Drexel University
Electrical and Computer Engineering Department

ECEC 672 - EDA for VLSI II

Statistical Static Timing Analysis Project

Andrew Sauber, Julian Kemmerer

Implementation Outline

Our implementation consists of the following steps: input file reading and heuristic parameter initialization, gate assignment initialization, SSTA initial run, global cell library sweep, local cell library sweep, and finally critical gate set inversion and cost sweep.

The input file reading step populates all of the major data structures used in our code. The data structures used are discussed in detail within the *Data Structures* section. The gate initialization section makes an initial selection of gates from cell library that are most optimal as evaluated by our cost-delay metric. Metrics are discussed in detail within the *Heuristics Metrics* section. At this time an initial run of SSTA can be completed. In the *Results* section the values produced by just this initial run are referred to as the 'basic' heuristic evaluation and will serve as the comparison point for the additional heuristic algorithms that were implemented. The global and local cell library sweeps in addition to the critical gate set inversion and cost sweep are discussed in detail within the *Heuristic Algorithms* section.

Data Structures

Our implementation makes use of several efficient classes and data structures. The most important aspect of representing the circuit within our program is the straightforward traversal and propagation of statistical timing values. To accomplish this, objects representing both wires and gates are created with linking pointers between the two, thus constructing the layout of the circuit. In addition, each wire and gate is able to store maximum and minimum arrival time distributions within the class. This allows for incremental SSTA to be implemented and subsequently cell library swaps for single gates are quick to evaluate.

In addition to using pointers to traverse the circuit, global vectors of the gate objects and wire objects are available. A gate or wire object allows for reverse lookup of its index in the vector, so that lists of gates (such as local output and input lists) can be generated and passed to into threaded functions. This allows us to avoid using shared memory among threads entirely. When gate types are swapped within a thread, it has it's own copy of the gate-list, which can propagate arrival times via input and output indices.

There are also useful mappings and data points constructed at file read in time. One of which is a hash map of gate operation (NAND, OR, ...etc.) to a list of gates of that type (the key to this hash map can be accessed on a gate-by-gate basis). This eliminates the need for searching the entire gate list each time a specific gate type needs to be swapped. A useful data point calculated is the number of downstream gates that a single gate has. This value is used in calculating metrics mentioned in the *Heuristic Metrics* section. Another optimizing data element is the list of which gates constitute the inputs and outputs of the circuit, so that they can be looped-over for propagation and access of circuit arrival times.

A final important data structure is the gate set. This was a custom implementation of a set (rather than a list), that allowed for set inversion within the universe of the circuit. A gate set is simply a mathematical set that consists of gates. These sets are used to represent accumulating critical path gates for use in heuristic algorithms.

Heuristic Metrics

Five metrics were implemented as part of this SSTA program. One aspect of our analysis is modular heuristics. Wherever a heuristic is used, it is a function call. This function can be easily modified to change the behavior of the entire program.

Delay

This metric's purpose is to reduce the complex canonical probability density function (CPDF) form into a single floating point value. It is important that this delay value take into account both the mean and variability of the delay. That is, a distribution with a low average delay should be penalized for having a large variance - the goal being to minimize both the mean and standard deviation. To test this hypothesis we ran the program with a delay metric that used *both* mean and standard deviation as well as the metric with just mean alone (Eqns 1, 2).

$$Delay = a_0 + \sigma \tag{1}$$

$$Delay = a_0 \tag{2}$$

Cost-Delay

This metric seeks to balance both the cost of a particular gate or circuit implementation with it's associated delay. The most immediate solution to this with the knowledge of wanting both a small delay *and* cost is shown in Equation 3. This equally weights both cost and delay. This is intentionally - the project did not specify a particular desire towards a lower cost or a lower delay. With this metric there is an attempt made to minimize both. To evaluate this decision we used additional cost-delay metrics that were either purely cost or purely delay based (Eqns 4, 5).

$$Cost - Delay = \frac{1}{Cost * Delay} \tag{3}$$

$$Cost - Delay = \frac{1}{Delay} \tag{4}$$

$$Cost - Delay = \frac{1}{Cost} \tag{5}$$

Operation Importance

This metric seeks to qualify how important a certain type of gate is - that is, NAND vs. OR vs. XOR ...etc. This is of particular use in our global cell library sweep where we are changing all gates that are of a certain operation. More details regarding the global cell library sweep are discussed in the *Heuristic Algorithms* section. Operation importance is judged purely on the information contained within the cell library. The following values are considered: number of gates of this type within the circuit, variance in the cost of this gate type, and variance in the delay of this gate type. The reasoning behind considering these values are 1) if there are a large number of these gates in the circuit then an early decision on an cost-delay optimal gate choice will have a large effect on future selections - conversely, if there are very few gates of a specific operation then it is ucost-delay metric over the initial gate selection method. Metrics that include both cost and dnlikely that a global swapping of gate will have a large effect on the circuit (i.e. it is unlikely, from a purely 1 out of many probabilistic standpoint that this type of gate will end up on the critical path of the circuit). 2) if there is a large variance in either the cost or delay then selections from the cell library will have large effects on the resulting cost-delay metric of the circuit. Conversely, if a gate has little variation in cost or delay then the end choice of the gate is relatively inconsequential to the end cost-delay metric.

