Balancing Structural Hazards and Hardware Cost of Pipelined Processors

Albert E. Casavant
C&C Research Laboratories, NEC USA, Inc.
Princeton, NJ

Abstract

In this paper, a tool to aid pipelined processor instruction set implementation is described. The purpose of the tool is to choose from among design alternatives a design that minimizes overall processor cost. In the proposed cost model, processor cost has two components, the cost of hardware necessary to realize the processor and the cost of degraded performance due to pipeline hazards as compared to an ideal pipelined processor. A previous paper detailed the optimization algorithm. This paper extends these results to handle enclosed pairs of instructions having structural hazards. The extended algorithm can produce an optimal result. This algorithm and several examples are presented.

1.0 Introduction

A major hurdle in pipeline design of programmable processors is pipeline hazards [1]. Often a designer is faced with a decision to accept the performance degradation that a pipeline hazard produces or incur additional hardware cost to prevent the hazard. This paper presents a tool to aid designers to make these decisions. Structural hazards are caused by contention for resources by two or more instructions executing simultaneously in the pipeline. If resources are increased appropriately, contention can be eliminated.

Integer instructions in RISC processors have very few structural hazards, due in large part to equal length implementations of those instructions. The same cannot always be said of floating-point instructions. In moderate performance single chip microprocessors, inexpensive general purpose coprocessor designs, and special purpose designs these instructions have different lengths and generally there are hazards present, as in the R4000 design [2]. CISC designs tend to have more structural hazards than RISC designs.

Very high performance state-of-the-art microprocessors may have enough silicon area to avoid the need to share any hardware in the floating-point unit. Very low performance microprocessors, such as those used in games have low performance floating-point units which may not overlap execution of instructions. Both of these categories of designs have no contention for resources in the pipeline and are not candidates for application of the tool. The tool currently handles in-order issue and in-order completion or out-of-order completion. It is not applicable to processors using dynamic scheduling. It is possible, however, to extend the tool to handle superscalar architectures.

This paper describes a tool called MIST [3] (*M*icroprocessor *I*nstruction *S*et *T*ool) to aid pipelined processor instruction set implementation. The tool should properly be viewed as a design aid because it does not synthesize the hardware, rather it selects from alternate designs that the tool user has provided. Alternatives can be generated automatically using shell scripts from a relatively small number of design templates. Instruction implementation possibilities are totally under the control of the designer and can contain any number of pipeline stages. Enumeration of combinations is practical only for small numbers of instructions having few implementation possibilities. The purpose of the tool is to choose from among user-provided design alternatives a design that minimizes overall processor cost. A design in this case is defined to be the choice of exactly one implementation for all instructions.

Cost has two components: the cost of hardware necessary to realize the processor and any hazard prevention hardware, and the cost of degraded performance due to uncorrected hazards as compared to an ideal pipelined processor. In addition to alternate designs, the tool user provides a trade-off factor representing the relative importance of hardware cost versus degraded performance cost, and the frequency of occurrence of pairs of instructions which can be simultaneously resident in the pipeline for typical programs to be run on the processor. The latter input is generated by compiling benchmark programs and automatically analyzing the resulting assembler code.

Previous work in this general area has concentrated on synthesis rather than analysis. In the Piper system [4] first a scheduling into a fixed number of pipeline stages is performed. Then instructions in the benchmark are possibly reordered and/or additional hardware is added to resolve hazards. No unresolved hazards are allowed to remain.

In the “snapshot method” [5], snapshots of instructions from a benchmark are ordered by frequency of occurrence and hardware in a pipelined processor is incrementally added to satisfy the hardware needs of each snapshot. Difficult to schedule and/or unimportant snapshots are allowed to generate stalls in the pipeline. Limitations of the snapshot method include equal length instructions only and long run times.

