Highest Utility First Search: a Control Method for Multi-level Stochastic Design[1]

Louis Steinberg
J. Storrs Hall[2]
Brian D. Davison
Department of Computer Science, Rutgers University
New Brunswick, NJ 08903, USA
{lou,davison}@cs.rutgers.edu

Abstract

An intrinsic characteristic of stochastic optimization methods, such as simulated annealing, genetic algorithms and multi-start hill climbing, is that they can be run again and again on the same inputs, each time potentially producing a different answer. When such algorithms are used in a design process with multiple levels of abstraction, where the output of one stochastic optimizer becomes the problem statement for another stochastic optimizer, we get an implicit tree of alternative designs. After each optimizer run we face a control problem of which level's optimizer to run next, and which design alternative to run it on. This problem is made more difficult by the fact that we generally can get a precise evaluation of the design alternatives only at the lowest level (the final results), and must make do at higher levels with only an estimate of how good a final design each alternative will lead to.

[1]The work presented here is part of the "Hypercomputing & Design" (HPCD) project; and it is supported (partly) by ARPA under contract DABT-63-93-C-0064. The content of the information herein does not necessarily reflect the position of the Government and official endorsement should not be inferred.

[2]Current address: Institute for Molecular Manufacturing, 123 Fremont Ave, Los Altos, CA. email: josh@imm.org

We present the Highest Utility First Search (HUFS) control algorithm for this problem. HUFS is based on an estimate we derive for the expected utility of starting the design process from any given design alternative, where utility reflects both the intrinsic value of the final result and the cost in computing resources it will take to get that result. This estimate is comparable across levels of the hierarchy. HUFS is essentially best first search where "best" is defined by this expected utility. We also present an empirical study applying HUFS to the problem of VLSI module placement, which demonstrates the superiority for HUFS over the common "waterfall" control method in this setting.

1 INTRODUCTION

An intrinsic characteristic of stochastic optimization methods, such as simulated annealing, genetic algorithms and random multi-start hill climbing, is that they can be run again and again on the same inputs, each time potentially producing a different answer. Some of the answers will be better, some worse. Thus, after each run of such an optimizer we have a choice: we can use the best of the results from the runs we have done so far, or we can run the optimizer again in the hope of getting a still better result.

The issue gets more complicated when the design task at hand is being done in a series of stages, or levels of abstraction. E.g., in designing a microprocessor, we might start with an instruction set, implement that as a set of register transfers, implement the register transfers as boolean logic, implement the boolean logic as a "netlist" defining how specific circuit modules are to be wired together, implement the netlist by choosing specific locations for the circuit modules and wires on the surface of a VLSI chip, etc. In many cases, such systems work by taking a design at one level, translating it to a correct but low-quality design at the next level, and running an optimizer on that design to produce a high-quality design.

Thus, the output of the optimizer is not a final design. Instead it is used as input to the next lower optimizer.[3] Thus, if the optimizers use stochastic methods we have an implicit tree of alternate designs. The root of the tree is

[3]Strictly speaking, it is input to a translator and it is the the output of the translator which is input to the next optimizer. However, since our focus in this paper is on the optimizers, we will ignore the translators and speak as if the output of one optimizer is used directly as input to the next optimizer.

the initial design specification, and the children of any parent node are all the designs that the optimizer for the next level could produce, given the parent node as input. Leaves of the tree are the concrete, ground-level designs, one of which will be final result of the overall design system.

Each time an optimizer at any level gives us an output, we can declare the best design we have at this level to be "good enough" and proceed to use it as input to the next level, we can rerun the optimizer in hopes of getting a better design at the current level, or we can go back up to some higher level in the tree and try again there.

Current practice is often to use a "waterfall" approach — run the optimizer at one level some fixed number of times, choose the best of the results from these runs to be the parent for the next level, and proceed downwards. This would be an optimal approach if we had a completely accurate way of evaluating designs at intermediate levels (i.e., evaluating how good a final design we will get from this intermediate design), but in practice all we normally have are heuristics that tell us *approximately* how good an intermediate design is, and the only way to get a fully accurate evaluation is to carry the design out all the way to the ground level. Doing so at each choice point amounts to exploring the whole tree and is infeasible. Furthermore, the huge branching factor of this implicit tree has made it hard to know how to do even partial lookahead in any automated way.

This paper presents an alternative to the waterfall control method, called "Highest Utility First Search" (HUFS), which explores the tree of alternatives in a much more flexible manner than waterfall. We will describe HUFS and present empirical data from one example design task showing that HUFS can be a significant improvement over waterfall search.

HUFS is based on the idea that a good control method is not one that finds the best design (since finding the absolute best design takes exponential time for many of these problems), but one that gives the best tradeoff between computation cost and design quality, and that therefore control decisions should be based on an analysis of the *utility* of each possible computation, i.e. the value of the result minus the cost of doing the computation. This approach was inspired by the "rational meta-reasoning" advocated by Russell and Wefald [Russell and Wefald, 1991].

In our context, we can use the notion of the utility of a computation to define the utility of a design alternative. The utility of a design alternative d, which we will refer to as $Udesign(d)$, is the expected utility of a design process that starts with d as input and returns a ground-level design from among the descendants of d. I.e., the utility of d is the expected difference between the value of the final design we will get if we use d as our starting point and the cost of the computation it will take to get this design. (Of course, the utility depends on the algorithm we will be using to do the design.)

The basic idea of HUFS is very simple:

> Find the design alternative d_{opt} with the highest *Udesign*, among all the design alternatives you currently have on all the levels, and generate one child from d_{opt}, that is, run the appropriate level's optimizer with d_{opt} as input. Repeat this process until d_{opt} is a ground-level design, then stop and return d_{opt} as the result

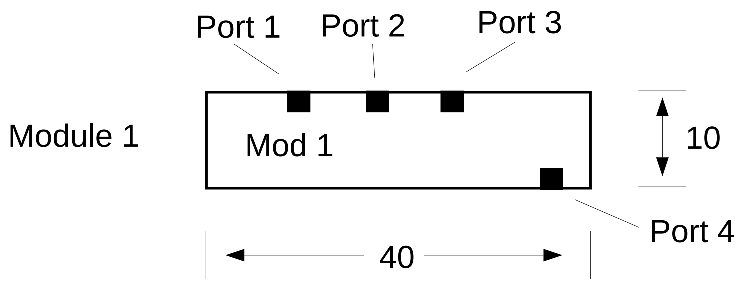
To do his, however, we need some way of estimating $Udesign(d)$. Section 3 below will explain how we do this, and will present HUFS in more detail. Section 4 will describe our empirical test of HUFS, Section 5 discusses some further issues, and Section 6 covers related work. In order to provide a concrete example to use in these sections, Section 2 will first describe the design problem we have used as the main testbed for our research on HUFS.

2 The Example Problem: Module Placement

In this section we will describe the example problem that we have been using to drive our work. This is the problem of positioning rectangular circuit modules on the surface of a VLSI chip: a given set of rectangles must be placed in a plane in a way that minimizes the area of the bounding box circumscribed around the rectangles plus a factor that accounts for the area taken by the wires needed to connect the modules in a specified way.

The input to the placement problem is a "netlist". A netlist specifies a set of modules, where each module is a rectangle of fixed size along with a set of "ports". A port is simply a location within the rectangle where a wire may be connected. In addition to giving the modules, a netlist specifies a list of "nets" specifying which ports of which modules must be connected by wires. (See Figure 1.)

The output from a placement problem is a location and orientation for each module. Modules may be rotated by any multiple of 90 degrees and/or reflected in X, Y, or both. (See Figure 2.) Rectangles may not overlap.



Modules:

    Module1: size: 10 x 40, Ports: Port1: $< 15, 10 >$, . . .
    Module2. . .

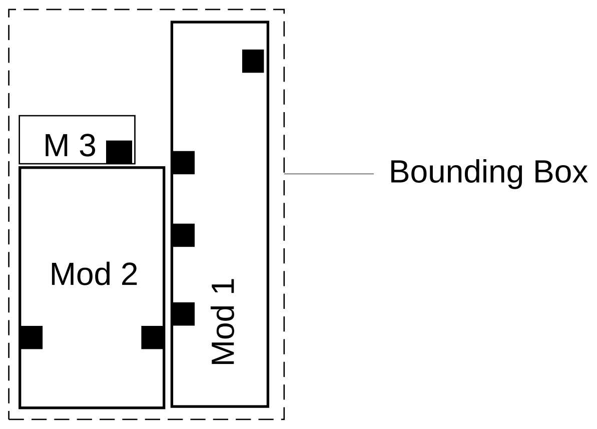
Nets:

    Net1: [Module1 Port1], [Module2 Port2], [Module3 Port1]
    . . .