$$Operation\ Importance = \#Gates * Cost\ Variance * Delay\ Variance \tag{6}$$

Gate Arrival Time Spread

This is a metric that looks to quantify the separation between arrival times at a gate. In a similar fashion as tightness probability is used in finding likely critical paths, spread looks to measure how separated the arrival times are at the inputs to a gate. This being done with the knowledge that highly separated distributions of arrival times are likely to result in larger downstream delays as their maximized resultant distributions will have larger variances. These largely variable distributions will show less of an effect to a change in the gate delay properties. Conversely, a low variation in arrival time would signal that a change in this gate’s delay would have a more directly correlated effect on the maximized arrival time distribution leaving that gate. Mathematically, spread is defined as the variance among arrival times at the input of a gate (with arrival times being quantified as CPDFs evaluated by Eqn 1 or 2).

$$Spread = var(\ Arrival\ Times\ ) \tag{7}$$

Gate Criticalness

This metric serves to evaluate how critical a gate is within the circuit - that is, how likely is this specific gate, if changed, to have a large effect on the cost-delay metric of the circuit. The following values are considered for a specific gate: number of downstream gates, current cost, current delay, and current spread. The reasoning behind these values is 1) a gate with a large number of downstream nodes has a large effect on critical and potentially critical paths within the circuit. 2) If the current cost or delay is very high it is important that this gate be considered for optimization. 3) As mentioned previously, a small spread of arrival times is indicative of a gate that will better ‘propagate a change’ in it’s delay properties.

$$\text{Gate Criticalness} = \#\text{downstream nodes} * \text{cost} * \text{delay} * \text{spread} \tag{8}$$

<u>Heuristic Algorithms</u>

Three algorithms were used in our implementation:

Gate Initialization

This is not considered one of ‘our algorithms’, rather, it is considered the baseline for results comparison. Using the cost-delay metric as defined in Eqn 3, 4, or 5, the optimal gate for the entire operation type is selected. As you will notice, this may seem similar to the global cell library sweep however, no SSTA is run here - the gate choice is based on the cost-delay metric of the gate as described in the cell library not the actual gate’s effect on the circuit as a whole. This is in an attempt to have a ‘middle ground’ starting point for the remaining heuristics.

Global Cell Library Sweep

As mentioned previously, this algorithm looks to make an optimal choice for all gates of a certain operation. The decision is made by evaluating the entire circuit cost-delay metric when each of the possible cell library choices when applied to all gates of that type. This processes is threaded per cell library choice - the options are evaluated in parallel. The order of which gate type is swept first is important and more ‘important’ gates might have larger effects on the circuit. Thus, the importance metric as described earlier - the order of gate operations to be swept across the cell library is decided by the operation importance as calculated during the file read in time (as it relies only on the properties of the cell library and circuit layout). During this process a gate set of all gates that ever appear on a critical path are accumulated (note: critical path is evaluated by tracing back the largest delay values starting at the output gates).

Local Cell Library Sweep

Once the global sweep is complete, a set of critical gates is ready (the gates that have appeared on critical paths). This set will be the first working set for this local cell library sweep iteration. This list of critical gates is now sorted by gate criticalness as defined previously. This allows for the most critical gate in the critical path(s) to be optimized first. The process of optimizing a single gate is similar to that of the global sweep except that only a single gate is being swept across choices within the cell library. It is important to note that this does not require a full re-run of SSTA but instead allows us to make use of our incremental SSTA data captured along the way. Again, this operation is threaded across cell library choices. As these gates are being optimized in their criticalness order, another new set of critical path gates is being accumulated. Once this original set of gates is optimized the difference between the current set and the new set is evaluated. If gates exists after this set difference operation then they are used for the next iteration. If no gates exist, that is, if no new critical paths appeared, then the iterations stop. Iterations also are limited by a user set number - three in our test cases.

Critical Gate Set Inversion

During both the global and local sweeps, a superset of all gates that have ever, at any time, appeared on any critical path is being accumulated. A set inverse operation is performed yielding all gates that have never appeared on any critical path and at time. For each gate in this set a final pass at optimization (in no particular order) is attempted. Again, this operation is threaded across cell library choices. A new cell library choice is only accepted if the resulting circuit cost-delay metric is greater than or equal to the previous value *and* the total circuit cost is lower. This serves as one final push towards reaching the lowest possible cost while avoiding modification of the circuit critical paths.

