Implementation of a Genetic Hybrid Algorithm for Improvement of Usability with MIPS Parallel Architecture

Michael A. DeLeo
Student of Penn State Erie, The Behrend College
Erie, US
Mad6068@psu.edu

I. Abstract

One of the most frustrating challenges to implementing Parallel Architectures is the flexibility of the system. It's a challenge to achieve a balanced work load for parallel processors, as well as creating a balanced parallel algorithm. This issue stems from the fact that a parallel architecture requires a certain programming style to work efficiently. This style becomes difficult to work with, especially when the system is scaled up. A proposed solution is to use a Genetic Hybrid Algorithm to convert any style of programming instructions into a more processor-friendly program that would be more efficient than using the original instruction set.

II. Introduction

A parallel Architecture is a set of computer architectures that run a given set of instructions simultaneously. This splits the load of the instruction set to reduce the time it takes to pass through [1]. So if there is an program P, let Program P be split into four processes List0, List1, List2, List3. The idea behind this is that for a given process, if it is programmed correctly, portions can be separated and delegated to different processes [1]. The type of programming associated with this is concern-oriented programming [1]. That is, the program is divided up by concerns. For example, four inputs set of A->C is given. A is delegated to List0 process, B to List1 etc.. This treats concerns and processes as orthogonal concepts, because they need to act independently to work efficiently in a parallel processor.

The issue with the way we design software, currently, is that we don't take into account the concept of a process, and how the process deals with it. This is attributed to abstraction. To take away abstraction and to think about how software interacts with hardware is the most attributably reason to why parallel architectures are unfavorable. This is why a genetic Algorithm may be a good alternative to use for any program that is written, versus designing a specific program for a parallel processor, as a genetic algorithm could convert any program into one that is concern-oriented and work with a parallel architecture.

Genetic Algorithms have been proven to be capable in solving impossibly complex problems. A genetic algorithm is an algorithm that models natural phenomena such as genetic competition in Darwin's theory for evolution [2]. Specifically, a "Genetic Hybrid Algorithm" would work best for a parallel programming [2]. A GHA is a genetic algorithm except it has capabilities with non-linear systems [2]. This could be effective because of the parallel nature of the problem. Specifically, a GHA would be implemented when a set of instructions are given, and would be delegated to decide the parameters for stalling, instruction order, and pipeline order in order to map the instructions as closely to one would for concern-oriented programming. This would give parallel architectures more flexibility for programmability.

Some have even created their own languages for dealing with Parallel Architecture Programming. At the Imperial College of Science and Technology, Susan Eisenbach and Chris Sadler view a computer as a machine and each line of code as a moving part that alters the state of the machine [3]. They discussed a programing language called HOPE (named after hope park square at Edinburgh University) created by Dave MacQueen (Bell Labs,) Rod Burstall and Don Sannella (Edinburgh) [3]. Hope was designed for their parallel machine Alice [3]. Hope may be an example that shows we may need to think outside of the box in order to solve the issue with programming for parallel architectures.

III. Problem Formation

The problem with programming instruction sets for parallel architectures is that they require a certain style of programming to work efficiently. There are different types that work, but for this project I focus on concern-oriented programming, which separates the concepts of processes and concerns. The ultimate goal is to one day allow someone to program as they normally would, and then have a parallel architecture compute that instruction set more efficiently than a single processor architecture. For this project, I will be designing a simulation of a four processor MIPS five stage

parallel architecture, and allow the simulation to print out the pipelines to a text file, and take an instruction set input via text file. There will be pseudo code commented for the GHA, it will not be fully implemented for this project.

I will design the simulation in C++, with each processor (Alpha, Bravo, Charlie, and Delta) assigned its own respective set of ten registers. Basic assembly instructions will be supported. The simulation will detect hazards and add stalls as necessary to allow the pipeline to function. Bypassing and Read/Write same cycle will not be implemented in this process.

A final product that I can expect is a program that simulates a four-processor parallel architecture, takes instruction set inputs, outputs a fixed pipeline, and has pseudo code to allow a future project to take on the GHA implementation as a continuation of the project.

