Tuning To Create Good Circuit Designs

Chris Tong*
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

Phil Franklin+
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

Abstract

The efficiency and optimality of a divide-and-conquer approach to design depends upon the associated hierarchy of implementation goals having sibling subgoals that interact only weakly or not at all. Unfortunately, in many domains the only easily acquirable design refinement knowledge leads to the formation of goal hierarchies that violate this assumption.

In this paper, we describe a learning method that incrementally transforms a search-based design system that spends much of its time recovering from the implicit (and mistaken) assumption that subproblems do not interact, into a compiler-like system that decomposes the original design problem into truly non-interacting subproblems. The improved system finds locally optimal solutions to its subproblems, which are composed into globally optimal solutions. By analyzing dependencies, the learning method re-parses a poor design decomposition into one with no subproblem interactions; it then generalises from the resulting decomposition, adding new refinement rules to the knowledge base.

We have implemented a design system called CPS that solves design problems of implementing boolean expressions as gate-level circuits. We have also implemented a learning program called SCALE that incrementally transforms CPS into an optimizing compiler.

*currently on leave at IBM Watson Research Center

*The research reported here was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract Number N00014-85-K-0116, in part by the National Science Foundation (NSF) under Grant Number DMC-8610507, and in part by the Center for Computer Aids to Industrial Productivity (CAIP), Rutgers University, with funds provided by the New Jersey Commission on Science and Technology and by CAIP's industrial members. The opinions expressed in this paper are those of the authors and do not reflect any policies, either expressed or implied, of any granting agency.

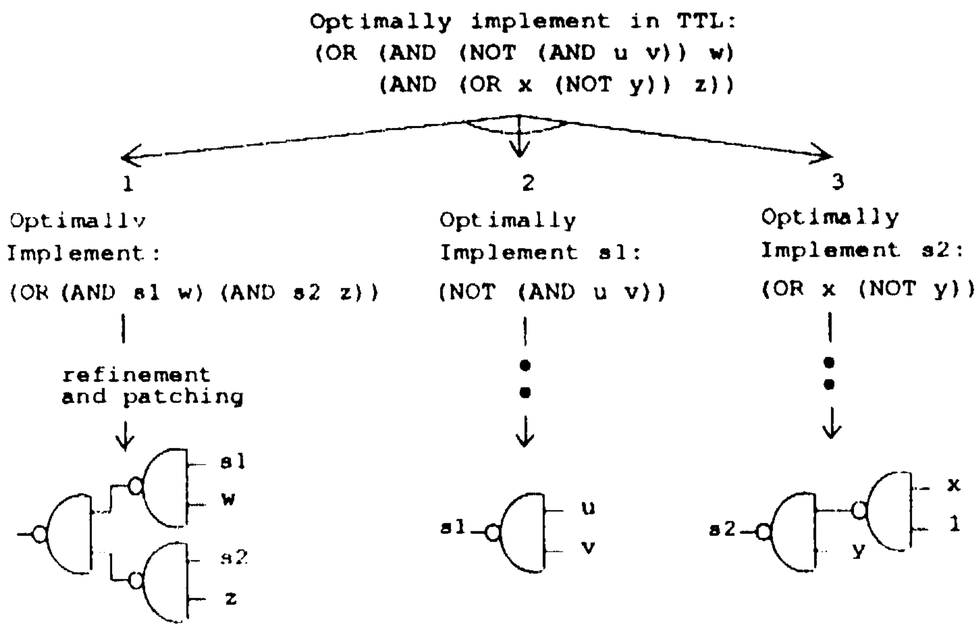

Figure 1: An implementation goal hierarchy

1 Introduction

In principle, design efficiency and solution optimality can be impacted by three factors: choice of design process model; the accuracy of the (possibly implicit) assumptions made by that design process about subproblem interactions; and the contents of the knowledge base. We will discuss these factors in turn, finally concluding that if a learning method focuses on improving the last of these, using a fixed, search-based model of design, the second issue will disappear over time.

1.1 Some simple models of design

Much research in knowledge-based design [Kant and Barstow, 1978, Stefik, 1981, Mittal, 1986, Steinberg, 1987, Tong, 1988, Brown and Chandrasekaran, 1989] has concentrated on design involving some form of *top-down refinement.* In many knowledge-based design systems, top-down refinement proceeds by repeatedly applying *refinement rules* that flesh out or decompose abstract functional specifications, until the leaves of the generated hierarchy are functions that are directly implementable in the target technology (using refinement rules we will call *implementation rules).* The functional component hierarchy is isomorphic to a goal hierarchy containing implementation goals and subgoals of the form, "Optimally implement component x". Figure 1 illustrates such a hierarchy in the domain which we will use to illustrate our ideas: boolean expression design.

In this paper, we will consider design processes that

perform not only refinement, but also *patching* and *backtracking;* we presume a depth-first search strategy. In the boolean function domain, a patcher might use a rule base containing *patching rules* such as P1: "If the circuit is a combinational circuit, and two inverters are in sequence, they can be replaced by a wire." Patching is used to correct constraint violations, and to optimise the design. If patching is insufficient for resolving some problem, backtracking will retract refinement and patching choices that have been made. Models that include other design operations are also possible. For example, many of the ideas about patching apply more generally to subproblem composition processes (including constraint-based reasoning). However, the simple elements just mentioned will suffice to demonstrate the basic intuitions behind our learning method.