Figure 1: The Module Placement Problem — Inputs

The quality of a placement is determined by two factors. One is the amount of wasted space in the placement, i.e., the area of a rectangular bounding box drawn around all the modules minus the total area of the modules themselves. The other factor is the estimated area taken by wires. Wire area is a the total estimated length of wire needed to connect the ports as specified by the netlist times a constant representing the wire width. Wire length for a single net is estimated as half the perimeter of a rectangular bounding box around the ports the net connects, and wire length for the entire netlist is the sum of the lengths for the nets. Note that the wire length is itself only a heuristic estimate, but for the purpose of this work we take it as our ground-level evaluation. The final "score" of a placement is a weighted sum of the wasted area and the wire area. The lower the score, the better the design.

We break the placement process into two stages. First we choose a structure called a "slicing tree". (Figure 3). A slicing tree is a binary tree. Each leaf is a module to be placed. Each non-leaf node represents a rectangular region containing all the modules that are its descendants. E.g., Node B in the tree represents the inner dashed box in the corresponding placement.



Module1: XY: (15, 0), Rotation: 90 degrees, Reflect X: false, Reflect Y: true
Module2: ...

Figure 2: The Module Placement Problem — Output

In particular, a non-leaf node specifies that the two rectangular regions represented by the node's two children will be placed next to each other in a specified relative configuration. E.g., Node A represents one child rotated and placed above the other. Figure 4 shows the four possible configurations, or "adjacencies" — all other combinations of above/beside and rotated/non-rotated can be attained from these by reflecting or rotating the node as a whole. As Figure 3 illustrates, a slicing tree can be seen as recursively slicing the overall circuit area into rectangular regions, subregions, etc.

While a slicing tree specifies relative rotations of its parts, it does not specify reflections. Thus, if you have a concrete placement that corresponds to a given slicing tree, you can reflect any of the subregions in X and/or Y, and get a different concrete placement that still corresponds to the same slicing tree. Figure 5 shows another concrete placement that corresponds to the slicing tree in Figure 3. Note that such reflections cannot change the bounding box area of the circuit, and hence cannot change the wasted area, but they can change the distances between ports, and thus the total wiring area.

We evaluate a slicing tree by a weighted sum of the wasted area and an estimate of the wire area. The wire area estimate is based on module sizes,

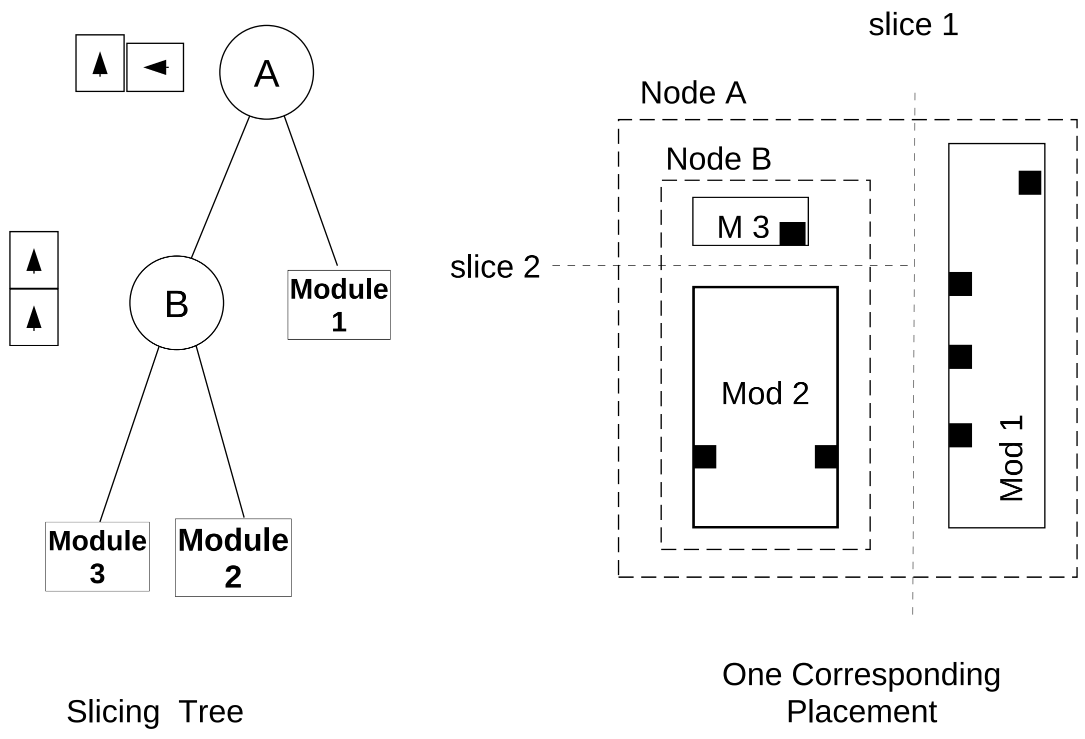


Figure 3: A Slicing Tree and One Corresponding Placement

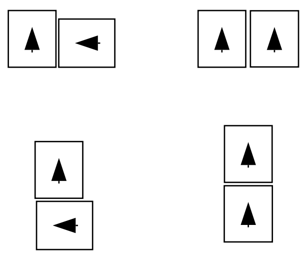


Figure 4: The Four Possible Adjacencies of Subnodes

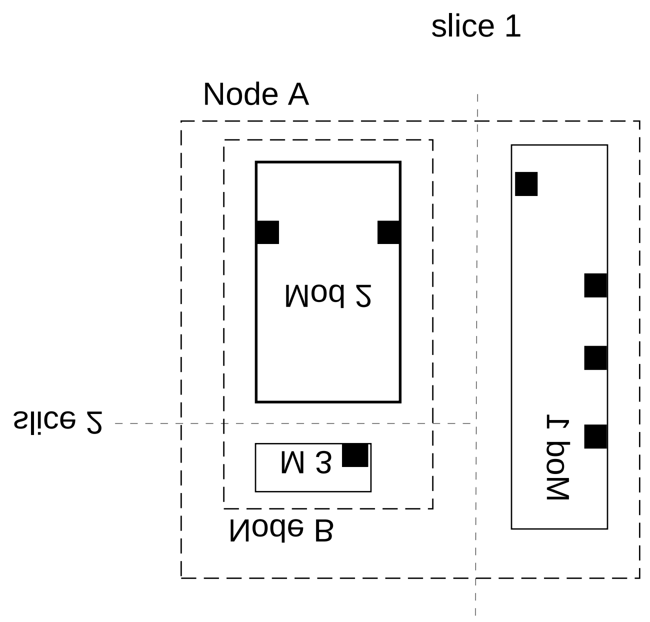


Figure 5: Another Corresponding Placement

and distances in the slicing tree.

We generate a slicing tree by first generating a binary tree with the specified modules as leaves, and then assigning the adjacencies in a bottom-up, greedy manner. We move from the leaves to the root, assigning at each node the adjacency that minimizes the area of the bounding box for this node, given the bounding boxes of the node's children.

The optimizer for slicing trees starts by generating a random binary tree. We define a set of neighbors as those trees that can be reached from the current tree by choosing two nodes and interchanging the subtrees rooted at those nodes, and then revising the adjacencies. At each step of the optimizer, we generate the neighbors of the current tree in a random order until we find a neighbor that is better than the current tree. When we find a better one, we make that the current tree and repeat. If no neighbor is better, the optimizer halts.

The second stage of the placement process converts a slicing tree into a concrete, specific placement by choosing a set of reflections (two bits: reflect in x? reflect in y?) for each node in the slicing tree. This gives us enough constraints to determine a specific location and orientation in the plane for each rectangle, i.e., a concrete placement. Reflections are optimized in the same way that slicing trees are, with one set of reflections defined to be a neighbor of another if they differ in only one bit, i.e. can be reached from each other by changing one of the two reflections at one node of the slicing tree.