IV. Preliminary Work

The features of the simulation I created are an Instruction Set Input via text file, removal of hazards from pipeline, pipeline creation from instruction set, Instruction Set Input via hard coding, assigning of instructions to multiple pipelines, printing singular pipeline with clock cycles via text file, printing of parallel pipelines with clock cycles to text file. The purpose of the simulation is to simulate a four pipeline MIPS parallel processor.

```
std::fstream input;
input.open("/Users/michaeldeleo/Documents/workspace/MIPS Processor/MIPS Processor/ISA.txt");

std::vector<Instruction> file_instructions = getISA(input);
Pipeline C(file_instructions);

input.close();

std::fstream myfile;
myfile.open("/Users/michaeldeleo/Documents/workspace/MIPS Processor/MIPS Processor/Pipe.txt");
C.CheckDepencies();
C.PrintPipeline(myfile);
```

Fig. 2 Method for getting an input set of instructions from a file and printing them as a pipeline

Fig. 6 Example of Single Pipeline Printed

```
.single
addi $t0, $t0, $t1 0
add $t0, $t0, $t0 0
sub $t5, $t0, $t5 0
```

Fig. 5 Example of Instruction Set input

For the input of an Instruction Set from a file reference figure 3. The set of instructions is taken as an input, as shown in figure 2, and turned into the Instructions class type with the Instruction class, then they are pushed into a vector and used to create a pipeline with the pipeline constructor. A constructor is a function that initializes all of the variables in a class when an instance of a class is made [5]. That pipeline is then checked for hazards, fixed and printed out as in Figure 1. The pipeline is printed out as in the format of a 5 stage MIPS Processor Pipeline [4]. An example of a 5 stage parallel architecture output is shown in Figure 10.

When a pipeline is created, it has a set of instructions within, and a pipeline generated from that list. The actual

```
                    t1   t2   t3   t4   t5   t6   t7   t8   t9
  add $a0,$a0,$a1   IF   ID   EX  MEM   WB
stall                    S    S    S    S    S
  add $a0,$a0,$b0             IF   ID   EX  MEM   WB
stall                             S    S    S    S    S
  add $a0,$a0,$c0                  IF   ID   EX  MEM   WB

                    t1   t2   t3   t4   t5
  add $b0,$b1,$b0   IF   ID   EX  MEM   WB

                    t1   t2   t3   t4   t5   t6   t7
  add $c0,$c0,$c1   IF   ID   EX  MEM   WB
stall                    S    S    S    S    S
  add $c0,$c0,$d0             IF   ID   EX  MEM   WB

                    t1   t2   t3   t4   t5
  add $d0,$d0,$d1   IF   ID   EX  MEM   WB
```

Fig. 10 Parallel Pipeline Output

pipeline sequence is the literal set of instructions. CheckDependencies() in Figure 4 goes through the whole pipeline and checks if there are any hazards. Each while loop is used for a different type of hazard (Data, Structural, and Control), Control has not been

```
void Pipeline::CheckDepencies(){
    bool doesNotPass = false;
    while (!doesNotPass){ //while the instructions adds one stall, recheck
        doesNotPass = true; //if there are no dependencies then the loop will break
        for (int i = 0; i < instructs.size(); i++){ //i is first iterator of pipeline y axis
            for (int j = 0; j < instructs.size(); j++){ //j is second iterator of pipeline y axis
                if (i == j || i > j){   //do nothing
                else if (DataisDependent(instructs.at(i), instructs.at(j)) ==RAW && abs(i-j) <= 3){...}
                else if (DataisDependent(instructs.at(i), instructs.at(j)) == WAW && abs(i-j) <= 1){...}
                else if (DataisDependent(instructs.at(i), instructs.at(j)) == WAR && abs(i-j) <= 1){...}
            }
        }
    }
    doesNotPass=false;
    //checking for structural hazards
    while (!doesNotPass){ //while the instructions adds one stall, recheck
        doesNotPass = true; //if there are no dependencies then the loop will break
        for (int i = 0; i < pipe.at(0).size(); i++){    //iterates through x axis of pipeline
            std::vector<stage> column_pipe = whatIsTime(i); //get a cross section of the pipeline
            //iterations through column of pipeline
            for (int k = 0; k < column_pipe.size(); k++){   //iterates through y axis at i of x axis
                for (int l = 0; l < column_pipe.size(); l++){   //second iterator through y axis of i at
                    x axis
                    if (k ==l){}       //nothing
                    else if ( (column_pipe.at(k) == column_pipe.at(l)) && column_pipe.at(l) != BLANK &&
                        column_pipe.at(l) != STALL){...}
                    else if ((column_pipe.at(k) == WB  && column_pipe.at(l) == ID) || (column_pipe.at(k)
                        == ID  && column_pipe.at(l) == WB)){...}
                }
            }
        }
    }

}
```