1.2 The accuracy of assumptions about subproblem interactions

Suppose we want the design process to create an implementation of a boolean expression design problem that is optimal with respect to gate count. We will consider three different solutions to the design problem of Figure 1; one has 10 gates, another, 8 gates, and a third, 6 gates. The 6 gate solution is an optimal one for this design problem. What sort of design processes might produce the suboptimal solutions?

"Non-interacting subproblems" assumption. A 10 gate solution could be produced by a purely refinement-based process. The design process creates the subproblem decomposition in Figure 2, and selects the (locally) best solution to each subproblem (depicted in the figure). Such a design process implicitly[1] makes the mistaken assumption that subproblems *do not interact]* that is, it assumes that locally correct and optimal solutions to the subproblems compose to form a globally correct and optimal solution.

"Weakly interacting subproblems" assumption. An 8 gate solution could be produced by a design process that supplements top-down refinement with optimization. Many programming language *compilers* have this sort of behavior. Repeated application of patching rule P_1 in the boolean function domain is an example of a *linear time optimizer,* a procedure that iteratively modifies solutions until it produces an optimal solution (in linear time). "Peephole optimization" is another example of a linear time optimizer. Both of these patching processes are applied to the flat, composed solution; hierarchical patching (e.g., the merge step of mergesort) can increase the complexity by a factor of *d,* the depth of the hierarchy.

Top-down refinement would initially produce the same 10 gate implementation as before. Subsequent hillclimbing toward a more optimal design via patching finds only a single opportunity to apply patching rule P1 (to subproblems 3 and 4), resulting in an 8 gate solution. This design process implicitly makes the mistaken assumption that the subproblems only *weakly interact,* i.e., it is possible to take *any* combination of subproblem solutions and *patch* it into a globally correct and optimal solution in linear time.

Strongly interacting subproblems. Subproblems *strongly interact* if some combination of subproblem solutions cannot be patched into a globally optimal solution in linear time. The subproblems in Figure 2 strongly interact (as illustrated by the particular solutions in the figure). Given the possibility of strongly interacting subproblems, we use a search-based model of design that permits backtracking to guarantee that an optimal solution is produced. We assume that the search is trying to optimize an evaluation function f (e.g., gate count), and that one of the problem inputs is a budget that must be met: *f(design) < bu.*

Because the search-based design process still divides problems into subproblems (via refinement) it still behaves as though the subproblems do not interact or only weakly interact, and only backtracks when that (implicit) assumption is proven wrong. Thus like the previous design processes, it selects the locally best solution to each subproblem it tries. Suppose it solves the subproblems of Figure 2 in the order {1,2,...,7}. If we set a budget of 6 gates, it will search about half the design space before it retracts the initial choice it makes for subproblem 1, and begins to converge on the optimal solution (Figure 1). The optimal solution is produced by selecting the locally *worst* solutions for subproblems 1 and 6, implementing each of these two ORs as a NAND gate and two inverters. This produces a design in which patch rule PI can be applied *three* times, bringing the gate count down to 6. As illustrated by this example, search finally finds the optimal solution, but generally does so in exponential time. Thus no leverage in efficiency was gained from the "problem decomposition".

1.3 The contents of the knowledge base

Mistaken assumptions are due to inappropriate rules. Problems of suboptimal designs or inefficient design processes occur when the goal hierarchy is *inappropriate;* the subproblems it creates interact, thus misleading a design process that presumes otherwise. Inappropriate goal hierarchies are evidence that the knowledge base contains inappropriate rules. For instance, pure top-down refinement would be sufficient for the boolean expression design task *if the* refinement rules were guaranteed to produce non-interacting subproblems. Instead of changing the nature of the design process, we can try to change the rules which produce the subproblems, so that they produce subproblems whose worst-case interactions are guaranteed to fall into a particular category (e.g., "weakly interacting subproblems").

Acquisition of design knowledge. The most obvious response to this observation would be to acquire rules that only create non-interacting subproblems; we will call these *good refinement rules.* One or more such rules might decompose our example design problem in the manner depicted in Figure 1 (instead of the decomposition of Figure 2). Because they are non-interacting, the locally optimal solutions to these subproblems compose to form a globally optimal 6 gate solution. We may arrive at these locally optimal solutions either by ap-

[1]We only consider design processes that do not explicitly reason about such assumptions.