Both tools described above require that a floating point add and floating point divide take the same number of pipeline stages to execute. Both tools synthesize the microprocessor data paths. Each new generation of microprocessors has architectural innovations in the data path and stringent timing requirements. These requirements cannot be compromised in a high volume, high performance part such as a microprocessor. Both requirements are incompatible with the approaches advocated in these tools; in fact, such considerations have prevented logic synthesis from making inroads in the critical path of microprocessor data paths. Using the approach described in this paper, the designer has complete control over database design options; the tool helps him/her by efficiently

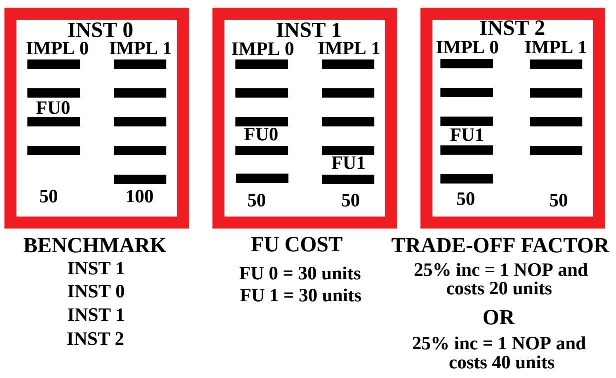

Figure 1. Small Example.

searching the design space (independent of design methodology) for good combinations of implementations which minimize overall cost of the processor.

2.0 Basic operation of MIST

A discussion of the processor model, tool inputs, stall strategy, basic integer programming formulation and constraint generation can be found in [3]. Some of these aspects of the tool will be illustrated instead (due to lack of space) by a small example shown in Fig. 1.

Four types of input data must be supplied by the designer. The first is a description of functional unit (FU) use in each execution stage of each candidate implementation of the instructions in Fig. 1. Shown are 3 instructions with 2 implementations each. Stage boundaries are shown as horizontal lines. MIST uses functional unit residency information to automatically generate all possible structural hazards.

FUs fall into two categories, *sharable* and *non-sharable*. Only sharable FUs can be involved in structural hazards, and these are shown as FU0 and FU1. Non-sharable FUs are assumed to have only one instantiation and thus no hardware contention is possible. The stages shown blank in Fig. 1 are populated with non-sharable FUs.

The second input is the cost of functional units. MIST

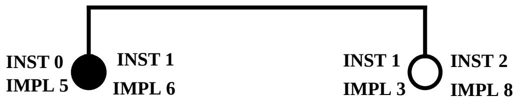

Figure 2. Hazard Edge.

adds up the cost of non-sharable FUs on a per implementation basis, and this cost is added to all hazard nodes associated with that implementation. This cost is shown at the bottom of each box in Fig. 1. The sharable FUs are represented by nodes and end up being variables in the linear programming formulation of the problem.

The third input is a benchmark composed of a machine instruction stream for typical programs which the microprocessor may be required to compute. MIST automatically analyzes this benchmark to determine frequencies of execution of instruction pairs at all feasible execution spacings in the pipeline, e.g. for the instruction pair FADD and FMULT, execution frequency is compiled for execution spacings of 1, 2, etc. up to the maximum possible execution spacing. Execution spacing is the number of clock ticks between instruction initiations in the pipeline.

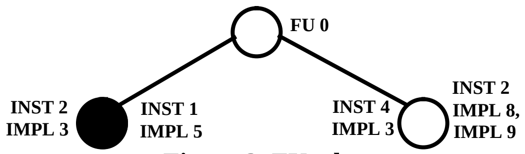

Figure 3. FU edge.

The fourth input is the trade-off factor. For the sake of simplicity in this small example, it is defined to be the cost in hardware units that the designer is willing to pay to prevent the addition of one nop to the benchmark. The trade-off factor puts the costs of pipeline hazards and machine resources on the same scale. In general (as in the results given later), the trade-off factor is defined to be the cost in hardware units required to prevent a one percent increase in benchmark execution time. A high trade-off factor yields relatively high hardware cost and relatively low nop cost, whereas a low trade-off factor yields low hardware cost and high nop cost. If the trade-off factor is set high enough a no stall configuration, i.e. having no structural hazards, can be found if one exists.