<u>Results</u>

All relative cost-delay metric values are relative to the basic heuristic value, which is the initial gate assignment technique described above.

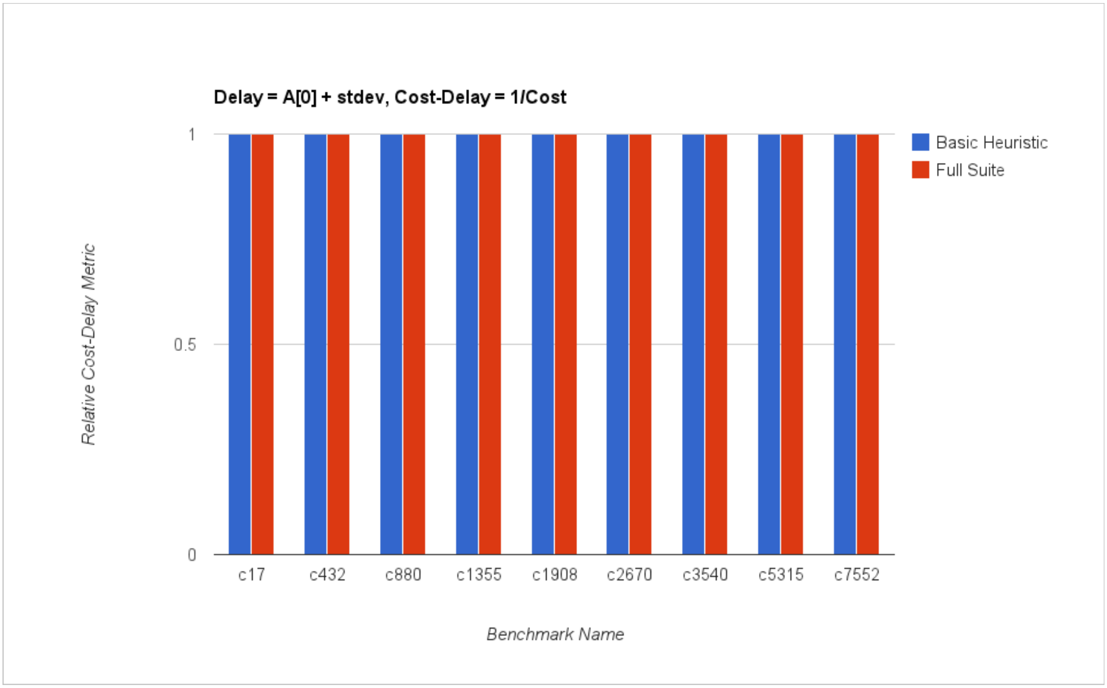

Figure 1. Comparison of the Basic Heuristic vs. Full suite when 1/Cost is the definitive metric. Delay is defined for these runs as A[0] + stdev.

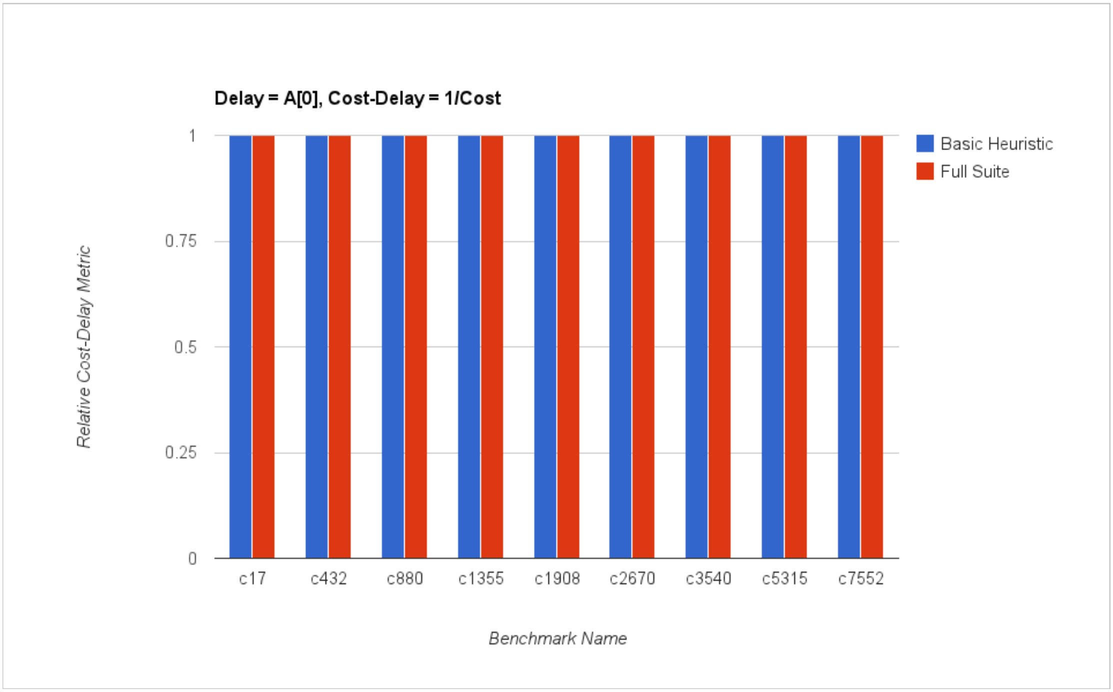

Figure 2. Comparison of the Basic Heuristic vs. Full suite when 1/Cost is the definitive metric. Delay is defined for these runs as A[0].