3 Expected Utility and Highest Utility First Search

This section will first explain the process HUFS uses to estimate $Udesign(d)$, the expected utility of design using alternative d as the starting point. We will see that some of the information HUFS uses in this process is not directly provided by HUFS' input, so we will then explain how HUFS gets this additional information. Finally, we will give the full HUFS algorithm.

3.1 Calculating $Udesign(d)$

Let us start by considering a simplified case. We will focus on a single-level problem such as that of generating concrete placements from a slicing tree. Also we will assume that HUFS has all of the following information:

- The *cost*, c, of running the optimizer. The cost might be measured, for example, by the CPU time the run takes. We assume that each run has the same fixed cost.
- A heuristic *Score function* $S(p)$, where p is a placement. S gives an estimate of how good a design p is intrinsically. It does not depend on aspects of the problem-solving context such as what other alternatives exist. For example, the score function we used for placements in our empirical tests is a weighted sum of the area of a bounding box around the circuit and an estimate of the total wire length based on distances between ports. For this score function, a lower score is a better design, and we will assume this to be true of all score functions to be used with HUFS.
- A *Value function*, $V(s)$, that tells us the value we place on having a child with score s. This value needs to be in the same units as c. We will assume that values are non-negative, and that a larger value is better.

Since lower scores have higher value, V is monotonic non-increasing. We will explain later why we separate V and S, rather than having a single function corresponding to $V(S(p))$.

- The *Child-Score Distribution* $CSD(s)$. This is a probability distribution. It gives the probability that if we generate a concrete placement its score will be s. The CSD will depend on which slicing tree we are using as input to the optimizer, but for the moment let us assume that we are dealing with one specific slicing tree and that we know its CSD.

Since for now we are considering a single level problem, there is no question of which optimizer to run or which input to give it — there is only one optimizer, and only one possible slicing tree to use as input. All that needs to be chosen is whether to do another run or to stop and declare the best design we have so far to be our final answer.

To make this choice, we consider the utility of doing one more run. If the utility of doing that one run is positive, i.e., if the value of doing the run exceeds the cost, it clearly makes sense to do that run, and if not it makes sense to stop. We assume we know c, the cost of a run, so to determine the utility we just need to determine the value of the run. We call this value the *expected incremental value*, EIV, of doing a run. It is the average expected increase in value (based on the information we have now) from the best design we have now to the best design we will have after the run.

The better the current best score is, the less likely it is that another run will do better, so the EIV depends on the best score we have so far. If s_b is the current best score and s_n is a random variable representing the score we will get on the next run,

$$\begin{aligned} EIV(s_b) &= E(\max(V(s_b), V(s_n)) - V(s_b)) \\ &= \sum_s P(s)(\max(0, V(s) - V(s_b)) \end{aligned}$$

Where E is Expected Value, s ranges over all possible child scores and $P(s)$ is the probability that the next child will have score s. Since the $P(s)$ is the CSD of the parent, and since V is monotonic decreasing,

$$\begin{aligned} EIV(s_b) &= \sum_s CSD(s)\max(0, V(s) - V(s_b)) \\ &= \sum_{s<s_b} CSD(s)(V(s) - V(s_b)) \end{aligned}$$

$$= \left(\sum_{s<s_b} CSD(s)V(s)\right) - V(s_b)\sum_{s<s_b} CSD(s)$$

Note that s_b cannot increase as we do more runs (since the best score is the lowest score we have seen), and that as s_b decreases, $EIV(s_b)$ cannot increase. Therefore, once the EIV is less than c it will stay less than c for all further runs, so the expected utility of doing any number of further runs is negative, and the rational control decision at this point is to stop.

In other words, if we define the *threshold score* s_t to be such that $EIV(s_t) = c$, then the optimal strategy for a single-level problem is to continue generating children until we get one whose score is less than s_t.

Now, given the stopping criterion, we can calculate the expected value and cost of the whole (single level) process. The expected value, EV, of the final concrete placement is the expected value of the first score we find that is less than s_t, which is just the average of the values of these scores weighted by the *relative* probability of each score, i.e., the probability of getting that score given that we got some score less than s_t:

$$EV = \sum_{s<s_t} V(s)CSD(s) / \sum_{s<s_t} CSD(s)$$

The chance of finding a score under s_t in one run is

$$\sum_{s<s_t} CSD(s)$$

so the average number of runs to find such a score is

$$1/\sum_{s<s_t} CSD(s)$$

and the expected cost, EC, is

$$EC = c/\sum_{s<s_t} CSD(s)$$

So the Expected Utility, $Udesign(d)$, of the overall design process will be

$$Udesign(d) = EV - EC = \frac{\sum_{s<s_t} V(s)CSD_d(s)}{\sum_{s<s_t} CSD_d(s)} - \frac{c}{\sum_{s<s_t} CSD_d(s)}$$

A curious property to note is that $EIV(s_t) = c$ implies (by algebra on the formula above for EIV) that

$$V(s_t) = \frac{\sum_{s<s_t} CSD_d(s)V(s) - c}{\sum_{s<s_t} CSD_d(s)}$$

which is just $Udesign(d)$. That is, the utility is just the value of the threshold score s_t. Since we stop for any score better than s_t the average value of the score we stop at will be better than $V(s_t)$. But the average utility is the average stopping value minus the average cost of the runs, and subtracting the cost of the runs brings us exactly back to $V(s_t)$.

In general, then, the expected utility of generating a final ground-level design from a design alternative d at the next higher abstraction level is

$$Udesign(d) = V(s_t(CSD, V, c))$$

where $s_t(CSD, V, c)$ is the score such that

$$(\sum_{s<s_t} V(s)CSD(s)) - V(s_t)\sum_{s<s_t} CSD(s) = c$$

The utility of a given non-ground object is determined by its CSD, so we can define the utility of the CSD itself to be

$$Ucsd(CSD) = V(s_t(CSD, V, c))$$

Then,

$$Udesign(d) = Ucsd(CSD) = V(s_t)$$

Note that the stopping criterion $s_b < s_t$ is equivalent to $V(s_b) > V(s_t)$. But $V(s_t) = Udesign(d)$. Also note that the utility of a ground level design is just its value; if we start with a ground level design there is no more computing needed, so the "cost" of starting with such a design is zero. Therefore, if p_b is the placement (i.e., ground level design) whose score is s_b and t is the slicing tree we are generating placements for, then the stopping criterion is equivalent to

$$Udesign(t) < Udesign(p_b)$$

That is, we generate children from t as long as it has a higher $Udesign$ than any other design alternative, and stop the design process when the design alternative with the highest $Udesign$ is at the ground level. When we look at it this way, we can see our single-level algorithm as a special case of the multi-level algorithm summarized briefly in Section 1.

In this subsection we have shown how to compute $Udesign(d)$ from CSD, V, and c in a single-level case. However, in general we do not know a priori what CSD is. Different slicing trees have different CSDs, and, especially in the multi-level case where there may be combinatorially many potential slicing trees, it is not reasonable to ask that their *CSD*s be provided as inputs to HUFS. Therefore, HUFS has to estimate the *CSD*s for itself. In following subsections we will discuss how it does so, and then show how the approach we have discussed for a single-level problem can be extended to multiple levels. Before we cover these topics, however, we will explain the notational conventions we use in the rest of this paper.

3.2 Notation

We will number the abstraction levels from the lowest level to the highest, with ground-level designs (e.g. concrete placements) being at level 0. So a slicing tree is at level 1 and a netlist at level 2. We will use superscripts to denote levels. Thus for our example problem either d^{tree} or d^1 would refer to a slicing tree design alternative, while d^0 would be a concrete placement.

S^i and V^i are the score function for level-i objects and the value function for level-i scores, and c^i is the cost of generating a level-i object.

The distribution $CSD(s)$ depends not only on the level but on which specific parent we are generating from. We will use subscripts to specify the parent a CSD refers to, so $CSD_{d^1}(s)$ is the probability that a child of alternative d^1 will have score s. We will use similar notation with other probability distributions we mention.

3.3 Estimating *CSD*

In general, we will not know the exact CSD for any design alternative. Instead, we make a heuristic estimate of the parent's CSD. If from this estimate it appears worthwhile to generate children we do so, and as we see the scores of these children we update our estimate of the CSD using a Bayesian method. This section will describe how we form our initial estimate of the CSD and how we update it.