The pipelined cost minimization problem can be formulated as a node packing problem [6]: $MAX \sum_{v \varepsilon V} w_v x_v$, $x_u + x_v \leq 1$ for all edges, $x \;\varepsilon \; \{0, 1\}$ for all nodes, where w_v is the weight (cost) of a node. This formulation is called the *edge formulation* of the node packing problem. Nodes (variables in linear programming) represent the presence (value 1) or absence (value 0) of a particular hazard in the solution, or they represent the absence (1) or presence (0) of a particular functional unit.

The active (colored) HAZ node in Fig. 2 represents a hazard between IMPL 5 of INST 0 and IMPL 6 of INST 1, i.e. if implementation 5 of instruction 0 AND implementation 6 of instruction 1 were chosen, there would be a structural hazard. When a node is *active*, its associated variable is a 1 and its cost is charged.

Edges (forming constraints in linear programming) are used for two purposes. Firstly, edges guarantee consistency of implementation choices. The edge in Fig. 2 guarantees that INST 1 may have only one of implementations 3 and 6 in the final solution.

Shown in Fig. 3 is an active (non-colored) FU node, FU0. When an FU node is active its associated variable is a 0 and its cost is charged. Note the contrast between definitions of active for HAZ and FU nodes.

The second use for edges is to guarantee correct cost charging for functional units. If a functional unit is used by any of the implementations represented by a HAZ node, an edge is placed between that node and the FU node. If any of the HAZ nodes using a particular FU are active, the FU node is active i.e. non-colored and its associated variable is 0. In Fig. 3, FU0 is used by at least one of the instruction implementations, say IMPL5 of INST1, and since that HAZ node is active, so is the FU node connected to it.

In the edge formulation, the optimization problem is to find a maximum weighted combination of 1 nodes. For the application presented here, the objective function is algebraically manipulated to form a minimization problem, because cost minimization is desired.

The hazard cost component of a HAZ node is: $\sum_{\text{spacings}} \text{freq} \times \#\text{stalls} \times \text{tradeoff\_factor}$, where *freq* is the frequency of occurrence of an implementation pair in the benchmark and *#stalls* is the severity of the hazard, i.e. how many clocks that the pipeline must be stalled due to the hazard. When appropriate, the non-sharable FU cost for an implementation is added to HAZ nodes, as explained below. The cost of an FU node is the hardware cost of the unit.

The *clique formulation* is equivalent to the edge formulation previously shown: $MAX \sum_{v \varepsilon C} w_v x_v$, $\sum_{v \varepsilon C} x_v \le 1$ for all cliques C, and $x_v \varepsilon \{0, 1\}$ for all nodes. It has been shown to generate fewer fractional answers and is the formulation used by MIST.

Returning to the example, Fig. 1 indicates two possible structural hazards, HAZ1: INST 1 followed by INST 0 involving FU 0, and HAZ2: INST 1 followed by INST 2 involving FU 1. In both cases, the second instruction must be delayed causing a stall.

Fig. 4 shows the corresponding node packing graph for the example in Fig. 1. HAZ nodes are arranged in *banks*. Banks represent all the hazards between pairs of instructions. In Fig. 4, banks are shown as 0 <-> 1 (i.e. hazards between instructions 0 and 1) and 1 <-> 2. Implementations are shown to the right and left of HAZ nodes, their position indicating which instruction of the instruction pair they are associated with. A potential implementation pair may appear in only one hazard node; however, one node may represent several implementation pairs. The node generation code of MIST is responsible for "condensing" multiple hazards into nodes [3]. Since only one node can be active in a solution, a clique is formed for each bank. These edges are shown as medium gray in Fig. 4.