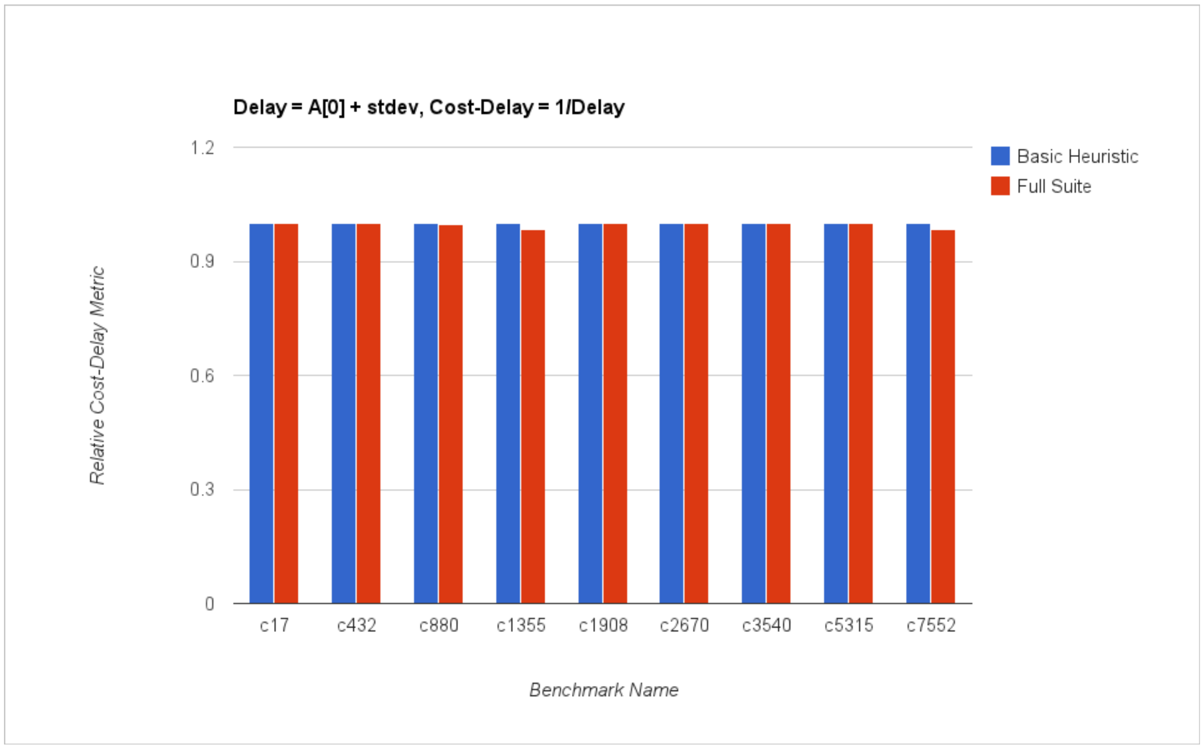

Figure 3. Comparison of the Basic Heuristic vs. Full suite when 1/Delay is definitive metric. Delay is defined for these runs as A[0] + stdev