Fig. 1 The CheckDepencies() function for pipelines

```
DATA_HAZARD Pipeline::DataisDependent(instruction x, instruction y){
    //checks for data hazards
    //Write after read
    if (x.getName() == stall || y.getName() == stall){
        return NO_DATA_HAZARD; //stall
    }
    else if (x.getSource1() == y.getDestination() || x.getSource2() == y.getDestination()){
        return WAR;
    }
    //Write after write
    else if (x.getDestination() == y.getDestination()){
        return WAW;
    }
    //Read after write
    else if (x.getDestination() == y.getSource1() || x.getDestination() == y.getSource2()){
        return RAW;
    }
    return NO_DATA_HAZARD;
}
```

Fig. 4 DataIsDependent() function for checking if two instructions are data dependent

implemented yet.

```
std::vector<stage> Pipeline::whatIsTime(int time){
    std::vector<stage> list;
    for (int i = 0; i < instructs.size(); i++){
        list.push_back(pipe[i][time]);  //pushes back stages going down column #time
    }

    return list;    //returns the instructions for that time
```

Fig. 3 WhatIsTime() is the function for bringing back all the stages at a point in time

If there is a hazard, a stall is inserted at the later instruction, and the pipeline sequence is refreshed. Data Hazards are handled by DataisDependent() in figure 5, and Structure Hazards are handled by WhatIsTime() in Figure 6.

Fig. 7 Instruction Class

```
class Instruction{//no compatability for LW yet
private:
    instruct_type name;
    reg  destination;
    reg source1;
    reg source2;
    int ISA_order;

public:
    Instruction();

    Instruction(instruct_type name, reg destination, reg source1, reg source2, int place)
    {this->name = name; this->destination=destination;this->source1=source1;this->source2=source2;this->
        ISA_order=place;}

    Instruction(instruct_type name) //stall constructor
    {this->name=name;destination=reg_none;source1=reg_none;source2=reg_none;this->ISA_order=-1;}

    instruct_type getName();
    reg getDestination();
    reg getSource1();
    reg getSource2();

};
```

In the simulation, an instruction is a data set that includes an Instruction Type, Destination Register, Two Sources, and a level for the ISA hierarchy. The ISA Hierarchy is used for indicating the level of the instruction when used for a parallel architecture; See Figure 7. A Pipeline is a set of instructions, with a pipelined list of said instructions. See Figure 9 for the pipeline class. A Generation is a set of four pipelines, see figure 8 for the Generation class.

```
class Generation : public Pipeline{
private:
    Pipeline Alpha;
    Pipeline Bravo;
    Pipeline Charlie;
    Pipeline Delta;
    //4 pipelines
    int dependencies;
    int time_Alpha,time_Bravo,time_Charlie,time_Delta;
    int time_elapsed;
public:
    void countDependencies();
    void countPipelineTime();
    void countTime();

    Generation(Pipeline Alpha, Pipeline Bravo, Pipeline Charlie, Pipeline Delta);

    void CheckAllDepencies();


    void PrintGeneration(std::fstream& myfile);

};
```

Fig. 8 Generation Class

```
class Pipeline : public Instruction{
private:
    std::vector<std::vector<stage>> pipe; //pipeline
    std::vector<Instruction> instructs; //List of instructions in pipeline

    void PrintHelper_regs(std::fstream & myfile, reg x);
public:
    Pipeline();
    Pipeline(std::vector<Instruction> x);
    Pipeline(Pipeline const & p);
    void addInstruction(Instruction x, int stage);
    void addStall(int stage);//at time t, and at stage s insert 5 stalls, and
    Instruction whatIsStage(int stage);
    std::vector<stage> whatIsTime(int time);
    void refreshPipeline(); //refreshes pipeline with instructions available

    DATA_HAZARD DataisDependent(Instruction x, Instruction y);     //TODO
    bool StructureisDependent(Instruction x, Instruction y);//TODO
    bool ControlisDependent(Instruction x, Instruction y);//TODO

    void PipelineDependency(Pipeline x);    //checks for depencies on the oth

    void CheckDepencies();

    void PrintPipeline(std::fstream& myfile);
};
```