HAZ edges are shown by light gray arcs and FU edges by dark arcs. Costs for nodes are shown in close proximity to the nodes, and in the case of hazard nodes are the sum of hazard cost (shown boxed) and non-sharable hardware cost. Since only one node per bank can be chosen, it is convenient to put non-sharable costs for implementations on these nodes. The best solution is shown by the coloring,

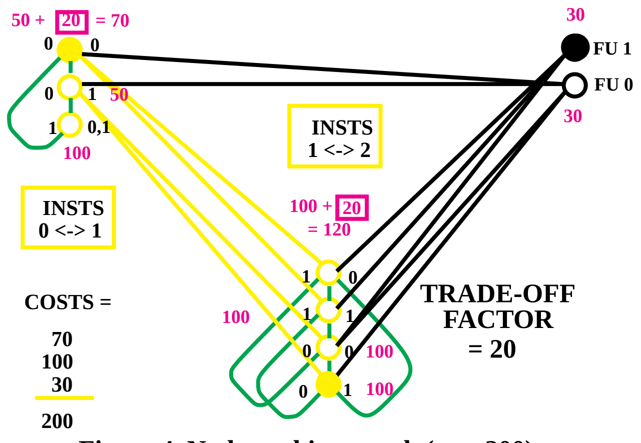

Figure 4. Node packing graph (cost 200).

and the resulting cost of 200 is shown on the lower left. When the trade-off factor is increased to 40 and HAZ node costs are adjusted accordingly, the same coloring yields a cost of 220. However, there is a better solution having cost of 210 but requiring two FUs instead of one. This example illustrates that sometimes it is better to use more functional units to avoid costly hazards. More details on the node packing formulation and constraint generation can be found in [3].

In the basic MIST optimization, many maximal clique constraints are systematically generated knowing the structure of the node packing graph. Linear programming is then employed to find the optimal solution. The solution is not guaranteed to be integer, but computational experience has indicated that is true over 90% of the time. Branch and bound could be used for solving non-integer problems.

3.0 Enclosed Pairs

As pointed out in [3] the basic MIST algorithm has a potentially serious drawback. It gives an optimal solution based on the occurrence frequencies of instructions in the benchmark. Unfortunately, if the optimal solution indicates that there should be some nops inserted into the benchmark, the original occurrence frequencies become inaccurate and thus the solution becomes suspect. What would be desirable is a way to modify the costs of hazard nodes to account for the changes in occurrence frequencies in such a way that the optimization yields an optimal solution with all nops inserted. The solution is then truly optimal. The remainder of the paper is devoted to a discussion of methodologies to accomplish this goal and computational experience using the technique.

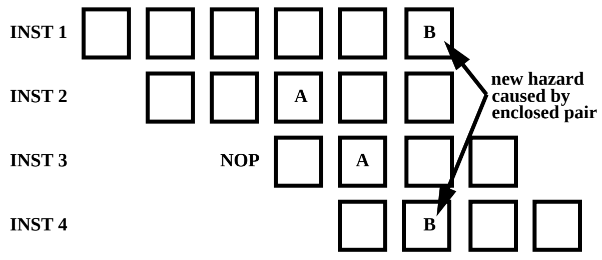

Figure 5. Enclosed Pairs.

An instruction pair is *enclosed* by another instruction pair if that pair appears in the instruction stream between the first and second member of an enclosing pair. Referring to Fig. 5, instruction pair INST 2 - INST 3 initially occurred in sequence with no intervening instructions. MIST determined that there should be a nop inserted between all occurrences of INST 2 followed by INST 3 at a spacing of 1. When the nop is inserted, the spacing between the enclosing pair, INST 1 followed by INST 4 at spacing 3 changed to spacing 4. Now functional unit B is in contention. This possibility was not considered in the original occurrence frequencies and hence the optimization may be in error.