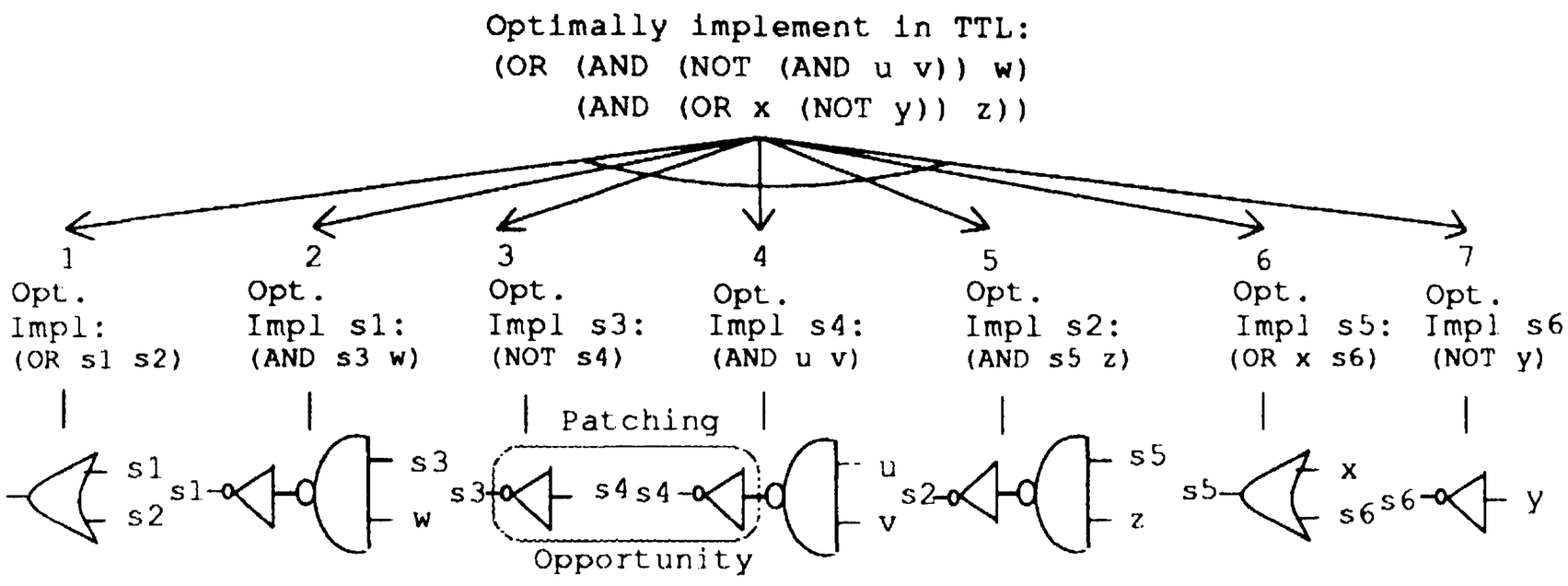

Figure 2: A poor implementation goal hierarchy

plying the "design as search" approach locally to solve each subproblem, or by acquiring three additional implementation rules that tell us how to optimally implement these subproblems.

In contrust, however, most prototype knowledge-based design tools or compilers that have been created for "real world" problems such as circuit or software design do not follow this strategy. Most create an initial, poor design based on an inappropriate goal hierarchy, and then optimize (e.g., [Knapp and Parker, 1986, Kowalski, 1985]). Far fewer (e.g., [Kant and Barstow, 1978, Steinberg, 1987]) actually use refinement rules that enable a polynomial time, divide-and-conquer approach.

The reason seems clear. Acquiring implementation rules sufficient for forming (probably inappropriate) goal hierarchies is generally a relatively easy and constrained task (see, e.g., Figure 3). Acquiring patching rules sufficient for limited optimisation is also relatively easy. On the other hand, acquiring good refinement rules is a rather unconstrained task. A combinatorial number of such rules may be necessary to get broad coverage. Such rules may not exist, may not be easily obtainable from domain experts, or may not be expressible in any intuitive form.

Learning good refinement rules from design experience. Let us reconsider the problem of inappropriate goal hierarchies from the standpoint of human designers. In a poorly understood design domain, human designers do not usually start out doing top-down design. Instead, they first do a lot of "exploratory design", using processes such as composition, hillclimbing, or analogical reasoning (see, e.g., [Kowalski, 1985]). We hypothesise that one of the primary purposes of such exploratory design is to form a more appropriate goal hierarchy out of an initially poor one, so that a reasonably efficient top-down refinement based approach to design can be pursued subsequently, and so that the process is reasonably likely to produce an optimal (or satisficing) solution. Human designers appear to learn appropriate goal decomposition hierarchies both while solving a particular problem, and more generally, so that later problem-solving behavior for a similar problem seems less search-like. We have modelled the "within-task" learning skill (given an initial budget for an evaluation function *f),* in a circuit design program called DONTE [Tong, 1988].

This paper focuses on modelling across-task learning of appropriate goal decomposition hierarchies that are fine-tuned for optimizing a particular evaluation function.

Implicit vs. explicit decomposition. The problem decompositions we have been discussing can be created directly by *decomposition rules* that match a design problem and produce a set of subproblems to be solved. Alternatively, a decomposition can be created indirectly by applying refinement rules that only match a subexpression *S* (e.g., (AND u v)) of the design problem *P* (e.g., (NOT (AND u v))). Such a rule implicitly decomposes *P* into two problems: "Implement S^n and "Implement everything in *P* but S^n (e.g., (NOT expr)). If *P* can be "parsed" into subexpressions all of which match some refinement rule, we say that *P* is *implicitly decomposed* by the set of matching rules. Ambiguity in the parse corresponds to alternative decompositions. In this paper, we will study the learning of *good* refinement rules that *implicitly decompose* a problem into *non-interacting* subproblems by matching its subexpressions.