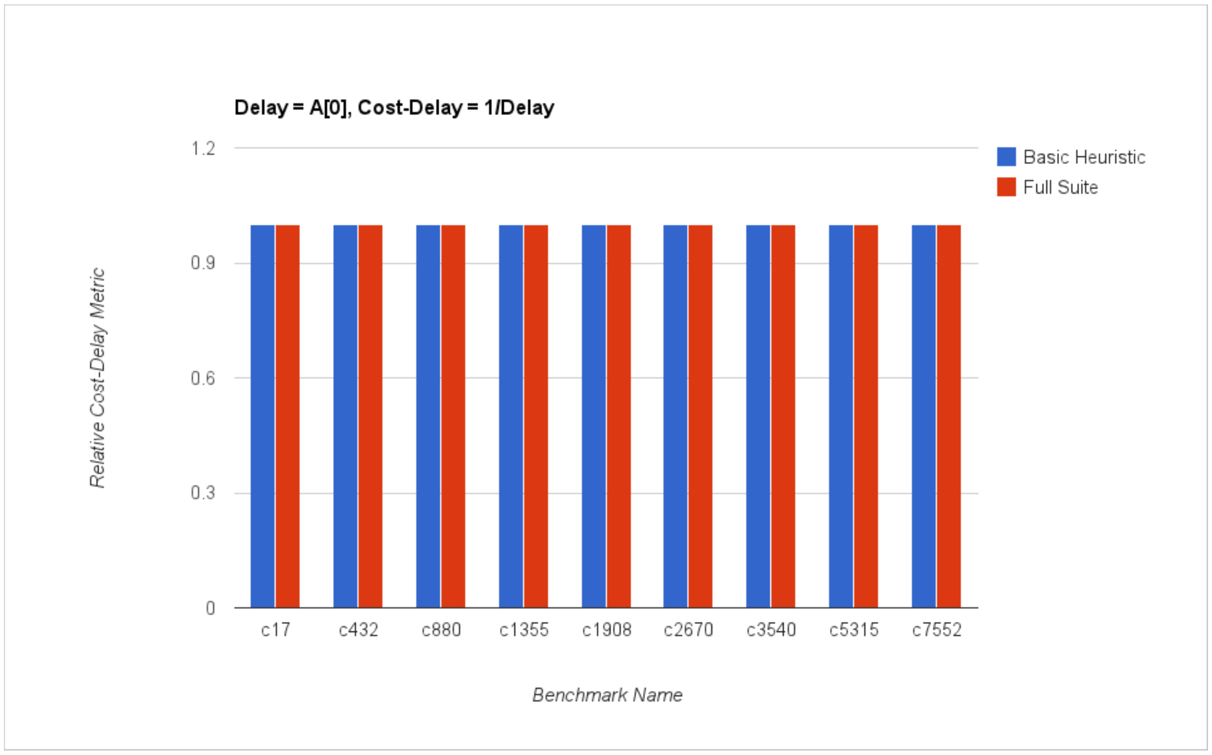

Figure 4. Comparison of the Basic Heuristic vs. Full suite when 1/Delay is definitive metric. Delay is defined for these runs as A[0].

From Figures 1 through 4 the initialization and total circuit cost-delay metric is calculated using Eqns. 4 or 5. The basic heuristic and our complex heuristic report nearly the same values. This is easily explainable: Gate initialization (the basic heuristic) is performed using the same cost-delay metric that is used to optimize the entire circuit. What this means is that when optimizing for cost alone, the task can be easily accomplished immediately after reading the cell library. Without any heuristics at all, one can pick the lowest cost gates for the entire circuit. Conversely, if you wish to optimize for delay alone then simply choose the gates with the lowest delay values. These simple steps are exactly what the basic heuristic accomplishes when initializing gates from the cell library. It is not until a balance between cost and delay is requested through using a cost-delay metric like Eqn. 3 that variations between the basic and complex heuristics emerge as seen in Figures 5 and 6. In short, optimizing for cost or delay alone is an ‘easy’ task and does not lend itself well to heuristic optimizations.

Two small anomalies occur for benchmark c1355 and c7552 when 1/Delay is the metric and Delay is defined as A[0] + stddev. This indicates a difference between using just A[0] to define delay and using A[0] + stddev. When using A[0] + stddev, different gates are chosen as based on “criticalness” during the local gate library sweeps (those being gates that have a higher standard deviation). This actually causes the full heuristic suite to result in a lower heuristic value than the initial assignment. A reason for this decrease in the heuristic could not be identified.