4.0 MIST Extensions

One of the goals of constraint generation is to minimize the number of nodes in the node packing graph, since each node becomes a variable in linear programming. A set

```
DONE = FALSE
FOUND_BETTER = FALSE
ACTIVE_BANK = 0
JAM SOLUTION_NODE in ACTIVE_BANK = 0

WHILE NOT DONE DO
  MIST OPTIMIZATION
  IF SOLUTION_CHANGED
    CALL PLUG_NOPS
    IF FOUND_BETTER
      REMOVE_DOMINATED_COLUMNS
      IF ACTIVE_BANK != 0
        REMOVE ZERO'd COLUMN
      ACTIVE_BANK = 0
  IF ACTIVE_BANK == LAST BANK
    IF INITIALIZE_MODE
      CALL NOP_INSERT || ITERATIVE
    ELSE
      IF PERTURB MODE
        DONE = TRUE
      ELSE /* EXTENDED_PERTURB MODE */
        IF !FOUND_BETTER
          DONE = TRUE
        ELSE
          CALL NOP_INSERT || ITERATIVE
  ELSE
    ACTIVE_BANK++
    SOLUTION_NODE in ACTIVE_BANK = 0
```

Figure 6. Perturb mode.

```
DONE = FALSE
FOUND_BETTER = TRUE
ACTIVE_BANK = 0
JAM FIRST NODE in ACTIVE_BANK to 1

WHILE NOT DONE DO
  MIST OPTIMIZATION
  CALL PLUG_NOPS
  IF BETTER_SOLUTION
    FOUND_BETTER = TRUE
    REMOVE_DOMINATED_COLUMNS
  IF NODES in ACTIVE_BANK !EXHAUSTED
    JAM NEXT NODE in ACTIVE_BANK to 1
  ELSE
    IF NOT LAST SUPERBANK
      SET ACTIVE_BANK = NEXT SUPERBANK
      JAM FIRST NODE in ACTIVE_BANK to 1
    ELSE
      IF FOUND_BETTER
        CALL PERTURB in EXTENDED_PERTURB MODE
      DONE = TRUE
```

Figure 7. Iterative mode.

covering heuristic is used for this purpose. The algorithms which will be presented below may require that the costs of hazards be updated to account for changes in occurrence frequencies. Since constraint generation requires (in many cases) more time than a linear programming optimization, hazards were combined into nodes using a method resulting in scalability with respect to occurrence frequencies. Hazards are combined into a single node if they:

1. have the same hazard cost within some designer specified tolerance, and

2. have the same number of stalls at the same spacings.

Requirement 2 ensures that two or more hazards combined into the same node will continue to have the same cost regardless of how occurrence frequencies change.

The following heuristics all share a common philosophy of perturbing a solution to look for better solutions. Unfortunately this seems to be the most viable solution method given that the costs of hazards between instruction pairs vary erratically with the insertion of nops in enclosed pairs. The insertion of a nop can result in either an increase or decrease in the hazard cost of enclosing pairs.

5.0 PERTURB mode

All optimizations start out with a call to PERTURB (Fig. 6). For nop_insert mode and iterative mode, PERTURB is called in initialize mode and before the NOP_INSERT and ITERATIVE code is called. PERTURB works by jamming 1-valued solution nodes to 0, attempting to find alternate solutions. If an alternate solution is found, PERTURB starts over again on the new solution, otherwise it quits and performs mode-dependent dispatching. Lists of nodes sorted by cost are kept and when a new solution is found, nodes which cannot possibly be in the solution are fathomed by REMOVE_DOMINATED_COLUMNS.