We model the *CSD*s for a given level as all coming from some parameterized family of distributions, e.g. the family of normal distributions with parameters specifying the mean and standard deviation. We assume the family is specified as part of the input to HUFS. Given the family we can specify a particular CSD by a tuple R containing a value for each of the family's parameters. Thus, $R =< 10, 2 >$ might represent a normal distribution with mean 10 and standard deviation 2. Also, we define the function $R(d)$ to mean the R tuple that corresponds to CSD_d, the CSD of design alternative d.

The user specifies the family of distributions by giving the function $CSDR$ (the "Child Score Distribution given R"), which maps an R tuple into the actual distribution. Then $CSDR_R$ is the specific distribution from the family that corresponds to R, and

$$CSDR_R(s) = P(S(child) = s | R(parent(child)) = R)$$

For example, if the family is the normal distributions, $CSDR_{<\mu,\sigma>}(s) = \sqrt{2\Pi}\sigma e^{-(s-\mu)^2/(2\sigma^2)}$.

The problem of determining CSD_d then becomes the problem of determining $R(d)$. We cannot determine exactly what $R(d)$ is, but we can estimate the probability that $R(d) = R$ for any specific R. This is equivalent to estimating the probability that d has any given CSD in the family of CSDs. We define the probability distribution $RD_d(R)$ (the "R Distribution of d") to be the probability that $R(d) = R$. We make an initial estimate of RD_d and use it to make our initial estimate of the CSD, then as we see scores of d's children we update our estimate of RD_d and use it to get our updated estimate of CSD. We will next discuss how how we get the initial estimate of RD_d, then how we update that estimate, and then how we derive an estimate of CSD_d from an estimated RD_d.

Our initial estimate is based on the parent's score. Just as we assumed we have a heuristic score function, S^0, on the children, we assume we have a similar function, S^1, on the parents. In our example, S^1, i.e. $S^{tree}(t)$, is the sum of the module area from the tree and an estimate of wire area based on the number of tree edges and the sizes of the modules between modules that must be connected.

So, in addition to the $CSDR$, the user needs to provide a function $RDS_s(R)$

$$RDS_s(R) = P(R(d) = R | S(d) = s)$$

Continuing our example, if $RDS_{100}(< 10, 2 >) = 0.1$, then a parent whose score is 100 has probability 0.1 of having a child distribution with mean 10 and standard deviation 2.

When we generate children from d, we update our estimate of RD_d by a Bayesian process, using $RDS_{S(d)}$ as our prior estimate and $RD(R)$ as our posteriori estimate, and the Bayesian formula,

$$P(R(d) = R | \text{child scores} = s_1 \ldots s_n)$$

$$= \frac{P(R(d) = R)P(\text{child scores} = s_1 \ldots s_n | R(d) = R)}{P(\text{child scores} = s_1 \ldots s_n)}$$

i.e.,

$$RD(R) = \frac{RDS_{S(d)}(R) \prod_{i=1}^{n} CSDR_R(s_i)}{\int_{R'} RDS_{S(d)}(R') \prod_{i=1}^{n} CSDR_{R'}(s_i) dR'}$$

While assuming we know the *CSDR* (i.e., the family of distributions) and the *RDS* may seem unrealistic, we will see below that in our example problem even a family that very roughly approximates the *CSD*s and a very approximate *RDS* give excellent results.

Now, given our estimate of *RD*, we can estimate *CSD* as

$$CSD_{RD}(s) = \int_R RD(R) CSDR_R(s) dR$$

That is, the probability of a child having score s is the average, over all R values, of the probability of score s given that $R(d) = R$, weighted by the probability that $R(d) = R$.

Since CSD_{RD} is only an estimate of the true *CSD*, and especially since it is an estimate that is changed as a result of generating children, the control strategy of stopping when $EIV(s_b) < c$ is not necessarily the optimal rational strategy. It ignores the fact that if we do another optimizer run we will update the parent's *RD*, and hence change the estimated *CSD*. In the previous subsection, where we assumed a known, fixed *CSD*, once the *EIV* was less than c, further optimizer runs could never increase the *EIV*, so there could be no point in continuing. Here, however, even if the current *EIV* is less than c another optimizer run could change the *CSD* in a way that increases the *EIV*, and thus making it rational to continue.

One way to look at this situation is to say that doing an optimizer run gets you an improved estimate of the parent's *RD*, that doing so has value, and this value may justify doing an optimizer run even when the *EIV* by itself does not. We would like to find a method to include this value in our accounting, but for now it is ignored.

In summary, we can estimate a design alternative's *CSD* from its *RDS*, its score, and the scores of any children we have generated. From the alternative's *CSD* we can estimate its *Udesign*. So far, however, we have assumed we had a single-level problem.

3.4 Multiple Levels: Estimating V

Now we turn to the task of calculating *Udesign* for a multi-level problem. If for each level i we knew V^{i-1}, S^{i-1}, c^{i-1}, and CSD_{d^i} we could apply our single level method to compute $Udesign(d^i)$:

$$Udesign(d^i) = V^{i-1}(s_t(CSD_{d^i}, V^{i-1}, c^{i-1}))$$

In fact, we assume we are given S^{i-1} and c^{i-1} as part of the input to HUFS. We can use the method discussed in the previous subsection to estimate CSD_{d^i} from $CSDR^i$, RDS^i, and S^i, all of which we assume are provided to HUFS, and from the scores of any children of d^i we have generated.

The only difficulty is in determining V^{i-1}. Since a ground-level design is the output the user is asking the design system for, we assume the user can tell us what a such a design is worth, so we assume V^0 is supplied to HUFS. But even if we know what, e.g., a concrete placement is worth, how can we determine the value of a design alternative at a higher level, e.g. a slicing tree?

In fact, in and of itself a slicing tree has no value — it only has value as a starting point for generating placements. Thus it makes sense to define its value in terms of the value of the placement we would ultimately end up with if we started with this slicing tree. Of course, we have to subtract from this value the cost of getting from the slicing tree to that placement. But the value of the resulting placement minus the cost of finding it is just the utility of the slicing tree. So, the value of a slicing tree is just its *Udesign*.

Now, when calculating $V^{tree}(s)$ we may not have a specific tree whose value is s. For instance, in calculating s_t we integrate $V(s)$ over a range of values of s. Thus, what we need is $V^{tree}(s)$, which takes a tree score, not a tree, as its argument. In other words, we need to be able to calculate the a priori value a hypothetical tree *would* have if its score were some given s. Since we know nothing about this tree but its score, we estimate its RD by just applying this level's RDS to the score. In general, for a hypothetical parent design at level i with score s,

$$CSDscore_s(sc) = \int_R RDS^i_s(R)\,CSDR^i_R(sc)dR$$

where sc is the child score whose probability we are calculating, $RDS^i_s(R)$ is the probability that a parent whose score is s will have as its the Child Score

Distribution the one specified by parameter vector R, and $CSDR^i_R(sc)$ is the probability of getting a child with score sc given that CSD.

From $CSDscore_s$ we can calculate the value of the hypothetical design alternative:

$$V^i(s) = Ucsd^i(CSDscore_s) = V^{i-1}(s_t(CSD_{RDS}(s)), V^{i-1}, c^{i-1}))$$

So we can compute V^{tree} from RDS^{tree}, $V^{placement}$, and $c^{placement}$. Given a netlist N, we can compute an a priori $RD^{netlist}$ for N from N's score and $RDS^{netlist}$, and update this RD from the scores of the trees we generate from N. From the RD we can compute CSD_{RD}, and from that, from the function V^{tree}, and from the cost c^{tree} of generating a slicing tree we can compute a threshold score $s^{tree}_t(N)$ for generating slicing trees from netlist N. This allows us to compute $Udesign(N) = V^{tree}(s^{tree}_t(N))$.

Furthermore, our definition of V^{tree} allows us to combine the two single-level analyses (one for generating trees from netlists and one for generating placements from trees) to show that $Udesign(N)$ is not only the utility of generating trees from N, it is also the utility of the whole multi-level process of generating placements from N.

We demonstrate this as follows: since $Udesign(N)$ is our estimate of the utility of using this netlist to generate trees, if EV^{tree} is the expected value of the final slicing tree we will wind up with and EC^{tree} is the expected cost to generate trees until we get that final one, then

$$Udesign(N) = EV^{tree} - EC^{tree}$$

But since the value of a slicing tree is the expected utility of using it to generate placements, if $EV^{placement}$ and $EC^{placement}$ are the expected value and cost of generating placements from the final slicing tree we got, then

$$EV^{tree} = EV^{placement} - EC^{placement}$$

so

$$Udesign(N) = (EV^{placement} - EC^{placement}) - EC^{tree}$$

which is just the expected utility of designing all the way down from the netlist N to a final placement, i.e. a final ground-level design.