Fig. 9 Pipeline Class

Added into the functionality of the simulation is the option to send instructions into a single pipeline or superscalar parallel pipeline. This is decided in the beginning of main, but it is implemented in the getISAPipe() function, see figure 10. This splits the instructions into four sets of pipelines. The split occurs with .alpha, .bravo, .charlie, .delta in the input text file. The instructions below these commands are placed into the subsequent pipeline of the generation.

```
////////////////////////////////////////////////////////////////////////
//Precondition: file stream in
//Postcondition: vector vector of instructions
//Function: takes the list of instructions from input, and seperates them into their pipeline
////////////////////////////////////////////////////////////////////////
std::vector<std::vector<Instruction>> getISA_Pipe(std::fstream & input){
    std::vector<Instruction> fullset = getISA(input);
    std::vector<Instruction> a,b,c,d;
    std::vector<std::vector<Instruction>> sets = {a,b,c,d};
    int cursor = 0;
    for (int i = 0; i < fullset.size(); i++){
        if (fullset[i].getName() != none){
            sets[cursor].push_back(fullset[i]);
        }
        else{
            cursor++;
        }
    }
    cursor++;
    return sets;
}
```

Fig. 10 Generation ISA input from file

```
////////////////////////////////////////////////////////////////////////
//Precondition: pipeline
//Postcondition: nothing
//Function: fixes interpipeline dependencies
////////////////////////////////////////////////////////////////////////
void Pipeline::PipelineDependency(Pipeline & x){
    int max_level_p1 = 0;//max_level is maximum level in the hierarchy of instructions inside a
        parallel pipeline
    int max_level_p2 = 0;
    int max_level = 0;
    //iterate through both pipelines and find max level
    for (int i = 0; i < x.instructs.size(); i++){
        Instruction temp = x.instructs[i];
        if (temp.getLevel() > max_level_p2){max_level_p2 = temp.getLevel();}
    }
    for(int i =0; i < instructs.size(); i++){
        Instruction temp = instructs[i];
        if (temp.getLevel() > max_level_p1){max_level_p1 = temp.getLevel();}
    }
    //now we have the maximum level of the hierarchy
    if (max_level_p1 > max_level_p2){max_level = max_level_p1;}
    else{max_level = max_level_p2;}
    //assign the supreme


    /*is any register used in current level being used in any upper level
    BEGINNING
    if no, no hazard
    if yes, there may be a hazard
          check the depencies of next level
                is it being used consequtively
                      if no, then focus on first hazard
                          fix hazard
                        if yes, then focus on first hazard
                            fix hazards until clean

     loop till clean
     increment next level
     check next level for current level depencies

     if next level has surpassed max level of pipeline
        increment current level
            loop back to beginning of checking for hazards. label BEGINNING
     if not, continue with procedure

     if done with current level iteration
        switch current level iterator to other pipeline
```