Simplifying assumptions. Design subproblems can interact in different ways. A *functional interaction* occurs if some combination of choices for solving the subproblems leads to a global design that fails to function correctly. A *resource usage interaction* occurs if some combination of choices for solving the subproblems leads to a design that is *suboptimal* with respect to a given resource budget. In this paper, we focus on resource usage interactions between subproblems, and presume that no functional interactions between subproblems can occur.

2 The design method

Choice of design process model. If the same design process model is to be used at any point during the course of learning, that model must be *robust* enough to recover from mistaken assumptions about subproblem interactions. On the other hand, after its *knowledge base* has been improved, the design process must be able to use the learned knowledge in such a way that its *performance* is improved. In this section, we describe a simple design process model that has both of these properties. This model has been implemented in a program called CPS. CPS is a search-based design system that performs top-down refinement, patching, and chronolog-

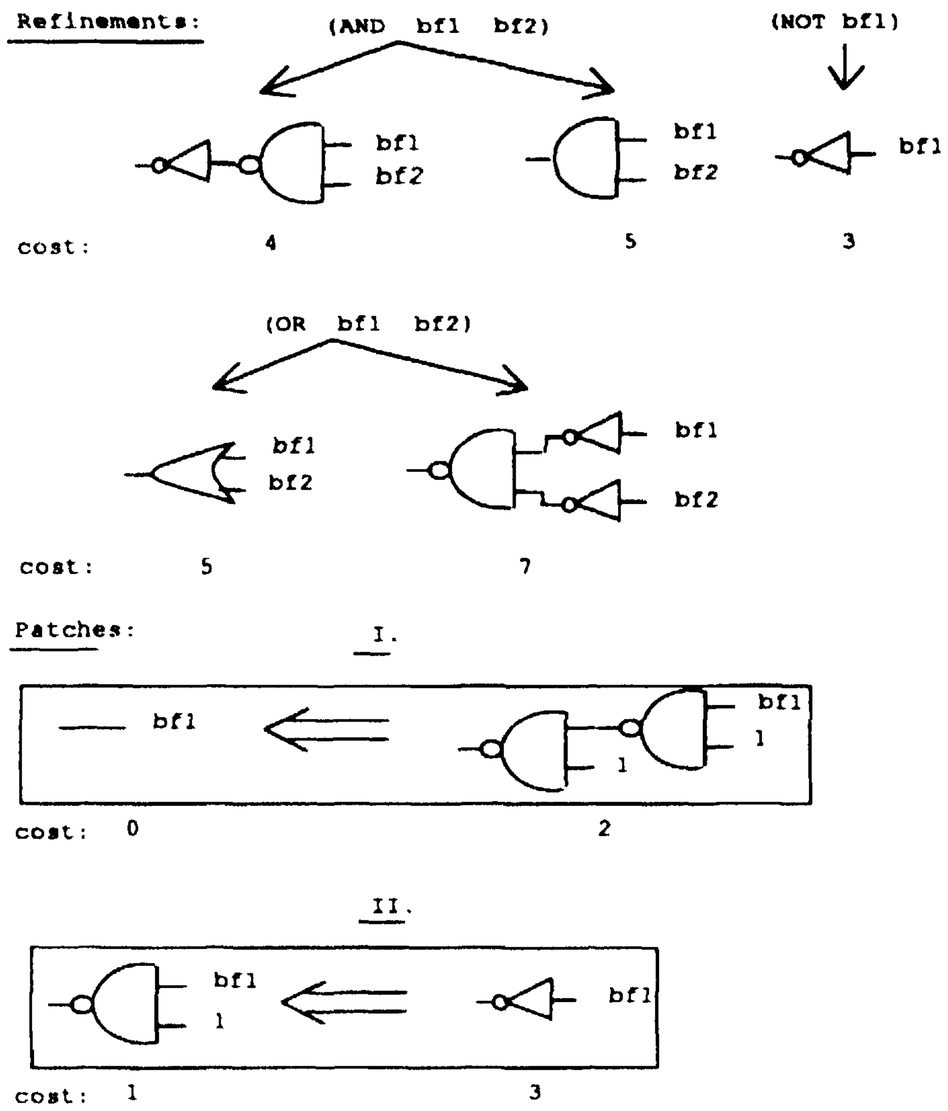

Figure 3: Initial knowledge base for CPS

ical backtracking. We will illustrate the behavior of CPS in solving the problem in Figure 1.

Input. The input to CPS is a boolean expression (composed of binary AND and OR functions, and the NOT function), an evaluation function that defines design optimality, and a global resource budget that must be achieved. For the purpose of simplifying the learning method (so it needn't verify that the design produced by the performance system is optimal), the budget we give CPS is the cost of an optimal solution. By *optimal solution* we mean the circuit in the space of circuits defined by the refinement and patching rule base (see below) that costs the least (with respect to the evaluation function).

Output. The output is an optimal cost, TTL gate-level circuit, composed of binary AND-gates, OR-gates, and NAND-gates, as well as inverters. Figure 1 gives an optimal solution to our example problem.

The knowledge base. The domain knowledge base is depicted in Figure 3.