Thus, in general, for level $i > 0$,

$$V^i(s) = V^{i-1}(s_t(CSD^i_{RDS^i_s}, V^{i-1}, c^{i-1}))$$

3.5 The General Formula for *Udesign*

The preceding sections introduce and motivate a number of formulas. Next we will put these formulas together into a complete method for computing *Udesign*.

We start by computing the V^i for $i > 0$. This can be done before we actually generate any design alternatives.

$$V^i(s) = V^{i-1}(s_t(CSDscore^i_s, V^{i-1}, c^{i-1}))$$

where $s_t(CSD, V, c)$ is the score such that

$$(\sum_{s<s_t} V(s)CSD(s)) - V(s_t)\sum_{s<s_t} CSD(s) = c$$

and

$$CSDscore^i_s = \int_R RDS^i_s(R)CSDR^i_R(s)dR$$

Note that V^i is defined in terms of V^{i-1}. V^0 is given by the user, so from this we can calculate V^1, from V^1 we get V^2, and so on.

Once we have the V^i's we can compute *Udesign*. To compute $Udesign(d^i)$ for a specific design alternative d at level i, we start by computing d's RD from the RDS of level i and the scores $s_j^{i-1}, 1 \leq j \leq n$ of any children we have generated from d:

$$RD_d(R) = \frac{RDS^i_{S^i(d)}(R)\prod_{j=1}^n CSDR^i_R(s_j^{i-1})}{\int_{R'} RDS^i_{S^i(d)}(R')\prod_{j=1}^n CSDR^i_{R'}(s_j^{i-1})dR'}$$

From RD_d and $CSDR^i$ we calculate the CSD:

$$CSD_d(s) = \int_R RD_d(R)CSDR^i_R(s)dR$$

Then

$$Udesign(d) = V^{i-1}(s_t(CSD_d, V^{i-1}, c^{i-1}))$$

3.6 The HUFS Algorithm

The HUFS algorithm is a best first search where "best" means "largest *Udesign*". We start with a single, top-level design alternative representing the initial problem specifications. At each step, we find the design alternative with the largest *Udesign*, generate one child from it, and compute the child's *Udesign*.

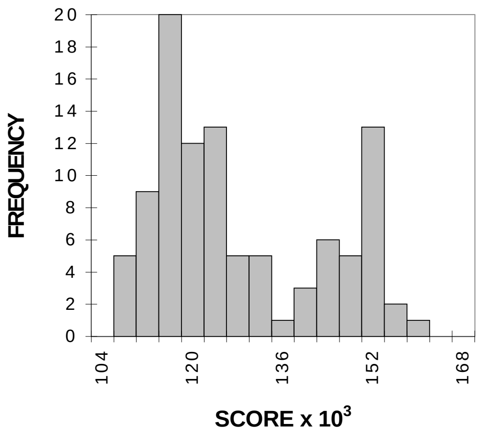
Now that the parent design alternative has a new child, we recompute the Bayesian update of the parent's *RD* using all the child scores including this new one, and compute a revised value, and hence a new *Udesign*, for the parent from the new *RD*. This new value for the parent is used in turn to revise the *RD* of *its* parent, and so on — we propagate the change in value through all the ancestors of the new child.

Note that our formula for the utility of an alternative implicitly assumes that it and the alternatives below it will be designed with a top down search. If we are using HUFS, we may be able to get a higher quality design and/or take less time to do the design, and thus the utility of an alternative will be higher than the value of this formula. Ideally, HUFS should be adjusted to take this into account.

4 Empirical Evaluation

We now turn to the empirical studies we did to evaluate HUFS. There were at least two concerns that we had regarding HUFS that we particularly wanted to investigate. The first and foremost concern was that HUFS depends on knowing the family of distributions that characterize the Child Score Distributions and on knowing *RDS*, i.e., the a priori probability that an alternative with a given score will have a *CSD* characterized by a given vector, *R*, of parameter values. Could we in fact get a good enough statistical model of a realistic problem solver to allow HUFS to work, without having to collect so much data as to make HUFS infeasible? The second concern was whether HUFS would actually result in a significant improvement in the design process, even given the right models.

In order to provide netlists both for calibrating the *CSD*s and *RDS*s and as test data for our experiments, we wrote a program to generate netlists with the modules' heights and widths, the number and location of ports, and the specific interconnections all chosen randomly. All netlists in these tests



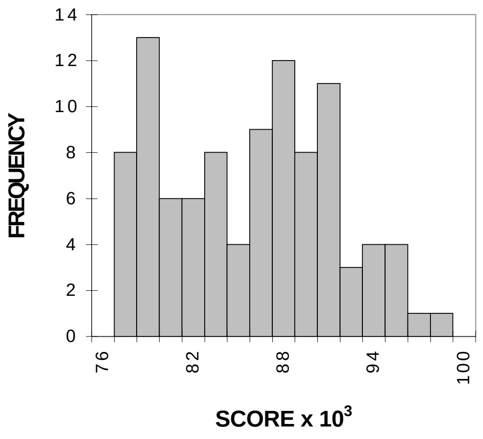


Figure 6: Child Score Distributions for Two Netlists

had 20 modules.

We will first discuss our implementation of HUFS on the example problem described above, then we will discuss the waterfall control method we used as our standard of comparison, and finally the tests we ran and their results.

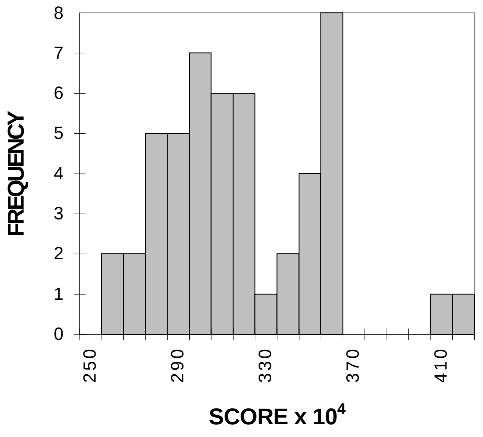
4.1 HUFS for the Placement Problem

To implement HUFS for the placement problem, we needed the costs of the optimizers, the value function for placements, the score functions at all levels, and the *CSDR* and *RDS* for each optimizer. As will be seen, we actually tested HUFS for a range of costs, although we kept the costs of the two levels equal. We rather arbitrarily set $V^{placement}(s) = 10^6 - s$.

The score functions for placements and slicing trees were the ones described above in Section 2. For the score function for netlists, we created a linear function predicting the average score of the slicing trees generated from a netlist from a set of parameters including the average and standard deviation of module area, the number of wires, etc.

To get data to calibrate the coefficients of this function and other functions mentioned below we ran the respective optimizers to generate 30 slicing trees from each of 8 netlists, and 30 placements from each of 8 slicing trees.

The next step in implementing HUFS was to determine what family of distributions would be used to model the Child Score Distributions at each level. Figure 6 shows the actual distribution of child scores for two netlists



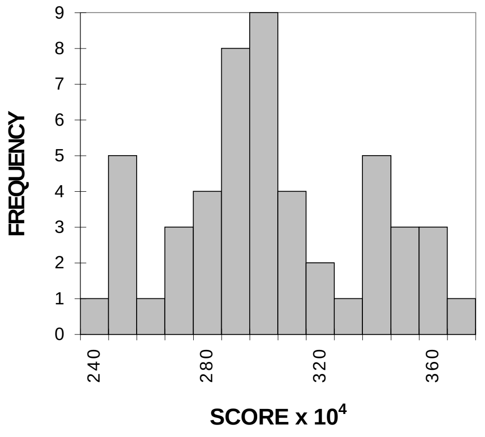


Figure 7: Child Score Distributions for Two Slicing Trees

(i.e., these are the scores of the slicing trees that are the netlists' children.) and Figure 7 shows the actual distribution of child scores for two slicing trees (these are scores for placements). The appropriate family of distributions to use is not immediately obvious. However, in the hope that the details of the distribution would not matter that much, especially in the high-score (low quality) region, we chose to model the distributions with a very simple family we call the "triangle distributions" (Figure 8). These are piecewise linear functions, with three parameters: l, m, and r. The function is a line sloping up from 0 probability at score $m - l$ to a peak probability at score m, and then a line sloping down from there to 0 probability at score $m + r$. The formula is

$$P(s) = \begin{cases} 2 * (s - (m - l))/((r + l) * l) & \text{if } m - l \le s \le m \\ 2 * (m + r - s)/((r + l) * r) & \text{if } m \le s \le m + r \\ 0 & \text{otherwise} \end{cases}$$

