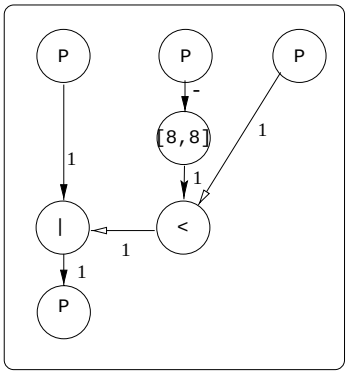

Figure 13: *A "<" operation with 1-bit operands which may be cast as a logical operator and merged with the logical op (OR) following it.*

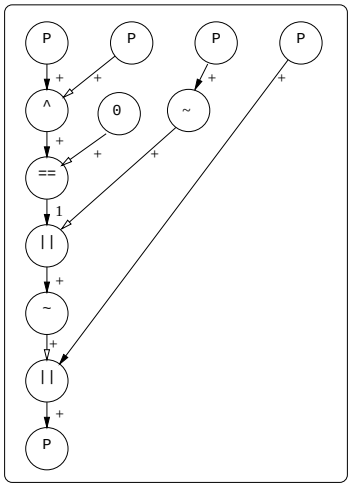

Figure 14: *A power-user patterns. The NOTs may be optimized away, and the entire pattern concisely mapped to 2 PEs on PipeRench using its control interconnect lines[6].*

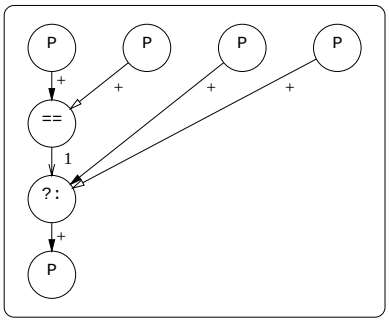

Figure 15: *A comparison feeding a mux. The 1-bit wire between the == and ? : may be allocated to a control interconnect line.*

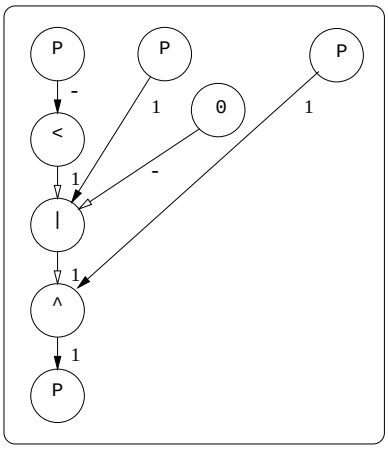

Figure 16: *Room for compiler optimization. A "< 0" may be implemented by passing the sign bit.*

tion [11].

Cordic is a 12 stage implementation of the Cordic vector rotation benchmark [5].

CSD implements a 16-bit canonical signed digit multiplier with the constant 123.

DCT is a one-dimensional, 8-point discrete cosine transform.

FIR is a FIR filter with 20 taps and 8-bit coefficients.

IDEA implements a complete 8 round International Data Encryption Algorithm with the key compiled into the configuration.

Nqueens is an evaluator for the Nqueens problem on an 8x8 board.

Over implements the Porter-Duff over operator used to joins images [1].

PopCount counts the number of 1's in a stream.

Square simply squares a 16-bit signed number.

Varpoly evaluates a polynomial of degree three in x. The coefficients and x are supplied.

4.2 Patterns Observed

Figures 11–16 are examples of high ranking patterns found in the kernels described above. The graphs are generated automatically by CPR. Figure 11 is the add-with-carryin described in Section 2.3. This pattern occurs frequently and having few inputs is highly ranked. It is the most important of the patterns found as evidenced by the frequency numbers shown in Table 2. Figure 12 is the conditional add operation also described earlier. Figure 13 shows a subgraph which indicate that technology mapping using the third input to the LUT would improve performance. Figure 14 shows a power-user pattern that may be heavily optimized using Boolean optimization techniques.

Figure 15 is an example of a pattern which, though it occurs often in FIR, causes FIR to increase in size when included in the ASM library. This is due to the fact that the extra inputs constrain place and route reducing overall utilization. This pattern is ranked very low by CPR and was added to the ASM library before CPR was available.

Finally, Figure 16 shows an example of a graph which could be optimized by a better strength reduction pass in the compiler. CPR found several patterns which expose compiler inadequacies and areas for future optimization.

Table 2 shows the frequency of occurrence of each of these subgraphs in all the kernels.

Kernel	Fig. 11	Fig. 15	Fig. 12	Fig. 14
DCT	71	0	0	0
IDEA	43	0	0	0
FIR	22	18	0	0
ATR	2	0	0	0
Nqueens	0	0	0	4
Over	4	0	28	0
Cordic	29	0	0	0
CSDMult	5	0	0	0
PopCount	0	0	0	0
Quantize	88	0	0	0
Square	18	0	0	0
Varpoly	0	0	35	0

Table 2: *The frequency of occurrence of various subgraphs seen across our benchmark suite.*