Refinement. Design (sub)problems are solved either directly, using a single implementation rule (e.g., for subgoal 4 in Figure 4, "Implement (AND u v)") or indirectly by (implicitly) decomposing the problem. In the latter case, the subproblems are determined recursively. CPS finds all refinement rules that apply to the current expression (initialized to be the entire problem). Of these applicable refinement rules, CPS chooses one whose "left hand side" is *maximally specific,* i.e., one that applies to the largest possible sub-expression of the current expression.[3] Thus rules that refine expressions of the form (OR (AND x y)(AND w ■)) will be selected over more general rules that refine expressions of the form (OR x y). In our example, based on the refinement rules in the initial knowledge base (all of whose "left hand sides" are primitive functions), the only rules that apply to the top-level OR simply have (OR x y) as their left-hand side. Subgoal 1 is created in response and added to the implementation goal hierarchy (see Figure 4). Then for each non-literal argument, the same process is repeated, creating subgoals 2 through 7.

In general, the result of refinement can itself be decomposable. However, in our simple boolean function domain, the subproblems of the original problem are directly implement able, because the refinement rules that implicitly construct the decomposition are all implementation rules. The implementations chosen for each subproblem (after some search) are illustrated in Figure 4.

Evaluation function. Faced with a choice among different refinement alternatives for the same subgoal, CPS will prefer the alternative that is locally optimal with respect to a given evaluation function. In our examples, the evaluation function is one that prefers NAND gate implementations:

5×#ANDGates+5×#ORGates+3×#INVs+#NANDGates

Selecting locally optimal refinement alternatives does not guarantee a globally optimal solution. The refinement choices that do lead to the optimal cost solution are illustrated in Figure 4. Only 5 of the 7 are locally optimal.

Patching. After CPS constructs a global solution, if it detects a budget violation, it repeatedly selects and executes applicable patch rules until no more patching is possible. Figure 4 illustrates the patching process for the subproblems in our example.

Backtracking. Backtrackable choice points are created whenever a particular implementation task has alternative solutions. Chronological backtracking occurs whenever a design has been *completely* implemented (and patched) and the global budget has not been met; an implementation choice and all patches that depend on it are retracted. CPS stops at the first complete solution that meets the budget. In the boolean function domain, given only the (easily acquired) knowledge in the initial knowledge base (Figure 3), we cannot backtrack when partial solutions violate the budget because, in many cases, further patching opportunities (leading to an optimal cost design) may become available only after *additional* choices have been made.

In our example (Figure 4), the partial solution after the implementation of subproblem 6 has a cost of 7, exceeding the global budget. Only when the final subproblem is implemented and the solution patched does the implementation meet the budget.

Termination. When all the implementation goals have been achieved, and the resulting design does not violate the budget, the design process is completed.

3 The learning method

SCALE is the learning component of our system. After CPS solves a problem, SCALE'S learning method ratio-

[2] The ARGO circuit design system [Huhns and Acosta, 1989] uses a similar heuristic.