From the calibration data we saw no reason not to use normal distributions for the RDS's, so we set the a priori probability of a parameter vector $< l, m, r >$ to

$$RDS(< l, m, r >) = Z(s, lm(s), ld) * Z(s, mm(s), md) * Z(s, rm(s), rd)$$

where $Z(s, m, d)$ is the normal distribution function with mean m and standard deviation d applied to score s. The functions $lm(s)$, etc, are linear functions of s (except for mm of the distribution of placements, which needed a

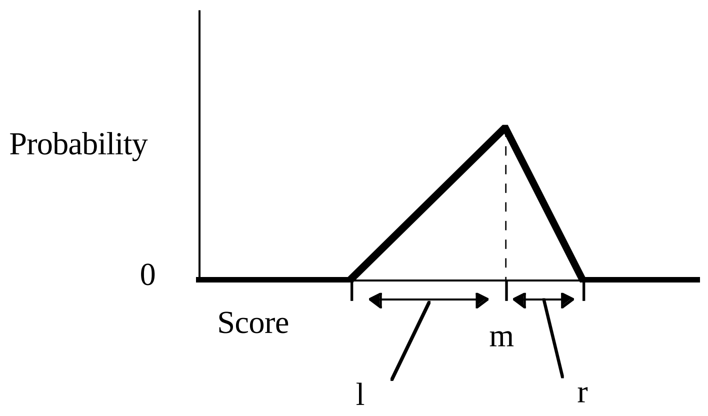


Figure 8: The Triangle Probability Distribution

quadratic function to fit the data well.) The parameters *ld*, etc., are constants. The values for these constants and for the coefficients of the mean functions were determined by fitting to the data we had collected. We then ran HUFS 5 times on each of the 8 netlists, and adjusted the parameters slightly (e.g., we had underestimated *md*). At that point we froze the parameters and proceeded to test HUFS.

HUFS is implemented in Common Lisp and takes about 15 seconds on a Sun Ultrasparc I to update the *RD*s after an optimizer has been run and then to choose the next design alternative to generate children from.

As a standard for comparison we used a waterfall search. This process took a netlist, generated some prespecified number of slicing trees from it, and chose the one that had the lowest score. It then generated the same number of placements from the chosen slicing tree, and chose the placement with the lowest score as its final result. We had the waterfall search generate equal numbers of children at each level because some preliminary experiments indicated that, for a given total number of children generated, the quality of the resulting designs was optimal when the ratio of children at the two levels was roughly one to one, and that the quality was quite insensitive to the precise ratio.

4.2 The Test

To test HUFS we ran it and waterfall on a set of 19 random netlists, which did not include any of the netlists we used for calibration. To save time, the tests were run on pre-generated data. For each netlist we generated 50 slicing trees, and for each of these 50 trees we generated 100 placements. When we ran HUFS or waterfall with this data, instead of calling the optimizer to generate a slicing tree we chose randomly (with replacement) one of the trees we had pre-generated for this netlist, and similarly for generating a placement from a tree.

Using this test data, we tested HUFS for each of 4 different settings of c, the cost per run of the optimizer: 1600, 3200, 6400 and 12800. The setting of 1600, for instance, means that the cost of doing one additional optimizer run would be justified by an increase in the value of our final placement of 1600. Given our $V^{placement}$, this means a decrease in placement score of 1600.

For each setting of c, we ran HUFS 100 times on each netlist, and took both the average score of the 100 resulting placements and also the "95th percentile" scores — the score that was achieved or surpassed by 95 percent of the runs. We believe the 95th percentile score is a more realistic measure than the average score. An engineer normally only designs a given circuit once, so the primary measure of merit should be the quality a tool can be *counted on* to produce each time it is used. We then averaged the 95th percentile scores of the separate netlists to obtain a combined 95th percentile score for the test set, and similarly we averaged the average scores. We did not take the 95th percentile of the separate netlist scores because some netlists are inherently harder than others to place, and it was not clear how adjust for this in determining what the 95th percentile netlist was.

We compared the number of optimizer runs and the resulting scores of HUFS and waterfall in two ways: by asking how many runs waterfall would need to match HUFS' score, and by asking what score waterfall would achieve if it used the same number of runs as HUFS.

To determine how many runs waterfall would need to match HUFS's score, we started with 2 optimizer runs per waterfall trial (i.e., one per level) and did 1000 trials on each of the 19 test netlists. As with HUFS we took the average (over the 19 netlists) of the 95th percentile (over the 1000 runs for a netlist) score. We repeated this with 4 optimizer runs per waterfall (2 per level), then 6, etc., until there were enough runs that waterfall achieved the same overall score that HUFS had gotten. Finally, we re-did the tests using averages in place of 95th percentile scores.

Figure 9 plots the 95th percentile results. Successive points from left to right represent results for the successive values of c, with 1600 on the left. Note that the designs are better, and hence optimizer runs higher, to the

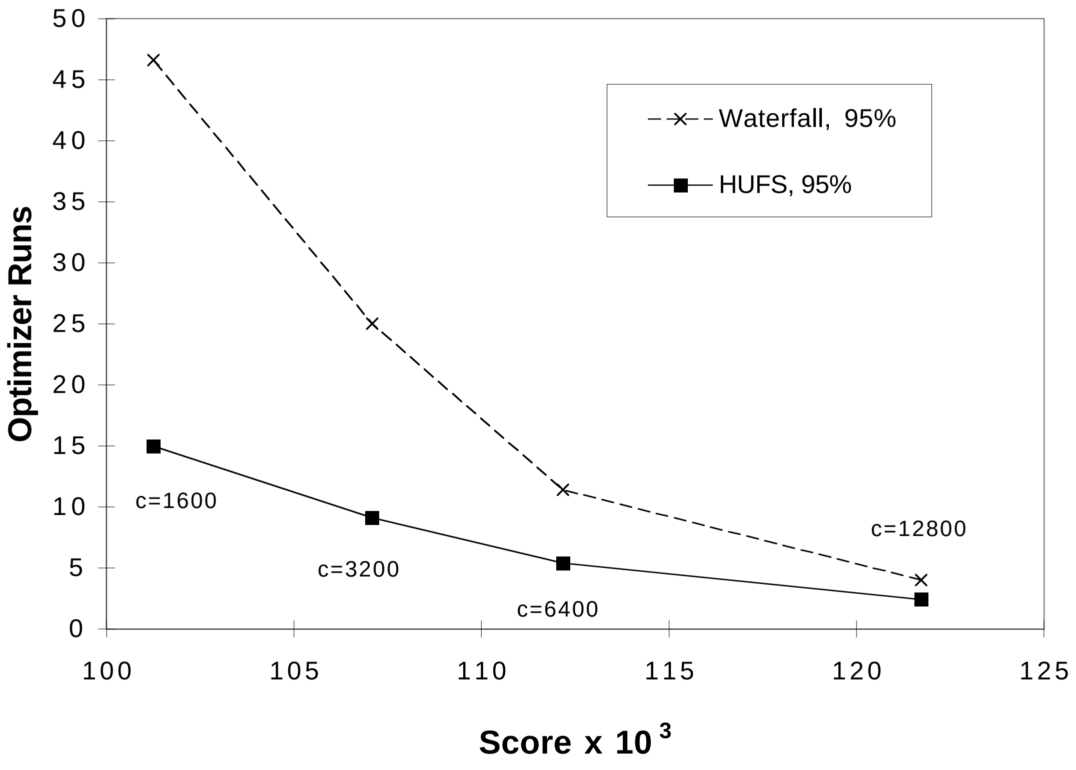


Figure 9: Optimizer runs vs. 95th percentile score for HUFS and waterfall

left.

Table 1 presents the same data. "Std. Dev." for HUFS Score and HUFS Runs is the the standard deviation across the 19 circuits. The column labeled "HUFS/WF" is the ratio of the number of optimizer runs taken by HUFS to those taken by waterfall for the same score.