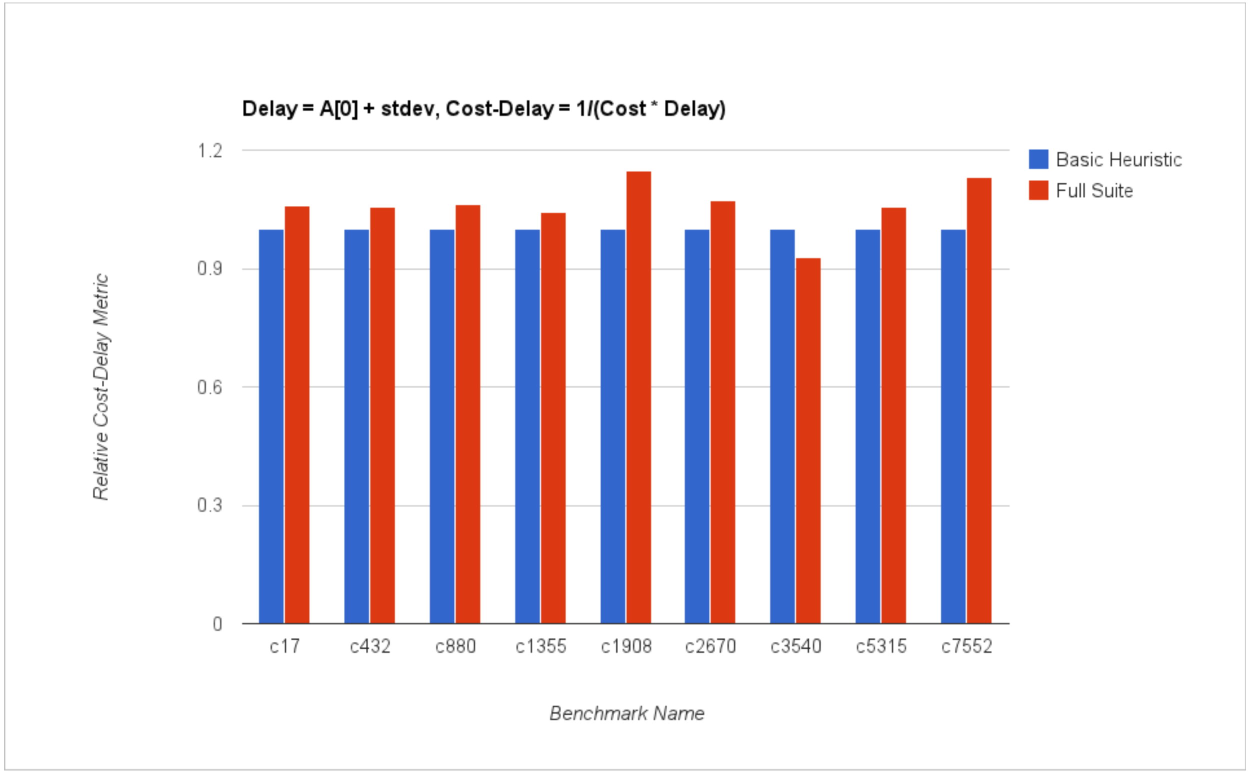

Figure 5. Comparison of the Basic Heuristic vs. Full suite when 1/(Cost*Delay) is the definitive metric. Delay is defined for these runs as A[0] + stdev.

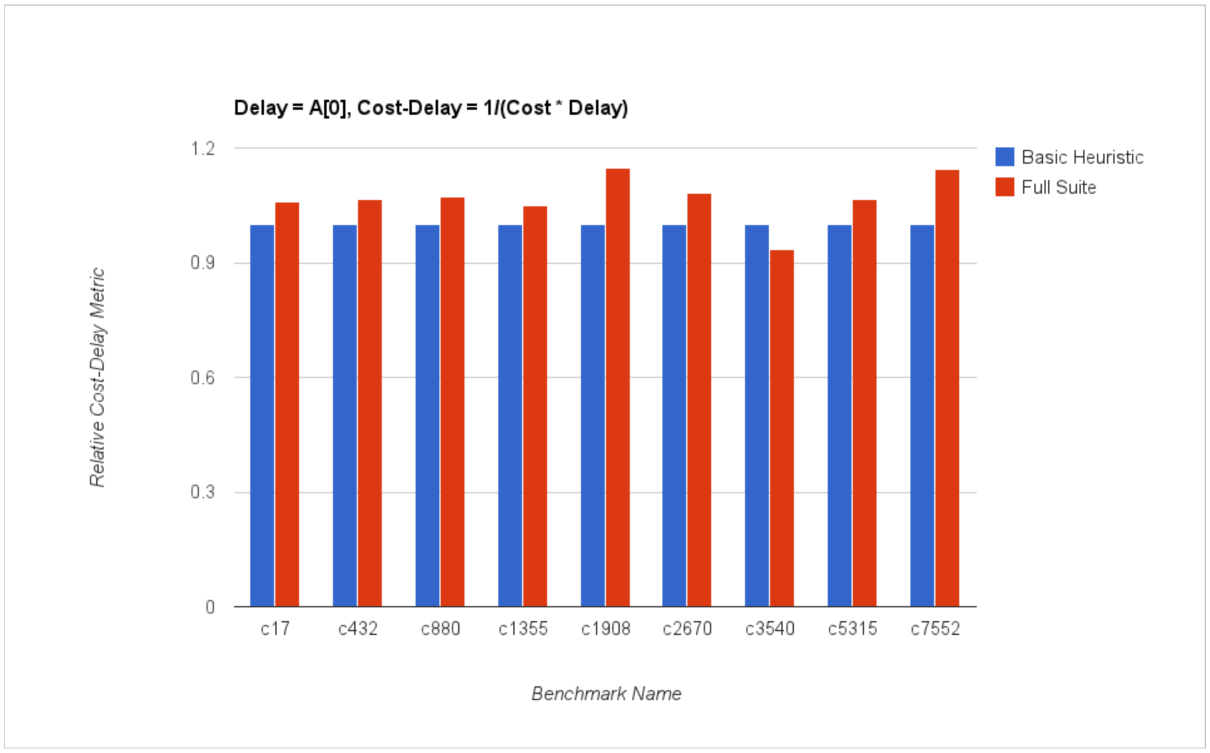

Figure 6. Comparison of the Basic Heuristic vs. Full suite when 1/(Cost*Delay) is the definitive metric. Delay is defined for these runs as A[0].