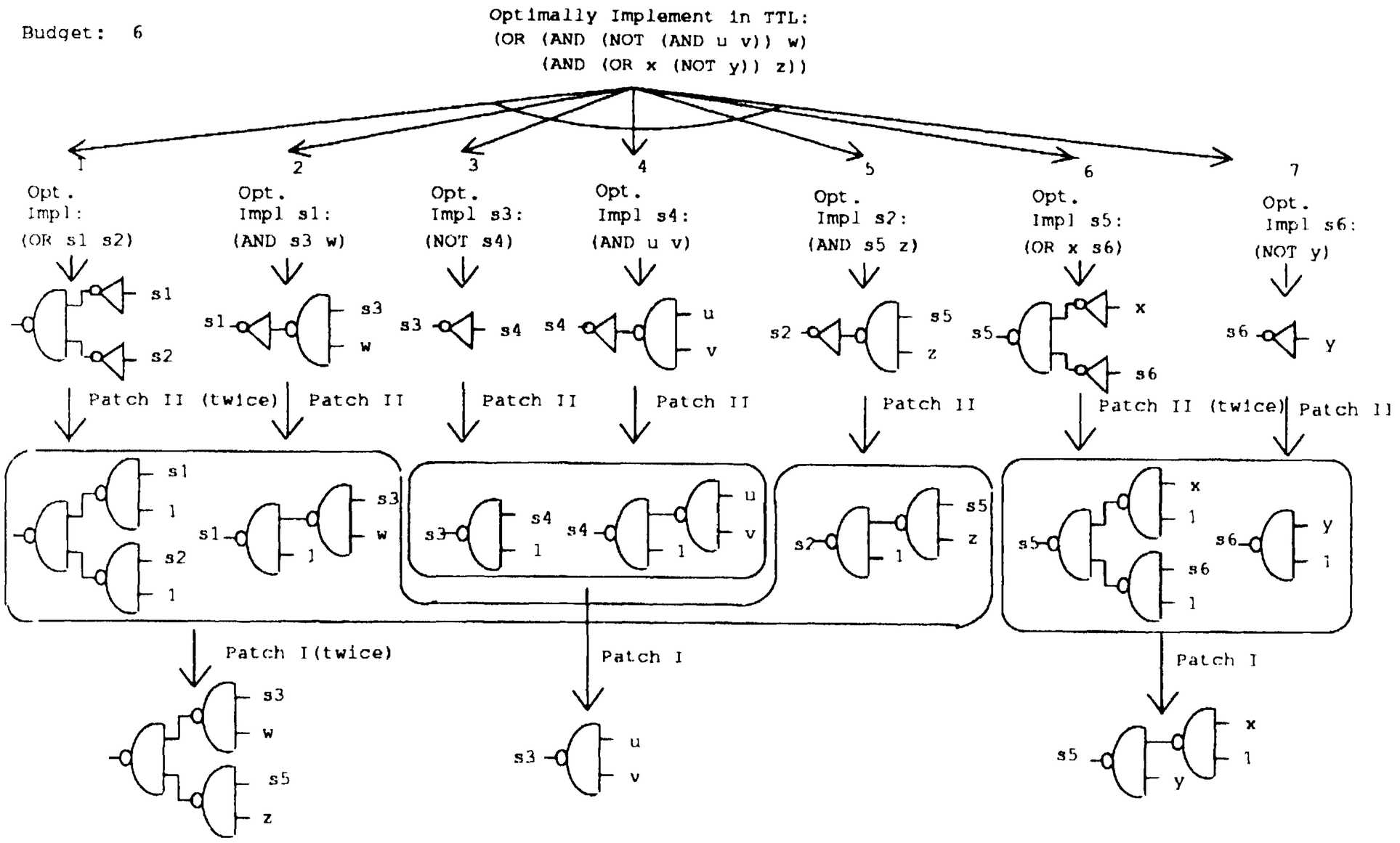

Figure 4: Solution tree produced by CPS

nally reconstructs the problem decomposition by analyzing dependencies. It then generalizes from the non-interacting subproblems in the reformulated decomposition; new refinement rules that identify and optimally implement similar non-interacting subproblems are created. The input to SCALE is a CPS problem-solving trace. The trace includes the original problem, the sequence of refinement and patch operations which led to the optimal solution, and the final implementation. To illustrate SCALE'S behavior, we will use the example of Section 1.

We use an explanation-based learning (EBL) approach [Mitchell *et* a/., 1986] to *explain* the training example and *justifiably generalize* from it one or more refinement rules:

- The *training example* (the input to SCALE) is a design problem and the trace of CPS solving it.
- The *domain theory* is the knowledge base and an evaluation function /.
- The *goal concept* is one or more "good" refinement rules, LHS—>RHS, where LHS is a generalized *cluster* (see below), and RHS is an implementation of LHS that is optimal with respect to /.
- The *operationality criterion* requires the LHS of the rule to be a boolean expression, and the RHS to be a network of gates.

Explaining subproblem interactions using clusters. By analysing dependencies in the final optimal design, SCALE identifies *clusters* of interacting subproblems. These clusters consist of one or more of the subexpressions that were created when the problem was originally decomposed by refinement rules that matched its sub-expressions. A cluster consists of *more than one* sub-expression when the successful application of a patch depends upon the existence of several sub-expressions. The patches can be thought of as "gluing" the sub-expressions together. If the same sub-expression is involved in several patches, all the "glued" sub-expressions are clustered. The cluster is a *single* sub-expression in the special case when CPS has derived a better implementation for a primitive function (AND, OR, NOT) than was originally provided by the initial set of refinement rules.

In our example, three clusters of interacting subproblems are detected (see the circled subproblem implementations in Figure 4, which match the subproblems in Figure 1). The first cluster groups subproblems 1, 2, and 5. Patch II converts two inverters created by the implementations for subproblems 1 and 2 into two NAND gates; since these two gates are in series, they are then eliminated by Patch I. These two patching rules apply in a similar way to subproblems 1 and 5. One cluster is formed for both patch applications (rather than two separate clusters) because both depend in part on subproblem 1. Several uncircled clusters are also formed; these clusters correspond to new, optimal implementations for OR, AND, and NOT.

The EBL goal concept requires that the RHS of a learned rule be an optimal implementation of the LHS. The global implementation is known to be optimal because CPS is required to meet an "optimal cost" budget. By construction, the clusters do not interact; the optimality of the global implementation implies the optimality of the cluster implementations.

Boolean expressions	Circuits
1. (AND x y)	1. (NAND-G (NAND-G x y) 1)
2. (OR x y)	2. (NAND-G (NAND-G x 1) (NAND-G y 1))
3. (NOT x)	3. (NAND-G x 1)
4. (OR (AND x y) z)	4. (NAND-G (NAND-G x y) (NAND-G z 1))
5. (OR (AND x y) (NOT z))	5. (NAND-G (NAND-G x y) z)
6. (OR x (AND y z))	6. (NAND-G (NAND-G x 1) (NAND-G y z))
7. (OR (NOT x) (AND y z))	7. (NAND-G x (NAND-G y z))
8. (OR (AND x y) (AND w z))	8. (NAND-G (NAND-G x y) (NAND-G w z))
9. (OR (NOT x) y)	9. (NAND-G x (NAND-G y 1))
10. (OR x (NOT y))	10. (NAND-G (NAND-G x 1) y)
11. (OR (NOT x) (NOT y))	11. (NAND-G x y)
12. (NOT (AND x y))	12. (NAND-G x y)

Table 1: Rules added to the initial knowledge base to form the closure

Cluster generalisation. The explanation process has parsed the solution tree into subproblem clusters and their optimal implementations. SCALE generalises each of these into a refinement rule. For example, the rule learned from the first cluster in Figure 4 has (OR (AND varl var2) (AND var3 var 4)) as its generalised LHS and (NAND-G (NAND-G varl var2) (NAND-G var3 var4)) as its generalised RHS. The LHS generalises the function arguments from the original sub-expressions at the leaves of the goal hierarchy, while the RHS generalizes the implementations of those same arguments.