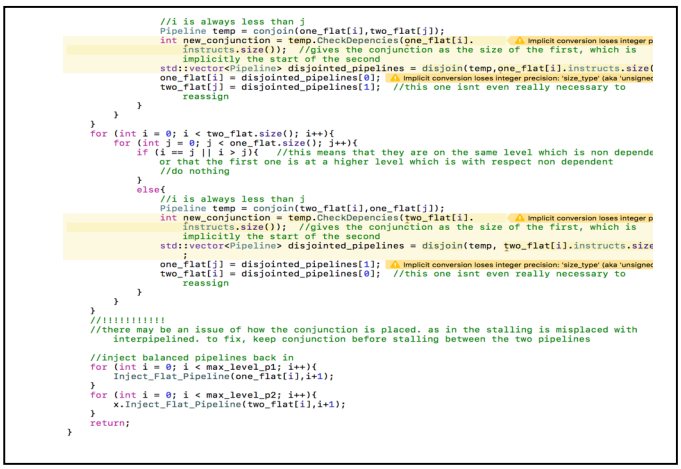
As shown in Figure 11, PipelineDependency() fixes any dependencies caused by another pipeline. The reference of the caller and called is irrelevant, they are both checked. In the first section of the function the maximum instruction level is determined for each pipeline, and then a supreme is found.

```
    */
    std::vector<Pipeline> one_flat;
    std::vector<Pipeline> two_flat;
    for (int i = 0; i < max_level_p1; i++){
        one_flat.push_back(Flatten_Pipeline(i+1));
    }
    for (int i = 0; i < max_level_p2; i++){
        two_flat.push_back(x.Flatten_Pipeline(i+1));
    }
    //stores the levels of the pipelines into two vectors. So each iteration of the vector is a sepera
        level
    //this is an iteration of the y cooordinate, as in the instruction list, of one part of a
    //flat pipeline and a flat part of another pipeline
    //now iterator one: 1 2 ... max   2 3 4 ... max
    //                  two: 0 0 0 ... 0   1 1 1 ... 1
    //continue until the current level is 1 less than the max
    //            one: 0 0 0 ... 0   1 1 1 ... 1
    //            two: 1 2 3 ... max 2 3 4 ... max
    //same here, except the current level is on 'one' pipeline
    //

    //within each iteratation there is a flat pipeline represented by the stage in the vector
    //so from there, check depencies as if the flat pipeline that is higher in level is after the lowe
        level pipeline
    //specific example
    /*Pipeline temp = conjoin(one_flat[0],two_flat[1]);
    int new_conjunction = temp.CheckDepencies(one_flat[0].instructs.size());  //gives the conjunction
        the size of the first, which is implicitly the start of the second
    std::vector<Pipeline> disjointed_pipelines = disjoin(temp, new_conjunction);
    one_flat[0] = disjointed_pipelines[0];
    two_flat[1] = disjointed_pipelines[1];  //this one isnt even really necessary to reassign
    //and now at these two levels they are working
    */

    //assumption that every level has something for the pipeline to do
    for (int i = 0; i < one_flat.size(); i++){
        for (int j = 0; j < two_flat.size(); j++){
            if (i == j || i > j){  //this means that they are on the same level which is non dependen
                or that the first one is at a higher level which is with respect non dependent
                //do nothing
            }
            else{
                //i is always less than j
                Pipeline temp = conjoin(one_flat[i],two_flat[j]);
                int new_conjunction = temp.CheckDepencies(one_flat[i].
                    instructs.size());  //gives the conjunction as the size of the first, which is
                    implicitly the start of the second
```

For the second of the three sections for the PipelineDepency() function. This section gets a pipeline list of N (number of instructions in the pipeline) instructions from L (L being the max level for that pipeline) levels of each pipeline. The reason for this is to build around the concept that there can be many instructions in each level. The comments in this section attempt to prepare the programmer and the reader how the dependencies will be addressed.



For the last section of the PipelineDependency() function refer to figure 12. This section of the function picks out a list of instructions from a level of one pipeline and a list of instructions from a level of the other pipeline, and if the former is at a lower level than the latter then the two lists are conjoined into one pipeline, and then checked for dependencies with existing functions. Then the fixed pipeline is split back into its constituent pipelines, and those are placed back into the pipeline level sets. At the end of the function, the levels are placed back into the original pipelines, by deleting the level existing and inserting the new version.

V. Conclusion

To summarize, I have created a C++ simulation of a MIPS processor that has the functionality of a text assembly input, and a pipeline output. There is also the functionality of using a superscalar parallel pipeline. The simulation can check for dependencies within a pipeline and fix them by inserting stalls. The simulation can also fix inter-pipeline dependencies by inserting stalls with regards to the level of the instruction in a hierarchy.

The purpose of this simulation is to enable someone to implement an algorithm to create a test case in which the simulation has a case which is correct, and the instructions are loaded in a certain way into the generations, and then examined with respect to the test case. The parameter being the level of the instructions, and the goal being to find the quickest superscalar parallel pipeline.

References

[1] F. Carvalho Junior, R. Lins, R. Corrêa and G. Araújo, "Towards an architecture for component-oriented parallel programming", *Concurrency and Computation: Practice and Experience*, vol. 19, no. 5, pp. 697-719, 2007.

[2] R. Östermark, "A multipurpose parallel genetic hybrid algorithm for non-linear non-convex programming problems", *European Journal of Operational Research*, vol. 152, no. 1, pp. 195-214, 2004.

[3] S. Eisenbach and C. Sadler, "Parallel architecture for functional programming", *Information and Software Technology*, vol. 30, no. 6, pp. 355-364, 1988.

[4] D. A. Patterson, J. L. Hennessy, and P. Alexander, *Computer organization and design: the hardware/software interface*. Amsterdam: Elsevier/Morgan Kaufmann, 2018.

[5] P. Prinz and U. Kirch-Prinz, *C: pocket reference*. Beijing: OReilly, 2009.