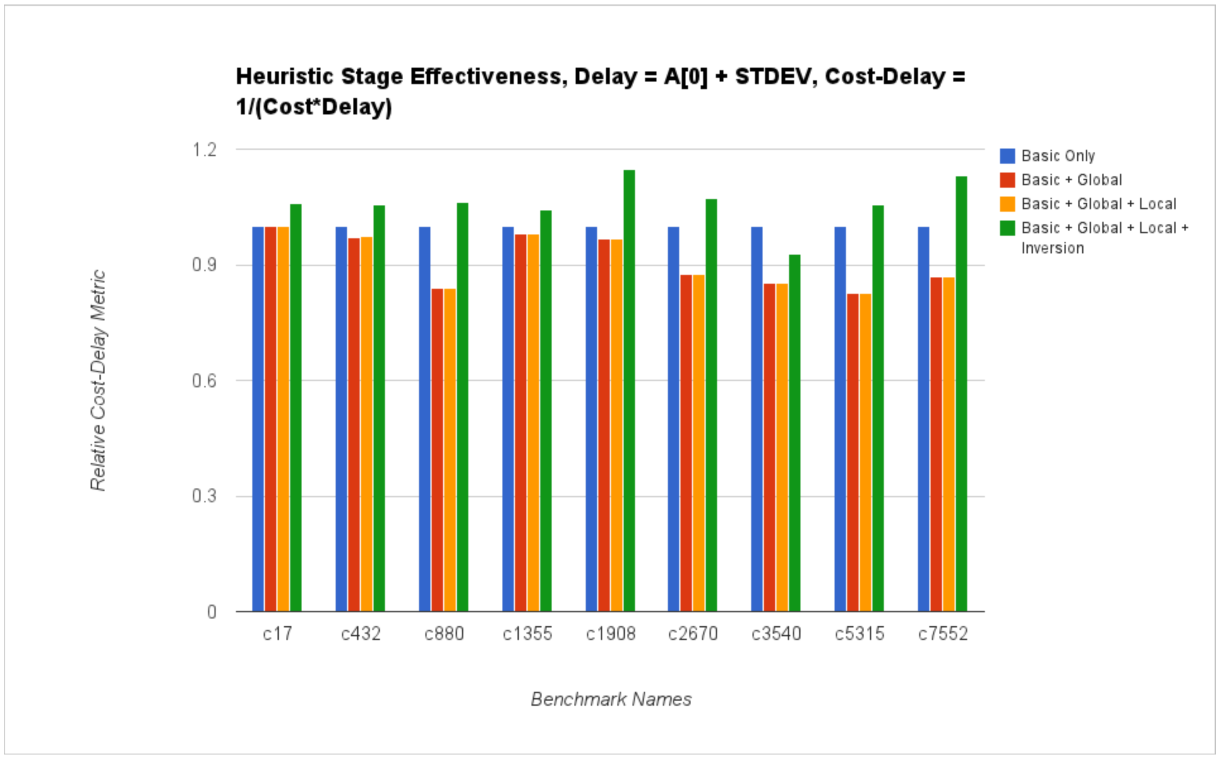

Figure 7. Comparison of the Cost-Delay metric after each stage of the heuristic. Cost delay defined as 1/(Cost*Delay). Delay defined as A[0] + std