PERTURB first performs a basic MIST optimization. Then the solution obtained is checked against the previous solution. If there has been a change, the nops associated with the new solution are inserted into the benchmark and a new exact cost is determined. Note that this may be different from the cost of the optimal solution found by the basic MIST algorithm because the new cost calculated is the actual true cost in terms of nops added to the benchmark rather than the possibly incorrect summation of node costs.

```
DONE = FALSE

WHILE NOT DONE DO
  MIST OPTIMIZATION
  IF SOLUTION_CHANGED
    CALL PLUG_NOPS
    IF SOLUTION_BETTER
      REMOVE_DOMINATED_COLUMNS
      REINSTATE ORIGINAL NODE COSTS
                for NON-DOMINATED NODES
      CALL PERTURB in EXTENDED_PERTURB MODE;
      DONE = TRUE
    ELSE
      CALL NOP_ANALYSIS
      SORT NOPS BY NOP STRENGTH
      INSERT STRONGEST NOPS IN BENCHMARK
      CALCULATE OCCURRENCE FREQUENCIES
      UPDATE NODE COSTS
```

Figure 8. Nop_insert mode.

6.0 ITERATIVE mode

Iterative mode (Fig. 7) works much like a complementary PERTURB mode. All nodes in selected banks are jammed to a 1 attempting to perturb the solution. If a new solution is found, PERTURB is called again. In Fig. 7, superbank refers to all banks where the second instruction is one greater than the first. See [3] for details.

7.0 NOP_INSERT mode

NOP_INSERT mode is more sophisticated than the previous methods. NOP_ANALYSIS (Fig. 8) is a procedure for grading the necessity of nops. After nops have been inserted into the benchmark by PLUG_NOPS, NOP_ANALYSIS removes each nop in turn and calculates how many instruction pairs are dependent on that nop to prevent their associated hazard. In doing so NOP_ANALYSIS considers all combinations of implementations for instruction pairs. Nops may do multiple duty by preventing more than one hazard by their presence in the benchmark. Each nop in the benchmark is graded and they are sorted by their ability to do multiple duty. A designer determined percentage of nops are permanently added to the benchmark on each iteration. Then new occurrence frequencies are calculated based only on permanent nops, and node costs are updated. In this method, the perturbation is to the benchmark rather than the solution nodes as in the previous methods. The benchmark incrementally evolves into the final benchmark.

8.0 Computational Experience

Table 1 shows problem information for 20 randomly generated problems. A random problem generator is given the following information: 1) the number of instructions involved in hazard generation, 2) the range of numbers of implementations, 3) the number of functional units, 4) the density of hazards, and 5) the trade-off factor. The problem generator randomly chooses a number of implementations in the range given for each of the instructions and then randomly chooses a number of stages in the range given for each implementation. It then randomly populates the implementations with functional units. The density of hazards limits the random generator in its attempts to randomly populate pipeline stages with functional units. A higher density yields stages populated with more FUs

The optimization times shown in Table 1 are given for perturb mode, nop_insert mode and iterative mode respectively. and are in seconds on a SPARC2. The starred times indicate non-optimal answers. The errors shown are the best achievable among all three methods. Most problems could be solved using only the perturbed mode method, but for problems 6 and 10, only iterative mode or nop_insert mode could produce the optimal solution. One problem generated only fractional solutions. The nop_insert mode was always superior or equal to the iterative mode in terms of accuracy and running times. Clearly, on these small problems it is better to exhaustively enumerate.

PROB	#VARS	# CONSTRAINTS	#MATRIX ENTRIES	GEN TIME	OPTIM TIME P / N / I	ERROR
P1	76	583	11,772	1	5 / 6 / 37	0
P2	83	1259	25,036	3	8 / 9 / 15	0
P3	86	1380	29,148	4	15 / 18 / 102	0
P4	210	4837	248,830	33	64 / 82 / 1594	0
P5	210	4608	239,474	33	43 / 50 / 205	0
P6	216	5797	286,144	37	*95 / 186 / 2471	0
P7	83	1259	25,036	3	10 / 33 / 35	0
P8	74	474	9264	2	3 / 5 / 21	0
P9	120	3130	86,078	11	15 / 15 / 21	0
P10	79	539	12,202	3	*3 / 6 / 22	0
P11	262	1206	40,907	44	*18 / *18 / *224	< 1
P12	262	1206	40,907	44	24 / 24 / 251	0
P13	320	13967	377,800	123	6022 / 7322 / 57674	0
P14	309	2970	83,648	92	149 / 324 / 1646	0
P15	309	2767	84,362	91	*28 / *29 / *1574	FRAC
P16	308	2894	80,306	85	*52 / *53 / *96	< 9
P17	305	7727	203,954	92	381 / 428 / 1648	0
P18	312	5885	159,262	92	97 / 98 / 157	0
P19	309	2891	75,938	86	*44 / *62 / *119	< 5
P20	310	4916	139,594	93	430 / 1005 / 6482	0