The long-term "goal concept" of SCALE'S learning is a *set* of refinement rules that implicitly decompose any problem in the domain into non-interacting subproblems (and create optimal implementations for these). SCALE'S goal concept is instead expressed in terms of learning a single rule, and cluster formation. The next section discusses some of the ideas behind why SCALE'S goal concept leads to achieving the long-term goal concept in the boolean function domain.

4 Discussion

Closure: completely learning a knowledge base. One view of SCALE is that it compiles the optimisation knowledge of the patch rules into new refinement rules. Learning these "good" rules is attractive when using knowledge bases like that of our example, for which only a finite number (a *closure)* of good rules exist. After a relatively small number of examples (less than twenty, on average, in fifteen experiments involving randomly generated boolean expressions), SCALE converges to the set of rules listed in Table 1. These rules have the form: IF boolean expression THEN implement as circuit. SCALE forms the closure by adding these rules to CPS's knowledge base.

Design as search: efficiency speed-up. Figure 5 illustrates the results of our experiments to measure the improvement in efficiency of a search-based design system due to repeated application of our learning method.

The "before learning" curve illustrates the behavior of CPS using the initial knowledge base (Figure 3) on a set of randomly generated test problems. Chronological backtracking, optimization patching, and a target optimal budget combine to control CPS's search for an optimal circuit. As discussed in Section 1, the time complexity of such a problem solver is exponential.

SCALE increases the efficiency of CPS by transforming it from a search-oriented knowledge-based design system (which is relatively easy to build) into an efficient (one pass) knowledge-based compiler which produces optimal designs. Because CPS uses the "maximally specific" heuristic, and draws its refinement rules from the knowledge base closure (guaranteed to produce non-interacting, locally optimal solutions), it has compiler-like behavior; because the resulting subproblems do not interact, CPS simply composes subproblem solutions, producing a global solution *without backtracking or patching.* The "after learning" curve in Figure 5 illustrates the behavior of CPS on the same problems as the "before learning" curve after the closure has been learned (the "after learning" knowledge base was expanded by learning from a separate, randomly generated problem set). It shows that CPS produces the target solutions in linear time by generating the fewest possible nodes (one for each subproblem in the decomposition).

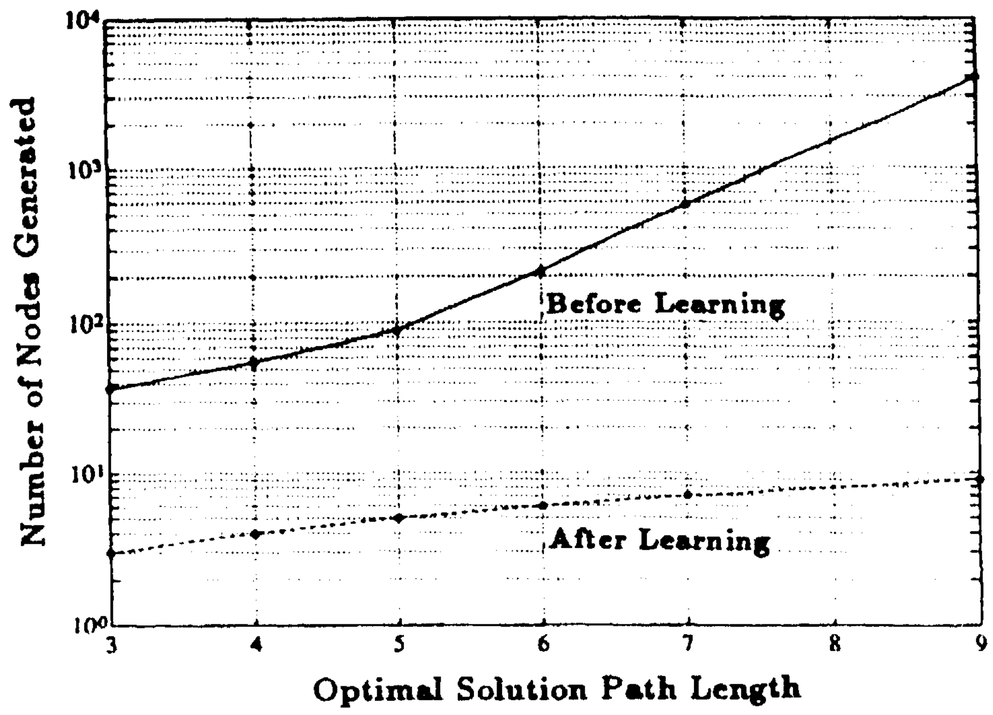

Figure 5: Design efficiency before and after learning the closure: The *x*-axis represents length of shortest path to an optimal solution in the search space defined by the closure of CPS's knowledge base. The *y*-axis is *logarithmic.*

Integration of knowledge. SCALE adds newly learned rules to the knowledge base, but does not remove existing rules. This raises two issues of *knowledge integration:* How does SCALE limit the new rules learned so as to minimize redundancy in the rule base? After new rules have been learned, how does the performance system select among competing rules, so that the overall performance after learning improves?