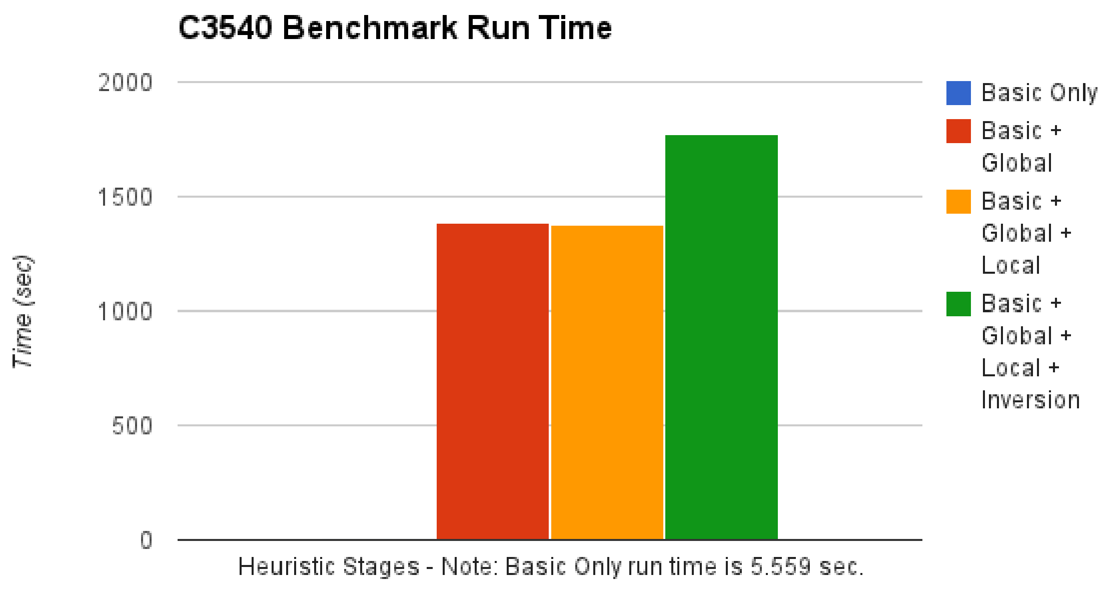

Figure 8. C3540 benchmark run time with varying stages of heuristics enabled. Cost delay defined as 1/(Cost*Delay). Delay defined as A[0] + stdev.

Note: Run time was collected using Intel i5 Quad Core Sandybridge 3.3 GHz PC.

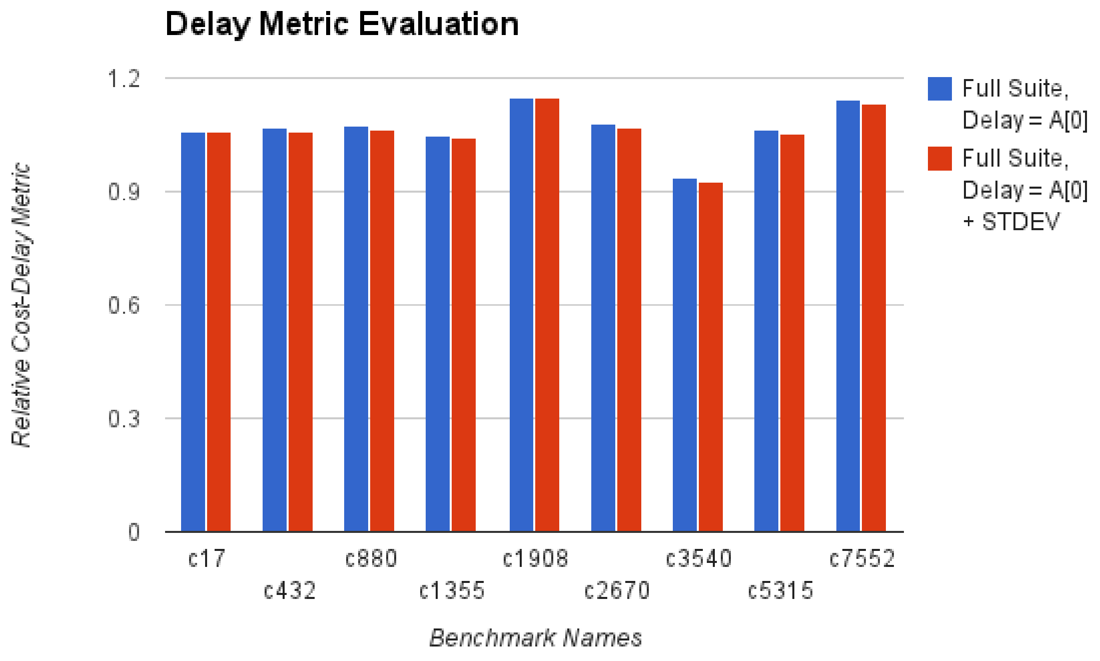

Figure 9. Relative cost-delay metric values with varied delay metric equations. Cost delay defined as 1/(Cost*Delay).

In conclusion, the heuristics were able to produce approximately 6% increase in cost-delay metric over the initial gate selection method. Metrics that include both cost and delay can take any number of forms and even allow for scaling of one parameter over another. This heuristic is of course definable by the user of the SSTA suite that has been created. In addition, the runtime can be tweaked by the number of inversion rounds. Future work would include analysis of various cost *and* delay including cost-delay metrics. Citing Fig. 9, it appears as though better relative cost-delay performance can be obtained by using a delay metric of just a0 rather than one that includes standard deviation. The most profound conclusion to draw from this experiment is that our global and local heuristics do not appear to significantly or positively impact the resulting cost-delay metric as expected. Future work would place a greater emphasis on evaluating and modifying the runtime heuristics. On the other hand, the inversion step improved our cost-delay metric by a large factor. It would appear that the best configuration for use of this SSTA at the current time would be to disable the global and local heuristic, perhaps using them only to generate a critical set of gates to invert. Using C++11 threading mechanisms was also an intriguing experience, and there are opportunities to improve our use of these primitives.