Table 1: Small Problem Optimization Results

Table 2 shows results from realistic examples. The codes for the example names are as follows: “md” and “lg” are medium and large respectively. “Ld” and “hd” refer to a stall strategy which produces low and high density of hazards respectively. Examples md_hd and md_ld are based on a 4 instruction floating point (fp) unit. The fp add-sub instruction exhibits 4 different add algorithms. This instruction has both inter and intra-instruction sharing of the exponent add and mantissa add hardware and different stage residencies, giving a total of 36 implementations. The fp mult uses 5 basic algorithms in both radix 4 and radix 8. The amount of sharing of carry-save adders is varied yielding different numbers of iterations through the carry-save stage of multiplication. There is also inter and intra-instruction sharing of the exponent add and final carry propagate add of the partial products, and different stage residencies, giving a total of 300 implementations. Radix 2 and radix 4 implementations of SRT division, with intra-instruction sharing of hardware in the basic loop give variation from 3 passes for one of the radix 8 implementations to 24 passes for one of the radix 2 implementations. The final carry propagate add is possibly shared with fp add-sub and/or fp mult. The number of implementations for both fp-div and fp-sqrt is 80.

Example lg_ld has 8 instructions and is a contrived example consisting of a concatenation of two md_ld examples with fewer implementations for each instruction. This gives a large realistic example where the optimal answer is known without requiring enumeration. All of the problems were solved in perturb mode.

PROB	VAR	CNST	MATRIX ENTRIES	COMBINATIONS	ENUM TIME	GEN/OPT TIME	ERR
md_hd	4058	5411	9,796,944	69,120,000	586,627	5768/4193	0
md_ld	1510	27,682	7,129,198	69,120,000	N.A.	12,665/3939	N.A.
lg_ld	759	41566	1,883,446	> 1E14	N.A.	728/15,077	0

Table 2: Large Problem Optimization Results

9.0 Conclusion

MIST provides an effective and efficient solution to the structural hazard problem in pipeline design. To make the tool more comprehensive and attractive to designers it should be expanded to include other architectural features of processor pipelines such as superscalar architectures, branch prediction, and cache considerations. The cost function would consequently need to be expanded to include performance factors other than pipeline hazards.

Data hazards should be considered, and extension of the tool to handle them appears feasible. The cost of control is considered only in the choice of stall strategy - other more direct approaches could be added. The categories of constraints generated should be expanded to make the tool give integer answers more reliably.

10.0 References

[1] J.L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc. 1990.

[2] G. Kane and J. Heinrich. *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ. 1992.

[3] A. E. Casavant, “MIST - A Design Aid for Programmable Pipelined Processors,” in 31st Design Automation Conference, pp. 532-536, 1994.

[4] I. J. Huang and A. M. Despain, “Hardware/Software Resolution of Pipeline Hazards in Pipeline Synthesis of Instruction Set Processors,” *International Conference on Computer Aided Design*, pp. 594-599, 1993.

[5] R.J. Cloutier and D. E. Thomas, “Synthesis of Pipelined Instruction Set Processors,” in *30th Design Automation Conference*, pp. 583-588, 1993.

[6] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*. Wiley Interscience, 1988.