A common problem of rule learning is *swamping,* learning rules that express the same knowledge redundantly (see [Fikes *et* a/., 1981]). A central contribution of our work is the selective learning of "good" refinement rules that produce non-interacting subproblems (rather than all possible rules for a knowledge base). SCALE limits redundancy because good rules express knowledge about refinement rule interactions; thus no refinement rule application can be re-expressed simply as an application of other refinement rules. Of course, the refinement rules will always be re-expressible as an application of the primitive refinement rules *and* patch rules.

The use of the "maximally specific" heuristic

gives CPS the desirable property of *non-interference:* subsequently-learned rules never interfere with CPS's ability to solve previously worked problems. This is because SCALE always learns rules from the maximal clusters of a problem trace. On subsequent attempts at solving the same problem, the "maximally specific" heuristic ensures that it will choose the refinement rules for the same maximal clusters.

Solution optimality. When restricted to boolean expressions whose literals are distinct positive logic values (i.e., *x*, *y*, *z,* etc.), CPS generates gate-level implementations of boolean expressions that are "optimal" with respect to the space of all designs it can generate. CPS cannot produce an optimal solution for *all* boolean expressions because it does no structure-sharing; for example, if there are two occurrences of (NAND-G x y) in the final implementation of a circuit, the output of a single NAND-gate could be directed to both inputs which require it.

Scope of the learning method. SCALE learns about *local resource usage interactions* between subproblems that do not interact functionally. It presumes that the patch rules improve the design with respect to the evaluation function. It does not reason about optimising by global structure-sharing.

SCALE could be applied to *any* initial knowledge base of refinement and monotonic patch rules for our example domain, including one where the results of refinement rules are expressed as optimal NAND-gate configurations. We chose to start with a sub-optimal knowledge base to show that we can relieve the human engineer of the requirement of providing an optimal base, and *still* produce a linear time optimizing compiler.

5 Conclusions

Summary. This paper has presented a method that incrementally transforms a search-based circuit design system (CPS) that uses only primitive (and non-redundant) refinement and patch rules into a compiler-like design system with the following properties:

- *Non-redundant rule-set.* No refinement rule application can be re-expressed as an application of other refinement rules.
- *No backtracking or patching.* The system produces a design that meets an initially specified resource budget without patching or backtracking.
- *Coincidence of local optimality with global optimality.* Composing locally optimal solutions to subproblems created by refinement rules results in a solution that is globally optimal.

When and what to learn is based on analyzing design step dependencies, and interactions of design steps with respect to optimization. This learning method has been implemented in the SCALE program, which converges on the "best possible" knowledge base (the "closure") from a simple initial knowledge base, using (on the average) less than twenty random design problems as training examples. For further details on the evolution of the CPS/SCALE system, see [Tong and Franklin, 1989].

Acknowledgements

We thank Jack Mostow, Mike Barley, Wes Braudaway, Dawn Cohen, Chun Liew, Sridhar Mahadevan, and Armand Prieditis for providing helpful comments on earlier drafts of this paper. We also thank the members of the Rutgers AI/Design Project for the stimulating environment they provide.

References

[Brown and Chandrasekaran, 1989] D. Brown and B. Chandrasekaran. Investigating routine design problem solving. In C. Tong and D. Sriram, editors, *Artificial Intelligence Approaches To Engineering Design.* Forthcoming, 1989.

[Fikes *et al.,* 1981] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. In B. L. Webber and N. J. Nilsson, editors, *Readings in Artificial Intelligence.* Morgan Kaufmann, 1981.

[Huhns and Acosta, 1989] M. Huhns and R. Acosta. Argo: An analogical reasoning system for solving design problems. In C. Tong and D. Sriram, editors, *Artificial Intelligence Approaches To Engineering Design.* Forthcoming, 1989.

[Kant and Barstow, 1978] E. Kant and D. Barstow. The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis. *IEEE Transactions on Software Engineering,* 9, 1978.

[Knapp and Parker, 1986] D. Knapp and A. Parker. A design utility manager: the ADAM planning engine. In *Proceedings of the 23rd Design Automation Conference.* IEEE, June 1986.

[Kowalski, 1985] T. Kowalski. *An Artificial Intelligence Approach to VLSI Design.* Kluwer Academic Publishers, Boston, 1985.

[Mitchell *et al.,* 1986] T. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning,* 1(1), 1986.

[Mittal, 1986] S. Mittal. Pride: An expert system for the design of paper handling systems. *IEEE Computer,* 19(7), July 1986.

[Stefik, 1981] M. Stefik. Planning with constraints (MOLGEN: Part 1). *Artificial Intelligence,* 16(2), May 1981.

[Steinberg, 1987] L. Steinberg. Design as refinement plus constraint propagation: The VEXED experience. In *Proceedings of the National Conference on Artificial Intelligence.* AAAI, 1987.

[Tong and Franklin, 1989] C. Tong and P. Franklin. Toward automated rational reconstruction: A case study. In *Proceedings of the Sixth International Machine Learning Workshop,* June 1989.

[Tong, 1988] C. Tong. *Knowledge-based circuit design.* PhD thesis, Dept. of Computer Science, Stanford University, 1988.