Kernel	Stripes		PEs		Time(s)	
DCT	220	190	2586	2398	41.5	35.8
FIR	78	78	1090	996	9.7	9.4
ATR	18	17	154	150	3.5	3.5
Over	26	24	294	292	3.8	3.6
Cordic	175	163	2107	1886	20.7	19.5
CSDMult	19	17	184	174	2.2	2.0
PopCount	5	5	39	39	0.3	0.3
Quantize	255	223	2872	2588	54.4	42.8
Square	42	48	501	480	6.2	5.6
Varpoly	27	27	242	242	2.4	2.4

Table 3: *Synthesis results on PipeRench using macros suggested by CPR. Two of the above macros were used: the PlusWithCarryin and the Conditional Add. For each metric, the first column on the left shows the results without using CPR recommended macros, while the column on the right shows the result after macro generation.*

4.3 Improved Results

We have implemented a few of the modules suggested by CPR, and show the results for all of the kernels when mapped to PipeRench in Table 3. Each PE is 8-bits wide and there are 16 PEs per stripe, where each stripe is a pipeline stage in the application. Thus, the performance of a kernel is determined by the number of stripes. The data shows that a very few macros, if properly chosen, can yield significant improvements in both configuration quality and compilation speed. However, in the case of Square although the ASM library reduces the number of PEs, the number of stripes actually goes up. This is the result of interactions between the ASM library and DIL's place and route algorithm. We are currently adjusting the place and route algorithm to handle ASM library modules.

5 Related Work

There have been many related research efforts in the areas of high-level synthesis and FPGA logic synthesis. These include the use of behavioral templates [8] and performance-driven template mapping for high-level synthesis [9]. The Garp architecture employs a

compiler tree-parsing tool for datapath module mapping [4]. FPGA logic minimization and technology mapping have been tightly-coupled in a scheme introduced in [12]. A common observation is that most technology mapping and module generation efforts in the past focus on graph matching using pre-built templates. As far as we know, this is the first tool that actually generates the templates constructively by searching the graph.

6 Conclusions

In this paper we have described a configuration profiling tool which identifies the most important parts of an application. The central algorithm for CPR is based on an efficient way to generate and count all subgraphs of the configuration graph. This algorithm is applicable both to dataflow graphs and general netlists. We have also shown that we can reduce the running time of CPR without missing any of the important patterns. To our knowledge CPR is the first tool of its kind.

Using the patterns found by CPR we construct an application specific macro library which improves both the running time and the overall quality for the final configuration. By focusing on only the key patterns we were able to decrease the overall configuration size by approximately 10% over a broad range of kernels with only a few macros. CPR was also helpful in highlighting areas of further work in our compiler.

Acknowledgements

The authors would like to thank Herman Schmit, Mihai-Dan Budiu, Matt Moe, Ron Laufer and Reed Taylor of the CMU PipeRench group for their help. Special thanks to Herman for his advice and suggestions and to Mihai for his work on the DIL compiler.

This work was supported by DARPA contract DABT63-96-C-0083 and Altera Corporation. The authors also wish to acknowledge the reviewers for their helpful comments.

References

[1] Jim Blinn. Fugue for MMX. *IEEE Computer Graphics and Applications*, pages 88–93, March-April 1997.

[2] Mihai-Dan Budiu and Seth Copen Goldstein. Fast compilation for pipelined reconfigurable fabrics. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 1999. to be published.

[3] Srihari Cadambi, Jeffrey Weener, Seth Copen Goldstein, Herman Schmit, and Don Thomas. Managing pipeline-reconfigurable fpgas. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 55–64, Monterey, CA, February 1998.

[4] Timothy J. Callahan, Philip Chong, Andre DeHon, and John Wawrzynek. Fast module mapping and placement for datapaths in fpgas. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 123–132, Monterey, CA, February 1998.

[5] Sanjaya Kumar et.al. Timimg sensitivity stressmark. Technical Report CDRL A001, Honeywell, Inc., January 1997. `http://www.htc@honeywell.com/projects/`.

[6] S.C. Goldstein, H. Schmit, M. Bidui, M. Moe, S. Cadambi, R. Taylor, and R. Laufer. Piperench: A coprocessor for streaming multimedia acceleration. In *International Symposium for Computer Architecture*, Atlanta, GA, June 1999. to be published.

[7] Xilinx Inc. Advanced carry logic techniques. *XCELL*, pages 42–44, Second Quarter 1996. `http://www.xilinx.com/xcell/xcell21.htm`.

[8] Tai Ly, Knapp D., Miller R., and MacMillen D. Scheduling using behavioral templates. In *Proceedings of the 32nd Design Automation Conference*, San Francisco, CA, June 1995.

[9] Corazao M.R., Khalaf M.A., Guerra L.M., Potkonjak M., and Rabaey J.M. Performance optimization using template mapping for datapath intensive high-level synthesis. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 15, No. 8*, pages 877–888, August 1996.

[10] David A. Patterson and John L. Hennessy. *Computer Architecture A Quantitative Approach, Second Edition*. Morgan Kaufmann Publishers, San Francisco, CA 94104, 1996.

[11] J. Villasenor, B. Schoner, K. Chia, and C. Zapata. Configurable computing solutions for automatic target recognition. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 70–79, Napa, CA, April 1996.

[12] Kang Yi, Seong Yong Ohm, and Chu Shik Jhon. An efficient fpga technology mapping tightly coupled with logic minimization. In *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Vol.E80-A, No.10*, October 1997.