To determine what score waterfall would achieve if it used the same number of optimizer runs as HUFS, for each trial of HUFS we did a corresponding trial of waterfall. If the HUFS trial did n optimizer runs, the corresponding waterfall trial did $n/2$ optimizer runs at each level. (If n was odd, we made a random choice, with equal probability, between using $n+1$ and $n-1$ in place of n.) Figure 10 plots the 95th percentile scores versus the number of optimizer runs for HUFS and waterfall in this test. Note that in this graph better designs are to the right. Table 1 presents the same data along with the standard deviations. All standard deviations are across the 19 circuits.

c	HUFS Scores		HUFS Runs		Waterfall	HUFS/WF
		Std.		Std.	Runs	Runs
	Average	Dev.	Average	Dev.	Average	
1600	101252.	19653.	15.6	11.5	50.0	0.31
3200	107086.	18629.	8.8	5.0	26.0	0.34
6400	112174.	18943.	5.6	3.4	12.0	0.47
12800	121729.	21842.	2.3	1.1	6.0	0.38

Table 1: Optimizer runs vs. 95th percentile score for HUFS and waterfall

c	HUFS Runs		HUFS Score		Waterfall Score		(WF-HUFS)/WF
		Std.		Std.		Std.	Score
	Average	Dev.	Average	Dev.	Average	Dev.	
1600	15.6	11.5	101252.	19653.	111978.	20921.	0.096
3200	8.8	5.0	107086.	18629.	115111.	20332.	0.070
6400	5.6	3.4	112174.	18943.	123046.	23278.	0.088
12800	2.3	1.1	121729.	21842.	132528.	29912.	0.081

Table 2: 95th percentile score vs. optimizer runs for HUFS and waterfall

The column labeled "(WF-HUFS)/WF" gives the improvement (decrease) in the score of HUFS relative to waterfall as a percentage of the waterfall score.

For the sake of completeness, Tables 3 and 4 give the same data as Tables 1 and 2 but using the average scores across the 100 trials rather than the 95th percentile scores.

As can be seen from the data, if the relative values of design quality and computation time justify even a small number of optimizer runs, HUFS produces an equivalent quality design using 30% to 40% of the optimizer runs

c	HUFS Scores		HUFS Runs		Waterfall	HUFS/WF
		Std.		Std.	Runs	Runs
	Average	Dev.	Average	Dev.	Average	
1600	92523.	18498.	15.6	11.5	26.0	0.60
3200	96272.	18800.	8.8	5.0	12.0	0.73
6400	99978.	18158.	5.6	3.4	8.0	0.70
12800	107958.	21501.	2.3	1.1	4.0	0.57

Table 3: Optimizer runs vs. average score for HUFS and waterfall

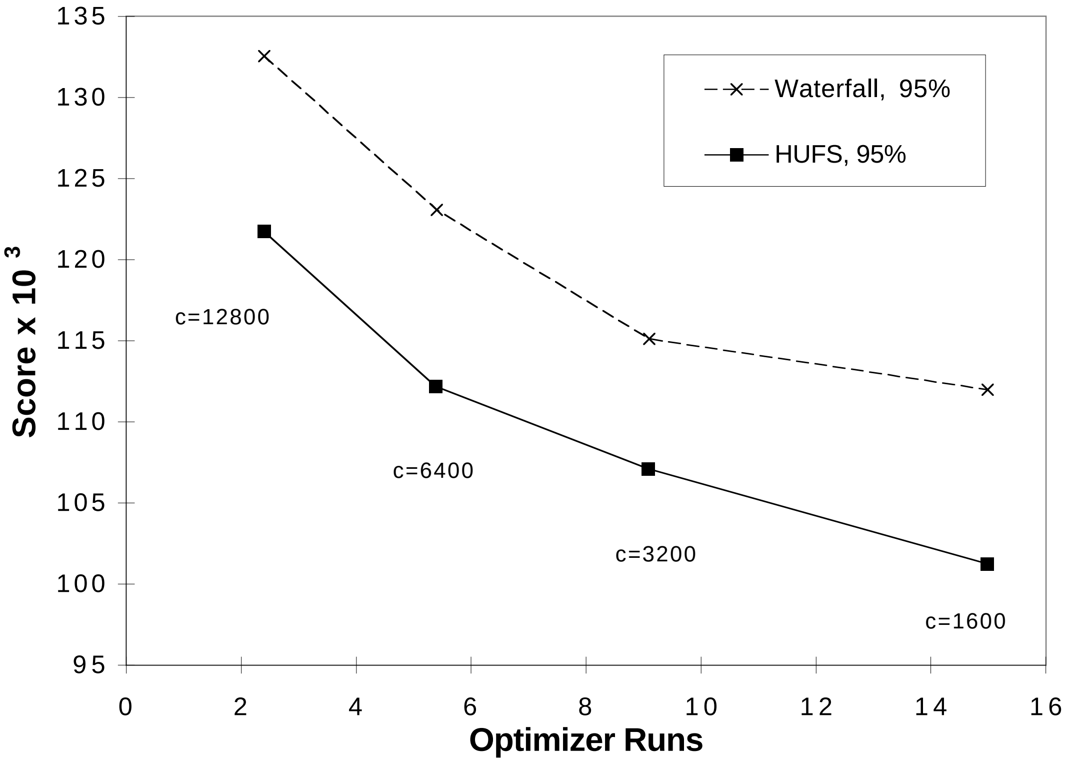


Figure 10: 95th percentile score vs. optimizer runs for HUFS and waterfall

c	HUFS Runs		HUFS Score		Waterfall Score		(WF-HUFS)/WF Score
	Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev.	
1600	15.6	11.5	92523.	18498.	95785.	19681.	0.034
3200	8.8	5.0	96272.	18800.	98958.	20058.	0.027
6400	5.6	3.4	99978.	18158.	102664.	20006.	0.026
12800	2.3	1.1	107958.	21501.	109555.	24796.	0.015

Table 4: Average score vs. optimizer runs for HUFS and waterfall

compared to waterfall. Or looking at it another way, for the same computational effort HUFS produces a score from 7% to 10% better than waterfall. These results demonstrate that, at least for this particular problem, HUFS is a significant improvement over waterfall. Furthermore, HUFS did this well even though we modeled the score distributions as "triangle" distributions, which did not correspond very closely to the actual distributions, and we used few enough optimizer runs in the calibration phase that calibration was very feasible.

5 Discussion

There are several additional issues worth discussing in regard to HUFS. This section will cover those.

5.1 Cost and Scalability of HUFS

It should be noted that our evaluation of HUFS considered only the numbers of optimizer runs and the resulting scores. We did not measure the actual time involved in doing designs with and without HUFS because the time HUFS takes to do its reasoning is unrelated to the cost of the optimizers it calls. Thus, if HUFS generally reduces optimizer calls we can simply apply it to a problem where optimizer run times are long in order to get a situation where total run time is less using HUFS than waterfall. Since HUFS takes only 15 seconds or so per optimizer run, this is easy to do.

This does raise the question, though, of what HUFS' run times *are* related to. The most significant factor is the number of parameters required to determine a specific distribution from the family of distributions we are using to model the *CSD*'s. Since computing the *CSD* from the *RD* requires integrating over the entire set of possible distributions, HUFS takes time that is exponential in the number of parameters, i.e. the number of dimensions over which we must integrate. Fortunately the simple "triangle" family of distributions, which seems to be sufficient at least for our placement problem, has only 3 parameters.

Another factor that affects the run time of HUFS is the number of levels. Each time HUFS updates the *RD* of any alternative, and hence its utility, it must propagate this change to all of the alternative's parents. Thus, HUFS should take time roughly linear in the number of levels. As HUFS is applied to problems with many levels, it may be useful to find strategies that can limit this upward propagation.

5.2 The Models of Value and Time Cost

Another question relevant to the practical usefulness of HUFS is whether it is realistic to assume that the user will be able to supply a value function mapping the score of a bottom-level alternative into its value, along with values for c_i, the costs of the optimizers, in the same units. We would argue that, whatever algorithm (or even intuition) is used to trade off computing time with design quality, such a mapping of optimizer run cost to design value must be made, at least implicitly. And, if the user of HUFS is unable to generate this information in explicit form, these inputs can still be viewed as arbitrary "knobs" that allow the user to control (albeit indirectly) the running time of the overall process.

A more serious problem lies in the assumption of a constant cost per unit of computer time. This is probably a reasonable model for accounting for actual computer cycles, but in many design situations, there is a cost to real time delay, e.g. in its impact on time-to-market, in addition to the cost of the cycles per se. In HUFS the cost of computer time acts as a stand in for all such costs, but for these other costs the true cost per unit time is probably not well modeled by a constant cost per unit time. We are investigating ways to generalize this model.

5.3 HUFS, Genetic Algorithms, and Simulated Annealing

The optimizers we used for our test of HUFS worked via random-restart hill climbing. These optimizers were implemented before we developed HUFS, so they were not specifically designed to make HUFS work well. However, random-restart hill climbing does have an advantage for HUFS over other stochastic methods such as Genetic Algorithms (GA) and Simulated Annealing (SA): both GA and SA methods typically have much less variance in the quality of their results than random-restart hill climbing. Thus, while rerunning a GA or SA optimizer will give different answers each time, the quality of the answers may be so similar that there is little to be gained from multiple runs. On the other hand, these methods invest a great amount of computer time and program complexity in achieving this low variance. If we can simplify a GA or SA so that it has greater variance but takes less time, and use HUFS to handle the variance, the total system may be faster than the more complex GA or SA alone.

6 Related Work

The two bodies of literature that are most relevant to our work on HUFS are the work on utility-based meta-reasoning by Russell and Wefald reported in [Russell and Wefald, 1991] and the work on monitoring anytime algorithms by Zilberstein and colleagues. Another relevant paper is [Etzioni, 1991].

The key points that Russell and Wefald make are that it is often impossible due to time constraints for a problem solver to do all computations that are relevant to the problem it is solving, and therefore it can be useful to reason explicitly (either at run time or at program design time) about the utility of alternate computations, and to use this reasoning to guide the choice of which computations to actually do. They point out that the utility of having a solution to a problem can depend both on the quality of the solution itself and on the delay in getting the solution. They also note that this utility can often be expressed as the difference between an *intrinsic utility* of the solution itself and a *time cost* that accounts for the decreased in utility as delay increases. Both our focus on utility and our formulation of utility as (intrinsic) value minus cost of computation time were inspired by this work.

Russell and Wefald also present applications of their ideas to game-tree search and to problem solving (state-space) search. The problem solving application is more relevant to our work than the game-tree search. They present an algorithm they call DTA* for searching "min-min" trees, i.e. trees where the value of a node is the minimum of the values of its descendants. However, they take an A*-like approach, assuming that a problem statement includes an initial state, a function that can recognize a goal state and a heuristic function that estimates the distance of any given state from the goal. They also assume that the purpose of the search is to find the shortest path from the initial state to the goal state, or at least as short a path as can be found with limited computing time. Our problem does not fit this model. The path from the root of our tree, the initial specifications, to any final design has a fixed length, equal to the number of levels of abstraction, so neither the heuristic function nor the goal of finding a short path is relevant.

Also, rather than assuming we are given a function to recognize a goal state, deciding when we are done is one of the main issues we are concerned with in our analysis.

Hansen and Zilberstein [Hansen and Zilberstein, 1996a], [Hansen and Zilberstein, 1996b] are concerned with *anytime algorithms* [Boddy and Dean, 1994], [Dean and Boddy, 1988]. An anytime algorithm is one that can be stopped after working for a variable amount of time. If it is stopped after working for a short time, it will give lower quality results than if it is stopped after working for a longer time. The single-level problem discussed above, i.e. repeated execution of one stochastic optimizer, is thus an anytime algorithm — if there is more time, more runs can be done and the average quality of the result will be better, and if there is less time fewer runs can be done and the quality will be worse. Two papers, [Hansen and Zilberstein, 1996a] and [Hansen and Zilberstein, 1996b], deal specifically with the issue of monitoring anytime algorithms, i.e. deciding how long to run them. Our problem here is a case of what they refer to as "active monitoring", basing the decision of when to stop in part on what occurs during the process, rather than making the decision before the run starts.

[Hansen and Zilberstein, 1996a] defines the "myopic expected value of computation" (myopic EVC) as "the expected utility of acting on the result that will be available after continuing the algorithm for exactly one more time step minus the expected value of acting immediately on the result currently available." This is equivalent in our terms to $EIV - c$, and their rule for stopping, stop when myopic EVC is negative, is equivalent to our rule, stop when $EIV < c$. However, Hansen and Zilberstein are concerned with the general case of anytime algorithms (and also with the cost of the monitoring, which we do not consider), and thus does not derive any more specific formula for myopic EVC. They also do not consider multi-level systems.

[Zilberstein, 1993] and [Zilberstein and Russell, 1996] primarily deal with composing anytime algorithms into larger anytime algorithms, but also deal with monitoring. They define a stopping rule similar to ours and prove that, under conditions similar to those that hold in our single-level case, it is optimal.

It is worth noting that while repeated stochastic optimization can be seen as an anytime algorithm, HUFS as a whole is not an anytime algorithm. If it is stopped before any ground-level design is produced, then it gives no answer at all. It would be interesting to see if HUFS could be turned into an anytime algorithm; this is related to the issue of our model of time cost discussed above.

Etzioni [Etzioni, 1991] describes an approach to a planning problem that is quite different from our problem here, but he uses a notion called “marginal utility”. Marginal utility is the incremental value divided by the incremental cost, and is analogous to our $EIV - c$ but is based on a model of utility as “return on investment” rather than our model of utility as “profit”. He also includes an interesting learning component to estimate means of distributions for cost and value.

7 Summary

In summary, we have presented a method for control of design systems that work by translating a design down through a hierarchy of abstraction levels, where each step also involves doing a stochastic optimization such as random-restart hill climbing. Since each optimizer, when run multiple times with the same input, gives a set of different outputs, the hierarchy of optimizers implicitly generates a tree of designs. The control problem amounts to the question of how we can efficiently search this tree, in such a way as to optimize the utility of the result, i.e. the value of the final design produced minus the cost of the computation time it took to produce it.

Our control method, Highest Utility First Search (HUFS), is based on a method for estimating, for any design alternative in the tree, what the average utility of our final design will be if we start with this alternative and produce the final design from it. At each point where we must choose a design alternative to translate and optimize, we simply choose the alternative with the highest estimated utility. When this alternative is at the lowest level abstraction level, i.e. is a leaf of the tree, we stop and return this alternative as our result.

We presented HUFS and its implementation for a two-level system that solves the problem of placing circuit modules on a VLSI chip, and showed that HUFS performed significantly better than the waterfall approach of working in a strict top-down, level by level manner.

Finally, we believe the general approach of combining a utility-based analysis with statistical measures such as the *CSD* shows great promise for many kinds of search problems, and we plan to explore the broader application of this approach.

8 Acknowlegements

We would like to thank Saul Amarel, Tom Ellman, Haym Hirsh, and Shlomo Zilberstein for their helpful comments on earlier drafts of this report.

References

[Boddy and Dean, 1994] Boddy, M. and Dean, T. (1994). Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67:245–285.

[Dean and Boddy, 1988] Dean, T. L. and Boddy, M. (1988). An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, Minneapolis, Minnesota.

[Etzioni, 1991] Etzioni, O. (1991). Embedding decision-analytic control in a learning architecture. *Artificial Intelligence*, 49:129–159.

[Hansen and Zilberstein, 1996a] Hansen, E. and Zilberstein, S. (1996a). Monitoring anytime algorithms. *SIGART Bulletin Special Issue on Anytime Algorithms and Deliberation Scheduling*, 7(2):28–33.

[Hansen and Zilberstein, 1996b] Hansen, E. and Zilberstein, S. (1996b). Monitoring the progress of anytime problem-solving. In *Proceedings of the 13th National Conference on Artificial Intelligence*, pages 1229–1234, Portland,Oregon.

[Russell and Wefald, 1991] Russell, S. and Wefald, E. (1991). *Do the Right Thing.* MIT Press.

[Zilberstein, 1993] Zilberstein, S. (1993). *Operational Rationality Through Compilation of Anytime Algorithms.* PhD thesis, University of California at Berkeley.

[Zilberstein and Russell, 1996] Zilberstein, S. and Russell, S. (1996). Optimal composition of real-time systems. *Artificial Intelligence*, 82(1-2):181–213.